Universidad Autónoma de Madrid
Escuela Politécnica Superior

Proyecto fin de máster

DISTRIBUTED CLUSTERING FOR VOLATILE SYSTEMS

Máster en Ingeniería Informática

Rafael Vindel Amor
1 de julio de 2016

DISTRIBUTED CLUSTERING FOR VOLATILE SYSTEMS

AUTOR: Rafael Vindel Amor
TUTOR: Héctor Menéndez Benito
Ponente: David Camacho Fernández

Applied Intelligence and Data Analysis
Dpto. de Ingeniera Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
1 de julio de 2016

Resumen

Resumen

Las técnicas de computación distribuida y las técnicas de análisis de información están adquiriendo una gran importancia en los últimos años. Las compañías tienen una gran cantidad de información que necesita ser analizada ahorrando en recursos. Estos recursos pueden ser desde tiempo hasta costes.

Es por esto que han surgido distintas técnicas y distintas herramientas que facilitan el análisis de estos datos. Generalmente, estos sistemas requieren de grandes recursos, a nivel computacional, para llevar a cabo estos análisis. Del mismo modo, disponer de estos recursos suele desembocar en una inversión importante de dinero.

A partir de estas técnicas y herramientas, se ha realizado un sistema de computación volátil altamente configurable. Este sistema permite, mediante una aplicación servidor y un conjunto dinámico de aplicaciones clientes, distribuir operaciones y analizar los datos empleando un determinado algoritmo. La gran ventaja que ofrece el sistema es la incorporación de nuevos clientes de forma dinámica lo que aumenta de forma directa el rendimiento de la aplicación.

Todo esto ha sido desarrollado aplicando metodologías ágiles y principios de patrones de diseño de *software*, asegurando de esta forma una elevada calidad y un sistema robusto y escalable.

Palabras Clave

volátil, efímero, sistema distribuido, algoritmo, k-means, map, reduce, scrum, tdd, solid

Abstract

Distributed computing techniques and information analysis techniques are gaining more importance in recent years. Companies have huge amounts of information that need to be analyzed saving resources. These resources can be from time to money.

Different techniques have emerged and different tools that facilitate the analysis of these data have emerged too. Generally, these systems require large resources, computationally, to carry out the analysis. Similarly, these resources have often led to a significant investment of money.

By using these techniques and tools, a highly configurable volatile system has been developed. This system allows, through a server application and a dynamic set of client applications, distribute operations and analyze data using a certain algorithm. The main advantage of the system is the addition of new clients dynamically increasing performance directly from the application.

All this has been developed using agile methodologies and software principles and software design patterns ensuring high quality and a robust and scalable system.

Key words

volatile, ephimeral, distributed system, algorithm, k-means, map, reduce, scrum, tdd, solid

Agradecimientos

Quiero dar las gracias, una vez más, a mis padres, ya que han sido los que me han apoyado durante todos estos años y los que han hecho que llegue hasta donde me encuentro ahora mismo. Sin ellos, nada de esto habría sido posible. Muchas gracias también a toda mi familia por animarme.

A esa persona especial que siempre está apoyando y dando ánimos. Gracias de nuevo Paty, ya que has estado apoyándome todo este tiempo y me has dado las fuerzas y las ganas necesarias para terminar con esta etapa.

A todos mis amigos. Hemos pasado muy buenos ratos todo este tiempo y han hecho que sea una muy buena etapa en mi vida.

A los compañeros del departamento y; en especial, a Héctor, ya que me ha tenido que aguantar una vez más, esta vez con el trabajo de final de máster. Ha sido un placer poder hacer este trabajo con él.

A Autentia, ya que, gracias a todas las personas que están allí, he aprendido y crecido como persona y como profesional. He tenido también la posibilidad de conocer a mucha gente increíble con las que pasar muy buenos ratos.

Muchas gracias a todos ya que sin vosotros nada de esto hubiese sido posible.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

Los entornos volátiles se encuentran presentes en la mayoría de las redes actuales, especialmente en las redes de comunicaciones. La capacidad de entender dichas redes unida a la capacidad de predecir estos entornos genera una explotación potencial dentro del campo del aprendizaje automático. Esta explotación consiste en emplear dichos entornos como sistemas de cómputo adaptando los algoritmos de aprendizaje automático para sobrepasar las limitaciones que puedan presentar.

El trabajo realizado está orientado en esta dirección. Como entorno volátil, se han empleado las comunicaciones HTTP que utilizan los navegadores a la hora de acceder a páginas web y, sobre esta capa de aplicación, se ha generado una capa de servicios que ofrece un sistema de análisis de datos altamente configurable. En concreto, como caso práctico de uso, se ha desarrollado el algoritmo K-Means, algoritmo que tiene como objetivo la agrupación de un conjunto de datos en función de su proximidad. La ventaja de la que parte este algoritmo es que sus operaciones se pueden distribuir de forma sencilla.

La capa de servicios generada está inspirada en los sistemas de Map-Reduce, desarrollados tanto por Hadoop[1] como por Spark[2]. Esta capa de servicios divide el cómputo en un sistema servidor y en un conjunto de clientes heterogéneos. El servidor realiza las tareas relacionadas con la división de los datos a procesar, a partir de ahora esta tarea será conocida como *splitting*, y la distribución, también conocida como *shuffle*, de los datos a cada uno de los clientes. Mientras tanto, los clientes son los encargados de realizar las operaciones no triviales de los cálculos derivados de Map-Reduce.

De esta forma, la principal carga de computación es delegada a los clientes, dejando las operaciones sencillas al servidor. Por lo tanto, se puede dedicar la mayor parte de los recursos a los clientes, que serán las unidades externas de la aplicación, teniendo un servidor con un consumo bajo de recursos.

Dado que el sistema se define como volátil, se ha optado por una comunicación basada en un intercambio de mensajes asíncronos. Para la realización de este sistema de intercambio de

[1] `http://hadoop.apache.org/`
[2] `http://spark.apache.org/`

mensajes de forma asíncrona, los clientes se comunican con el servidor invocando a una serie de servicios REST, a través de una URL mediante una serie de métodos, implementados en la parte del servidor. Como respuesta a esta llamada, a los clientes se les proporciona una serie de información en formato JSON con las siguientes características: las instrucciones de la tarea a realizar, los datos correspondientes a dicha tarea que se tienen que analizar y, por último, la información del contexto del algoritmo que se está ejecutando. Con esta información, los clientes, que tienen la implementación de las operaciones de *map* y de *reduce* del algoritmo, realizan los cálculos correspondientes y devuelven el resultado al servidor. La respuesta también se produce de forma asíncrona mediante la comunicación con los servicios REST presentes en el servidor. Los clientes siguen procediendo de esta forma hasta que no hay más operaciones disponibles en el servidor, que será cuando la ejecución del algoritmo haya finalizado y se puedan obtener los resultados.

Este sistema supone una primera aproximación a un nuevo modelo de computación donde no es necesario generar ningún tipo de instalación en los clientes, dado que el código se recibe mediante el acceso a una página web con un navegador y el navegador es capaz, por tanto, de interpretar dicho código.

Unido a todo lo anterior, la incorporación de nuevos clientes se produce de forma inmediata, dado que la comunicación entre los clientes y el servidor es vía servicios REST de forma asíncrona. Esto significa que se pueden gestionar de forma dinámica las unidades de cálculo del sistema. Con todo esto, se puede generar un *cluster* con una elevada capacidad de computación que combina todo tipo de dispositivos, incluyendo: teléfonos móviles, ordenadores portátiles, televisores, ordenadores de sobremesa y cualquier tipo de sistema que tenga un sistema capaz de interpretar JavaScript.

Además de todo lo anterior, la implementación basada en Map-Reduce reduce el esfuerzo a la hora de adaptar los algoritmos a un sistema de cómputo basado en paralelización. Finalmente, los experimentos de la actual implementación del algoritmo K-Means, dentro de este sistema, han demostrado resultados equivalentes a los generados por otras implementaciones populares, siendo, a su vez, computacionalmente competitivos. La principal diferencia que existe es que el sistema desarrollado cuenta con una mayor facilidad para añadir una mayor capacidad de computación en forma de nuevos clientes.

Unido a todos estos resultados, se une la metodología de desarrollo que se ha llevado a cabo. Es importante destacar que, a la hora de desarrollar nuevos proyectos, es necesario tener en cuenta, desde el primer momento, la calidad de los mismos. Es por esto que el uso de una buena metodología, que permita dotar de una mayor calidad al proceso, es beneficioso. Se han utilizado metodologías ágiles debido a su alta capacidad de adaptación a los cambios.

Del mismo modo, se han empleado patrones y principios de diseño de *software* con el objetivo de conseguir realizar un proyecto con una elevada calidad, permitiendo así facilitar su mantenimiento y extensión con nueva funcionalidad. La estructura del proyecto también se ha visto beneficiada por la aplicación de estas buenas prácticas.

Por lo tanto, gracias al uso de metodologías ágiles, al uso de principios y patrones de diseño de *software* y a una serie de técnicas de desarrollo guiado por pruebas el resultado final, a parte de ser un proyecto computacionalmente competitivo, también ha resultado ser un proyecto con una gran robustez, presentando buenas capacidades para extender la funcionalidad añadiendo nuevos algoritmos, teniendo un gran nivel de cobertura del código mediante distintos tipos de pruebas y mejorando la calidad.

1.1. Motivación

Los sistemas volátiles y el procesamiento de datos son unos mecanismos que están adquiriendo una gran importancia en los últimos años. Es por esto que han surgido tecnologías dedicadas única y exclusivamente al procesamiento de grandes cantidades de información de forma óptima. De forma adicional, el auge de lo conocido como *Big Data* está suponiendo un cambio en las formas de procesamiento y tratamiento de los datos.

Al existir un mayor número de datos para procesar, es importante descubrir nuevas técnicas que permitan optimizar el uso de recursos con el objetivo de reducir el tiempo y los costes del análisis de dichos datos. Así, el beneficio de todo este proceso será mayor.

Unido a todo esto, las metodologías de desarrollo de productos *software* están cambiando, dando una mayor importancia a las metodologías ágiles sobre las metodologías tradicionales. Debido a este cambio, se quiere profundizar en este tipo de metodologías, aplicándolas directamente al desarrollo de la aplicación para beneficiarse de todos sus aspectos positivos.

De forma adicional, las empresas cada vez son más conscientes de que es necesario realizar proyectos que cumplan con su cometido pero que también cumplan con unos mínimos de calidad. Para esto, existen numerosos patrones y principios de desarrollo de *software* que será necesario aplicar para asegurar la calidad del resultado final.

Con todo esto, la principal motivación de este proyecto no es sólo realizar un proyecto competitivo, a nivel computacional, sino también competitivo a nivel tecnológico que esté dotado de una elevada calidad.

1.2. Objetivos

Los objetivos que se pretenden cubrir con la realización de este trabajo son los siguientes:

- **Diseñar un sistema de computación volátil**. El objetivo principal de este proyecto es la realización de un sistema de computación volátil donde, a través de un servidor, se distribuyan una serie de operaciones que serán procesadas por un número variable de clientes heterogéneos. El principio de distribución de operaciones está basado en Map-Reduce. El diseño del sistema debe contemplar la configuración variable de clientes y la no disponibilidad total de cada uno de ellos. Del mismo modo, no sólo se contemplará la funcionalidad de la aplicación sino también la buena estructura de la misma, la capacidad de escalabilidad del sistema, la robustez del código y la facilidad de añadir nuevos algoritmos.
- **Emplear tecnologías actuales e innovadoras**. Otro de los principales objetivos de este proyecto es la implementación del mismo empleando tecnologías actuales que permitan un desarrollo con un alto valor tecnológico. El proyecto debe ser competitivo dentro de las principales tecnologías empleadas en la actualidad. El uso de tecnologías actuales favorece la evolución de la aplicación. Las tecnologías empleadas, aunque actuales, deberán ser tecnologías con un amplio recorrido y validadas por la comunidad.
- **Emplear una metodología de desarrollo de *software* que mejore la calidad del proceso**. El desarrollo de un proyecto que obtenga buenos resultados no es lo único a tener en cuenta para que sea un proyecto satisfactorio, sino que también es importante emplear una buena metodología de desarrollo que favorezca la calidad del proceso. De

esta forma, el objetivo es conseguir aplicar metodologías ágiles durante la realización del proyecto para, de esta forma, conseguir una mejor calidad en el proyecto.

- **Utilizar principios y patrones de diseño de *software* con el objetivo de implementar una solución mantenible y fácilmente extensible**. Unido al uso de una buena metodología, también se debe mencionar el uso de una serie de herramientas que aseguren que el resultado final sea mantenible y tenga una estructura correcta. Estas herramientas son los principios y patrones de diseño de *software*, que proporcionan una serie de directrices a tener en cuenta durante el desarrollo del proyecto.

- **Emplear herramientas de seguimiento y medición de la calidad del producto**. Para comprobar la calidad del producto obtenido al final del desarrollo, se emplearán herramientas de medición de la calidad con el objetivo de asegurar el resultado final de la aplicación. Estas herramientas están basadas en el análisis estático del código que, mediante una serie de reglas, permiten obtener ciertas métricas con las que clasificar el proyecto en función de su calidad.

- **Aplicar mecanismo de distribución de operaciones mediante Map-Reduce y programación de algoritmos de forma distribuida**. Gracias a la operativa Map-Reduce, la distribución de las operaciones se produce de una forma más sencilla. Utilizando este mecanismo, se implementará de forma distribuida un algoritmo con el objetivo de probar el funcionamiento de la aplicación desarrollada. El algoritmo a implementar será el algoritmo K-Means, cuya implementación de forma distribuida es sencilla.

- **Obtener resultados competitivos con otros sistemas reconocidos**. Una vez diseñada e implementada la solución del sistema, es importante destacar que los resultados obtenidos por el sistema deben ser competitivos con otro tipo de sistemas ya existentes dedicados al mismo campo. El sistema tiene la ventaja de poder dedicar una unidad de poco procesamiento para el análisis puesto que las principales operaciones son realizadas por los clientes. Es por esto que, aunque no sea competitivo en tiempos, puede ser competitivo en cuanto a recursos emplados. Unido a esto, además, la elevada capacidad de configuración de un mayor número de clientes deberá ayudar a conseguir a esta tarea.

2

Trasfondo Teórico

Esta sección tiene como objetivo proporcionar una visión general del trabajo relacionado con los entronos volátiles y las técnicas de *clustering*. Este punto se centra especialmente en *clustering* y una visión teórica de Map-Reduce y de algoritmos genéticos, especialmente los generados para computación efímera.

2.1. Sistemas Volátiles y Sistemas de Computación Efímera

Los entornos o ecosistemas efímeros son descritos como aquellos entornos dinámicos donde las estructuras están en constante cambio. La adaptación del aprendizaje automático a estos entornos, en función de cómo los algoritmos bio-inspirados se adaptan a ellos, es extremadamente útil y tiene una aplicación directa para el análisis de grandes cantidades de datos, como podría ser crear un *cluster* de ordenadores utilizando sus navegadores como clientes.

En la actualidad, todos los dispositivos tienen un navegador web y la red definida por estos navegadores se puede considerar como un sistema efímero, ya que está en constante cambio. Utilizando estos ecosistemas, se necesitan adaptar las técnicas de aprendizaje automático para crear *clusters* de ordenadores más grandes combinando varios dispositivos heterogéneos y, creando así, un sistema de bajo coste distribuido, donde el cliente no necesita ningún proceso de instalación, ya que puede trabajar con código JavaScript. Los principales problemas de adaptación son: los algoritmos básicos de aprendizaje automático, el análisis en tiempo real, las diferentes representaciones de datos (imágenes, textos, redes, ...) y los grandes volúmenes de datos.

Los entornos reales están compuestos por entidades efímeras. Estos sistemas emergentes se basan en individuos que están cambiando constantemente la estructura general. Esto es típico, por ejemplo, en las colonias de bacterias, los animales o las sociedades humanas. Las dos características principales de estos sistemas son:

1. El propio sistema emergente.
2. La volatilidad de dicho sistema.

Estos sistemas han sido analizados en diferentes áreas, tales como: la biología, la psicología y la sociología, entre otras. Desde el punto de vista de la informática, han sido especialmente estudiados en el área de redes, donde la robustez de la red se mide en función de su comportamiento efímero. Recientemente, los sistemas efímeros han sido estudiados desde la perspectiva de la optimización, especialmente enfocado en computación bioinspirada y; más concretamente, en la computación evolutiva [14].

Los algoritmos bio-inspirados imitan un sistema biológico con el fin de optimizar las soluciones de problemas específicos. Por ejemplo, los algoritmos genéticos se utilizan para evolucionar una población de soluciones, guiadas por una función de aptitud, que intenta encontrar soluciones óptimas a un problema determinado. En los últimos años, los sistemas efímeros han sido utilizados con el fin de desarrollar versiones distribuidas de estos algoritmos genéticos [14].

Utilizando la computación efímera como idea base, se podría extender al área del *Machine Learning*, de manera similar a la que los algoritmos de optimización de colonias de hormigas o genéticos se extienden a los algoritmos de aprendizaje automático. Con estas extensiones, se quiere diseñar un paradigma de aprendizaje automático para trabajar en ambientes efímeros y; como consecuencia, se tendría que adaptar el entorno efímero para ser capaz de ejecutar algoritmos de aprendizaje automático. Ésto sería análogo a la extensión Map-Reduce para *Machine Learning* [4], pero centrado en dispositivos heterogéneos, en lugar de los dispositivos homogéneos, como hacen actualmente Hadoop[1], Yarn[2] y Spark[3].

Es por esto que, de forma similar a como Map-Reduce supone un importante paso adelante para el aprendizaje automático, EphemeML tiene como objetivo desarrollar un paradigma que parta de la filosofía de Map-Reduce y la transporte a todos los posibles recursos actuales, tales como: teléfonos móviles, ordenadores, televisores, etc. Éste es el caso de uso principal: la elaboración de un *cluster* en el navegador.

Este caso de uso crea un sistema distribuido de bajo coste sin necesidad de la instalación de *software* en los clientes. Del mismo modo, se podría extender fácilmente a varios dispositivos, ya que los clientes se encuentran conectados a un servicio que distribuye las tareas por las que está compuesto el algoritmo. Debido a la naturaleza efímera y volátil de la solución, cualquier tipo de error que se produzca en el navegador tendrá que ser gestionado. Es por esto que, de la misma forma, también necesita algunas de las características de los algoritmos bio-inspirados.

2.2. El Paradigma Map-Reduce

El análisis de grandes cantidades de datos, como se ha mencionado anteriormente, es un desafío tanto para los algorimos de *clustering*, como para cualquier otro método de minería de datos en general.

Las bases de datos actuales contienen gran cantidad de instancias que necesitan ser analizadas en un tiempo razonable. Los procesos de *clustering* clásicos utilizados para analizar grandes conjuntos de datos han sido adaptados con el fin de mejorar su capacidad de análisis y su escalabilidad. Esta filosofía se ha podido poner en práctica inicialmente gracias al algoritmo de Map-Reduce y a su implementación en Hadoop [18]. Estas herramientas han seguido evolucionando hacia modelos como Spark [22], que mejora varias características de Map-Reduce.

[1] `http://hadoop.apache.org/`
[2] `https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html`
[3] `http://spark.apache.org`

Map-Reduce es un modelo utilizado para procesar y analizar grandes conjuntos de datos utilizando un algoritmo paralelizado y distribuido entre varias máquinas. Este algoritmo se divide en dos etapas principales:

- ***Map***. Los datos de entrada se dividen en subproblemas más pequeños. Estos subproblemas se procesan en paralelo en cada nodo del *cluster*. Una vez que un nodo ha completado el proceso, se envía la respuesta a su nodo maestro. Es importante destacar que cada dato tiene asociada una clave con el objetivo de identificarlo de forma única. De esta forma, es sencilla la distribución de operaciones.
- ***Reduce***. El nodo maestro combina todas las respuestas de los subproblemas para generar la solución final.

Hay varias adaptaciones de los algoritmos de *clustering* en el modelo Map-Reduce, los más relevantes están relacionados con los algoritmos: K-means [23], EM [4] y Spectral Clustering [3].

2.3. Aprendizaje no Supervisado y *Clustering*

El aprendizaje automático, generalmente, se clasifica en cuatro áreas principales, donde cada área tiene una serie de características descritas a continuación:

- **Aprendizaje supervisado**. El proceso de aprendizaje tiene como objetivo obtener un clasificador que imite el conocimiento de un experto.
- **Aprendizaje no supervisado**. El proceso de aprendizaje tiene como objetivo identificar patrones a ciegas.
- **Aprendizaje por refuerzo**. Un agente pretende aprender utilizando un conjunto limitado de información proporcionada durante el proceso de aprendizaje.
- **Aprendizaje basado en teoría de juegos**. Varios agentes tratan de aprender, de forma cooperativa o competitiva, mediante una estrategia de aprendizaje por refuerzo

Este trabajo está enfocado, especialmente, en el aprendizaje no supervisado, aunque se puede generalizar fácilmente al resto de las áreas. El campo no supervisado se divide en diferentes estrategias, donde una de los más famosas es el agrupamiento o *clustering*.

El *clustering* se define como un proceso que pretende maximizar la similitud de los datos agrupados siguiendo un criterio específico. Este proceso se lleva a cabo, por regla general, sin una retroalimentación durante el aprendizaje, sin embargo, se necesita algún tipo de supervisión humana para definir los criterios.

Formalmente, la agrupación se describe como una función de coste que el algoritmo tiene que minimizar. Sea $X = \{x_1, \ldots, x_n\}$ un conjunto de datos con n elementos y $C = \{c_1, \ldots, c_m\}$ el conjunto de los *clusters* donde los datos van a ser agrupados, se podría definir una función de coste J basada en una métrica específica.

Por ejemplo, para la métrica Euclídea, el objetivo se define como:

$$min_C J(X) = \sum_{i=0}^{n} \min_j \{(x_i - c_j)^2\} \tag{2.1}$$

2.4. K-Means y Map-Reduce

K-Means ha sido uno de los primeros algoritmos de *clustering* optimizados a través de la filosofía de Map-Reduce. El algoritmo se escala en las dos etapas principales de K-Means:

- **Asociar los datos a los centroides más cercanos**. Para ello, se toma como clave cada uno de los datos y se asigna, mediante una función para calcular la distancia, al centroide más próximo.
- **Calcular el nuevo centroide**. Para cada uno de los centroides, se obtienen los datos asociados a cada uno y se calcula el nuevo centroide en función de esta información.

La versión del algoritmo de Map-Reduce se divide en tres etapas principales:

1. **Inicialización**. El conjunto de datos se divide en bloques y se establecen los centroides iniciales. Para establecer los centroides se pueden usar distintas técnicas, desde utilizar los propios datos como centroides hasta generar centroides de forma dinámica.
2. **Asignación de datos (*Map*)**. Los datos de cada bloque se asignan al centroide más cercano. En este caso, cada nodo asocia los datos de un bloque y todos los nodos comparten el conjunto de centroides.

Algorithm 1 K-means Mapper

Require: $MB^i = b_1^i, \ldots, b_N^i$ data block and $C = c_1, \ldots, c_k$ centroid
Ensure: $F^i = f_1^i, \ldots, f_N^i$ mapping MB^i to C
1: **for** $b_j^i \in B$ **do**
2: $\quad f_j^i = \text{mín}_k (b_j^i - c_k)^2$
3: **end for**
4: **return** $F^i = f_1^i, \ldots, f_N^i$

3. **Actualización del centroide (*Reduce*)**. Cada nodo recalcula un centroide, recibiendo todos los datos que se han asignado al mismo. Con ésto, se actualiza la posición del centroide.

Algorithm 2 K-means Reducer

Require: $RB^i = b_1^i, \ldots, b_M^i$ data block.
Ensure: c_i centroid.
1: **return** $c_i = \frac{1}{M} \sum_{j=0}^{M} b_j^i$

Después del tercer paso, los centroides establecidos se actualizan y se repiten el segundo y tercer paso hasta que el algoritmo converge o se alcanza un número máximo de iteraciones.

Algorithm 3 K-means Map-Reduce

Require: $X = x_1, \ldots, x_n$ data.
$nBlocks$ number of mapping blocks.
$iterations$ number of iterations.
Ensure: $C = c_1, \ldots, c_k$ centroids.

```
C = sample(X, k)
MB = split(X, nBlocks)
for i = 0 to iterations do
    for MB^i ∈ MB do
        F^i = mapper(MB^i, C)
    end for
    for c^i ∈ C do
        RB^i = suffle(X, F, i)
        c_i = reduce(RB^i)
    end for
end for
return C
```

3

Metodología de Desarrollo

Lo más importante, de cara empezar un nuevo desarrollo, es escoger una buena metodología y un buen conjunto de buenas prácticas que permitan llevar a cabo con éxito el nuevo proyecto.

Antes de nada, se puede definir como metodología al conjunto de procedimientos que se siguen, de forma ordenada, para llevar a cabo un determinado objetivo. De esta forma, las metodologías de desarrollo de *software* son el conjunto de procedimiento que se llevan a cabo para obtener, como resultado final, un determinado proyecto.

En los principios de la informática, los proyectos se desarrollaban sin seguir ninguna metodología. Esto era debido a la novedad del mundo de la informática y a las pocas referencias existentes sobre los distintos procesos que se tenían que llevar a cabo para poder realizar un proyecto de forma organizada, lo que asegura en parte una mayor calidad en el resultado final y un menor número de problemas. Esto provocó, unos años más tarde, que se produjese la denominada **Crisis del *Software*** [19].

La **Crisis del *Software*** es el momento a partir del cual se reconoce que los mecanismos que se estaban utilizando no eran los correctos y que se tenían que tomar medidas para evitar que los futuros desarrollos terminasen en fracaso. A partir de este momento, se empezaron a definir procesos para desarrollar los programas. También, a partir de este momento, aparece la **Ingeniería del *Software*** [20, 5].

La **Ingeniería del *Software*** es la ciencia que tiene como objetivo proporcionar distintos mecanismos para asegurar la realización de los proyectos de una forma sistemática, favoreciendo así el desarrollo de proyectos con una mayor calidad, disminuyendo los costes y el número de errores. A partir de este momento, surgen también las metodologías de desarrollo de *software*.

Las primeras metodologías de desarrollo que surgieron son las denominadas **metodologías tradicionales**. Las **metodologías tradicionales** son aquellas que dan una mayor importancia a la documentación, planificación y gestión de todos los aspectos relacionados con el proyecto en lugar de centrarse en la fase de desarrollo del propio proyecto. Los aspectos sobre los que se ejerce una mayor importancia pueden llegar a ser: los requisitos funcionales, los requisitos no funcionales, la gestión del riesgos, la gestión de la incertidumbre y la gestión de la desviación, entre otros.

Como se ha mencionado anteriormente, una metodología ofrece una serie de fases que se deben cumplir para llevar a cabo el desarrollo del proyecto de forma correcta. Las fases principales por las que están compuestas las metodologías tradicionales son las siguientes:

- **Análisis de requisitos**. Durante esta fase, se obtienen las distintas funcionalidades que deben ser cubiertas por el proyecto. Es muy importante destacar que existen distintos tipos de requisitos: los requisitos funcionales que son aquellos relativos al funcionamiento de la aplicación; y los requisitos no funcionales, que son aquellos relativos a las condiciones sobre las que se va a ejecutar el proyecto.
- **Diseño del sistema**. Durante esta fase es donde se planteará la estructura de la aplicación en cuanto a los módulos por los que está compuesto y el diseño interno de sus componentes.
- **Codificación de la aplicación**. Durante esta fase es donde se implementan cada uno de los requisitos funcionales obtenidos durante la primera fase en función de la arquitectura pensada en la segunda fase.
- **Pruebas**. Durante esta fase es donde se prueba el proyecto para comprobar si se cumplen todos los requisitos, tanto funcionales como no funcionales, que han sido contemplados en la fase inicial.
- **Mantenimiento**. Por último, durante esta fase es donde se ofrecen los desarrollos correctivos sobre la aplicación solucionando aquellos problemas que pueda haber y extendiendo la funcionalidad de la aplicación en caso de que fuese necesario.

Dentro de estas metodologías tradicionales, se pueden encontrar distintas metodologías que se diferencian, principalmente, en el orden en el que se ejecutan las distintas fases mencionadas anteriormente y en el número de veces en el que se ejecutan.

Una de las primeras metodologías es la **metodología de desarrollo en cascada** [16], metodología que está basada en el desarrollo de un proyecto en una serie de fases en las que se pasará de una a otra sin posibilidad de volver atrás. Este proceso se puede observar en la figura 3.1.

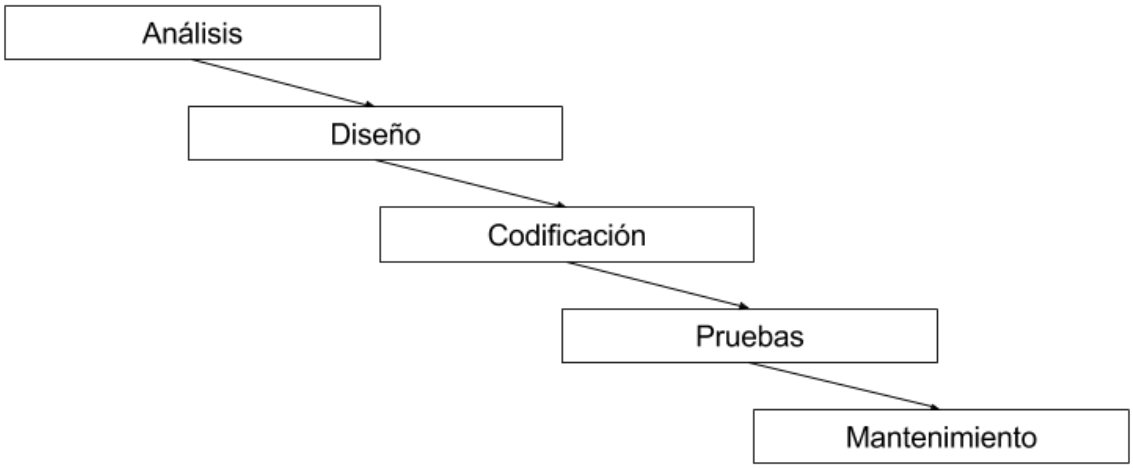
El problema claro que plantea esta metodología es la poca flexibilidad que proporciona y que un fallo en la estimación inicial del proyecto puede desembocar en un fracaso absoluto del mismo.

Una evolución de esta metodología es la **metodología de desarrollo incremental o creciente** [16]. Esta metodología plantea un desarrollo iterativo en el que se van repitiendo una serie de ciclos de desarrollo consistentes en las fases ya vistas anteriormente. Este proceso se puede observar en la figura 3.2.

Esta metodología mejora la anterior ya que permite realizar ciclos en los que, en cada uno, se van realizando mejoras sobre el ciclo anterior. Aún así, también tiene problemas evidentes como son la clara separación de las fases de desarrollo, lo que dificulta la toma de decisiones en el proyecto.

Con el tiempo, aparecen nuevos proyectos de una duración más o menos corta que, con el uso de estas metodologías, fracasaban debido a que gran parte del tiempo se empleaba en temas ajenos a la codificación de la solución final. Esto, unido a la poca formación de los programadores y al poco uso de principios y patrones de diseño, hacía que la calidad del proyecto no fuese suficiente. Con estos proyectos, es necesario realizar un desarrollo más ágil, reduciendo

Figura 3.1: Metodología de desarrollo en cascada



el tiempo en las fases de gestión y planificación y aumentando la interacción con el cliente y la validación del producto. Estas metodologías aportan una mayor interacción con el cliente dándole la opción de priorizar aquellas tareas por las que está compuesto el proyecto.

Del mismo modo, durante la evolución de los lenguajes de programación también aparecen una serie de principios y de patrones de diseño del *software*. Estos principios y patrones de diseño ofrecen herramientas a los programadores para codificar proyectos con una mejor estructura. Esto favorece el desarrollo de proyectos con una mayor calidad, calidad que se hace efectiva a la hora de la realización del mantenimiento de la aplicación y en la disminución en el número de errores.

Unido a esto, también aparecen técnicas que favorecen el desarrollo de productos de elevada a calidad. Técnicas que permiten realizar las aplicaciones robustas que tienen una gran cantidad de código cubierto por pruebas, asegurando de esta forma el correcto funcionamiento de la aplicación.

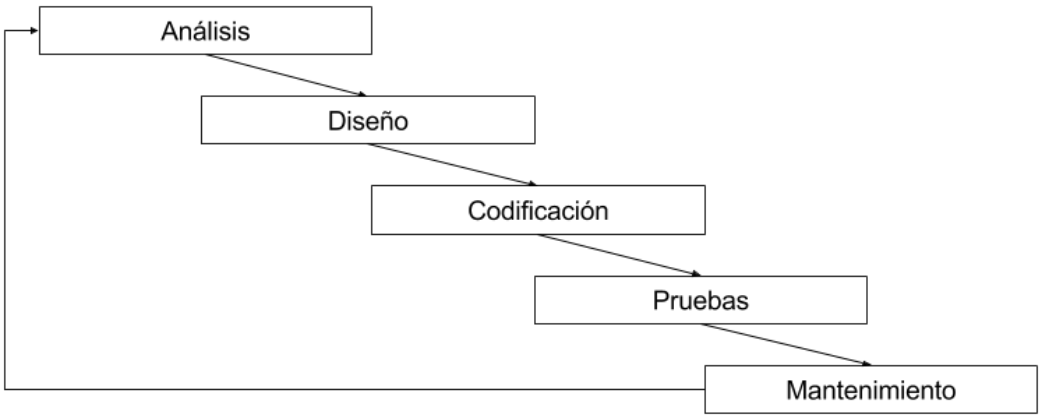
Por último, también aparecen herramientas con las que se puede medir la calidad de los proyectos y así, de esta forma, obtener resultados objetivos e informes de forma automática.

3.1. Metodologías Ágiles: SCRUM

Una de las metodologías ágiles que más impacto han tenido estos últimos años es la metodología de desarrollo SCRUM.

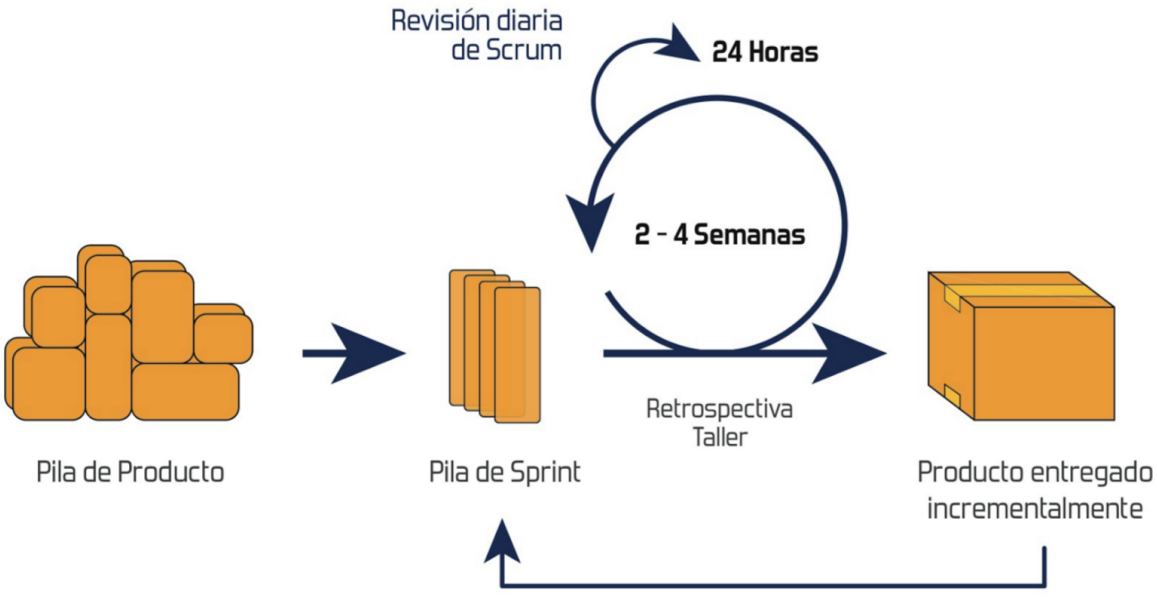
SCRUM [17] es una metodología de desarrollo de *software* iterativa e incremental donde el producto se divide en las denominadas historias de usuario. Estas historias de usuario son organizadas y ordenadas en función de su prioridad de tal forma que en cada una de las iteraciones se van desarrollando las historias de usuario de mayor prioridad. De esta forma, el producto resultante al final de cada iteración tiene la funcionalidad más prioritaria desechando así la funcionalidad menos importante hasta las últimas iteraciones.

Figura 3.2: Metodología de desarrollo iterativa



SCRUM favorece el desarrollo de productos con mayor calidad debido a la forma en la que se organiza el desarrollo y a las reuniones de seguimiento que se producen a lo largo del mismo. Unido a esto, los equipos de SCRUM están dotados de unos conocimientos mucho mayores en varios campos, no sólo en el campo de la programación sino también en campos de gestión y diseño. Este ciclo del proceso de desarrollo de puede observar en la figura 3.3.

Figura 3.3: Ciclo de desarrollo de SCRUM



En el siguiente apartado, se van a mostrar las distintas fases por las que está compuesta esta metodología y por qué estas fases aportan una mayor calidad al proceso y al producto final desarrollado.

3.1.1. Definición del *Product Backlog*

En esta primera fase es donde se van a definir todas las historias de usuario por las que estará formado el proyecto. Las historias de usuario no son otra cosa más que las funcionalidades que se deben desarrollar en el proyecto orientadas a cada uno de los usuarios de la aplicación.

Estas historias de usuario están ordenadas en función de su prioridad. De esta forma, las primeras historias de usuario serán abordadas antes que aquellas con una menor prioridad; es decir, aquellas que aporten un mayor valor al negocio. Esta ordenación por prioridad es realizada por el *Product Owner*.

Al ser el *Product Owner* el encargado de realizar esta ordenación, y siendo el *Product Owner* el cliente, se asegura que el proyecto va a tener siempre la funcionalidad más deseable realizada lo antes posible. De esta forma, si al final el desarrollo no se han completado todas las historias de usuario, se sabe que las historias de usuario más prioritarias sí estarán realizadas, aportando un mayor valor el producto final.

Ésta es una de las diferencias principales con las metodologías tradicionales. Las metodologías tradicionales no plantean una organización de las tareas sino que se van acometiendo, generalmente, por capas. Al acometer las tareas por capas; es decir, primero implementando los accesos a base de datos, luego la lógica de negocio y, por último, la interfaz de la aplicación, es posible que se llegue al final del tiempo del proyecto y no haya nada totalmente funcional.

3.1.2. *Sprint Planning*

El *Sprint Planning* es la reunión de planificación inicial que se produce en cada una de las iteraciones. En este caso, a cada una de las iteraciones se les denomina *sprint*. En esta reunión se definirán las historias de usuario que serán acometidas durante la iteración. Las historias de usuario serán aquellas con mayor prioridad dentro del *backlog*.

Antes de empezar con la realización de cada una de las historias de usuario, se definen los criterios de aceptación de cada una de las historias de usuario. Los criterios de aceptación se pueden definir como las condiciones que se tienen que cumplir para que la historia de usuario sea considerada como válida. Estos criterios de aceptación son propuestos por el *Product Owner*.

Asimismo, el equipo de desarrollo también se compromete a desarrollar esas historias de usuario en el tiempo propuesto para ello. Es importante escoger un número de historias de usuario suficiente como para garantizar una ligera presión y evitar, sobretodo, la relajación del equipo de desarrollo.

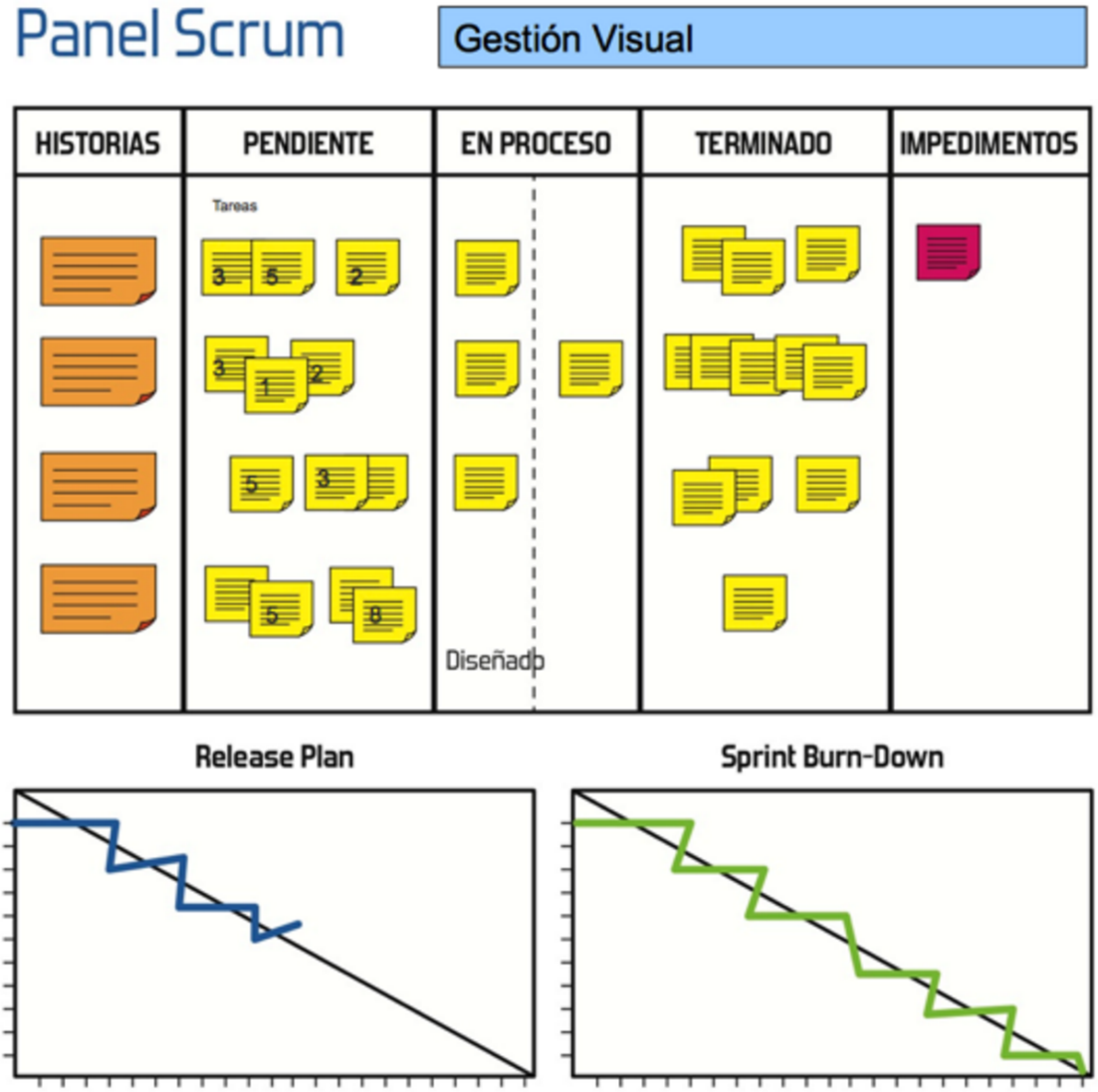
Esta forma de plantear el desarrollo, mediante iteraciones y donde las historias de usuario son elegidas por el *Product Owner*, ofrecen ventajas sobre las metodologías tradicionales. Estas ventajas van desde una mayor motivación del equipo de desarrollo hasta una mayor calidad en el resultado final y la obtención de un producto viable antes en el tiempo.

3.1.3. *Daily Meeting*

Para realizar el seguimiento diario de la evolución del proyecto se realiza el *Daily Meeting*. El *Daily Meeting* es una reunión de seguimiento diaria donde todos los integrantes del equipo de SCRUM comentan lo que han realizado hasta el día de la reunión, lo que van a realizar y si han tenido algún problema. De esta forma, los problemas aparecen antes y se puede encontrar una solución a tiempo. Todo el mundo conoce la ocupación de cada integrante.

También, al finalizar la reunión, se actualizan las gráficas de seguimiento del proyecto. De esta forma, se mantiene actualizado de forma diaria. Estas gráficas son: el *SCRUM Panel*, panel donde se encuentran todas las historias de usuario, divididas en tareas, por las que está compuesto el sprint; y el *Sprint Burndown*, gráfica de seguimiento donde se mide la desviación entre el trabajo ideal del *sprint* y el trabajo actual. Es muy importante que estas gráficas se encuentren visibles para toda la organización. Un ejemplo de estas gráficas se puede ver en la figura 3.4.

Figura 3.4: SCRUM Panel y Sprint Burndown



Estas reuniones, en contraposición a las metodologías tradicionales, permiten conocer el estado del proyecto día a día, favoreciendo así la resolución de los problemas. El equipo también está más motivado debido a que se comparten los progresos y las soluciones tomadas.

3.1.4. *Sprint Review*

El *Sprint Review* es la reunión que se produce entre el equipo de desarrollo y el *Product Owner*. En esta reunión es donde se comprueba si el trabajo que se ha realizado está realizado correctamente o no.

En esta reunión, el equipo expone, a modo de demostración, el funcionamiento de las historias de usuario que se han desarrollado a lo largo del *sprint*. El *Product Owner* es el encargado de aceptar estas historias de usuario en función de los criterios de aceptación de cada una de ellas.

Lo más importante de esta reunión es que, periódicamente, se comprueba el funcionamiento de la aplicación y se pueden corregir las desviaciones que se produzcan. Comparando esto con las metodologías tradicionales, donde se muestra al final el producto íntegro, esta forma permite adaptarse al cambio y modificar, sobre la marcha, los posibles requisitos iniciales de la aplicación.

Con esta reunión, el equipo también es consciente de los posibles problemas que puedan ir surgiendo. Normalmente, al realizar un proyecto, existen ciertas dudas técnicas sobre la implementación de ciertas funcionalidades. Estas dudas se van disipando a medida que se van completando *sprints*. También, se discuten las cosas que han ido bien y las cosas que han ido mal para intentar encontrar una solución.

3.1.5. *Sprint Retrospective*

Por último, al final del *sprint*, a parte de realizar la reunión del *Sprint Review*, también se produce la reunión de *Sprint Retrospective*. A diferencia de la reunión anterior, esta reunión se produce únicamente entre los miembros del equipo de desarrollo.

Esta es una reunión en la que los miembros del equipo de desarrollo intercambian opiniones sobre la evolución del proyecto con el objetivo de mejorar la calidad del proceso.

3.2. Principios y Patrones de Diseño para el Desarrollo de *Software*

Tanto los principios como los patrones de diseño para el desarrollo de *software* tienen como objetivo dotar a los desarrolladores de una serie de directrices y herramientas con las que puedan desarrollar las aplicaciones dotándolas de una elevada calidad.

La calidad en los proyectos es un aspecto fundamental ya que no sólo afecta al funcionamiento de la aplicación sino que también afecta al mantenimiento y a la extensibilidad de la misma.

3.2.1. Principios de Diseño de *Software*

Como se ha mencionado durante este apartado, los principios de diseño de *software* son una serie de directrices que se deben tomar para mejorar la calidad de los desarrollos de *software*. Los desarrolladores son los principales encargados de seguir estas directrices y aplicarlas en los desarrollos.

En muchas ocasiones, se le otorga una mayor importancia a otras tareas dentro del proceso de desarrollo de *software* olvidándose de que, al fin y al cabo, la tarea encargada del desarrollo de la propia aplicación es la programación de la misma. Con esto no se quiere decir que el resto de

tareas no sean importantes, sino que hay que dotar de la misma importancia a la programación de la solución y contar con personal cualificado.

En numerosas ocasiones el personal dedicado a la realización de esta tarea no cuenta con los conocimientos necesarios ni con una buena metodología de desarrollo, lo que desemboca en la realización de aplicaciones con una baja calidad y poco mantenibles. Esto, a parte de los problemas obvios a corto plazo, se le añaden los problemas a largo plazo a la hora de realizar evolutivos sobre la aplicación o con tareas de mantenimiento.

Para evitar esto, surgen los principios de desarrollo de *software*, principios que deben conocer los desarrolladores para poder así realizar mejores aplicaciones con una mejor calidad.

En el siguiente apartado es donde se van a ver los principales principios de diseño de *software*.

DRY - *Don't Repeat Yourself*

El objetivo principal de este principio de desarrollo de *software* es el de evitar la repetición de información, de cualquier tipo, dentro de las aplicaciones, sobretodo si se trata de aplicaciones multicapa donde es más probable que se repitan ciertas estructuras de almacenamiento o de propagación de información. Este principio es formulado de la siguiente forma: "Cada porción de conocimiento tiene que tener una única forma, y no ambigua, de representación dentro de la aplicación". Este principio ha sido formulado por Andy Hunt y Dave Thomas en su libro The Pragmatic Programmer [1].

Gracias al uso de este principio de diseño de *software* dentro de los desarrollos, el código resultante tiene un menor nivel de código duplicado y un menor nivel de viscosidad, lo que aumenta de forma directa la calidad del producto final. Un menor nivel de viscosidad quiere decir que, para realizar un cambio de la lógica dentro de la aplicación, lo único que será necesario es modificar la entidad representativa de esta unidad de información sin necesidad de modificar los objetos que no tienen que ver con este cambio.

Lo contrario a este principio es una solución *WET* (*Write Everything Twice*, *We Enjoy Typing* o *Waste Everyone's Time*). Las soluciones *WET* son muy comunes dentro de los proyectos multicapa donde, un simple cambio, puede afectar tanto a la parte encargada de la representación visual, como a la parte de la lógica de negocio e; incluso, a la parte encargada de gestionar la conexión con la base de datos. *DRY*, en cambio, aísla este tipo de conocimiento evitando que la información quede duplicada dentro de la aplicación.

KISS - *Keep It Simple Stupid*

El objetivo principal de este principio de desarrollo de *software* es el de conseguir realizar la solución más simple en lugar de optar por soluciones más complejas. Este principio también comparte significado con el principio metodológico de La Navaja de Ockham, el que enuncia lo siguiente: "En igualdad de condiciones, la explicación sencilla suele ser la más probable". Este principio, aplicado al entorno de desarrollo de *software*, indica que no se debe sobredimensionar el problema sino remediarlo con la solución más simple posible.

En muchas ocasiones, a la hora de plantear la solución a cualquier problema, se piensa en realizar una solución genérica o una solución que abarcar más funcionalidad de la deseada en un primer momento. Esto, a parte de incrementar el tiempo de desarrollo y los costes directos del proyecto, también incrementa de forma directa la complejidad de la aplicación. Al aumentar la complejidad de la aplicación, y no usar una buena metodología de desarrollo, aumenta la

dificultad de las labores mantenimiento. De forma adicional, también aumenta el número de fallos de la aplicación ya que aumentan los puntos susceptibles de fallar al tener un mayor número de líneas de código.

Por lo tanto, al tener en cuenta este principio, lo que se está consiguiendo es simplificar en todo lo posible el desarrollo, haciendo que se pueda incluso reducir el tiempo estimado. Del mismo modo, se reduce la complejidad de la aplicación, favoreciendo el mantenimiento de la misma y reduciendo sus costes.

SOLID

SOLID es un acrónimo que hace referencia a un conjunto de principios de desarrollo de *software* de lenguajes orientados a objetos que tienen como objetivo mejorar la calidad de la aplicación desembocando en una mayor mantenibilidad y en una mayor capacidad para extender la aplicación en un futuro.

Estos principios fueron enunciados por Robert C. Martin [8] a principios de los años 2.000. Pretenden eliminar los denominados malos olores dentro del código empleando técnicas de refactorización.

El proceso de refactorización [6] es uno de los procesos más importantes de cara a mejorar la calidad del código presente en la aplicación. Mediante la refactorización, lo que se pretende conseguir es reestructurar el código de tal forma que mejore su claridad y de tal forma que no modifique el comportamiento interno de la aplicación. También pretenden disminuir la complejidad y hacer el código más legible, favoreciendo el mantenimiento de la misma.

Para realizar la refactorización de una aplicación existen distintas técnicas, organizadas en distintos grupos según su objetivo:

- Técnicas para favorecer la abstracción.
  - **Encapsular campo**. Fuerza el acceso a un atributo mediante objetos para definir su valor y para recuperarlo.
  - **Generalización de tipos**. Permite generalizar el tipo de una determinada estructura con el objetivo de admitir más tipos de datos y aumentar su funcionalidad.
  - **Reemplazar la comprobación de tipos mediante distintos estados o estrategias**.
  - **Reemplazar el uso de condicionales mediante el uso de polimorfismo**.
- Técnicas para dividir el código en distintas partes lógicas.
  - **Definición de componentes**. Permite la división del código en partes más pequeñas, favoreciendo la legibilidad del código y la mantenibilidad. Es muy importante tener en cuenta que las clases no deben ser muy grandes para evitar que contemplen más de una funcionalidad.
  - **Extraer clase**. Permite extraer cierto contenido de una clase para que forme otra clase nueva. De esta forma se consigue separar la funcionalidad en componentes distintos.
  - **Extraer método**. Permite dividir el código en fragmentos más pequeños. Esto, a parte de mejorar en gran manera la legibilidad del mismo debido a métodos mucho más pequeños, aumenta la reusabilidad de ciertos componentes.

- Técnicas para mejorar los nombres y la organización del código.

  - **Mover método o mover campo**. Permite mover un método o un campo a una clase donde tenga más sentido, favoreciendo así la claridad y legibilidad del código.
  - **Renombrar método o renombrar código**. Permite cambiar el nombre de un método o un campo para mejorar la claridad y la legibilidad del código.
  - **Desplazar hacia arriba**. Permite mover tanto un método como un campo a una clase padre. Es importante destacar que esta técnica de refactorización sólo está disponible en aquellos lenguajes orientados a objetos.
  - **Desplazar hacia abajo**. Permite mover tanto un método como un campo a una clase hija. Es importante destacar que esta técnica de refactorización sólo está disponible en aquellos lenguajes orientados a objetos.

Una vez analizados ciertos términos a tener en cuenta, se mencionan los cinco principios:

- **SRP - *Single Responsibility Principle*** [13]. El principio de responsabilidad única enuncia que todos los módulos o clases tienen que ser responsables sobre una funcionalidad concreta de la aplicación y que esa funcionalidad tiene que estar encapsulada íntegramente por la clase.

  Este término fue introducido por Robert C. Martin. Martin define responsabilidad como una razón para cambiar y concluye con que una clase debe tener una y sólo una razón para cambiar.

- **OCP - *Open/Closed Principle*** [10]. El principio de abierto/cerrado enuncia que todos los módulos o clases tienen que estar abiertos a extensiones pero cerrados a modificaciones. Esto, de forma resumida, indica que una clase puede ver extendido su comportamiento sin modificar el código original.

  Para cumplir con este principio se debe usar la herencia como técnica principal. La herencia es uno de los mecanismos básicos para conseguir los objetivos de mantenibilidad y extensión dentro de los proyectos *software*. La herencia, en término generales, es una relación que existe entre una clase general y otra más específica. De esta forma, al emplear la herencia, la clase más específica adquiere los campos y los métodos de la clase general y puede, de forma fácil, adaptar su comportamiento en función a la interfaz general.

- **LSP - *Liskov Substitution Principle*** [7]. El principio de la sustitución de Liskov enuncia que: "Sea q(x) una propiedad comprobable sobre los objetos x de tipo T. Entonces q(y) debe ser verdad para los objetos y del tipo S donde S, es un subtipo de T". Esto, a nivel general, lo que indica es que si se tiene una clase que hereda de otra, se puede usar como el tipo de su padre sin necesidad de que se conozca algún comportamiento adicional.

- **ISP - *Interface Segregation Principle*** [11]. El principio de la segregación de interfaces indica que un cliente no tiene que verse obligado a depender de métodos que no usa. El objetivo de este principio es dividir aquellas interfaces que son demasiado grandes en otras interfaces más pequeñas y con un comportamiento más específico. De esta forma, aquellos clientes que necesiten implementar cierta interfaz implementarán las interfaces necesarias con los métodos mínimos necesarios.

- **DIP - *Dependency Inversion Principle*** [9]. El principio de la inversión de dependencias indica que un módulo de bajo nivel no debe ser responsable de instanciar el módulo

de alto nivel que se usa. De esta forma, se disminuye el acoplamiento entre los distintos módulos de la aplicación.

Este conjunto de principios están destinados a mejora la cohesión, disminuir el acoplamiento entre las distintas partes de la aplicación.

SoC - *Separation of Concerns*

El objetivo principal de este principio de desarrollo de *software* enuncia que una aplicación se debe dividir en distintos módulos y que cada módulo sea el encargado de la realización de un determinado concepto [15].

Un programa que tiene una buena separación de conceptos se dice que es modular. Un ejemplo típico de un sistema modular es aquel que está dividido en distintas capas, donde cada una de las capas representa una funcionalidad.

De esta forma, al estar dividido en capas, aumenta la reusabilidad de cada uno de los componentes.

YAGNI - *You Ain't Gonna Need It*

Este principio de desarrollo de *software* indica que no se debe realizar nada que no sea necesario dentro de un programa.

Este principio está relacionado, en parte, con el principio *KISS*, donde las soluciones propuestas deben ser simples, sin entrar dentro de sobredesarrollos ni de implementaciones de funcionalidades innecesarias.

En general, el desarrollo de funcionalidades innecesarias puede tener las siguientes desventajas:

- El tiempo empleado en el desarrollo de estas funcionalidades incrementa al dedicar tiempo al desarrollo de una funcionalidad adicional.
- Del mismo modo, al realizar nuevas funcionalidades, estas funcionalidades también tienen que ser documentadas y validadas, haciendo que las labores de mantenimiento aumenten.
- Es probable que una de las funcionalidades extras desarrolladas puedan tener un efecto colateral sobre otra de las funcionalidades necesarias por la aplicación. Pueden entrar en conflicto y, en este caso, sería necesario volver a dedicar tiempo y esfuerzo en corregir este fallo.
- Al aumentar la funcionalidad del programa, también aumenta la complejidad, dificultando las labores de mantenimiento.

BS - *Boy Scout*

Este principio tiene como objetivo el de desempeñar tareas de mantenimiento y de mejora del código siempre que sea posible. En el momento en el que se encuentre un defecto en el programa, debe ser solucionado de forma inmediata.

3.2.2. Patrones y Antipatrones de Diseño de *Software*

Los patrones de diseño de *software* [21] son herramientas que proporcionan a los desarrolladores distintos métodos para solucionar los posibles problemas que puedan surgir. Para que una solución sea considerada un patrón debe haberse comprobado su eficiencia y su capacidad de reutilización en distintos escenarios.

De forma análoga, también existen antipatrones, que son formas que informan sobre cómo no se deben solucionar los problemas. Es tan importante conocer el cómo se deben hacer las cosas como saber cómo no se deben hacer. Muchas veces se implementa una solución que se cree que es correcta y, gracias a los antipatrones, se pueden utilizar mecanismos para conocer si dicha solución es válida desde el punto de vista de diseño de las aplicaciones.

Patrones de Diseño de *Software*

En función de su naturaleza, existen distintos tipos de patrones, aportando cada uno de ellos distintos mecanismos para la resolución de los problemas. Se van a enumerar los distintos tipos de patrones de diseño que existen y se nombrarán algunos de los principales ejemplos de cada uno de ellos.

Los distintos tipos de patrones son los siguientes:

- **Patrones creacionales**. Las patrones creacionales son aquellos destinados a solucionar los problemas derivados de la creación de nuevas instancias. De esta forma, la elección de la instancia a crear y el mecanismo de creación serán diseñados de la forma más eficiente y reutilizable posible.

  A continuación, se muestran los principales ejemplos de patrones creacionales:

  - ***Object Pool***. Este mecanismo indica que, para crear nuevas instancias de los objetos, en lugar de crearlas desde cero, se clonen los atributos de otro elemento. Este mecanismo se emplea cuando es más costoso la creación de un nuevo elemento que la clonación.
  - ***Builder***. A la hora de crear objetos complejos que están compuestos por numerosos atributos, esta lógica de creación se abstrae y se localiza en un único punto. Con la aparición de las API's fluidas, este tipo de mecanismos ven mejorada de forma considerable su legibilidad.
  - ***Factory Method***. Este mecanismo permite trabajar con la creación de distintos tipos de objetos desde un mismo punto. De esta forma, se abstrae toda la lógica de creación de objetos y de selección del tipo y se ubican en una una única parte, de forma que la reutilización es posible en otros lugares de la aplicación.
  - ***Prototype***. Este mecanismo, al igual que el *Object Pool*, permite crear nuevas instancias mediante la clonación de un determinado objeto.
  - ***Singleton***. Se garantiza la existencia de una única instancia de una determinada clase para toda la aplicación. De esta forma, esta instancia puede ser usada desde distintos puntos de la aplicación. Es muy importante tener en cuenta la concurrencia en este aspecto.
  - ***Model View Controller***. Sirve para separar, en tres niveles lógicos, la arquitectura de la aplicación. De esta forma, existe una capa de modelo de datos, una capa de

controladores donde se ejecutan las operaciones y la capa de vista donde se muestra la información de la aplicación.

- **Patrones estructurales**. Los patrones estructurales son aquellos que están destinados a la resolución de problemas de composición entre objetos.

  A continuación, se muestran los principales ejemplos de patrones estructurales:

  - ***Wrapper***. Permite adaptar una determinada interfaz para que pueda ser usada por otro objeto que, sin este mecanismo, no podría ser usada.
  - ***Composite***. Este mecanismo permite tratar un conjunto de objetos como si fuese uno solo, aplicando una serie de funciones de forma global a todo el conjunto.
  - ***Decorator***. Permite añadir, de forma dinámica, funcionalidad a un determinado objeto.
  - ***Facade***. Facilita la tarea de acceder a un conjunto de interfaces proporcionando un único punto de acceso para la comunicación con todas ellas.

- **Patrones de comportamiento**. Los patrones de comportamiento son aquellos que están destinados a la resolución de problemas de interacción entre objetos.

  A continuación, se muestran los principales ejemplos de patrones comportamiento:

  - ***Chain of Responsibility***. Permite definir una cadena en la que los mensajes se van propagando entre sus elementos para desempeñar unas determinadas operaciones.
  - ***Command***. Este patrón permite encapsular una determinada acción de tal forma que pueda ser llamada sin necesidad de conocer la implementación de la misma.
  - ***Iterator***. Dada una colección de elementos, permite recorrerlos independientemente de su topología y de su tipología.
  - ***Observer***. Permite tener una dependencia con una serie de objetos que, cuando cambien alguna de sus propiedades, es informado de este cambio para realizar algún tipo de operación.
  - ***Strategy***. Permite disponer de varios tipos de soluciones ante un mismo problema y elegir, de forma dinámica, la solución a adoptar.

Antipatrones de Diseño de *Software*

A diferencia de los patrones de diseño de *software*, los antipatrones ofrecen ejemplos de operativas que no se tienen que seguir a la hora de desarrollar las aplicaciones.

No sólo existen antipatrones orientados al diseño de *software*, sino que también ofrecen antipatrones que se pueden aplicar a distintas áreas. Estas áreas pueden ser desde la gestión hasta la metodología.

Unido a esto, la capacidad de desarrollar un código legible también es importante [12].

A continuación, se muestran algunos de los antipatrones más relevantes:

- **Código espagueti**. Este antipatrón hace referencia a aquellos sistemas cuya estructura es difícilmente comprensible.

- **Confianza ciega**. Consiste en no comprobar el resultado de las llamadas a las subrutinas confiando en la realización correcta de la operación.
- **Objeto todopoderoso**. Este antipatrón aparece cuando, dentro de la aplicación, existe un elemento que tiene conocimiento sobre demasiadas situaciones.
- **Lava seca**. Mantener código que ya no se usa dentro de un sistema que evoluciona constantemente.
- **Números mágicos**. Incluir en los algoritmos determinados números que no tienen conexión con la ejecución del propio algoritmo.

3.3. Desarrollo Guiado por Pruebas

Las pruebas, en Ingeniería del *Software*, son los procesos que permiten verificar y revelar la calidad verdadera del producto. Con cada una de las pruebas, se lleva a cabo la ejecución de un programa que, mediante técnicas experimentales, trata de evitar errores que se producirían en tiempo de ejecución.

Hay muchos tipos distintos de pruebas: pruebas unitarias, donde se comprueba el comportamiento atómico de los componentes de la aplicación; pruebas de integración, donde se comprueba la comunicación entre los distintos componentes de la aplicación, pruebas funcionales, pruebas de aceptación, pruebas de regresión y un conjunto de pruebas generales del sistema.

El hecho de tener pruebas sobre el código, asegura que lo que funciona hoy seguirá funcionando mañanda; sobre todo si la ejecución de las mismas está automatizada e integrada dentro del ecosistema de desarrollo, con el soporte de un servidor de integración continua.

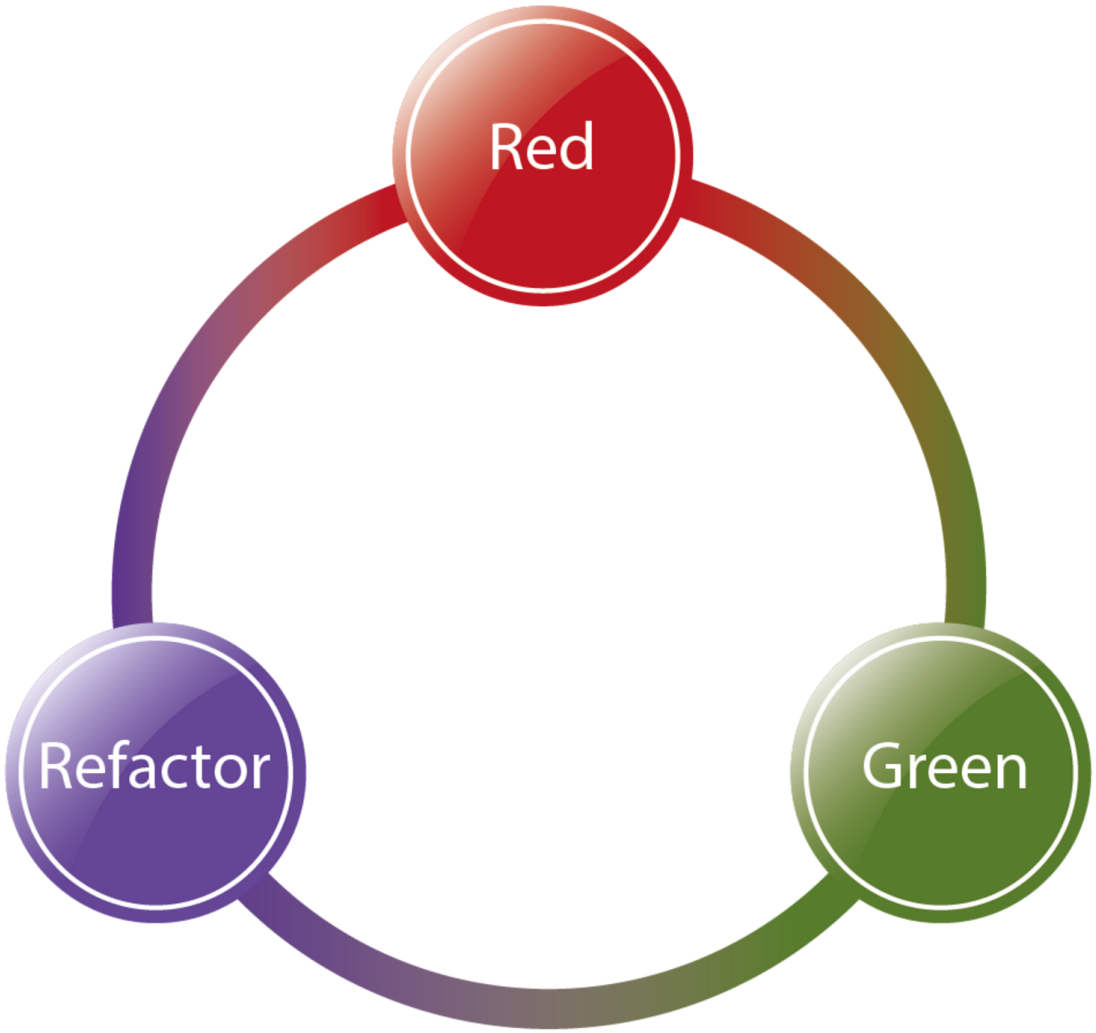
Sin una buena batería de pruebas, cualquier modificación en el código puede ser el origen de un nuevo fallo. Con pruebas, se pierde el miedo al cambio y, cuanta mayor cobertura del código, mayor seguridad a la hora de afrontar cambios dentro de la propia aplicación.

Las pruebas deben ser también un medio para realizar el diseño de la funcionalidad de negocio de la propia aplicación, usando TDD [2] (Diseño Dirigido por Tests) y pasando por cada una de las siguientes fases, como se puede ver en la figura 3.5:

- ***Red***. Primero se comienza escribiendo el código de la prueba, que no compilará puesto que aún no se ha escrito ni siquiera el fuente de las clases y no pasará porque no tiene lógica de negocio.
- ***Green***. Después, se escribe el código de las clases de negocio para que la prueba compile y pase. Este código que se ha escrito es el mínimo código posible para que la prueba sea superada con éxito.
- ***Refactor***. Por último, se elimina la redundancia y se mejora la implementación, con técnicas de refactorización y principios *SOLID*.

Siguiendo la técnica del *Red - Green - Refactor*, se asegura que no se escribe una línea de código que no esté probada mediante una prueba y; con ello, que no se escribe una línea de código innecesaria.

Figura 3.5: Ciclo de la metodología TDD



Siguiendo con la misma filosofía, sólo se deberían generar pruebas que cubran la funcionalidad de historias de usuario o casos de uso, con lo que no se generará más código del estrictamente necesario para cubrir la funcionalidad de negocio.

Las propias pruebas deben perseguir también un buen diseño, para evitar que la propia infraestructura de pruebas se convierta en un problema, debería cumplir con el principio FIRST:

- ***Fast***. Las pruebas deben ser de rápida ejecución, por eso se debe poner especial énfasis en implementar pruebas unitarias y sólo pruebas de integración en aquellos casos en los que realmente se necesite el contexto de un sistema externo para ser ejecutados.
- ***Independent***. Para facilitar la tarea de detección de errores es muy importante que las pruebas sean independientes las unas de las otras. Para lograrlo, se debe evitar que las salidas de unas pruebas se utilicen como entradas de otras y no debería importar el orden en el cual se vayan a ejecutar las pruebas, ya que cada ejecución puede tener, de hecho, una ordenación distinta.

Si se tiene una batería de pruebas de integración contra base de datos, se debe mantener la transaccionalidad en la operaciones de las mismas, de modo tal que el entorno siempre quede consistente tras su ejecución.

- ***Repeatable***. Deben funcionar en cualquier entorno, sin tener dependencias físicas. Estas dependencias físicas se deben poder simular, con bases de datos empotradas o a través de la simulación de sistemas (como pueden ser servicios remotos, por ejemplo).
- ***Self-validating***. Deben ser autoevaluables, es decir, que la propia prueba identifique si la prueba ha funcionado correctamente o no y no requiera inspección visual o comprobación manual por el usuario de la salida de la ejecución de la prueba. Esta autoevaluación se realiza mediante comprobaciones.
- ***Timely***. Deben escribirse en el momento oportuno; es decir, antes del código de producción y el motivo es muy simple: es más fácil hacer pruebas para un código que todavía no está escrito que para uno que ya ha sido creado, del mismo modo que es más fácil hacer crecer recto un árbol que todavía no ha brotado con una guía, que enderezar uno que tiene varios metros de altura.

Toda prueba debería tener tres secciones claramente diferenciadas:

- ***Arrange* o preparación**. Esta fase implica una serie de tareas de inicialización de las clases de servicio o preparación de los datos previo a la invocación a la lógica de negocio.
- ***Act* o actuación**. Esta fase consiste en invocar a la lógica de negocio con los datos previamente preparados.
- ***Assert* o afirmación**. Por último, la fase de comprobación de los resultados. Una prueba sin comprobaciones no es una prueba autoevaluable.

Por último, a la hora de realizar pruebas con objetos que emplean otros objetos externos, se utilizan dobles de prueba. Los dobles de prueba permiten engañar al código para que se crea que colabora correctamente con otras clases.

Existen los siguientes tipos de dobles, ordenados de menor a mayor complejidad:

- ***Dummy***. Se pasa cuando no importa cómo se colabora con este objeto. Por ejemplo, cuando se sabe que no se va a usar en absoluto. La implementación de los métodos de estos dobles no hacen nada.
- ***Stub***. Es como un *dummy* pero la implementación de los métodos devuelven valores fijos. Por ejemplo, un método de autenticación devolvería siempre autenticación correcta y así se podría usar este doble para probar todos los escenarios donde la autenticación ha sido correcta, sin necesidad de hacer de verdadera la autenticación.
- ***Spy***. Es como un *stub* pero que espía a quien lo llama. Esto permite luego comprobar si se ha llamado correctamente, el número de veces correcto que se ha invocado a un determinado método y una serie de información adicional. Estos dobles son peligrosos porque acoplan la prueba con la implementación concreta, lo que provocará que, si se cambia la implementación, aunque no cambie el comportamiento, la prueba fallará. Son pruebas frágiles.

- ***Mock***. Es como un *spy* que sabe lo que está probando exactamente. Así al propio *mock*, en la sección de aserciones, se le preguntará si ha ido bien o mal la prueba. El *mock* sabe el comportamiento de cómo se debe llamar al doble, cuántas veces y con qué parámetros.
- ***Fake***. Es un tipo totalmente distinto a los anteriores. Un *fake* implementa los métodos con lógica de negocio, es como un simulador que puede ser muy sencillo o extremadamente complicado. Por ejemplo, si se usa una base de datos en memoria para simular una base de datos real, esta base de datos en memoria es un *fake*.

Con todo esto, se consigue una metodología de elaboración de pruebas que permiten probar el sistema de forma íntegra, con todas las interacciones entre los distintos sistemas. Esto aporta un claro beneficio que es el de la seguridad de saber que todo funciona correctamente siempre y cuando todas y cada una de las pruebas se ejecuten de forma correcta. Otro beneficio adicional es la capacidad para realizar modificaciones dentro del código sin añadir ningún error.

4

Arquitectura

En este apartado se va a describir la arquitectura que se ha empleado para desarrollar la aplicación. Es importante destacar que la definición de una buena arquitectura es esencial para una futura evolución de la aplicación y para mejorar el mantenimiento de la misma. De forma adicional, una buena arquitectura posibilitará en un futuro escalar el funcionamiento de la aplicación de forma correcta aumentando de forma considerable el rendimiento y la capacidad de computación.

La arquitectura de una aplicación establece la forma en la que se van a distribuir los componentes y las relaciones que se van a producir entre cada uno de ellos. En la actualidad, a la hora de desarrollar una nueva aplicación, existen mecanismos para identificar cuál es la mejor arquitectura que se debe utilizar en función de las distintas necesidades de la propia aplicación. Este hecho hace que plantear una buena arquitectura sea una tarea más fácil debido al gran número de ejemplos de aplicaciones existentes y debido también a la publicación de este conocimiento. Este hecho era muy distinto en los primeros años del desarrollo del *software* ya que no había mucha información y los proyectos no estaban bien estructurados.

Es importante destacar también el papel de la tecnología empleada a la hora de desarrollar esta arquitectura elegida. La elección de una buena tecnología que no limite el desarrollo y que proporcione herramienta que lo faciliten es esencial. Esta tecnología tiene que tener un largo recorrido, ser fiable y tener una gran comunidad que permita la identificación y corrección de los posibles errores. A la hora de elegir una tecnología los sistemas con los que tiene integración también aumentan su funcionalidad.

Con todo esto, en los sucesivos apartados se va a analizar, de forma específica, la arquitectura de la aplicación y los distintos módulos por los que está compuesta. Del mismo modo, también se va a analizar la tecnología que se ha empleado para desarrollar cada uno de estos módulos y las ventajas que proporciona la tecnología elegida. Con todo esto, se mostrará el funcionamiento general del sistema de distribución de operaciones propuesto y las distintas opciones de configuración presentes en cada una de las partes. Finalmente, mediante un caso práctico, se mostrará el funcionamiento de la aplicación.

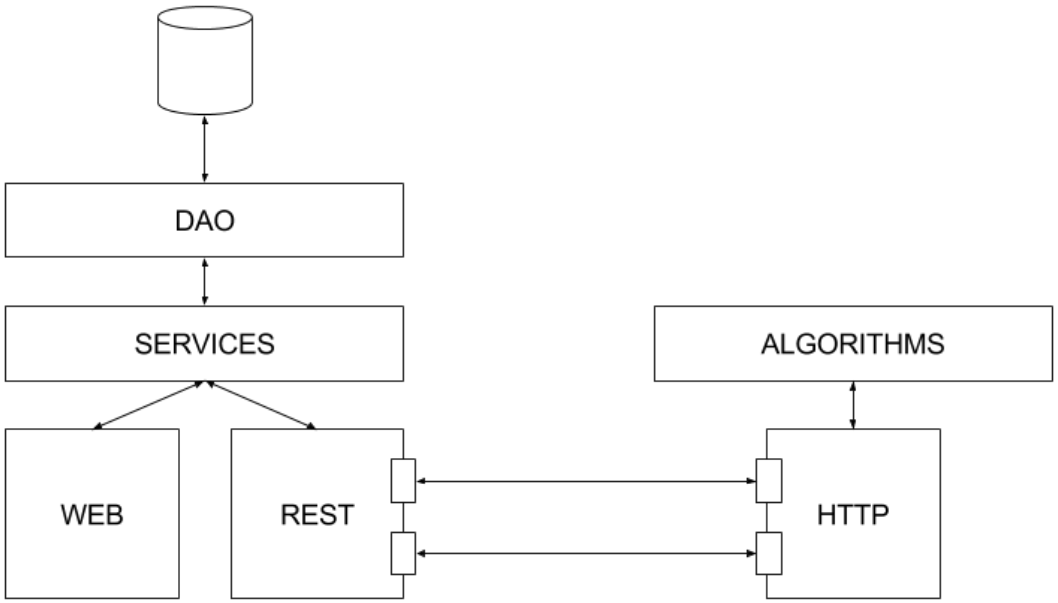
4.1. Arquitectura de la Aplicación

En el siguiente apartado se van a analizar los distintos componentes por los que está compuesta la aplicación. Cada uno de estos componentes, a su vez, están divididos en distintos módulos en función de la operativa que realice cada una de ellos. Esta operativa puede variar desde gestionar la comunicación con la base de datos hasta proporcionar un mecanismo de representación para los datos residentes en la aplicación.

El objetivo principal de la aplicación es gestionar la distribución de una serie de operaciones para la ejecución de un determinado algoritmo sobre un conjunto de datos. También permite configurar, de forma sencilla, la manera en la que se van a distribuir esta serie de operaciones y los algoritmos que se deben aplicar sobre los datos. Esta serie de operaciones son distribuidas mediante una aplicación residente en un servidor. Para la ejecución de las operaciones, los clientes se comunican con el servidor y solicitan la operación a realizar. Una vez realizada la operación, se vuelven a comunicar con el servidor para devolver el resultado y; en caso de que haya más operaciones, obtener la siguiente o finalizar la ejecución del proceso.

La arquitectura resultante de la aplicación es la mostrada en la figura 4.1.

Figura 4.1: Arquitectura de la aplicación



4.1.1. Parte Servidor

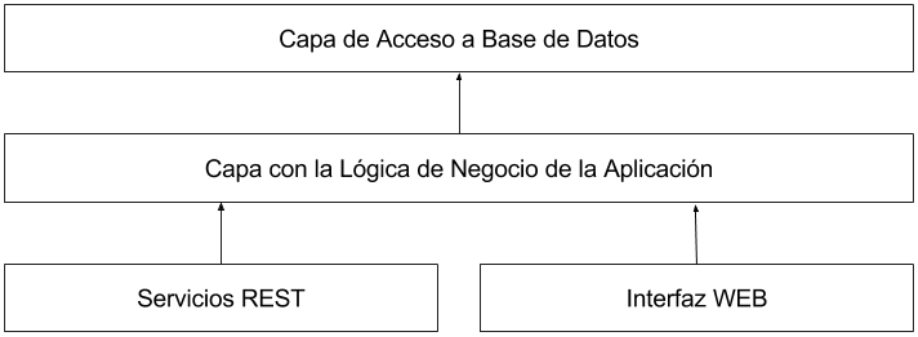
Como se ha visto, en esta parte es donde reside todo el sistema de gestión y orquestación de distribución de operaciones a los clientes. Ésta es la pieza angular del sistema. Es por esto que se tiene que realizar un buen diseño de tal forma que se permita extender de forma sencilla la funcionalidad y para que el mantenimiento de la aplicación no suponga un problema.

Para desarrollar la parte del servidor se ha empleado una arquitectura JEE. Dentro de esta arquitectura basada en Java, se han definido distintas capas con el objetivo de separar, a nivel lógico, las distintas funcionalidades que existen dentro de la aplicación. Éste, por tanto, es un claro ejemplo de arquitectura definida por capas de tres niveles (nivel de acceso a base de datos, nivel donde reside la lógica de negocio de la aplicación y nivel de representación visual de los datos de la aplicación, donde se pueden tener distintas formas de representación de los datos residentes en la aplicación).

El objetivo principal de la utilización de una arquitectura basada en tres niveles es la separación de las distintas partes lógicas, y totalmente independientes, que existen dentro de la aplicación. Para conseguir separar de forma correcta la aplicación en capas, es muy importante la definición de interfaces de comunicación entre ellas. Estas interfaces, al ser el punto de comunicación de cada una de las capas por las que está compuesta la aplicación, permiten la reutilización, de forma clara, de cada uno de los distintos módulos ya mencionados por parte de otros tipos de aplicación, en caso de que fuese necesario, o por evoluciones que se realicen en la misma aplicación.

Un ejemplo de arquitectura de tres capas es la que se puede observar en la figura 4.2.

Figura 4.2: Arquitectura de la parte servidor



A continuación, se pasa a detallar cada una de las distintas capas presentes en la aplicación:

- **Capa de acceso a Base de Datos**. Este módulo es el encargado de establecer la comunicación con la base de datos de la aplicación. En la base de datos es donde se almacenan las estructuras y la información a partir de la cual se realizan las operaciones. Por tanto, dentro de este módulo, se encuentran los objetos que contienen la representación de los objetos almacenados en la base de datos y los métodos de acceso a la información almacenada.

  Es muy importante destacar que, dentro de este módulo, también está configurada y gestionada la comunicación con la base de datos. Esta configuración no sólo incluye la conexión con la base de datos, sino que también incluye el sistema de gestión de la sesión y de las transacciones realizadas. Es muy importante este aspecto ya que, de forma general, uno de los principales problemas en las aplicaciones es una mala configuración y gestión de la comunicación con la base de datos. La configuración de un *pool* de conexiones correcto es algo a tener en cuenta para un correcto funcionamiento de la aplicación.

Para que la configuración de los parámetros de conexión sea óptima, esta configuración se encuentra externalizada en un fichero de propiedades que, a la hora de proceder con la instalación en otro sistema, es donde se definirán dichos parámetros para que la aplicación gestione la conexión con la base de datos. De esta forma, la configuración queda desacoplada del propio proyecto aportando mayor flexibilidad de configuración.

Para asegurar la correcta implementación de este módulo y para comprobar que, en caso de realizar cambios en el mecanismo de acceso a la información, no se altere el comportamiento del mismo, se han desarrollado una serie de pruebas de integración automáticas. Dentro de estas pruebas de integración se comprueba que la información recuperada de base de datos es la correcta en función de cada uno de los métodos desarrollados. Al ser automáticas, estas pruebas podrán ejecutarse en cualquier momento.

Como ya se ha mencionado en apartados anteriores de la memoria, las pruebas de integración comprueban la correcta comunicación con un sistema externo, en este caso un servidor de base de datos. Es por esto importante que estas pruebas se realicen sobre el mismo entorno final en el que se desplegará la aplicación.

- **Capa con la lógica de negocio de la aplicación**. Este módulo es el encargado de contener toda la lógica de negocio de la aplicación y los objetos con los que se modelará esta lógica.

  Este módulo es muy importante tenerlo de forma independiente ya que, de esta forma, se tiene centralizada en un único punto la lógica de negocio de la aplicación, pudiendo reutilizarla en cualquier otra aplicación o módulo de la misma aplicación. Es buena práctica realizarlo de esta forma ya que, en el caso de tener la lógica de negocio distribuida en varios componentes, a la hora de realizar un cambio, éste afectaría a todos y cada uno de los componentes. En cambio, al tenerlo de forma independiente, al modificar esta lógica, no sería necesario realizar ningún cambio adicional. Del mismo modo, al reducir el número de puntos donde se encuentra la lógica de negocio, los fallos debidos a errores de esta lógica se reducen y corregirlos es una labor más sencilla. Con esto se mejora la mantenibilidad de la aplicación.

  Esta capa de negocio de la aplicación se ha implementado siguiendo los principios y patrones de diseño analizados en el punto anterior de la memoria. De esta forma, el resultado es un diseño fácilmente extensible y mantenible, aportando una mayor calidad al sistema. El diseño basado en interfaces favoreciendo el polimorfismo permite un bajo acoplamiento y una elevado cohesión entre todos sus componentes.

  Del mismo modo que en el caso de módulo de acceso a base de datos, para asegurar la correcta implementación de este módulo, se han realizado una serie de pruebas automáticas. Estas pruebas automáticas son pruebas unitarias; es decir, prueban el comportamiento aislado de cada uno de los métodos. Al igual que en el caso anterior, al ser automáticas también facilita su ejecución.

- **Capa con la publicación de servicios REST**. Este módulo es el encargado de ofrecer una serie de servicios externos con los que aplicaciones clientes se puedan comunicar con el sistema.

  Para la comunicación, se han utilizado servicios REST con formato JSON. Se han ofrecido servicios REST ya que se espera que las comunicaciones no sean mantenidas ni síncronas. Estas comunicaciones se establecerán mediante el protocolo HTTP. Se ha elegido el formato JSON ya que es bastante ligero al no contener información irrelevante.

A la hora de asegurar, en parte, la seguridad en cuanto al acceso de estos servicios, se ha configurado de forma restrictiva los permisos de CORS. De esta forma, los permisos sólo podrán ser accedidos desde un determinado dominio y las cabeceras serán las mínimas indispensables. Las solicitudes HTTP de recurso cruzado se producen cuando, desde un cliente web, se realizan llamadas a una serie de recursos o servicios residentes en un dominio distinto al que reside la aplicación. Por defecto, estas llamadas son bloqueadas por el propio navegador a no ser que se configure de forma correcta al nivel de acceso de los recursos permitiendo que puedan ser accedidos desde otro dominio distinto.

Una parte fundamental de la realización de servicios externos que sirven para que los clientes se comuniquen con el sistema, es la documentación de estos servicios. También, es necesario que esta documentación se mantenga actualizada. Estos servicios han sido documentados y, para comprobar que se cumple con la especificación de la documentación, se han realizado pruebas de integración donde se valida la documentación realizada sobre los propios servicios desarrollados. De esta forma, lo que se consigue es asegurar que los servicios están sincronizados con la documentación.

- **Capa con la publicación de la interfaz de gestión web**. Por último, este módulo es el encargado de proporcionar una herramienta de administración donde se pueden dar de alta los distintos procesos que gestiona la aplicación y visualizar el estado de ejecución de cada uno de ellos.

  Éste es el punto de entrada de la aplicación en cuanto a la creación de nuevos procesos. Se proporciona una interfaz web con una serie de formularios donde el administrador del sistema puede dar de alta nuevos procesos. Para ello, es necesario introducir una serie de información básica y una serie de parámetros de configuración para los distintos procesos.

  Los parámetros de configuración varían de forma dinámica en función del tipo de algoritmo a aplicar en el proceso. De forma adicional, también es necesario proporcionar los datos que forma el propio proceso. Estos datos tienen que tener un determinado formato para que puedan ser procesados de forma correcta por la aplicación.

  En cuanto a las pruebas realizadas sobre este módulo, en este caso, no se ha realizado ningún tipo de prueba. Esto es debido a que este módulo es puramente una interfaz gráfica.

4.1.2. Parte Cliente

En esta parte es donde residen las distintas operaciones que se realizan sobre los datos de cada uno de los procesos. Esta parte se comunica con la parte servidor para obtener la operación a procesar y, una vez procesada, envía el resultado de nuevo a la parte servidora.

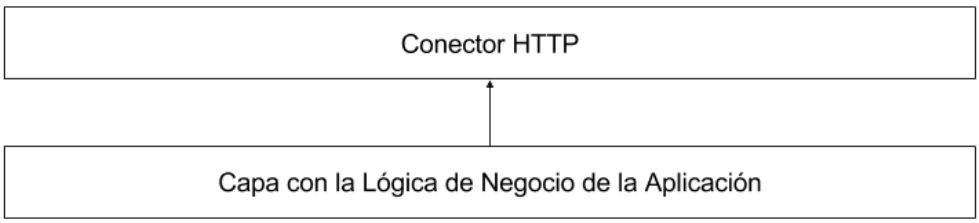
En la figura 4.3 se puede ver un esquema de la distribución de la arquitectura de la aplicación cliente.

Para realizar cada una de estas operaciones, se ha dividido la aplicación de la parte cliente en distintos módulos:

- **Módulo de comunicación HTTP**. Este módulo es el encargado de ofrecer una capa de abstracción sobre la comunicación con los servicios REST de la aplicación residente en el servidor.

  Este módulo permite tener centralizada en un único punto toda la configuración relacionada con la comunicación con los servicios REST y toda la parte relacionada con el

Figura 4.3: Arquitectura de la parte cliente



tratamiento de las respuestas y la composición de las llamadas. De esta forma, a la hora de realizar estas llamadas dentro de la aplicación bastará con usar este módulo sin necesidad de preocuparse de nuevo por la configuración.

- **Módulo de tratamiento y procesamiento de la información**. Este módulo es el encargado de, dada la operación devuelta por el servidor, procesar dicha operación y calcular el resultado.

  Dentro de este módulo se encuentran, por tanto, todos los objetos y métodos necesarios para realizar cada una de las operaciones y el sistema de decisión para escoger la operación determinada.

  Debido a que dentro de este módulo residen métodos cuyo comportamiento es vital para la aplicación, comprobar que el funcionamiento es el correcto es esencial. Es por esto que es necesario tener algún tipo de mecanismo que permita validar estos métodos. Para validarlos, se ha procedido a realizar una serie de pruebas automáticas unitarias, asegurando así el correcto funcionamiento de la aplicación.

4.2. Stack Tecnológico

La elección de un buen conjunto de tecnologías que no limiten el desarrollo y que lo faciliten es esencial. En la actualidad, existen numerosas tecnologías, cada una con unas características distintas, lo que permite poder elegir entre un espectro mucho mayor. Hay que tener en cuenta que no se debe acoplar la aplicación a la tecnología empleada, en la medida de lo posible, para que, en caso de encontrar algún tipo de problema, se pueda cambiar sin comprometer el desarrollo.

4.2.1. Parte Servidor

A la hora de elegir una buena tecnología para la parte de la aplicación residente en el servidor, es importante tener en cuenta las características que debe tener. La aplicación debe ser una aplicación sólida y con una gran capacidad de evolución. También, debe tener una gran capacidad de integración y una serie de *frameworks* que permitan mejorar el desarrollo y hacerlo más fácil. Debe ser una tecnología de largo recorrido pero actual.

En esta parte, se van a analizar las distintas tecnologías que se han empleado de forma común a los distintos módulos que componen la aplicación:

- **Lenguaje de programación**. Se ha escogido **Java 8** ya que es un lenguaje orientado a objetos y fuertemente tipado unido a que existe una gran cantidad de *frameworks* que tienen integraciones con este lenguaje. Además, es multiplataforma, lo que indica que se puede ejecutar en diversas máquinas, independientemente del entorno.

  Existen numerosas diferencias entre los lenguajes orientados a objetos y los lenguajes no orientados a objetos. Al ser un lenguaje orientado a objetos, la estructura del código mejora frente a un lenguaje no orientado a objetos. Esto es porque este tipo de lenguajes favorecen el modelado de los objetos en función de conceptos. De forma adicional, existen numerosos principios y patrones de diseño para los lenguajes orientados a objetos que facilitan la creación de estructuras con el objetivo de mejorar la calidad de la aplicación y el diseño.

  La ventaja de ser un lenguaje fuertemente tipado es que se mejora de forma considerable la detección de errores. Al ser un lenguaje fuertemente tipado, la detección de tipos incorrectos se detecta en tiempo de compilación, dando opción a solucionar este tipo de problemas antes de que la aplicación se encuentre funcionando. Es por esto que son más restrictivos, lo que hace que el número de errores sea menor. En cambio, los lenguajes no tipados no tienen esta comprobación de tipos, lo que hace que los errores no puedan ser detectados hasta que la aplicación se encuentre funcionando.

  Se ha elegido la versión 8 de este lenguaje ya que ofrece mejoras tanto en rendimiento como en funcionalidad considerables que favorecen y mejoran el desarrollo de aplicaciones. Con esta versión del lenguaje, se han realizado mejoras considerables en los mecanismos internos del funcionamiento de la máquina virtual de Java, lo que aumenta el rendimiento. Se han añadido nuevas funcionalidades, como la programación funcional, dentro del lenguaje lo que aporta más flexibilidad a la hora de desarrollar.

- **Herramienta de gestión del proyecto**. Debido a que la aplicación está dividida en distintos módulos, es necesario disponer de un sistema que permita gestionar estas dependencias y relaciones de forma correcta.

  Del mismo modo, de forma adicional, se usan una serie de dependencias externas, o librerías, dentro de cada uno de los módulos que es necesario también tenerlas controladas. Estas dependencias externas cubren cierta funcionalidad no cubierta por el lenguaje u ofrecen mejores alternativas. Es importante gestionar las versiones empleadas de estas dependencias de forma correcta.

  Asimismo, se debe dotar a la aplicación un ciclo de vida de desarrollo que sirva para estructurar en fases la compilación, ejecución de las pruebas, despliegue y documentación.

  Para todas estas tareas, se ha elegido emplear **Apache Maven** [1]. **Apache Maven** es una herramienta de gestión de construcción de proyectos y de generación de documentación. Es una herramienta de uso libre que permite mejorar de forma considerable el mantenimiento de las dependencias de los proyectos y mejorar el ciclo de vida del desarrollo de las aplicaciones. De forma adicional, tiene un sistema de extensiones que permiten generar documentación e informes sobre el código en función de una serie de parámetros.

[1] `https://maven.apache.org/`

- **Contenedor de inversión del control e inyección automática de dependencias**. Es muy importante destacar que, para la realización de un correcto diseño, se ha empleado un sistema de inyección de dependencias y un contenedor de inversión del control.

  La inyección de dependencias es el procedimiento mediante el cual, en lugar de ser la clase la encarga de construir cada una de sus dependencias, estas dependencias son pasadas a esta clase ya construidas. De esta forma, se consigue desacoplar el sistema, con todos los beneficios que esto tiene. Gracias a la inyección de dependencias, se permite también simplificar el código resultante y tener el código más autocontenido, ya que las clases no tienen que ser responsables de conocer todos los mecanismos para crear sus dependencias.

  La inversión del control es el mecanismo mediante el cual se invierten la lógica de ejecución de las sentencias del programa.

  Para automatizar este proceso, se ha utilizado **Spring** [2]. **Spring** es una tecnología que ofrece su propio contenedor de inversión del control y su propio sistema de inyección de dependencias. De esta forma, se respeta uno de los principios de diseño de *software*.

  Mediante estos dos conceptos, el código está menos acoplado, lo que mejora considerablemente el diseño y el mantenimiento de la aplicación. Todos los módulos realizados en la aplicación en la parte del servidor emplean **Spring** para la inyección de dependencias.

- **Realización de pruebas**. Como ya se ha mencionado, en todos los módulos de la aplicación se han desarrollado una serie de pruebas con el objetivo de validar el comportamiento de la aplicación.

  Para la realización de las pruebas unitarias, se ha empleado **JUnit** [3]. **JUnit** es un *framework* que permite la realización de pruebas dentro del código de la aplicación. Este *framework* ofrece una serie de métodos con los que se puede comprobar el funcionamiento. A estos métodos se les denomina *Asserts*. Con estos métodos de comprobación, se pueden realizar pruebas lo suficientemente sólidas como para que informen en caso de que, debido a una modificación del código, si algo ha dejado de funcionar o no.

  Para aumentar la legibilidad de las pruebas, existen distintos *frameworks* que mejoran la sintaxis a la hora de escribir los casos de prueba. En este caso, se ha empleado **AssertJ** [4]. **AssertJ** es un *framework* que, mediante su API fluida, mejora la sintaxis de los casos de prueba.

  Por último, para el caso en el que se necesario comunicarse con una entidad externa y seguir realizando pruebas unitarias, se ha empleado **Mockito** [5]. **Mockito** es un *framework* que permite la creación de dobles de prueba con los que poder realizar pruebas unitarias en lugar de pruebas de integración. Es muy útil cuando se quiere probar de forma aislada un componente en particular que depende de otros componentes.

A continuación, se va a analizar la tecnología no común empleada en cada uno de los distintos módulos de la aplicación:

- **Capa de acceso a Base de Datos**. Para realizar la comunicación y la conexión con la base de datos se ha empleado **MyBatis** [6]. Como base de datos se ha empleado un

[2] `https://spring.io/`
[3] `http://junit.org/junit4/`
[4] `http://joel-costigliola.github.io/assertj/`
[5] `http://mockito.org/`
[6] `http://www.mybatis.org/mybatis-3/es/`

contenedor de **Docker** [7] con una base de datos **PostgreSQL** [8]. Como se ha mencionado anteriormente, todo el sistema está gestionado con **Spring**.

MyBatis es un *framework* de persistencia de base de datos que soporta SQL. SQL es el lenguaje principal de acceso a base de datos para realizar consultas y para insertar, modificar y eliminar datos. **MyBatis** ofrece una capa de abstracción por encima de JDBC, que es el sistema de conexión con base de datos de Java, de tal forma que toda la gestión de la comunicación y la configuración está implementada por **MyBatis**. Gracias a esta abstracción, se simplifica mucho el desarrollo de este módulo.

Las consultas realizadas a base de datos se han desarrollado mediante ficheros XML. Toda la gestión de la seguridad también está gestionada por el propio *framework*, el que ofrece por defecto el uso de sentencias preparadas con las que se puede evitar el ataque por *SQL Injection*.

De forma adicional, el *framework* incorpora por defecto dos sistemas de caché con los que mejorar los tiempos de respuesta de la base de datos. Un sistema de caché es un sistema que tiene como objetivo almacenar la información de respuesta de un sistema y proporcionar dicha información en un tiempo menor, de tal forma que la comunicación es más rápida. El primer nivel de caché siempre está activo de forma automática y el segundo sistema de caché se puede activar de forma manual.

En cuanto a la base de datos, como se ha mencionado, se ha empleado **PostgreSQL**. Esta base de datos es una base de datos relacional, aunque también es posible configurarla de tal forma que se comporte como una base de datos no relacional. Es una base de datos gratuita y de elevado rendimiento lo que la convierte en una de las grandes opciones. Para la instalación de la base de datos, a modo de desacoplar la instalación y proporcionar un entorno fiable, se ha considerado emplear **Docker**.

Docker es un proyecto de código libre que proporciona una serie de herramientas para desplegar aplicaciones dentro de contenedores de *software*. Estos contenedores son virtualizaciones a nivel de sistema operativo Linux. También es posible realizar estas virtualizaciones sobre el resto de sistemas operativos mediante herramientas adicionales. Lo que se pretende conseguir con esto es desacoplar la instalación y ofrecer un mecanismo rápido para instalar esta base de datos con toda la configuración ya realizada.

- **Capa con la lógica de negocio de la aplicación**. Este módulo, al ser el que contiene la lógica de la aplicación, también se encuentra gestionado por **Spring**.

  Para una mejor integración, se han desarrollado una serie de interfaces para mejorar el sistema de inyección de dependencias de **Spring** y dar vía libre a la extensión del proyecto mediante la implementación de estas interfaces.

- **Capa con la publicación de servicios REST**. Para la realización de los servicios REST se ha utilizado **Jersey** [9].

  Jersey es un *software* de código libre para el desarrollo de servicios REST. Se basa principalmente en las implementaciones de referencia JSR-311 y JSR-339.

  Para realizar la documentación de los servicios REST, se ha empleado **API Blueprint** [10]. **API Blueprint** es un sistema de documentación de servicios REST que permite la

[7] `https://www.docker.com/`
[8] `http://www.postgresql.org.es/`
[9] `https://jersey.java.net/`
[10] `https://apiblueprint.org/`

colaboración sobre el desarrollo de los mismo. A partir de esta documentación, aparecen una serie de herramientas que permiten comprobar si la documentación se corresponde a los servicios desarrollados. Una de estas herramientas es **Dredd** [11].

Dredd es un programa realizado con NodeJS que, dado un fichero con la documentación de una serie de servicios REST y una ruta con el endpoint de la implementación de estos servicios, permite comprobar que los servicios cumplen con la documentación proporcionada. De esta forma, mantener la documentación actualizada es más fácil ya que se puede comprobar contra los propios servicios desarrollados.

- **Capa con la publicación de la interfaz de gestión web**. Para la realización de esta capa, a parte de **Spring** como ya se ha mencionado, se ha empleado **JSF** y **PrimeFaces** [12] como librería de componentes.

  JSF (JavaServerFaces) es un *framework* que permite el desarrollo de interfaces de usuario en Java de una forma mucho más sencilla. Mediante un sistema de plantillas se pueden construir las distintas páginas por las que estará formada la aplicación. **JSF** proporciona sus propios componentes visuales con los que ya se aplican determinados comportamientos y estilos de manera automática.

  Para mejorar estos componentes, se ha empleado a conocida librería de componentes **PrimeFaces**. **PrimeFaces** ofrece multitud de componentes visuales con los que dotar a la interfaz una mayor funcionalidad de forma rápida y sencilla. Estos componentes van desde la representación de gráficas hasta componentes para la subida de ficheros.

Por último, es necesario desplegar esta aplicación en un contenedor de *servlets*. Para ello, se ha empleado **Apache Tomcat** [13]. La arquitectura, con la tecnología resultante, se puede observar en la figura 4.4.

4.2.2. Parte Cliente

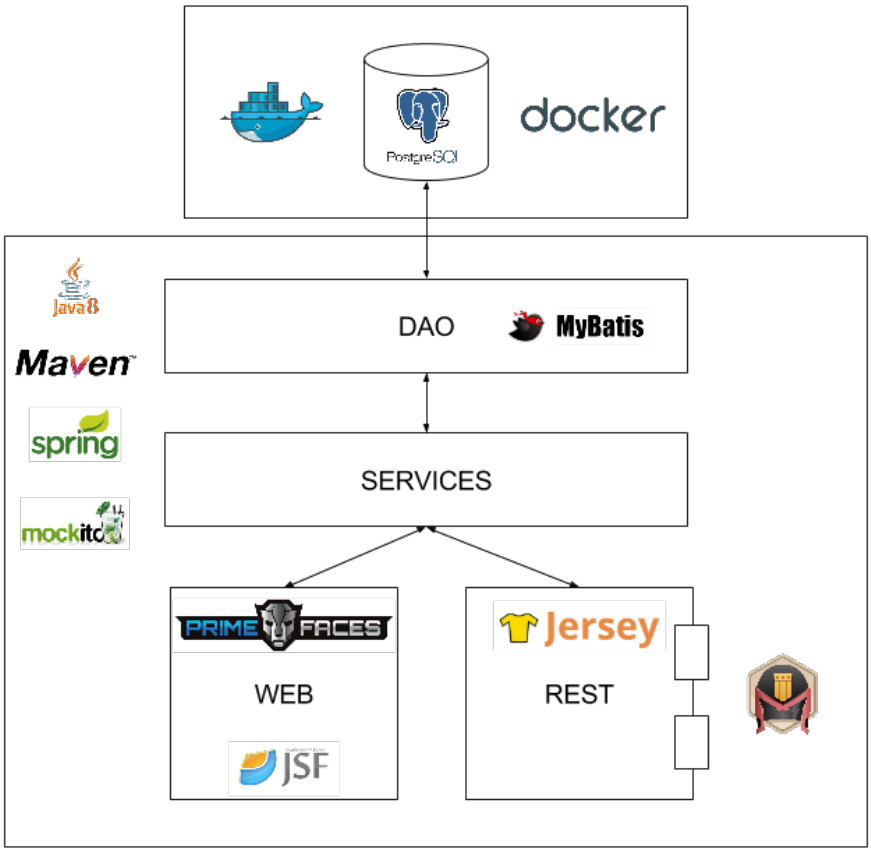
A la hora de elegir una buena tecnología para la aplicación del lado de cliente se han analizado las distintas tecnologías predominantes a la hora de desarrollar aplicaciones web. Entre estas tecnologías, la que más destaca tanto por su rendimiento como por su capacidad de evolución es **Angular2** [14]. Es por esto que toda la aplicación del lado de cliente se ha realizado con **Angular2**.

Gracias a **Angular2**, el desarrollo de aplicaciones del lado de cliente se ha mejorado enormemente ya que, a parte de todas las ventajas del propio *framework* se le añade la principal ventaja de **TypeScript** [15]. Con **TypeScript** se dota de tipado al código JavaScript, código que anteriormente no era tipado, lo que provocaba numerosos errores en tiempo de ejecución de las aplicaciones.

Actualmente, los navegadores no soportan directamente el código implementado en **TypeScript**, pero esto se soluciona mediante el proceso de transpilado. El proceso de transpilado es el proceso mediante el cual se pasa de código en TypeScript a código JavaScript compatible con los navegadores. De forma similar, este procedimiento es bastante parecido al proceso de compilado.

[11] `https://github.com/apiaryio/dredd`
[12] `http://primefaces.org/`
[13] `http://tomcat.apache.org/`
[14] `https://angular.io/`
[15] `https://www.typescriptlang.org/`

Figura 4.4: Tecnología de la parte servidor



Angular2 también aprovecha las novedades de **ECMAScript6**, nueva versión del lenguaje que trae numerosas mejoras que afectan a diversas características desde el rendimiento hasta la funcionalidad ofrecida por el lenguaje.

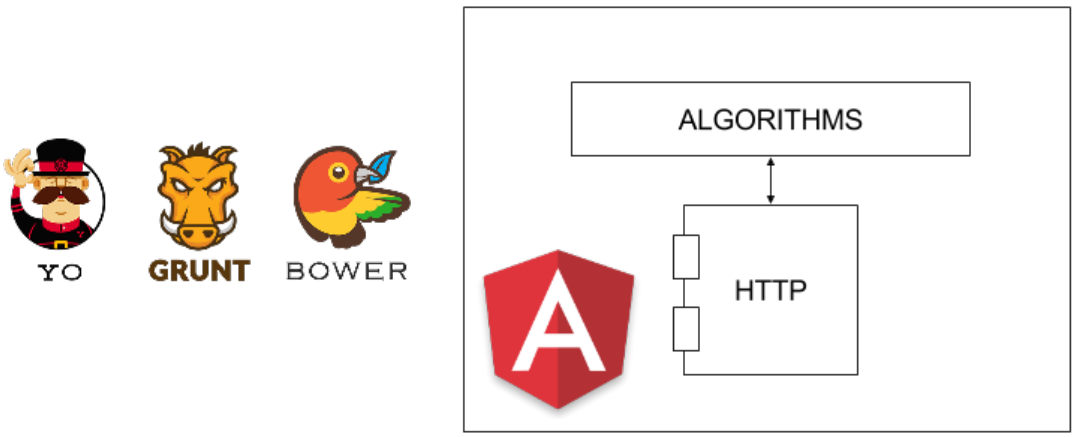
Afortunadamente, **Angular2** también se integra con numerosos *frameworks*. Dentro de este conjunto de *frameworks* también se han utilizado aquellos que favorecen la realización de pruebas en el código. Estos *frameworks* son **Karma** [16] y **Jasmine** [17]. **Karma** es un *framework* que proporciona un entorno de ejecución para los distintos métodos de prueba y **Jasmine** proporciona el lenguaje para programar cada uno de los distintos métodos de prueba. Con todo esto, el sistema gana en robustez.

También, para la gestión de todas estas dependencias y para el despliegue y compilación de la aplicación se han empleado **Grunt** [18], **Bower** [19] y **Yeoman** [20]. Gracias a este conjunto de *frameworks* se pueden obtener las dependencias, se puede compilar el proyecto, ejecutar las pruebas de forma automática y se puede guardar la configuración del proyecto.

[16] `https://karma-runner.github.io/1.0/index.html`
[17] `http://jasmine.github.io/`
[18] `http://gruntjs.com/`
[19] `https://bower.io/`
[20] `http://yeoman.io/`

Por último, se adjunta el esquema con la arquitectura resultante en la figura 4.5.

Figura 4.5: Tecnología de la parte cliente



4.3. Operativa de la Aplicación

En este apartado se va a analizar la operativa de la aplicación y cada una de las piezas importantes por las que está compuesta.

El objetivo de la aplicación es distribuir una serie de operaciones a una serie de clientes para que las procesen, realicen los cálculos que sean necesarios y envíen el resultado al servidor. De esta forma, mediante la distribución de las operaciones, lo que se pretende conseguir es un sistema que, dado un determinado algoritmo, se distribuyan las operaciones de este algoritmo entre una serie de clientes para paralelizar el proceso.

Para ello, la aplicación se basa en el funcionamiento de Map-Reduce, algoritmo bastante extendido dentro de la computación distribuida de grandes cantidades de datos. Para emplear Map-Reduce, será necesario realizar un análisis de cada uno de los algoritmos e implementar el algoritmo de forma distribuida, con todas las consecuencias y consideraciones que eso implica.

Dentro de la aplicación, se tienen distintos mecanismos para distribuir las operaciones y para realizar las acciones de *map* y *reduce*.

4.3.1. Fase de *Split*

Cuando llegan los datos a la aplicación, es necesario distribuirlos. Para distribuirlos, es necesario realizar agrupaciones. Para la elaboración de estas particiones es donde entra en juego el *splitter*. El *splitter* es el encargado de particionar los elementos y gestionar el acceso a ellos para evitar colisiones.

El *splitter* tiene varios parámetros de configuración. Uno de ellos es la cantidad de datos de cada agrupación. De esta forma, se puede variar para obtener mejores rendimientos en distintos entornos. Del mismo modo, también se puede configurar el tiempo en el que un dato estará bloqueado. Los datos se bloquean al ser seleccionados para formar parte de una partición. Esto es así para evitar que un dato sea enviado a dos clientes de forma simultánea. Si cuando se seleccionan los datos de una partición se seleccionan sólo aquellos datos que no han sido marcados y cuya marca ya haya expirado, se garantiza evitar las colisiones.

La implementación del *splitter* reside en la aplicación del lado del servidor y es común para todos los algoritmos de la aplicación.

4.3.2. Fase de *Map*

Cuando desde el cliente se obtienen los datos, en función del estado del proceso, se realizarán una serie de operaciones u otras. En el caso de las operaciones de *map* deben ser implementadas de forma específica para cada uno de los algoritmos de la aplicación.

La implementación del *mapper* reside en la aplicación del lado del cliente y es independiente para todos los algoritmos de la aplicación.

4.3.3. Fase de *Reduce*

Cuando desde el cliente se obtienen los datos, en función del estado del proceso, se realizarán una serie de operaciones u otras. En el caso de las operaciones de *reduce* deben ser implementadas de forma específica para cada uno de los algoritmos de la aplicación.

La implementación del *reducer* reside en la aplicación del lado del cliente y es independiente para todos los algoritmos de la aplicación.

4.4. Algoritmo K-Means

En este apartado se va a mostrar el funcionamiento de la aplicación para un determinado algoritmo, desde el momento en el que es creado dentro del sistema hasta el momento en el que se ha obtenido el resultado final. El algoritmo elegido es el algoritmo K-Means.

El algoritmo K-Means es un algoritmo de *clustering* que tiene como objetivo agrupar una serie de datos, n, en una serie de grupos, k, en función de la cercanía de los datos de cada uno de los grupos.

4.4.1. Creación del Proceso

En este primer punto es cuando se va a dar de alta el proceso dentro de la aplicación para que pasen a distribuirse sus operaciones con el objetivo de obtener las asignaciones finales de las agrupaciones.

En este caso, mediante la interfaz web proporcionada por la aplicación del servidor, rellenando un formulario se podrá dar de alta este nuevo proceso. El formulario contiene los siguientes campos:

- **Nombre**. Éste será el nombre que introducirá el usuario para identificar el proceso. De esta forma, cuando se busque información sobre este proceso bastará con referenciar este nombre. Por tanto, este nombre debe ser único dentro de la aplicación. Este campo del formulario es obligatorio rellenarlo independientemente del tipo de algoritmo que se seleccione.
- **Algoritmo**. Éste será el tipo de algoritmo que se va a ejecutar sobre los datos proporcionados. Es importante destacar que en función del algoritmo seleccionado el formulario solicitará una serie de parámetros de configuración u otros. En este punto, la aplicación sólo soporta el algoritmo K-Means.
- **Número de iteraciones**. Número de iteraciones que se deberán completar hasta llegar al resultado final por el algoritmo. Este es un parámetro particular del algoritmo. En este caso, K-Means es un algoritmo que se basa en iteraciones donde se van actualizando los centroides. Este parámetro indicará este número de iteraciones.
- **Centroides**. Número de centroides que marcarán las agrupaciones del algoritmo. Éste es un parámetro particular del algoritmo. En función de esta configuración se obtendrá un resultado u otro.
- **Fichero con los datos a procesar**. Por último, será necesario introducir un fichero donde se encuentren los datos a procesar por el algoritmo. Este fichero tiene que tener un determinado formato para que sea aceptado por la aplicación. En este caso, deberá contener cada muestra en una línea con los atributos separados por comas.

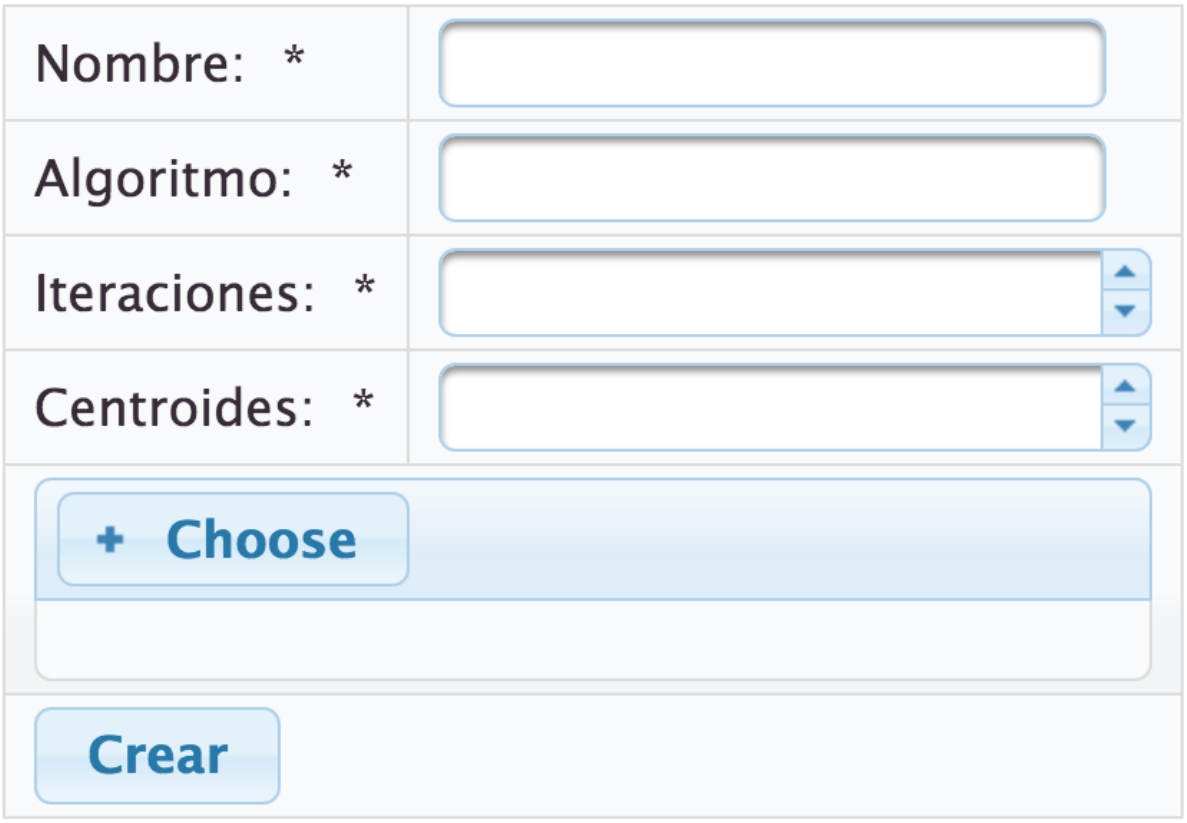
A continuación, en la figura 4.6, se puede observar la pantalla referente al formulario de la aplicación de gestión del servidor.

Una vez introducidos los datos y haber pulsado el botón de crear, se procesará la solicitud para dar de alta este nuevo proceso dentro de la aplicación. El procedimiento para dar de alta este nuevo proceso consiste en una serie de pasos que serán ejecutados de forma transaccional para evitar que cualquier posible error durante el proceso deje en un estado erróneo la base de datos:

1. Se dará de alta en base de datos el nuevo proceso con la información introducida por el usuario. De forma adicional, se definirá el número de la iteración actual y el estado inicial del proceso. Es importante mencionar en este punto que los estados del proceso puede ser *mapper* o *reducer*. Estos estados ya se vieron en el apartado anterior.
2. Una vez dado de alta el proceso, se crearán las tablas de almacenamiento temporal de la información necesaria para ejecutar las operaciones del proceso. Estas tablas almacenarán la información introducida en el fichero como los datos a procesar y también se creará una tabla para almacenar la información de los centroides.
3. Acto seguido, se insertarán los datos leídos del fichero en la tabla de almacenamiento temporal de los datos. A parte de los datos, esta tabla contiene información sobre la asignación del dato al centroide y sobre el tiempo en el que ha sido procesado el dato.
4. Por último, se seleccionarán los centroides y se insertarán en la tabla temporal. El mecanismo de selección de centroides consiste en, dados los datos obtenidos, ordenarlos aleatoriamente y escoger un determinado número de ellos de forma aleatoria.

En este punto, el proceso estará dado de alta correctamente en el sistema y estará listo para poder empezar a ser procesado.

Figura 4.6: Formulario de alta de procesos en la aplicación



4.4.2. Fase de Operaciones del Estado *Mapper*

En este punto, con el proceso creado, es el turno de la aplicación cliente de empezar con el proceso. Para ello, lo que hará es solicitar la operación que tiene que realizar a la aplicación servidora.

En el estado de *mapper*, la respuesta del servidor contendrá una serie de datos proporcionados por el *splitter* y los centroides. Además, también se incluye información general relativa al propio proceso. Es importante destacar que, para evitar que se procesen los mismos datos por varias aplicaciones clientes, en el momento de devolver los datos, son marcados con la fecha de solicitud. De esta forma, sólo se devolverán los datos que no estén marcados o cuyo tiempo de marcado sea anterior al minuto.

La aplicación cliente realizará los cálculos necesarios para asignar cada uno de los datos al centroide más cercano. Cuando se haya completado, enviará los resultados al servidor. Para ello, enviará una lista con las asignaciones producidas entre los datos y los centroides.

Cuando esta parte ha concluido, la aplicación cliente vuelve a solicitar la siguiente operación.

4.4.3. Cambio de Estado de *Mapper* a *Reducer*

Cuando todos los datos han sido asignados a un centroide, la aplicación del servidor cambiará el estado del proceso a *reducer* para comenzar con la siguiente fase. En esta fase se reiniciarán

los valores de las fechas de marcado de los datos.

4.4.4. Fase de Operaciones del Estado *Reducer*

En el estado de *reducer*, la respuesta del servidor contendrá una serie de datos proporcionados por el *splitter* y las asignaciones a los centroides. Además, también se incluye información general relativa al propio proceso. Es importante destacar que, para evitar que se procesen los mismos datos por varias aplicaciones clientes, en el momento de devolver los datos, son marcados con la fecha de solicitud. De esta forma, sólo se devolverán los datos que no estén marcados o cuyo tiempo de marcado sea anterior al minuto.

La aplicación cliente realizará los cálculos necesarios para hacer la media parcial de cada una de las asignaciones y devolver este resultado al servidor.

Cuando esta parte ha concluido, la aplicación cliente vuelve a solicitar la siguiente operación.

4.4.5. Cambio de Estado de *Reducer* a *Mapper*

Cuando todos los centroides se han actualizado, la aplicación del servidor cambiará el estado del proceso a *mapper* para comenzar con la siguiente iteración. Por tanto, la iteración actual del proceso aumentará. En esta fase se reiniciarán los valores de las fechas de marcado de los datos.

4.4.6. Fase de Finalización del proceso

Cuando se han repetido las fases anteriores hasta que el número de iteraciones se haya cumplido, el proceso finalizará. El resultado se podrá obtener obteniendo la información de las tablas de base de datos.

5

Validación del Sistema

Una vez se ha analizado la metodología que se ha empleado para desarrollar el proyecto y la arquitectura resultante, el siguiente paso es validar el correcto funcionamiento del sistema.

El objetivo es comprobar que los resultados obtenidos con el sistema desarrollado son equivalentes, desde un punto de vista estadístico, con los resultados que se pueden obtener de otros sistemas empleados para el análisis de datos.

Del mismo modo, también se pretenden analizar los tiempos obtenidos con el sistema y compararlos con el de las otras soluciones. De esta forma, se podrá saber si se tratan de unos tiempos competitivos o, en cambio, son tiempos muchos peores, lo que dificultaría su posibilidad de uso.

Para realizar todas estas pruebas, es necesario disponer de unos datos con los que realizarlas y un sistema de análisis de datos adicional al desarrollado.

Por ello, debido a que se trata simplemente de una prueba de validación, los datos que se han elegido son los datos correspondientes al conjunto de muestra Iris-Setosa. Este conjunto de datos se trata de un conjunto de datos multivariante que contiene ejemplos de tres especies distintas de flores, cada ejemplo con una serie de características o atributos. El conjunto de datos contiene 150 datos en total divididos en tres especies con 50 datos para cada una de ellas. Estas especies son: Iris setosa, Iris virginica e Iris versicolor. Cada uno de estos datos tiene, a su vez, cuatro propiedades: largo de los pétalos, largo de los sépalos, ancho de los pétalos y ancho de los sépalos. A estos datos, se les aplicará el algoritmo K-Means con tres centroides, debido a que hay tres clases posibles de datos, y se comprobará que el resultado final de los centroides es estadísticamente equivalente.

En cuanto al sistema de análisis, concretamente, para realizar la validación, se va a emplear el programa Weka[1]. Weka es un programa que contiene un conjunto de algoritmos para realizar análisis de datos. A través de una interfaz gráfica, el programa permite importar un conjunto de datos y aplicar ciertos algoritmos sobre ellos. Al aplicar el algoritmo, permite configurarlo de distintas formas, indicando el número de iteraciones o, en el caso de K-Means, el número de centroides.

[1] `http://www.cs.waikato.ac.nz/ml/weka/`

Una vez se ha escogido el conjunto de datos sobre los que aplicar el algoritmo y el sistema con el que comparar la solución, se va a proceder a realizar cada uno de los procesos de comprobación.

5.1. Comprobación de la Calidad de los Resultados

Como se ha mencionado, lo primero que se va a tratar es la comprobación de si los resultados obtenidos son estadísticamente equivalentes a los obtenidos mediante otro sistema.

Para este caso de uso, se van a comparar los resultados obtenidos de la ejecución del algoritmo K-Means con Weka con los resultados obtenidos de la ejecución del algoritmo K-Means con EphemeML.

Para comprobar esta equivalencia estadística, se van a comprobar las ejecuciones de las dos herramientas: EphemeML y K-Means y se van a analizar los resultados.

Para la ejecución de Weka, se ha configurado con tres centroides y tres iteraciones. Los resultados obtenidos se pueden observar en la tabla 5.1.

Tabla 5.1: Resultado centroides Weka

	Weka			
centroide 1	6.1	2.9	4.7	1.4
centroide 2	6.2	2.9	4.3	1.3
centroide 3	6.9	3.1	5.1	2.3

Para la ejecución de EphemeML, se ha configurado de la misma forma que Weka, con tres centroides y tres iteraciones. Los resultados obtenidos se pueden observar en la tabla 5.2.

Tabla 5.2: Resultado centroides EphemeML

	EphemeML			
centroide 1	6.0	2.7	4.8	1.6
centroide 2	6.4	3.0	4.1	1.1
centroide 3	7.2	2.9	5.2	2.4

Como se puede observar, los resultados no son iguales, pero son similares. Esto indica que el modelo propuesto por la aplicación EphemeML proporciona resultados dentro del rango válido de soluciones para el cálculo de los centroides del algoritmo. Del mismo modo, las asignaciones sobre los datos han resultado ser las mismas, por lo que se corroboran los resultados obtenidos.

5.2. Comprobación del Tiempo de los Resultados

Como se ha mencionado, lo segundo que se va a tratar es la comprobación de si los resultados obtenidos tienen tiempos competitivos.

Para este caso de uso, se van a comparar los resultados de los tiempos obtenidos de la ejecución del algoritmo K-Means con Weka con los resultados de los tiempos obtenidos de la ejecución del algoritmo K-Means con EphemeML.

Los tiempos de la aplicación EphemeML puede separarse en dos tiempos, los tiempos obtenidos en la aplicación cliente, con el cálculo de las operaciones de tipo *map* y las opera-

ciones de tipo *reduce*; y los tiempos obtenidos en la aplicación servidor, que son todos aquellos dedicados a controlar el acceso a datos.

Los tiempos obtenidos en la aplicación cliente (en milisegundos), para un determinado número de datos a procesar por partición, se puede observar en la tabla 5.3.

Tabla 5.3: Tiempos de la aplicación cliente de EphemeML para un tamaño determinado de split

	EphemeML (ms)									
Split: 150	3	5	7	11	2	4	4	7	9	5
Split: 1.500	21	34	27	25	23	29	22	24	21	21
Split: 15.000	65	67	64	58	67	61	70	57	68	62

Los tiempos obtenidos en la aplicación de Weka (en milisegundos), para un determinado número de datos a procesar por partición, se puede observar en la tabla 5.4.

Tabla 5.4: Tiempos de Weka para un tamaño determinado de split

	Weka (ms)									
Split: 150	1	1	1	1	1	1	1	1	1	1
Split: 1.500	2	4	3	2	4	3	2	5	4	3
Split: 15.000	13	14	11	10	20	14	16	13	11	18

Con esto lo que se puede observar es que los tiempo en cómputo de operaciones son mayores en la aplicación cliente de EphemeML. A estos tiempos, a parte, hay que añadirle los tiempos derivados de la comunicación con el servidor y los procesos de bases de datos. Pese a esto, la ventaja principal del sistema es que estos datos se pueden distribuir, mejorando en parte el tiempo de procesamiento.

De forma adicional, pese al tiempo, otra de las ventajas que ofrece EphemeML es el poco consumo de recursos, ya que todas las tareas se distribuyen en los clientes. Es por esto que, aunque se tarde más en realizar el análisis sobre los datos, es importante destacar que el consumo se realiza sobre los clientes, que serán externos en el sistema.

6
Conclusiones

Con la realización de este proyecto se ha demostrado el correcto funcionamiento de la aplicación desarrollada. El objetivo de la aplicación es, mediante la técnica de Map-Reduce, distribuir una serie de operaciones a través de un servidor a una serie de clientes para que realicen los cálculos necesarios.

Esto se ha conseguido de forma satisfactoria, con un rendimiento parecido, cuando se emplean particiones pequeñas, a algunas de las soluciones actuales y con la obtención de los mismos resultados. Por tanto, se puede decir que el sistema ha sido validado y proporciona resultados correctos.

Este proyecto también ha sido desarrollado empleando metodologías ágiles. Del mismo modo, también se han aplicado principios y patrones de diseño de *software* a la codificación del mismo.

Con todo esto lo que se ha conseguido es un proyecto con una alta calidad. El sistema es un sistema robusto, escalable y con una buena capacidad de extensión. Esto es importante ya que, como se va a ver en el trabajo futuro, las siguientes evoluciones del programa requieren que el propio programa esté preparado para ellas.

7

Trabajo Futuro

Existen distintas vías para aumentar la funcionalidad y la estabilidad del proyecto. Estas vías de trabajo son las siguientes:

- **Aumentar el número de algoritmos soportados por el sistema**. El sistema actual sólo tiene soportado un algoritmo, el algoritmo de K-Means. Para un futuro, el objetivo es aumentar el número de algoritmos soportados por el sistema. Para ello, se debe realizar un análisis sobre la forma de distribuir las operaciones de los algoritmos que vayan a estar presentes en la aplicación y un sistema para optimizar la distribución de dichas operaciones.
- **Mejorar la capacidad de configuración de los algoritmos con un mayor número de propiedades**. Actualmente, las únicas propiedades que se pueden configurar son aquellas relacionadas con el algoritmo K-Means. Al añadir un mayor número de algoritmos, cada uno tiene asociadas una serie de propiedades. Por tanto, se debe gestionar esto de forma dinámica para cada uno de los algoritmos.
- **Mejorar el procesamiento de los ficheros de datos para realizarlo en un proceso asíncrono**. Actualmente, la subida de ficheros y el procesamiento del mismo se produce de forma síncrona. Este mecanismo es válido para ficheros que no tengan un número elevado de datos. Para estos casos, es necesario que el procesamiento se realice de forma asíncrona para evitar hacer esperar al usuario. Para conseguir esto, se puede emplear un sistema de colas donde se vayan procesando los ficheros mediante un consumidor.
- **Implementación de un panel de control completo donde poder obtener las estadísticas de cada uno de los procesos**. Actualmente, la información de los procesos sobre las estadísticas es bastante limitada. Una importante mejora sería añadir un panel de control que aportase información más detallada sobre el estado de ejecución de los procesos y una serie de estadísticas derivadas.
- **Permitir la generación de documentación sobre el resultado final de la ejecución de cada uno de los procesos**. Para obtener el resultado final, es necesario

consultar directamente la base de datos. La evolución del sistema pasaría por desarrollar un mecanismo de notificaciones que permitan avisar al usuario de la finalización del proceso y envíen los resultados obtenidos.

- **Permitir configurar el sistema para que, dados una serie de procesos de alta, se gestionen las operaciones de forma automática**. El sistema permite el procesado de varios procesos siempre y cuando sea referenciado cada uno en las llamadas de los clientes. Una posible mejora sería la unificación de las llamadas para la obtención de las operaciones en un único punto, de tal forma que la gestión no recaiga en los clientes sino en el propio servidor.
- **Mejorar la comprobación de la validez de los datos enviados por los clientes**. No se están comprobando si los clientes envían datos correctos o incorrectos. Se podrían implementar un mecanismo, no muy costoso, que permita detectar este tipo de errores.
- **Implementación de un sistema de lista negra para descartar clientes con una tasa elevada de errores en los cálculos**. Al poder existir clientes con un elevado número de errores, es importante poder filtrar estos clientes para que no afecten al resultado final del proceso.

Glosario de acrónimos

- **TDD:** Test Driven Development
- **CORS:** Cross-origin resource sharing
- **HTTP:** Hypertext Transfer Protocol
- **JSON:** JavaScript Object Notation
- **SRP:** Single Responsibility Principle
- **OCP:** Open/Closed Principle
- **LSP:** Liskov Substitution Principle
- **ISP:** Interface Segregation Principle
- **DIP:** Dependendy Injection Principle
- **SQL:** Structured Query Language

Bibliografía

[1] Hunt Andrew and Thomas David. The pragmatic programmer: From journeyman to master, 2000.

[2] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[3] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and Edward Y Chang. Parallel spectral clustering in distributed systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(3):568–586, 2011.

[4] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[5] Tsui Frank Karam, Orlando. Essentials of software engineering. 2007.

[6] Joshua Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[7] Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[8] Robert C Martin. Principles of ood. *Von butunclebob. com: http://butunclebob. com/ArticleS. UncleBob. PrinciplesOfOod abgerufen*, 1995.

[9] Robert C Martin. The dependency inversion principle. *C++ Report*, 8(6):61–66, 1996.

[10] Robert C Martin. The open-closed principle. *More C++ gems*, pages 97–112, 1996.

[11] Robert C Martin. Design principles and design patterns. *Object Mentor*, 1:34, 2000.

[12] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[13] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[14] Juan J Merelo-Guervos, Antonio Mora, J Albert Cruz, Anna I Esparcia-Alcazar, and Carlos Cotta. Scaling in distributed evolutionary algorithms with persistent population. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE, 2012.

[15] Richard J Mitchell. *Managing complexity in software engineering*. Number 17. IET, 1990.

[16] Roger S Pressman and Jose Maria Troya. *Ingeniería del software*. Number 001.64 P74s. McGraw Hill, 1988.

[17] Ken Schwaber. *Agile project management with Scrum*. Microsoft press, 2004.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[19] Frank Tsui, Orlando Karam, and Barbara Bernal. *Essentials of software engineering*. Jones & Bartlett Publishers, 2013.

[20] Hans Van Vliet. Software engineering. 2008.

[21] Pree Wolfgang. *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.

[22] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[23] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In *Cloud computing*, pages 674–679. Springer, 2009.