Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

FRAMEWORK EN PHP PARA EL DESARROLLO Y MANTENIMIENTO DE APLICACIONES EMPRESARIALES

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Enrique Rodríguez Pérez

Tutor: M.ª Carmen Penadés Gramage

2016 - 2017

Quisiera dedicar este trabajo final de grado a varias personas que han estado conmigo en todo momento:

A mi pareja Nahid por ayudarme con los aspectos formales y apoyarme en todo momento. Su presencia ha sido clave para que este trabajo saliera adelante.

A mis tutores de la empresa, Enrique y Chusa, por hacerme sentir su casa como si fuera la mía y por todo el tiempo invertido en mí.

A mi madre y a mi hermano, por estar ahí a las buenas y a las malas.

A mi tutora Mª Carmen Penadés por sus pautas y correcciones que han sido imprescindibles en la realización de este trabajo.

Muchas gracias,

Enrique Rodríguez.

Resumen

Este Trabajo Final de Grado en Ingeniería Informática es el resultado de las prácticas en empresa en Ferrocarrils de la Generalitat Valenciana (FGV). El trabajo aquí expuesto trata de la programación orientada a objetos y consta de dos partes. En la primera parte se estudia el concepto teórico de *framework* y el análisis e interpretación de un *framework* dado; y en la segunda parte se aborda el desarrollo de una aplicación real empresarial a partir del *framework* dado, con las consiguientes extensiones de sus funcionalidades. La metodología empleada es Scrum. Este trabajo muestra las posibilidades de uso y aplicación del *framework*, así como sus posibles mejoras e implementaciones.

Palabras clave: *framework*, objeto, Scrum

Abstract

This Bachelor Thesis is the result of my internship in Ferrocarrils de la Generalist Valenciana (FGV). This paper addresses object-oriented programming and consists of two parts. In the first one a study of the theoretical concept of framework and an analysis of a given framework is developed. In the second part, the development of a real company application is addressed, along with the improvements required by the framework that has been used. The applied methodology is Scrum. This paper shows not only the framework usage and applicability, but also its possible enhancements and implementations.

Keywords: framework, object, Scrum

Resum

Aquest Treball Final de Grau en Enginyeria Informatica és el resultat de les practiques en empresa en Ferrocarrils de la Generalitat Valenciana (FGV). El treball ací exposat tracta de la programació orientada a objectes i consta de dues parts. En la primera part s'estudia el concepte teòric de *framework* i l'anàlisi i interpretació d'un *framework* dona; en la segona part s'aborda el meu desenvolupament d'una aplicació real empresarial a partir del *framework* donat, amb les consegüents extensions de la seua funcionalitat. La metodología empleada és Scrum. Aquest treball mostra les posibilitats d'ús i aplicació del *framework,* aixina com les seus posibles millores e implementacions.

Paraules claus: *framework*, objecte, Scrum

Tabla de contenidos

1 Introducción ... 6
1.1 Motivación ... 6
1.2 Objetivos ... 7
1.3 Estructura del documento ... 8
2 Estado del arte ... 10
2.1 Consideraciones teóricas ... 10
2.1.1 Framework ... 10
2.1.2 Patrones de diseño ... 11
2.2 PHP Vs JavaScript ... 14
2.3 NetBeans 8.2 ... 16
3 Framework Core ... 18
3.1 Características ... 18
3.2 Composición ... 20
3.2.1 Núcleo ... 20
3.2.2 Clases primitivas ... 22
3.2.3 Gestión interna ... 24
3.2.4 Vista ... 25
3.2.5 Bases de datos ... 26
3.2.6 Pruebas ... 28
3.3 Alcance del proyecto ... 29
3.4 Metodología ... 29
4 Entrega I ... 31
4.1 Sprint I ... 31
4.2 Sprint II ... 32
5 Entrega II ... 35
5.1 Sprint III ... 35
5.2 Sprint IV ... 36
5.3 Sprint V ... 38

5.4 Sprint VI 39
5.5 Sprint VII 41
6 Próximas entregas 43
7 Conclusiones 44
8 Bibliografía 46

1 Introducción

Este estudio corresponde al Trabajo Final de Grado en Ingeniería Informática de la Universidad Politécnica de Valencia. El trabajo surge a partir de mi experiencia en la empresa Ferrocarrils de la Generalitat Valenciana (FGV) como alumno en prácticas durante 450h. El *framework* aquí analizado ha sido originalmente creado y diseñado por Don Enrique Ruiz, técnico en el departamento de sistemas y mi cotutor. En algunos apartados, Enrique Ruiz ha tenido peso en la implementación ya que yo poseo menos habilidades en la programación y me era complicado o casi imposible resolver ciertos problemas.

1.1 Motivación

Desde los años 80, la popularización de la programación orientada a objetos no ha dejado de aumentar. Los objetos son encapsulaciones de código con identidad propia, teóricamente independientes, pero que necesitan de otros objetos en cooperación para generar funcionalidades más complicadas. La naturaleza del objeto ayuda a establecer sus límites, ya que debe guardar también concordancia semántica. Esto es, si no tiene sentido, no debería estar ahí. Por ejemplo, si tenemos dos clases Niño y Pelota, tendría sentido Niño.botar() pero no Niño.rebotar(). Esto último debería ser trabajo de Pelota. Gracias a esta característica semántica, los objetos nos ayudan a guardan una estructura y una organización a lo largo de un proyecto.

En cuanto a la decisión de crear un *framework* propio o trabajar con uno ya existente en el mercado, existen argumentos válidos en los dos casos. Las posibles ventajas de usar un *framework* del mercado son:

- **Solución correcta y validada**. Las herramientas y los procesos existentes en el *framework* funcionan correctamente, con lo cual no sería objeto de estudio ni análisis, ahorrando tiempo.

- **Aprendizaje**. Si bien no dedicaríamos tiempo a desarrollar el *framework*, deberíamos hacerlo a aprender cómo funciona y, que es más importante aún, cómo usarlo.
- **Documentación y consulta**. La documentación disponible, si bien el tamaño depende de qué *framework* se elija, casi siempre estará completa y bien explicada. El nivel de información de consulta en la web también dependerá del *framework* elegido, pero probablemente habrá mucha disponible en la red.

Con respecto a la decisión de desarrollar y usar nuestro propio *framework*, se pueden destacar las siguientes ventajas:

- **Aprendizaje avanzado del lenguaje y experimentación**. Si bien es necesario bastante tiempo para cumplir este requisito, lo vemos más como una ventaja ya que alcanzaremos altos grado de habilidad y soltura con el lenguaje.
- **Ajustado a nuestro entorno**. A diferencia de otros *frameworks*, el nuestro estaría aplicado y personalizado a nuestro entorno de trabajo, lo cual ayuda a establecer un estándar común propio en todo el sistema.
- **Grupos de trabajo muy reducidos**. En nuestro caso, los grupos de trabajo serán normalmente de 1 persona, 2 como mucho. Esto es una gran ventaja ya que el tiempo de aprendizaje total del framework se reduce bastante.
- **Código propio**. La elección de desarrollar nuestro propio *framework* nos permite ser los propietarios y decidir sobre él.

1.2 Objetivos

El principal objetivo de este trabajo es el estudio de las interacciones entre los objetos en un *framework* y la disposición de herramientas para el programador que le ayuden y orienten en el proceso de codificación de aplicaciones empresariales. Este objetivo puede desglosarse en varios objetivos secundarios:

a. Estudiar los fundamentos teóricos de la programación orientada a objetos en un entorno empresarial e identificar los problemas inherentes a estas arquitecturas.
b. Profundizar en la arquitectura y la filosofía aplicadas en el *framework* base que me ha sido entregado por la empresa.
c. Extender nuevas funcionalidades para la simplificación del proceso de creación y mantenimiento de aplicaciones.
d. Desarrollar mediante el *framework* final una aplicación empresarial real.

1.3 Estructura del documento

Vamos a introducir brevemente en lo que consistirá cada punto del trabajo a partir de este. Esto servirá para hacernos un mapa mental del trabajo y si es necesaria la consulta sobre algo, saber a qué apartado ir.

El capítulo 2, Estado del arte, es una introducción al ámbito del trabajo y a sus principios teóricos. El punto 2.1 es una breve explicación de todos los conceptos tratados a lo largo del trabajo: *framework* y patrones de diseño. Correspondiendo al punto 2.2, hacemos una breve comparación de los dos posibles lenguajes sensibles de ser usados: PHP y JavaScript. Justificamos nuestra elección de usar PHP. Más adelante, en el punto 2.3, describimos brevemente algunas ventajas y características del IDE usado por nosotros, antes, durante y después del trabajo realizado, NetBeans.

Ya en el capítulo 3, concretamente en el punto 3.1, detallamos las características que definen el *framework* en estudio. Posteriormente, en el 3.2, profundizamos en la explicación de la composición del framework, las partes que lo componen y como se distribuyen y se intercomunican. En este punto existen sub-apartados donde se separan las clases por funcionalidades y se explican cada una de ellas. En el punto 3.3, especificamos el alcance del trabajo, y finalmente, en el punto 3.4, explicamos la metodología seguida en el desarrollo del trabajo elegido

Ya en el capítulo 4 y en el capítulo 5, se describen detalladamente las entregas por la cual se define la metodología seguida. Existen 2 entregas, subdivididas en diferentes *sprints*. Para cada *sprint* se detalla: una tabla de UT’s con información complementaria;

el trabajo dedicado al framework y el dedicado a la aplicación de forma narrativa. Esto último está adjunto en el anexo.

En el capítulo 6 hablaremos de las posibles aplicaciones y funcionalidades del *framework* sensibles a desarrollar en el futuro.

En el capítulo 7 detallaremos las conclusiones a las cuales hemos llegado desde el trabajo realizado.

Finalmente, en el capítulo 8, detallamos la bibliografía.

2 Estado del arte

2.1 Consideraciones teóricas

Antes del desarrollo del objeto de nuestro estudio, vamos a hacer una breve introducción a los conceptos y términos usados a lo largo de todo el trabajo.

2.1.1 Framework

El término *framework* es una palabra inglesa, cuya traducción literal es “marco de trabajo”. Según Dirk Riehle, entendemos *framework* como un modelo de clases que define la colaboración de los objetos (del *framework*) [**12**]. Adicionalmente, y de forma más abstracta, añadimos que un *framework* define el esqueleto de la aplicación e introduce un conjunto de procesos y buenas prácticas a seguir en las aplicaciones *software*. También aporta una normalización de los datos del sistema para que éstos puedan ser reconocibles y legibles en cualquier punto de la aplicación, entre otras cosas. El profesor Riehle explica que los *frameworks* orientados a objetos prometen una mayor productividad y disminuyen el tiempo de diseño e implementación a través de la reutilización de diseño y de código [**12**] Es por esto por lo que cada vez más empresas opten por usar *frameworks* que mejoren sus estructuras y procesos software. El uso de los *frameworks* orientados a objetos está aumentando día a día, ya que las estructuras y soluciones software van aumentando de tamaño y complejidad, y éstos ayudan a guardar unas formas comunes a lo largo del proyecto, mejorando la comunicación entre los integrantes del proyecto. Las posibles ventajas de usar uno se describen a continuación [**10**]:

- Reutilización de código sensible de ser usado en más de un punto del sistema, reduciendo el tiempo de codificación del programador.
- Abstracción, lo que permite más inmersión en implementaciones concretas y puntuales.

- Otorga herramientas para la implementación de trabajos más avanzados que, sin ayuda del *framework,* posiblemente el desarrollador no sería capaz de realizar, o realizar estos con mucha menos eficiencia.
- Velocidad en la codificación. La curva de aprendizaje de un *framework* en sus inicios es lenta, pero con la experiencia necesaria permite al programador alcanzar cotas de producción mucho más altas que si no se hiciera uso de un *framework.*

La palabra *framework* engloba distintos niveles de abstracción, desde una idea estructural (como lo puede ser por ejemplo el patrón *Model – View – Controller*, que declara separar el código por tres capas bien diferenciadas) hasta un conjunto grande de herramientas y procesos específicos (por ejemplo, Microsoft .NET). La mayoría de los *frameworks* se basan en los patrones de diseño, los cuales son en gran parte responsables de las mejoras cualitativas que ofrecen los *frameworks.*

2.1.2 Patrones de diseño

En innumerables momentos, como programadores, nos hemos encontrado en la tesitura de tener que aportar una solución a un problema concreto. No importa cuán específico y puntual sea el problema por resolver, es muy probable que alguien se haya enfrentado previamente al mismo problema y haya creado una solución válida. En este punto, no encontramos razones que eviten que reutilicemos esta solución, ya que ha sido demostrada su efectividad y eficiencia.

El concepto de "patrón de diseño" no tuvo su origen en el ámbito de la programación; fue acuñado, en cambio, por el arquitecto Christopher Alexander en 1979, en su libro *The Timeless Way of Building*. En él, Alexander proponía un conjunto de patrones para la construcción de edificios que otorgarían a éstos mayor calidad [**1**].

Dos informáticos, Kent Beck y Ward Cunningham, encontraron la similitud entre el trabajo de Alexander y el paradigma orientado a objetos (OO) De esta forma, en 1987, publicaron el artículo "Using Pattern Languages for OO Programs" donde se desarrollaba la idea aún primitiva que hoy se conoce como "patrón de diseño" en la programación

software [**3**]. No será hasta 1994 cuando se publica un libro de referencia hasta hoy día, escrito por los conocidos como la *Gang of Four* (GOF), cuyo título reza *Design Patterns* (Gamma, Helm, Jonhson y Vlissides, 1994). En él, se introducen y describen un conjunto de 23 patrones de diseño, que siguen siendo vigentes en la actualidad y su uso está ampliamente extendido [**13**].

2.1.2.1 MVC (Model - View - Controller)

El patrón *Model - View - Controller* (Modelo - Vista - Controlador). Este concepto fue acuñado por primera vez por Trygve Reenskaug en Smalltalk 76, un lenguaje antiguo orientado a objetos. No obstante, no es hasta 1988 cuando aparece como concepto teórico en "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80", de Glenn Krasner y Stephen T. Pope. MVC es un patrón de diseño basado en la estructuración y en la reutilización de código, donde los componentes de un sistema son separados por la función que realizan [**8**].

Esta clasificación se compone, como su nombre indica, de tres partes bien diferenciados [**2**]:

- **Modelo**. Representa y especifica la lógica de negocio y el tratamiento de datos, ya sea inserción, extracción o modificación, entre otros. Este se encarga de suministrar, ya sea al controlador o a la vista, los datos reclamados por el controlador.
- **Vista**: Describe el diseño y la apariencia de la interfaz que va a mostrar el sistema. Muestra y formatea los datos procedentes del modelo y retroalimenta al usuario cualquier tipo de información.
- **Controlador**: se encarga de recibir las órdenes del usuario y de ejecutarlas. Organiza el flujo de trabajo del sistema.

2.1.2.2 Inyección de dependencias

Como hemos visto en el punto 2.1.1, la funcionalidad de un *framework* reside en la cooperación entre los objetos. Esto es, los objetos se comunican entre sí. Para llevar a cabo esto, las clases, que definen el comportamiento de los objetos, deben contener referencias de las otras clases con las que se comunican. A esto es lo que llamamos dependencia, cuando un objeto depende de otro para alcanzar sus objetivos. Estas referencias habitan en las clases y pueden ser creadas por ellas. Tomemos dos clases A y B, donde A necesita de B para acarrear su funcionalidad:

```
public A () {
B b = new B*();
…
}
```

Denota la posibilidad de que B implemente B

Esta implementación implica ciertas restricciones y problemas:

- “A” debe conocer todos los detalles del constructor de la clase “B”, esto es, parámetros y clase que implementa “B”, si “B” se tratara de una interfaz.
- Dificulta a “b” contener una referencia del mismo objeto de la clase “A”.
- Complica la mantenibilidad ya que, cualquier cambio de “B” se debería reflejar también en la implementación de “A”.

La inyección de dependencias es un patrón de diseño introducido por Martin Fowler, ingeniero de software británico, conocido también por otros patrones como inversión de control o y *Model - View View-Model* (MVVM), entre otros [**6**]. Este patrón declara que los objetos que necesite una clase, las dependencias, deben de ser suministrados. Esto se lleva a cabo mediante la implementación de un setter de la referencia que inyecte la dependencia de forma externa [**6**]. Volviendo al ejemplo de antes:

```
public A{
  B b;
  public setB(B b){
        this.b = b;
}
}
```

Esto independiza a A de la implementación de B, siendo la instanciación de B llevada a cabo en otro lugar del sistema. Pero ¿quién es el encargado de la tarea? En el libro antes mencionado, *Design Patterns*, los llamados "Patrones arquitectónicos" daban una solución a esta pregunta. Estos proponían definir factorías, que se encargaban de instanciar los objetos [**7**].

2.1.2.3 Service Locator

El patrón Service Locator, introducido por Martin Fowler, daba una solución a la pregunta antes formulada: ¿en qué lugar instancio las clases que van a ser suministradas? Este patrón es comúnmente implementado mediante los llamados Inversion Of Control Containers (IOC Containers). La traducción en español sería algo como "contenedores que invierten el control". Se trata de una entidad que es la encargada de suministrar las dependencias de cada clase a lo largo del sistema software. La información de las dependencias de cada clase es normalmente especificada en un archivo de configuración [**6**].

2.2 PHP Vs JavaScript

Vamos a realizar una pequeña comparación entre los dos lenguajes más extendidos de desarrollo web, con lo cual justificaremos nuestro uso de PHP. La elección final del

lenguaje se basa en comodidades o experiencias previas con él, ya que cualquiera de los dos nos permitiría alcanzar las funcionalidades más tarde detalladas.

PHP es un lenguaje de script, localizado en el servidor, de propósito general para la web. Fue desarrollado por Rasmus Lerdorf en 1994. PHP es un acrónimo de "PHP: Hypertext Processor". Es un lenguaje no tipado, lo cual significa que el procesador no verifica los tipos de las variables. El código PHP se encapsula en código HTML mediante etiquetas que definen el comienzo y el final del script [**13**]. Fue, indiscutiblemente, el lenguaje más usado para desarrollo web hasta la aparición de NodeJS.

JavaScript (JS) es un lenguaje de script también no tipado, localizado y ejecutado por el navegador. Fue creado por Brendan Eich en 1995. Debe su nombre a su similitud con el lenguaje Java. Su ejecución es asíncrona, esto es, no bloqueante, o, en otras palabras, una acción lanzada por el intérprete que no ha terminado no impide a la siguiente acción ser lanzada [**5**].

Durante muchos años, la tupla <PHP servidor, JavaScript cliente> ha sido la principal composición de los sistemas software en la web. Pero en 2009, con el lanzamiento de Nades, código JavaScript del lado del servidor, ha disminuido el uso de la tupla antes mencionada. En esta comparación, cuando hablemos de JavaScript nos referiremos a su arquitectura back-end, NodeJS, ya que queremos compararlos equitativamente, y, por ende, no tendría sentido comparar dos lenguajes destinados a realizar trabajos diferentes. A continuación, nombraremos algunas de las diferencias más notorias entre estos dos lenguajes:

- **Simpleza.** PHP se presenta como un lenguaje de programación mucho más intuitivo y fácil de usar cuando no se tiene mucha experiencia, mientras que JavaScript, se muestra como un lenguaje mucho más complicado y difícil de leer,
- **Documentación y consulta.** Debido a la carrera de PHP es bastante más larga, nos encontramos en las páginas de consulta de desarrolladores (p. ej. StackOverflow)

mucha más información para PHP que para JavaScript. En términos de documentación, ambos lenguajes acompañan documentación extensa y completa.

- **Sintaxis.** Parafraseando el primer punto, el principio de la curva de aprendizaje de PHP es sencillo, pero en determinados puntos de madurez del desarrollador JavaScript empieza a ser el lenguaje más cómodo con el que trabajar. Además, la sintaxis de JS es similar a la de C y Java, por lo tanto, programadores habituales de estos dos últimos lenguajes, entre otros, tendrían mucha más facilidad en el aprendizaje básico de JS que de PHP.
- **Asincronía.** Como antes hemos mencionado, JS es un lenguaje no bloqueante, es decir, una línea de código que no ha terminado no evita a la siguiente comenzar su ejecución. Esto quizás sea confuso si el desarrollador no está acostumbrado a este tipo de programación.
- **Métodos mágicos.** PHP contiene una gestión de un conjunto de funciones que pueden ser llamadas implícitamente.
- **Código libre.** Existen compiladores OpenSource para ambos lenguajes.

Habiendo repasado las principales características de los dos lenguajes de programación web, decidimos escoger **PHP**, ya que hemos tenido experiencias previas con este lenguaje y tenemos bastante soltura y conocimientos sobre este.

2.3 NetBeans 8.2

Respecto a la elección de qué *IDE* usar, *Integrated Development Environment* (en español, Entorno de Desarrollo Integrado) es mucho más flexible. Depende casi en su totalidad del desarrollador y a qué IDE está acostumbrado a usar. Nosotros hemos usado NetBeans 8.2 ya que tenemos bastante experiencia en él. A continuación, nombraremos algunas características de este conocido IDE:

- **Interfaz configurable.** Los menús y barras de herramienta son flexibles y puedes cambiarlo al gusto del desarrollador. Puedes elegir el color del IDE,

posibilitándonos usar el fondo negro para que sea menos agresivo para los ojos, lo que nos gusta mucho.

- **Ligereza.** Comparado con otros *IDE's*, NetBeans se descubre como uno bastante más rápido y liviano. Esto se debe a que no ofrece tanta funcionalidad como otros como Visual Studio o Eclipse. Pero a nivel personal, contiene todo lo que necesitamos de un IDE.
- **Código abierto.** Nos encanta *OpenSource*, ¿qué más se puede decir?

Encontramos estas razones decisivas para elegir **NetBeans 8.2** como IDE para trabajar en nuestro proyecto y en futuros trabajos.

3 Framework Core

3.1 Características

El *framework* Core contiene una lista de características que definen de forma global cómo funciona la herramienta y marcan un estilo de trabajo con él. Estas características han sido base fundamental desde los orígenes del *framework* hasta la actualidad. Se enumeran a continuación:

- **Inyector de dependencias**. El eje principal del *framework*. Existe una gestión concreta, mediante el patrón Service Locator (final del apartado 2.1.2), que se encarga de instanciar las clases del sistema y relacionarlos entre ellos.
- **Accesores transparentes**. Mediante el uso de los métodos mágicos *get* y *set*, conseguimos un acceso a las variables de los objetos limpio y nos evita usar los accesores tradicionales, lo que nos ahorra mucho código repetitivo.
- **Arquitectura MVC**. La estructura del *framework* responde a una arquitectura MVC, antes explicada.

Como hemos mencionado en el punto 2.1, un *framework* no es otra cosa que las relaciones que existen entre unos unas entidades determinadas. Estos no son triviales, sino que se complementan y su funcionalidad reside en la cooperación entre estos. Como vemos en la ilustración 1, la estructura de clases que define el *framework,* las clases no contienen ni una sola relación de asociación. Esto reduce considerablemente el acople entre las clases, lo que permite mejor mantenibilidad y reusabilidad.

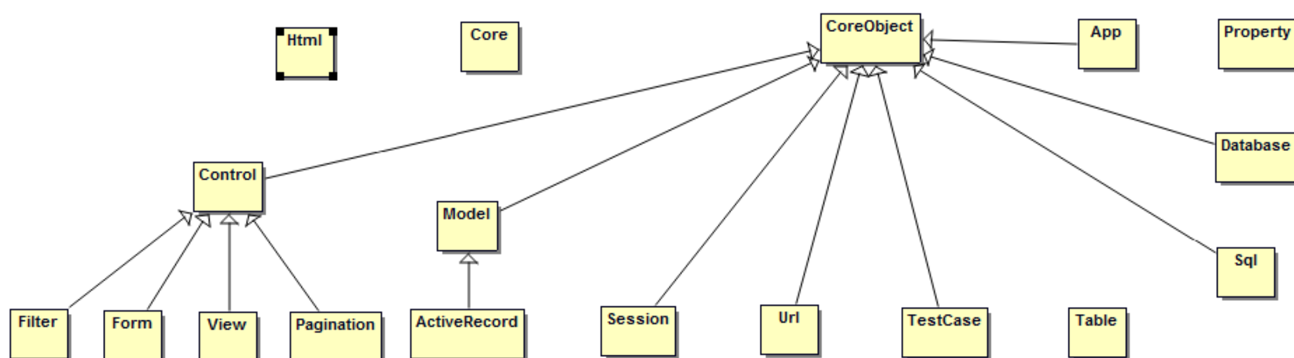

Ilustración 1. Diagrama UML de las clases que conforman el framework Core.

La funcionalidad proviene de la interacción de los objetos con las dependencias que sus clases definen. La clase Core, que explicaremos más adelante, es la encargada de llamar a los constructores y gestionar las **dependencias**. Más adelante, las clases del *framework* serán explicadas de forma resumida en el siguiente apartado.

Desde el comienzo de la implementación del *framework* Core, se han seguido unas pautas comunes que ayudan al desarrollo del mismo y la organización con el trabajo. Definimos estas pautas como la "filosofía del *framework",* y la desglosamos como sigue:

- El principio DRY (*Don't Repeat Yourself*), "no te repitas", se trata de evitar escribir código repetitivo. En otras palabras, en la práctica, si dos líneas son iguales, deberíamos de crear una función para encapsular el código sensible de ser repetido. Esta idea está bastante extendida en el entorno de ingeniería del software, y aunque suene obvio, queríamos incidir en la aplicación de esta filosofía. Es una característica inherente a los *frameworks*.
- "Escribe lo que necesitas". Esta frase se refiere a que, obviaremos siempre una solución propuesta a un problema que no se ha formulado. Es decir, una funcionalidad del *framework* será codificada cuando necesitemos aplicarla en las aplicaciones que están desarrollando.
- De estos dos puntos, se concluye otra idea: si una aplicación desarrollada mediante el *framework* necesita de una funcionalidad no implementada en la herramienta, y existe la posibilidad de generalizarla para su futuro uso, esta funcionalidad se diseña y se codifica en ese momento. Esto ayuda al desarrollo

tanto del *framework* como de la aplicación usando el tiempo de una forma eficiente.

3.2 Composición

Tras comentar las características del *framework* Core, nos disponemos a desglosar y descomponer las partes que conforman la herramienta para aclarar sus funcionalidades y la manera de usarlas.

3.2.1 Núcleo

Vamos a explicar y detallar el funcionamiento de las clases principales que representan la esencia del *framework*.

3.2.1.1 Core

La clase Core representa el eje central y encapsula la esencia del *framework*. Ésta se encarga de la creación y el suministro de los objetos solicitados como dependencias de cada clase. Es una versión de la implementación del patrón *Service Locator,* el *IOC Container.* Es la puerta de entrada para acceder a la clase App, encargada de lanzar el sistema y ejecutar el código, la cual desarrollaremos posteriormente.

Existen 2 tipos de relaciones entre objetos: 1) Un objeto crea otro, donde el objeto creador representa una factoría. 2) Un objeto está relacionada con otro.

Cuando la clase Core actúa como una factoría, estos son los pasos que realiza para alcanzar los objetivos:

1. Cada clase define los objetos producto de la factoría, donde se especifica el nombre de la variable que contiene el objeto dependencia y la ruta relativa de la clase.

2. Core recoge la información de las factorías y las guarda.
3. Cada vez que la factoría se ejecuta, éste llama a su constructor y devuelve el objeto dependiente para su tratamiento.

En cambio, cuando la instancia de la dependencia es única en todo el sistema, o lo que es en otras palabras, que implementan el patrón *Singleton*, los pasos a seguir son:

1. Previamente, algunas dependencias necesitan configuración adicional, que será especificada en el archivo de configuración, y referenciadas como variables globales.
2. Las clases definen sus dependencias, de forma similar al anterior caso.
3. La clase Core instancia las clases globales definidas por el archivo de configuración y las suministra a las clases que las haya definido como dependencia.

3.2.1.2 App

Si la clase Core representa el *framework*, la clase App representa la aplicación que es soportada por el framework. Es accesible de forma estática desde la clase Core. Esta clase se encarga de las siguientes funcionalidades:

- Es la responsable del inicio de la aplicación.
- Se encarga de la ejecución de cada petición por parte del controlador y del tratamiento de los parámetros pasados.
- Lee la url activa en el navegador y la suministra a la clase Url para su posterior manipulación.
- Ofrece herramientas para la creación de links para navegar a lo largo de la aplicación.
- También es la encargada de diferenciar los módulos, agrupaciones de clases donde puedes definir prerrequisitos o eventos para un tratamiento específico de la funcionalidad.

3.2.1.3 Archivo de configuración

En el archivo de configuración, *config.php,* se definen mayoritariamente parámetros de los objetos globales y dependencias. Se especifican los parámetros de construcción de las clases que van a suplir dependencias a lo largo del sistema. Estas dependencias es casi exclusivamente configuración de las conexiones a bases de datos.

3.2.1.4 Core_inc

Este es un archivo que inicializa los elementos y las funciones básicas de Core. Introduce las variables globales de las partes más importantes del *framework* y también algunos parámetros y métodos de asistencia y de configuración ajena al ámbito del programador.

3.2.2 Clases primitivas

En el *framework* existen dos clases padre, las cuales definen la estructura básica de los dos tipos de elementos primitivos: CoreObject, que representa cualquier objeto identitario único del sistema, y Property, que define las propiedades que pueden contener los objetos.

3.2.2.1 CoreObject

CoreObject define la estructura mínima de cada clase del sistema. Todas las clases no auxiliares (sin métodos estáticos de asistencia) heredan de esta clase. Éstas son las funcionalidades otorgadas por esta clase:

- Como hemos explicado anteriormente (filosofía del framework), CoreObject define una serie de accesores transparentes al usuario, comúnmente llamados métodos mágicos. Éstos permiten ser llamados implícitamente. Esto ahorra mucho código, ya que gracias a estos métodos no es necesario declarar los *getters* y *setters de cada clase*, y, por lo tanto, no es necesario llamar al método cada vez, tan solo nombrar la variable que se quiere acceder/modificar.
- Gestión de eventos. Es posible en cada elemento del *framework* programar un *lis tener*, ya sea genérico o específico, que se ejecutará cuando y como el usuario desee, proveyendo de una gran y flexible funcionalidad a la aplicación.

Si observamos los dos diagramas mostrados al principio de este capítulo con detenimiento, observamos que solo en las clases que extiendan CoreObject pueden existir dependencias. Esto se debe a que, esta clase implementa un método *dependencies*, que será la herramienta de las clases para especificar sus dependencias en todo el *framework*.

3.2.2.2 Property

Property representa una propiedad de un objeto. Éste contiene una serie de atributos que se define los métodos mágicos para el acceso/modificación de los atributos de Property, de forma que, análogamente con CoreObject, los *getters* y *setters* sean llamados implícitamente. Un objeto Property puede tener a su vez una serie de atributos. Nótese que, los accesores y modificadores de vectores no pueden ser llamados mágicamente, y, por lo tanto, tendrán que ser ejecutados explícitamente.

3.2.2.3 Model

Model es la clase primitiva de cualquier clase que represente un modelo en el sistema. Cada modelo contiene una lista de propiedades que definen o complementan el modelo. Estas propiedades a su vez contienen atributos. La clase Model tiene métodos

para el acceso y la modificación de cualquier atributo antes mencionado. También implementa una función para leer los comentarios escritos en el modelo (cualquier clase que extienda Model)

3.2.3 Gestión interna

A continuación se explican las clases encargadas de controlar el flujo del programa y los parámetros internos.

3.2.3.1 Control

Control representa la clase padre de cualquier clase que represente un controlador. Ésta define las dependencias genéricas que un controlador básico tiene. Contiene métodos que gestionan la comunicación de los datos a través del sistema y la puerta de acceso a la clase Url. El método más importante de esta clase es el llamado *call*, usado frecuentemente por el programador para redirigir el flujo del programa a otro trozo de código diferente al actual.

3.2.3.2 Url

Clase encargada del tratamiento de sus homónimas, las urls. Se introducen en la barra de navegación del navegador para indicar a que parte de la aplicación se quiere acceder. Se pueden introducir manualmente, o ser introducida por la aplicación tras haber realizado alguna acción en la web. Estas rutas relativas del sistema son traducidas a/desde la tripla {controlador, acción, parámetros *get*}, formato comprensible por el *framework,* que representa, *¿quién lo ejecuta? ¿que ejecuta?* y *¿de qué forma?* Cuando el antes nombrado método *call* de la clase Control es lanzado, Url trata estas tres variables y construye la url que representa, a la cual después será redirigido el flujo de la aplicación.

3.2.3.3 Session

Session es la encargada de guardar parámetros de configuración del estado de la aplicación. La clase define métodos de acceso para los valores y un sistema para diferenciar unos parámetros de sesión de otros, lo que nos permite separar los datos de configuración por IP y usuario.

Cabe mencionar, el antes nombrado método de Control *call*, usado para dirigir el flujo del programa a otro punto, recibe cuatro parámetros: controlador, acción, parámetros *get* y parámetros post. Hemos mencionado también que cuando la función es llamada, este pasa tres parámetros de los cuatro a Url para ser traducido a la ruta *url* correspondiente. Esto es debido a que los parámetros *get* se reflejan directamente en la *url*, mientras que los parámetros post son transmitidos mediante la variable global $_POST de PHP.

3.2.4 Vista

En este apartado, vamos a explicar las clases encargadas de gestionar la parte visual, es decir, la encargada de tratar los archivos HTML.

3.2.4.1 View

La clase View es la encargada de estructurar la parte visual del *framework*. En ella se encuentran variables y métodos para la gestión de la vista. La forma genérica de emplearla es mediante una plantilla HTML que será común a todas las pantallas de la aplicación, donde la parte intercambiable de la plantilla serán las distintas pantallas gráficas del sistema. El objeto View va añadiendo la información a la plantilla que el programador desea mediante sus métodos. La función más relevante de esta clase es *show,* la cual permite mostrar una pantalla cargada anteriormente.

3.2.4.2 Form

La clase Form permite construir un formulario HTML de forma cómoda. La gestión de los diferentes campos del formulario es definida por un objeto Model contenido en el formulario, donde el programador define una serie de atributos. Cada atributo representa un valor del formulario que luego será enviado al código PHP para su uso. Estos atributos también pueden contener propiedades que definan los campos de formulario que representan.

3.2.5 Bases de datos

Aquí introduciremos las clases responsables de la comunicación con las bases de datos y de gestionar la entrada o salida de los datos.

3.2.5.1 Database

Database es una clase que representa una conexión a base de datos. Éste es construido con los parámetros normales de enlace con base de datos. Estos parámetros son especificados en el archivo de configuración de la aplicación y se recupera el objeto llamando a la variable global donde se han definido los parámetros. También contiene una serie de métodos para la ejecución u obtención de información relacionada con una conexión a base de datos. La función relevante de esta clase es la llamada *sql*, que permite enviar comandos SQL al driver para el tratamiento de los datos contenidos.

3.2.5.2 Sql

Sql es una clase de asistencia para la construcción de instrucciones SQL. Su pilar principal es la gestión eficaz de los parámetros de las instrucciones SQL. También define

varias formas de extracción de los datos devueltos por la base de datos tras realizar una consulta y un sistema de reconocimiento de tipos para la inserción o recuperación de los datos formateados correctamente.

3.2.5.3 Drivers

Los drivers representan cada tipo de base de datos que acepta el *framework*, Oracle o MySQL. De esta forma, mediante una interfaz que especifique la mínima funcionalidad que debe de ofrecer un driver, la conexión se abstrae de su implementación. También define funciones auxiliares para las instrucciones *SQL* más comúnmente utilizadas.

3.2.5.4 Active Record

La clase ActiveRecord hace referencia a un modelo que representa una tabla en la base de datos, de forma que, mediante comentarios en la clase modelo, se relacionan las columnas de las tablas con las variables de la clase. De este modo, las transacciones con la base de datos del modelo referenciado son automáticas. La clase contiene métodos para el reconocimiento de los comentarios en la clase y las funciones CRUD (*create, read, update, delete*)

3.2.5.5 Table

Table es la clase puente entre la clase Database y la clase ActiveRecord, encargada de ejecutar las instrucciones enviadas a ActiveRecord

3.2.6 Pruebas

Adicionalmente, existe un par de clases para obtener información acerca de la eficacia (TestCase) o de la eficiencia (Benchmark).

3.2.6.1 TestCase

TestCase es una clase para ejecutar pruebas en cualquier punto del framework o de la aplicación. Su funcionamiento es muy similar a otros frameworks de testeo del mercado. Contiene métodos para la validación de las pruebas y para la retroalimentación al usuario de los errores o validaciones no exitosas que ha habido.

3.2.6.2 Benchmark

Benchmark es una simple clase encargada de informar al usuario de la consumición de memoria por parte de la aplicación o del tiempo de ejecución. Para su funcionamiento, es recomendable llamarlo desde el archivo php inicial, es decir, index.php.

Este ha sido la explicación de la composición del *framework* en el momento en que llegué a la empresa. Los dos siguientes capítulos hacen referencia a las dos entregas que realicé en el marco de mis prácticas, siendo descompuesto en los diferentes *sprints* que conforman la entrega. La explicación del trabajo dirigido por los *sprints* tendrá una forma narrativa, explicando paso a paso las decisiones para cumplimentar los requisitos.

3.3 Alcance del proyecto

Una vez hemos explicado y detallado el funcionamiento actual del *framework*, tenemos que especificar el alcance del trabajo a realizar. Este se define como la implementación desde cero de una aplicación empresarial orientada a mejorar los procesos de la empresa y los consiguientes cambios y mejoras en el *framework*. En el desglose del alcance del trabajo a ejercer se diferencian:

- Desarrollo integral de la aplicación empresarial que se ajuste a las necesidades y los requisitos exigidos por la empresa.
- Este desarrollo debe ser realizado mediante la herramienta Core antes expuesta.
- A la vez, este progreso de la aplicación debe conllevar una mejora de las partes que conforman la herramienta.
- El trabajo aquí expuesto no debe durar más de siete semanas.

3.4 Metodología

Para el desarrollo de la aplicación hemos sido dirigidos por una metodología SCRUM, donde el número de integrantes del grupo es de 1. El tiempo asignado a cada sprint es de 1 semana, con reuniones intermedias para la resolución de dudas que han ido surgiendo y con reuniones con cada inicio/fin de cada sprint. El proceso de definición de cada sprint esta detallado a continuación:

1. Se define una lista de requisitos empresariales que tendrán que ser satisfechos.
2. Se especifica el trabajo y desarrollo necesario en el *framework* para la implementación de las soluciones a los problemas antes nombrados.
3. Para cada punto, se calcula un tiempo estimado, y si debe contener interfaz gráfica, se dibujará un boceto inicial (MOCK UP)

El número de entregas finales asciende a 2. Cada entrega corresponde a la entrega de una aplicación empresarial demandada con su correspondiente estado del *framework*.

Cada entrega está compuesta por una o más iteraciones o *sprints*. Cada *sprint* está compuesto por aproximadamente 22 horas, pudiendo ser ampliable mínimamente según las necesidades. En cada tabla de la información de cada *sprint* se detalla su fecha de comienzo, las unidades de trabajo, las pruebas de aceptación que lo componen, las horas estimadas y reales de cada una y el total de horas trabajadas en el *sprint.* Nótese que, en cada descripción de hora real, están incluidas el tiempo dedicado a las pruebas.

Las características que hemos descrito anteriormente provocan las siguientes consecuencias:

- No existe un reparto del trabajo definido cada sprint, ya que el número de integrantes del grupo es de 1. Éste, irá eligiendo los puntos a desarrollar y los irá tachando conforme los vaya acabando. También anotará el tiempo que le ha llevado finalizar el trabajo.
- Al contrario de como dictan las metodologías ágiles, en nuestro caso el programador y el tester son la misma persona. El tutor del trabajo, Don Enrique Ruiz, actuará puntualmente de tester, pero la mayor responsabilidad como tester recaerá sobre el integrante del grupo SCRUM.
- Inexistencia del *backlog*, ya que, como antes hemos comentado, los requisitos a cumplimentar recaían directamente en las necesidades del momento del inicio del *sprint*.

4 Entrega I

Vamos a proceder a explicar las iteraciones necesarias que hemos necesitado para llegar a la primera entrega, que se corresponde con la aplicación SIAC. Para esta entrega han sido necesarios 2 *sprints*.

4.1 *Sprint* I

Al ser el primer *sprint*, aunque el trabajo a realizar sea de poca extensión, hay que conllevarlo con la introducción y aprendizaje del uso del *framework*. El trabajo pedido en este *sprint* se resume en:

- Clase Tarjeta para el tratamiento de tarjetas en la base de datos.
- Formulario de *login* para los empleados de la empresa.

Sprint I				12-02-17	
ID	UT	Dominio	PA	Horas estimadas	Horas reales
1	Modelo Tarjeta	A	Construir clase Tarjeta para el tratamiento en base de datos	3	3
			Añadir métodos para tratamiento específico de tarjetas	10	12
2	Vista Login	A	Crear formulario con dos campos para el login	6	6
				Total	21

Tabla I. Tabla con información del sprint I.

La primera clase por crear en esta aplicación es una perteneciente al modelo (recordemos aplicar siempre la arquitectura MVC). Creamos las tres carpetas que definen la arquitectura, y dentro del modelo construimos nuestra primera clase que se llamará Tarjeta. Ésta será la encargada de conectar con la base de datos pertinente y realizar los tratamientos requeridos. Codificamos los métodos de forma que construyan las sentencias Sql que realizan las consultas necesarias, en nuestro caso son procedimientos que tendremos que llamar de la base de datos, pero no profundizaremos en los detalles de los tratamientos de la base de datos. En esto no hubo problemas, ya que hemos estudiado previamente el lenguaje Sql y solo tuvimos que preguntar sobre la sintaxis de los

procedimientos de la empresa. Resta sólo realizar el tratamiento de errores si los supuestos datos son erróneos.

Posteriormente, debemos crear un formulario que sirva de puerta de entrada para los usuarios de la aplicación donde introduzcan sus credenciales que le permita el acceso a la aplicación, siendo el *mockup* el que se muestra. Ya que antes no habíamos creado un formulario a través del *framework*, tardamos relativamente mucho en realizar. Los pasos para llegar a su construcción son como siguen: creamos el objeto y rellenamos el modelo interno del formulario que, como antes hemos explicado, define los diferentes campos del formulario. Tras esto, entre las llamadas al objeto Form *begin* y *end*, que definen el comienzo y el final del formulario, se ejecutan los métodos que dibujan los campos, introduciendo el nombre de cada campo que antes hemos especificado en el modelo. Para el botón de acceso, adicionalmente, hay que detallar la tripla {controlador, acción, parámetros *get*}, es decir, la ruta que sigue la pulsación del botón.

Ilustración 2. Mockup de la UT con ID 2.

Hasta aquí el trabajo realizado para el primer *sprint*, a simple vista poco trabajo, pero como antes hemos comentado, son nuestros primeros pasos en la implementación de una aplicación mediante el *framework* y hemos tenido que dedicar bastante tiempo en el aprendizaje del mismo.

4.2 *Sprint* II

El trabajo demandado para este sprint es el siguiente:

- Controlador para los enrutados de la aplicación.
- Vistas de las diferentes páginas y sus formularios.

- Extracción del servidor LDAP de los credenciales.

Sprint II					19/02/2017
ID	UT	Dominio	PA	Horas estimadas	Horas reales
4	Vista Tarjeta	A	Diseñar pantalla por cada caso de uso	4	5
5	Controlador Tarjeta	A	Añadir métodos para las rutas de las pantallas	3	3
			Tratar los errores de formulario de tarjeta	2	2
6	Controlador Login	A	Crear métodos para controlar el login	4	3
			Tratar los errores de login	3	4
7	Modelo Login	A	Extraer credenciales de servidor LDAP y comparar con introducidos	3	5
				Total	22

Tabla 2. Tabla con información del sprint II.

Habiendo realizado el modelo de nuestra aplicación, la clase Tarjeta, ahora necesitamos completar el funcionamiento con el controlador y las vistas. En primer lugar, ya que conocemos como crear los formularios, creamos un archivo php para cada pantalla que contiene nuestra aplicación, las cuales serán 3, una por cada caso de uso de tratamiento específico de tarjetas. Definimos los formularios en cada archivo, las cuales son muy simples, ya que conocemos perfectamente cómo hacerlas.

Ilustración 3. Mockups de la UT con ID 4

Actualmente, necesitamos el controlador que enrute todas las acciones en nuestra aplicación. Construimos la clase Main en su carpeta controller, que extiende de Control y define sus dependencias. Necesitamos un método para cada vista y para cada acción de la aplicación, ya que el controlador es el encargado de transmitir los deseos de la vista (del usuario) al modelo para que los ejecute. También necesitamos los tratamientos de errores de los datos transmitidos.

En este momento, necesitamos la parte del modelo del *login,* capaz de extraer las credenciales del servidor LDAP, compararlos con los introducidos y permitir o denegar el acceso a la aplicación. Creamos el modelo de *login* y definimos los métodos encargados de ello, que depende de la estructura del servidor. Adicionalmente, codificamos el método encargado de preguntar al objeto Session si se ha realizado el *login*, y si no se ha realizado se dirige el flujo del programa a la pantalla de *login*. Esto permite que no se pueda tener acceso a la aplicación sin las credenciales necesarios.

5 Entrega II

La segunda entrega que debemos preparar trata de una aplicación orientada a gestionar los acuerdos que la empresa tiene con diferentes entidades de la comunidad. Esta parte de la aplicación la llamaremos **GdA.**

5.1 *Sprint* III

Tras la primera entrega de la aplicación junto con el *framework*, debemos seguir con la implementación de la aplicación dados los requisitos que se nos facilitan. Esta semana, debemos realizar:

- Clase Acuerdo para la gestión y tratamiento de acuerdos.
- Hacer que esta nueva aplicación este contenida dentro de la otra.
- Hojas de estilo para la aplicación SIAC.

Sprint III					26-02-17
ID	**UT**	**Dominio**	**UT**	**Horas estimadas**	**Horas reales**
8	Modelo Acuerdo	A	Construir clase que representa un acuerdo en la base de datos	8	9
			Crear métodos para tratamiento específico de tarjetas	6	6
9	Organización	A	Subcontener de GdA en SIAC	1	1
10	Vista Login y Tarjeta	A	Añadir hojas de estilo para SIAC	3	4
				Total	20

Tabla 3. Tabla con información del sprint III.

El objetivo principal de este *sprint* es la creación de la clase Acuerdo, que será similar a la clase Tarjeta de la aplicación SIAC. Ésta extiende ActiveRecord, y define una variable por cada columna contenida en la tabla de acuerdos que define la base de datos. También define unos métodos necesarios para el tratamiento de los acuerdos en la base de datos, acordados previamente.

Por otro lado, fusionar las dos aplicaciones es sencillo. Creamos una subcarpeta en SIAC, la nombramos como GdA, creamos las tres carpetas básicas que definen el patrón MVC, y, ya que de momento solo tenemos el modelo creado, introducimos en la carpeta modelo la clase que hemos implementado anteriormente.

Finalmente, se nos pide que construyamos las hojas CSS para la aplicación anterior, ya que el estilo que previamente hicimos no era apropiado. Mediante unas cuantas directrices, construimos unas hojas de estilo acordes con los gustos y necesidades del personal de empresa.

Curiosamente, aunque el trabajo más extenso a realizar en este *sprint* es la clase Acuerdo, los problemas más significantes los tuvimos al definir las hojas de estilo para la aplicación SIAC, ya que encontramos especialmente difícil estampar las ideas de apariencia en código CSS. Con un poco de ayuda de un compañero conseguimos estilizar la aplicación SIAC como se nos había pedido.

5.2 *Sprint* IV

En esta nueva semana, el trabajo demandado a realizar se desglosa en:

- Formulario para la edición o inserción de acuerdos.
- Añadir *JQuery* a la aplicación para los campos de fechas en el formulario.
- Tabla para mostrar todos los acuerdos.

Sprint IV					05-03-17
ID	**UT**	**Dominio**	**PA**	**Horas estimadas**	**Horas reales**
			Formulario para edición o inserción de acuerdos	6	7
			Tratar el formulario según edición o inserción	3	2
11	Vista Acuerdo	A	Ejecutar cambios introducidos	3	4
			Añadir Jquery para los campos con fecha	2	2
			Diseñar pantalla para mostrar todos los acuerdos contenidos la base de datos	3	3
12	Cambios Form	F	Añadir método para campo de formulario invisible (*hidden*)	1	1
13	Cambios View	F	Añadir función para crear enlaces (*link*)	1	1
				Total	20

Tabla 4. Tabla con información del sprint IV.

El trabajo más notorio de esta semana se trata del formulario para los acuerdos. Este debe definir un campo para cada propiedad de un acuerdo. Como sabemo_s, debemos crear el modelo del formulario primero y definir un atributo para cada campo, estableciendo si es requerido, el nombre de la etiqueta que lo acompañará y el tipo de cada uno. Tras esto, se complica un poco, ya que debemos de tratar el formulario dependiendo si es de edición de uno ya existente o uno nuevo. Ayudándonos de la clase Session, logramos llevar a cabo este requisito. También debemos de cargar todos los títulos existentes en la base de datos, ya que cada acuerdo tiene un título relacionado.

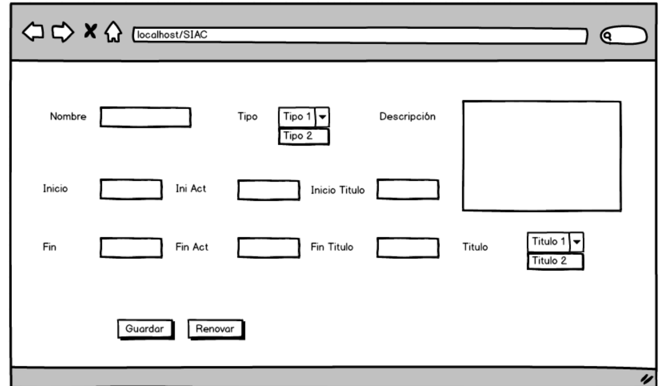

Ilustración 4. Mockups de la UT con ID 11

En este momento, necesitamos ejecutar los cambios realizados en el formulario, por lo tanto, codificamos el formulario de forma que cuando se pulse el botón de OK los cambios sean reflejados en la base de datos. nos damos cuenta de que necesitamos crear un campo “invisible” en el formulario, ya que se debe de guardar el ID del formulario para saber cuál es el acuerdo referenciado. Creamos un nuevo método en la clase Form llamado *hidden* que nos otorga esta funcionalidad.

En este punto, necesitamos el JQuery para poder seleccionar una fecha en los campos *date* del formulario. Con un poco de investigación, escribimos las líneas necesarias en la cabecera del fichero HTML que define la vista para agregar JQuery a la aplicación

Finalmente, necesitamos una tabla que muestre todos los acuerdos existentes en la base de datos. La tabla contiene el nombre y la fecha de los acuerdos. El nombre debe ser un link que nos lleve hasta el formulario de acuerdo antes implementado. Como esta función no ha sido codificada en el *framework*, añadimos la función *link* a la clase View, que nos otorgue esta funcionalidad. También debemos de enviar el ID del acuerdo mediante el objeto Session, para que cuando pulsemos algún acuerdo, la aplicación sepa que acuerdo es y recupere toda la información de la base de datos necesaria para rellenar los campos del formulario.

5.3 *Sprint* V

Terminado el *sprint* IV, nos dirigimos a organizar el trabajo para el *sprint* V, el cual se divide en:

- Buscador de usuarios dado un parámetro.
- Formatear los acuerdos para que se muestren en distintas páginas.
- Exportación o inserción de usuarios desde/a archivos csv.

Sprint V					26-03-17
ID	**UT**	**Dominio**	**PA**	**Horas estimadas**	**Horas reales**
14	Vista Acuerdo	A	Diseñar pantalla para consultar acuerdos por parámetros	2	2
			Crear pantalla para mostrar los resultados	2	2
15	Controlador Acuerdo	A	Tratar el control para los acuerdos consultados	3	3
16	Modelo Acuerdo	A	Añadir métodos para consultar acuerdos por parámetros	3	3
17	Creación Pagination	F	Dibujar barra para la navegación entre las distintas páginas	2	1
			Codificar métodos para la extracción de los datos según la página actual	4	5
18	Cambios Sql	F	Añadir método que devuelva los datos dependiendo de la página actual	1	1
19	Vista Acuerdo	A	Instalar clase Pagination en la pantalla de muestra de todos los acuerdos	1	1
			Instalar clase Pagination en la pantalla de muestra de los acuerdos consultados	1	1
20	Creación OÁmbito	F	Añadir método para la exportación de archivo csv	2	3
				Total	22

Tabla 5. Tabla con información del sprint V.

Necesitamos una pantalla adicional para buscar los usuarios adscritos a uno o más acuerdos que concuerden con los parámetros previamente introducidos. Para ello creamos un archivo php que defina la estructura de la página, como ya habíamos hecho recurridamente antes. Tras la creación del formulario, creamos una función en el controlador que se encargue de dirigir el flujo del programa en este caso, haya habido error o no. Los datos son recogidos y mostrados por otra pantalla diferente. En este momento nos damos cuenta de que la lista devuelta es sumamente larga, y que debíamos de separar los resultados por páginas. Cómo no sólo existe este problema aquí, sino también en la lista que muestra todos los acuerdos en la base de datos, nos preparamos para añadir otra clase al *framework*.

Decidimos nombrarla Pagination. Mediante un objeto Sql, el objeto Pagination se encarga de dibujar la barra de las distintas páginas y de devolver los datos que correspondan dependiendo de la página actual. Ya que se trata de la primera clase del *framework* que codificamos en su totalidad, su implementación se complica debido al nivel de abstracción que requiere. También tenemos que crear una función nueva en la clase Sql, llamada *fetchPartial*, encargada de ejecutar la consulta encapsulada en el objeto y devolver un número determinado de registros partiendo desde un registro determinado. Esto es, dado un número máximo de registros que se deben mostrar en una página (*nMax*) y el número de página actual (*nActual*), podemos llamar la función de esta forma *fetchPartial (nMax * nActual, nMax),* permitiéndonos extraer solo los datos de la página a mostrar. Tras esto, debemos instalar el uso de la clase Pagination en aquellas partes que lo necesiten hasta el momento.

Habiendo terminado el trabajo más extenso del *sprint*, necesitamos insertar o extraer los usuarios adscritos a un acuerdo desde/a un archivo .csv (*Comma-separated values*). Los archivos csv nos permiten tratar con archivos Excel (Office) o Calc (OpenOffice) de forma muy simple: cuando son formateados a archivos con extensión csv, los campos en diferentes columnas son separados por comas y en diferentes filas son separados por retornos de carro. De esta forma es muy fácil leer o escribir desde/en un archivo de Excel o Calc.

Para la extracción, como se puede hacer de forma genérica, decidimos crear otra clase en el *framework* llamada Out, que contiene un método *csvToFile*, encargada de crear un archivo csv y escribir los datos formateados en csv. Para la inserción, en cambio, creamos un método en el modelo de la aplicación GdA capaz de entender el formato y crear las sentencias Sql.

5.4 *Sprint* VI

Para este nuevo *sprint*, sabiendo que el fin de las prácticas en la empresa llegaba, decidimos enfocar estas dos últimas semanas en implementar los requisitos aún

pendientes, realizar las pruebas sobre las aplicaciones y perfilar su apariencia. Este es el trabajo el cual nos proponemos hacer para esta semana:

- Establecer un filtro para las tablas de resultados
- Construir la hoja de estilo del formulario de acuerdo

Sprint VI					02-04-17
ID	UT	Dominio	PA	Horas estimadas	Horas reales
21	Creación Filter	F	Añadir método para dibujar el formulario que recoge la palabra clave	2	3
			Diseñar el tratamiento a la consulta Sql para filtrar los datos dada la palabra clave	4	6
22	Cambios Pagination	F	Añadir la consulta Sql que extrae los datos	1	1
23	Modelo Acuerdo	A	Añadir métodos para consultar acuerdos por parámetros	2	2
24	Vista Acuerdo	A	Construir CSS para formulario de acuerdo	6	6
			Construir CSS para pantallas de casos de uso de tratamientos de tarjetas	2	2
			Construir CSS para pantallas de muestra de acuerdos	1	1
				Total	21

Tabla 6. Tabla con información del sprint VI.

Debido a que la base de datos de la empresa es muy extensa y la lista que se generaba era demasiado larga, buscar un valor en la tabla es un trabajo tedioso e innecesario. Por esta razón, necesitamos la implementación de un tipo de filtro en las tablas que muestran la información demandada. De esta forma, decidimos crear una clase Filter que otorgara esta funcionalidad. Este objeto devuelve en su construcción el formulario donde se debe introducir la palabra clave sobre la que se desea buscar y mediante el método *filter*, se recupere la sentencia Sql que devuelve los datos ya filtrados. Aunque la clase es corta de extensión, su codificación nos ocupó realmente mucho tiempo ya que muchas de las ideas que teníamos para el filtro no funcionaban del todo bien en la práctica o tenían problemas colaterales.

En este momento, cuando intentamos instalar el funcionamiento de Filter, nos damos cuenta qué la clase Pagination también necesita unos cambios ya que no conecta adecuadamente. La solución es introducir también la consulta Sql en Pagination, de esta forma, bajo cualquier circunstancia o flujo de la página, ésta funciona correctamente. Sabemos y conocemos acerca de los problemas de esta implementación, las clases que manipulan la vista se acopla con los objetos Sql, y esto puede conllevar a futuros problemas. Dejaremos esta implementación eficaz pero no eficiente para posteriores trabajos de mejora del *framework*.

Perfilado el funcionamiento de Filter, necesitamos una hoja de estilo CSS para el formulario de acuerdos y las distintas tablas. Tras unas pocas horas, conseguimos tener una hoja de estilos que diseñe y organice las páginas de acuerdo con las necesidades de la empresa. Como es habitual, no entraremos en detalles sobre el trabajo requerido y realizado para ello.

5.5 *Sprint* VII

Llegados al séptimo y último *sprint* de la segunda entrega y del proyecto completo, necesitamos dar los últimos retoques a la aplicación. El trabajo se desglosa en:

- Gestor para subir archivos al servidor.
- Descargar e instalar localmente JQuery.
- Menú para navegar entre las páginas

Sprint VII				30-04-17	
ID	UT	Dominio	PA	Horas estimadas	Horas reales
25	Modelo Acuerdo	A	Añadir métodos para la subida de archivos al servidor	7	9
			Localizar Jquery (antes se descargaba)	2	2
26	Vista Acuerdo	A	Realizar cambios en CSS	5	7
			Diseñar menú para navegar entre las pantallas de la aplicación	2	2
				Total	20

Tabla 7. Tabla con información del sprint VII.

Ya en la recta final de la codificación de la aplicación, necesitamos que los usuarios de la misma puedan subir los archivos al servidor, ya que, como en el anterior sprint hemos implementado, la aplicación tiene una funcionalidad de lectura de archivos csv. Para ello, en la clase Acuerdo, implementamos los métodos necesarios para ello. Cabe destacar que conseguimos hacerlo tras investigar mucho por internet y de probar varias formas, lo cual nos tomó un tiempo considerable.

Para perfilar la aplicación necesitamos un menú que nos permita navegar entre las diferentes páginas, por esto creamos un botón por cada pantalla y lo añadimos a la plantilla para que sea común en toda la aplicación.

Por otra parte, la aplicación descarga JQuery, implementado en el *sprint* IV, lo que provocaba un importante tiempo de ejecución. Por esto, decidimos localizar el archivo JavaScript que lo implementa. Creando una subcarpeta y cambiando las urls de internet a urls locales conseguimos resolver este requisito.

Para finalizar el último *sprint*, que cierra la segunda entrega y con ello el final de nuestras prácticas, solo nos restaba realizar unos cambios a las hojas de estilo, ya que a nuestros superiores no les convencía. Tras realizar los cambios, concluimos con el *sprint.*

6 Próximas entregas

Como el trabajo realizado ha dependido exclusivamente de las necesidades empresariales de cada *sprint*, hablar de próximas entregas de la aplicación carece de sentido. En cambio, referente al *framework* Core, consideramos como posible trabajo futuro:

- Continuación del estudio realizado al *framework* Core, ya que solo se han analizado las clases con las que hemos tratado y modificado.
- Remodelación de la clase Filter y Pagination. Hasta el momento, las dos clases realizan el trabajo de manera satisfactoria, pero contiene a su vez varios defectos. En primer lugar, deberíamos desacoplar las dos clases con Sql, ya que éstas se encargan de construir directamente la sentencia Sql y ejecutarla. Esto podría ser realizado mediante una clase pivote que las abstraiga de los datos a mostrar.
- Formulario automático desde una clase ActiveRecord. Otra funcionalidad pensada a implementar en próximas entregas es la automatización de creación de un formulario si ya conoce los campos y sus tipos. Esta idea ha estado rondándonos durante las prácticas, pero su codificación nos ha resultado tan complicada que hemos decidido dejarlo para trabajos futuros.
- Filtrado por ActiveRecord. Otorgar a la clase Filtro la capacidad de leer los atributos de un modelo ActiveRecord y poder crear el filtro automáticamente. Esta consideración requiere ser posterior a la anterior, ya que el formulario sería construido automáticamente desde un modelo.

7 Conclusiones

Habiendo expuesto las características y las partes que conforman el *framework*, junto con el trabajo ejercido tanto para la herramienta como para la aplicación, nos disponemos a sacar las conclusiones generadas a lo largo del estudio. Podemos desglosar las conclusiones extraídas en tres diferentes.

En primer lugar, existen muchos lenguajes de programación en el mercado capaces de satisfacer nuestras necesidades. Cómo nuestro ámbito se reduce a la programación web, el lenguaje usado, PHP, se amolda a nuestros requisitos, ya que se encapsula en código HTML y nos otorga herramientas muy útiles como los accesores mágicos, que nos ahorran una cantidad considerable de código. Pero esto no reduce a PHP al único lenguaje de nuestra elección, ya que cada uno tiene sus propios puntos fuertes que hubiéramos explotado para aumentar la capacidad funcional de nuestro *framework*.

Continuando con la segunda conclusión, el significado del concepto de *framework* es muy amplio. Ajustarse a uno disponible del mercado tiene tanto ventajas como defectos, su completitud es mayor a la par que el aprendizaje integral del uso puede ser costoso. El paso previo a elegir la herramienta con la que trabajar es definir claramente de las necesidades que necesitan ser cubiertas y el límite máximo de tiempo dedicado a su aprendizaje. En nuestro caso, tenemos un *framework* capaz de satisfacer nuestros requisitos empresariales con un tiempo de aprendizaje corto.

En último lugar, como hemos visto durante la narración de los *sprints* reflejada en los capítulos 4 y 5, la totalidad del trabajo generado en siete semanas refleja una capacidad de aprendizaje del *framework* bastante fluida. La aplicación resultado de nuestro trabajo cubre todas las necesidades que nos han ido surgiendo durante estas semanas, lo que argumenta la efectividad y la eficiencia del desarrollo de la aplicación mediante la herramienta Core.

Expuestas las conclusiones del trabajo, cabe destacar los puntos fuertes del *framework* y el desarrollo efectuado mediante él:

- Herramientas básicas para la construcción de aplicaciones web, como hemos visto. Si los futuros trabajos a realizar concuerdan con los requisitos en este trabajo definidos, volveríamos a usar la herramienta.
- La curva de aprendizaje resulta en sus principios compleja, pero con un poco de práctica, hemos aprendido a usar todas las herramientas de forma sencilla y directa.

En cambio, los puntos débiles del trabajo han sido:

- Nuestra intención, expuesta en el apartado 3.1 Características, la filosofía, de usar el tiempo de manera eficiente en lo que el desarrollo del *framework* a la par que el de la aplicación se refiere ha resultado ser ineficaz. En la mayoría de implementaciones, por ejemplo, en la clase Filter, el tiempo necesario en codificar la clase ha sobrepasado en exceso, bajo nuestra opinión, al tiempo necesario para crear un filtro independiente en cada página.
- El *framework* tratado se encuentra en fases muy tempranas de desarrollo. Debido a esto cualquier *framework* disponible en el mercado implementa, generalmente, más funcionalidad que Core. Esta es la razón de no haberle realizado ninguna comparación.

Y con esto concluye la narración del trabajo realizado, su exposición y las conclusiones del mismo.

8 Bibliografía

[1]. Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press

[2]. Álvarez, M.A.. *Qué es MVC.* https://desarrolloweb.com/articulos/que-es-mvc.html [Consulta 10 de mayo]

[3]. Beck, K. y Cunningham, W. (1987). “Using Pattern Languages for OO Programs”. En *OOPSLA*, n. 87.

[4]. Buckler, C. *SitePoint Smackdown: PHP vs Node.js.* https://www.sitepoint.com/sitepoint-smackdown-php-vs-node-js/ [Consulta: 5 de mayo de 2017]

[5]. Eich, B. (2010). *A Brief History of JavaScript.* En https://brendaneich.com/2010/07/a-brief-history-of-javascript/ [Consulta: 6 de mayo de 2017]

[6]. Fowler, M. (2004). *Inversion of Control Containers and the Dependency Injection pattern.* https://martinfowler.com/articles/injection.html [Consulta: 15 de mayo de 2017]

[7]. Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Londres: Addison-Wesley.

[8]. Krasner, G. E.y Pope, S.T. (1988). «A cookbook for using the model-view controller user interface paradigm in Smalltalk-80». En *The JOT*. SIGS Publications.

[9]. Leiva, A. *Patrones de diseño de software.* https://devexperto.com/patrones-de-diseno-software/ [Consulta: 3 de mayo de 2017]

[10]. Orix Systems. *¿Qué es un framework y para qué se utiliza?* https://www.orix.es/que-es-un-framework-y-para-que-se-utiliza [Consulta: 10 de mayo de 2017]

[11]. Reenskaug, T. (2003). “The Model-View-Controller (MVC) Its Past and Present”. En Java Zone, Oslo, 18 y 19 de septiembre.

[12]. Riehle, D. (2000). Framework Design: A Role Modeling Approach. Thesis. ETH Zürich.

[13]. The PHP Group (2001-2017). “Historia de PHP”. En http://php.net/manual/es/history.php.php [Consulta: 5 de mayo de 2017).