UNIVERSITAT DE BARCELONA

Trabajo de fin de grado

GRADO EN INGENIERIA INFORMÁTICA

Facultad de Matemáticas e Informática
Universidad de Barcelona

Modelo de Arquitectura de Software para Aplicaciones iOS basado en Clean Architecture

Autor: Jonattan Nieto Sánchez

Director:	**Dr. Sergio Sayago**
Realizado en:	**Universidad de Barcelona**

Barcelona, **22 de junio de 2017**

Resumen

El desarrollo de software para la plataforma iOS se ha convertido en estos últimos años en una parte importante dentro del sector del desarrollo de software. La velocidad a la que ha crecido la complejidad del software a desarrollar, ha provocado que los modelos de arquitectura utilizados hasta la fecha hayan quedado obsoletos para suplir las necesidades de un desarrollo de larga duración. En este trabajo se estudiarán y documentarán los modelos de arquitectura existentes para el desarrollo de aplicaciones para iOS. Tras ello, se utilizarán patrones de diseño de software para diseñar y documentar una propuesta de arquitectura con la que tratar de garantizar la sostenibilidad de un proyecto de software para iOS. Finalmente se aplicará el modelo de arquitectura propuesto en un desarrollo de software a modo de ejemplo, dando pie a una aplicación para iOS, intentando así garantizar la comprensión de este trabajo, y aportar todo mi análisis junto al modelo propuesto, al paradigma del desarrollo de aplicaciones para iOS.

Abstract

The development of software for the iOS platform have become in recent years an important part of the software development sector. The speed at which the complexity of the software to be developed has grown, has made the architecture models used to date obsolete to meet the needs of a long-term development. In this paper, the existing architecture models for the development of iOS applications will be studied and documented. After that, software design patterns will be used to design and document an architecture proposal with which to try to ensure the sustainability of an iOS software project. Finally, the proposed architecture model will be applied in a software development by way of example, giving rise to an application for iOS, trying to guarantee the understanding of this work, and contribute all my analysis with the proposal model, to the paradigm of application development for iOS.

Resum

El desenvolupament de software per a la plataforma iOS s'han convertit en aquests darrers anys en una part important dins el sector del desenvolupament de software. La velocitat a la qual ha crescut la complexitat del software a desenvolupar, ha provocat que els models d'arquitectura utilitzats fins a la data hagin quedat obsolets per suplir les necessitats d'un desenvolupament de llarga durada. En aquest treball s'estudiaran i documentaran els models d'arquitectura existents per

al desenvolupament d'aplicacions per iOS. Després d'això, s'utilitzaran patrons de disseny de software per a dissenyar i documentar una proposta d'arquitectura amb què tractar de garantir la sostenibilitat d'un projecte de software per iOS. Finalment s'aplicarà el model d'arquitectura proposat en un desenvolupament de software com a forma d'exemple, donant peu a una aplicació per iOS, intentant així garantir la comprensió d'aquest treball, i aportar tot el meu anàlisi juntament amb el model proposat, al paradigma del desenvolupament d'aplicacions per iOS.

Agradecimientos

A mis padres, por sacrificar el presente que consiguieron, para darme un futuro que no tuvieron.
A Salvador Martín, por su incondicionalidad, su apoyo técnico y su amistad.
A Eloi Guzman, por debatir conmigo todas y cada una de las ideas que plasmo en este trabajo.
Y a mi hermano Pablo, a quien le dedico este trabajo.

"Sometimes you gotta run before you can walk"
- Tony Stark

Índice general

1. Introducción **1**
- 1.1. Contexto y motivación . . . 1
- 1.2. Objetivo principal . . . 2
- 1.3. Metodología . . . 2
- 1.4. Resumen del trabajo desarrollado y principales resultados . . . 3
  - 1.4.1. Diseño del modelo de arquitectura . . . 3
  - 1.4.2. Documentación del modelo de arquitectura . . . 4
  - 1.4.3. Aplicación del modelo de arquitectura . . . 4

2. Estado del arte **5**
- 2.1. Model View Controller original . . . 5
- 2.2. Model View Controller propuesto por Apple . . . 6
- 2.3. Model View Controller realmente propuesto por Apple . . . 7
- 2.4. Model View ViewModel . . . 8
- 2.5. Arquitectura tradicional de capas . . . 9
- 2.6. Onion Architecture . . . 9
- 2.7. La idea Clean Architecture . . . 10
  - 2.7.1. Regla de la dependencia . . . 12
  - 2.7.2. VIPER . . . 13
  - 2.7.3. Mejoras respecto al resto de modelos de arquitectura documentados . 14
  - 2.7.4. Carencias de VIPER . . . 15

3. Arquitectura de software **17**
- 3.1. Ciclo de desarrollo de una arquitectura . . . 18

4. Definición de requisitos **21**
- 4.1. Requisitos de la estructura . . . 21
- 4.2. Requisitos del funcionamiento . . . 22
- 4.3. Requisitos de la interacción . . . 22

5. Diseño y documentación del Modelo de arquitectura **25**
- 5.1. Áreas del modelo de arquitectura . . . 26
  - 5.1.1. Business Rules . . . 26
  - 5.1.2. Interface Adapters . . . 26

5.1.3. Drivers y Frameworks . . . 27
5.2. Conceptos del modelo de arquitectura . . . 27
5.2.1. Tipos de Objetos . . . 27
5.2.2. Relación entre Componentes . . . 27
5.2.3. Comunicación entre Componentes . . . 28
5.3. View . . . 28
5.3.1. Salida de ViewEvents . . . 29
5.3.2. ViewEventsHandler Interface . . . 29
5.3.3. Entrada de ViewUpdates . . . 29
5.3.4. ViewUpdatesHandler Interface . . . 29
5.4. Presenter . . . 30
5.4.1. Objetos de tipo ViewModel . . . 31
5.4.2. Entrada de ViewEvents . . . 31
5.4.3. Salida de ViewNavigations . . . 31
5.4.4. NavigationsHandler Interface . . . 32
5.4.5. Salida de BusinessRequests . . . 32
5.4.6. RequestsHandler Interface . . . 32
5.4.7. Salida de ViewNavigations . . . 33
5.4.8. ResponsesHandler Interface . . . 33
5.4.9. Salida de viewUpdates . . . 34
5.5. Router . . . 34
5.5.1. Entrada de viewNavigations . . . 35
5.6. Interactor . . . 35
5.6.1. Entrada de businessRequests . . . 36
5.6.2. Objetos de tipo Entity . . . 36
5.6.3. Salida de businessResponses . . . 37
5.6.4. Salida de transactionRequest . . . 37
5.6.5. Entrada de transactionResponse . . . 37
5.7. Repository . . . 37
5.7.1. Objetos de tipo DTO . . . 38
5.7.2. Entrada de transactionRequest . . . 38
5.7.3. Entrada de providerRequest . . . 39
5.7.4. Mensajes entrantes de tipo providerResponse . . . 39
5.7.5. Entrada de transactionResponse . . . 40
5.8. Provider . . . 40
5.8.1. Mensajes entrantes de tipo providerRequest . . . 41
5.8.2. Entrada de providerResponse . . . 41
5.9. External DataSource . . . 42

6. Aplicación del Modelo de arquitectura **43**
6.1. Proyecto desarrollado . . . 43
6.2. El concepto Módulo . . . 43
6.3. Templates para la automatización de la creación un Módulo . . . 44
6.3.1. Builder . . . 44
6.3.2. View . . . 45

6.3.3. Presenter 46
6.3.4. Router 47
6.3.5. Interactor 47
6.4. Módulos implementados 48
6.4.1. LaunchScreen Module 48
6.4.2. SignIn Module 49
6.4.3. SignInWithLinkedIn Module 49
6.4.4. Welcome Module 51
6.4.5. Offers Module 52
6.4.6. BeaconsCoordinator 54
6.5. Estructura Lógica del Proyecto 55

7. Conclusiones **57**

8. Perspectivas de futuro **59**

Referencias **63**

Capítulo 1

Introducción

1.1. Contexto y motivación

Actualmente trabajo como desarrollador de software. En mi trabajo, me enfoco principalmente en el diseño y desarrollo de aplicaciones móviles para la plataforma iOS.

Uno de los principales problemas con el que me he encontrado, y que es bastante comentado en charlas y workshops de desarrollo para iOS, está relacionado con el diseño, la documentación e implementación del modelo de arquitectura para aplicaciones iOS.

Si bien es cierto que Apple propone oficialmente un modelo de arquitectura a seguir, también es cierto que la comunidad de desarrolladores ha acabado desestimando ese modelo de arquitectura por los problemas que conlleva su aplicación, ya que no da solución a todas las casuísticas de un desarrollo real.

Además, la implementación de las partes que se requieren en prácticamente todos los proyectos suele resultar un tanto repetitiva. Esto representa otro problema, ya que la repetición conlleva una pérdida de tiempo en el desarrollo, lo cual se traduce en costes innecesarios si existiera una forma de optimizarlos.

En el desarrollo de aplicaciones móviles para iOS, se aplica por defecto el modelo de arquitectura Modelo Vista Controlador propuesto por Apple, para aplicaciones sencillas. Con la evolución del sector y las tecnologías, las aplicaciones requieren de transmisión de datos, conexión a servidor y una lógica de negocio cada vez más compleja, por lo que el modelo de arquitectura propuesto por Apple resulta insuficiente para satisfacer esas necesidades.

Pese a que la comunidad de desarrolladores ha optado por aplicar otros modelos de arquitectura, como por ejemplo Model View ViewModel, ni se ha conseguido abarcar todas las partes que contempla un desarrollo de software, ni se ha llegado a establecer un consenso que defina un modelo de arquitectura completo.

Recientemente, ha aparecido una idea de arquitectura llamada Clean Architecture, la cual agrupa una serie de ideas con el fin de solventar el problema mentado previamente y ha aumentado el número de workshops relacionados con esta idea a los que los desarrolladores iOS hemos acudido en busca de una posible solución. No obstante, la interpretación y aplicación de esta idea no ha sido llevada a cabo correctamente por nadie, ya que tan solo se han llevado a cabo implementaciones parciales, con lagunas y algo ambiguas.

1.2. Objetivo principal

Con el fin de simplificar el proceso de desarrollo de un proyecto enfocado a la plataforma iOS, el objetivo de este Trabajo de Fin de Grado consiste en diseñar, documentar y aplicar un modelo de arquitectura de software que permita:

i) desarrollar el código de manera modular y reutilizable, respetando los patrones de diseño de software necesarios para garantizar la calidad del propio software así como garantizar su facilidad de mantenimiento y ampliación.

ii) crear e implementar automáticamente las clases necesarias en el desarrollo de una funcionalidad concreta del proyecto.

iii) garantizar que su uso resulte óptimo en cuanto al tiempo dedicado al desarrollo o bien a la refactorización de un desarrollo existente.

1.3. Metodología

Para alcanzar el objetivo de este Trabajo de Fin de Grado, se ha desarrollado un proyecto que sirve como prueba de concepto y que muestra el modelo de arquitectura propuesto aplicado, así como la implementación y uso de las plantillas que facilitan y automatizan la creación de las clases necesarias en el desarrollo.

La metodología de desarrollo ha sido una pequeña adaptación de la filosofía Test Driven Development, en la cual se utiliza el ciclo Red, Green, Refactor para implementar los diferentes casos de uso que contendrá el proyecto:

- Sign In mediante una cuenta de LinkedIn.
- Obtención de ofertas desde la API de InfoJobs.
- Detección de ofertas mediante proximidad a iBeacons.

Esta filosofía se aplica normalmente a los Test, normalmente unitarios, en la cual:

En el paso Red, se falla el test de forma intencionada.
En el paso Green, se pasa el test de la forma más fácil posible.

En el paso Refactor, se realiza un refactor para optimizar el código.

En este trabajo y debido a la simplicidad de la aplicación de ejemplo, se ha aplicado de forma que inicialmente se obtenía la funcionalidad en forma de código poco limpia, utilizando un modelo de arquitectura Modelo Vista Controlador, y tras ello se refactorizaba para obtener el modelo de arquitectura diseñado en este trabajo. De este modo, también se ha podido comprobar que el tiempo de refactorización de un código con una arquitectura más simple hacia uno con la arquitectura propuesta es óptimo, y la mejora en cuanto a la calidad del código es significativa.

Estos casos se han escogido debido a que abarcan todo el flujo de datos de una aplicación móvil, desde la interacción con el usuario hasta la obtención de datos por parte de un agente externo, y más concretamente en estos tres casos, cada uno de ellos obtiene los datos de un agente externo diferente, por lo que la arquitectura planteada debería poder ser aplicada a todos ellos independientemente de este detalle.

1.4. Resumen del trabajo desarrollado y principales resultados

1.4.1. Diseño del modelo de arquitectura

En este trabajo se ha realizado una investigación del estado del arte relacionado con los modelos de arquitectura de software utilizados en el desarrollo de software para iOS.

Tras ello se han analizado teniendo como referente el objetivo de obtener un modelo de arquitectura óptimo y se ha documentado dicho análisis, con el fin de plasmar un evolutivo de modelos de arquitectura hasta llegar al propuesto en este trabajo como resultado final.

Se ha diseñado y tras ello documentado un modelo de arquitectura de software para aplicaciones iOS basado en la idea Clean Architecture, propuesta por Robert C. Martin, ampliado y modificado acorde con el objetivo de este trabajo de fin de grado.

Para realizar el diseño y documentación del modelo de arquitectura, se ha investigado acerca de qué es una Arquitectura de Software y cuál es su ciclo de desarrollo, ampliando los conocimientos dados en la asignatura de Diseño de Software del Grado en Ingeniería Informática.

Además, se ha recopilado información acerca del ciclo de vida del desarrollo de una arquitectura de software, incluyendo una descripción de los patrones de diseño aplicados en este trabajo para el diseño del modelo de arquitectura propuesto.

Respecto al análisis del modelo de arquitectura a diseñar y documentar, se han definido los requisitos de la estructura, los requisitos de funcionamiento y los requisitos de

interacción.

Para definir estos requisitos se han recopilado y documentado todos los patrones de diseño utilizados en el diseño y aplicación del modelo de arquitectura propuesto en este trabajo.

Tras ello se ha realizado una interpretación de la idea Clean Architecture, y se ha diseñado el modelo de arquitectura acorde con la interpretación realizada, añadiendo las ampliaciones y rectificaciones convenientes.

1.4.2. Documentación del modelo de arquitectura

Una vez diseñado el modelo de arquitectura, se ha documentado cada uno de sus componentes, así como los objetos involucrados y las relaciones entre ellos y los componentes.

1.4.3. Aplicación del modelo de arquitectura

Finalmente se ha implementado la arquitectura en un caso práctico, desarrollando un proyecto de software para iOS a modo de ejemplo, en el cual se han tenido en cuenta los aspectos más interesantes de la aplicación del modelo de arquitectura.

Se ha desarrollado el proyecto de software para iOS utilizando el lenguaje de programación de Apple Swift 3.0 y el IDE Xcode 8.3.

El proyecto de software se ha desarrollado inicialmente aplicando el modelo de arquitectura Model View Controller, y posteriormente se ha refactorizado aplicando el modelo de arquitectura propuesto.

Se han implementado unas plantillas, denominadas *templates* para automatizar la creación de los componentes que forman el modelo de arquitectura propuesto.

Capítulo 2

Estado del arte

Lo más coherente al pretender iniciar un desarrollo enfocado a una plataforma en particular es verificar si el proveedor de la misma proporciona algún tipo de indicación referente a su arquitectura e implementación, por lo que el punto de partida de este trabajo será analizar el modelo de arquitectura que propone Apple para el desarrollo de aplicaciones en la plataforma iOS y se proseguirá siguiendo el evolutivo de modelos de arquitectura agrupados en la referencia [1].

2.1. Model View Controller original

Apple propone como modelo de arquitectura el uso del patrón de arquitectura Modelo Vista Controlador, pero no es el patrón original, por lo que a continuación se especifica el original para entender su evolución.

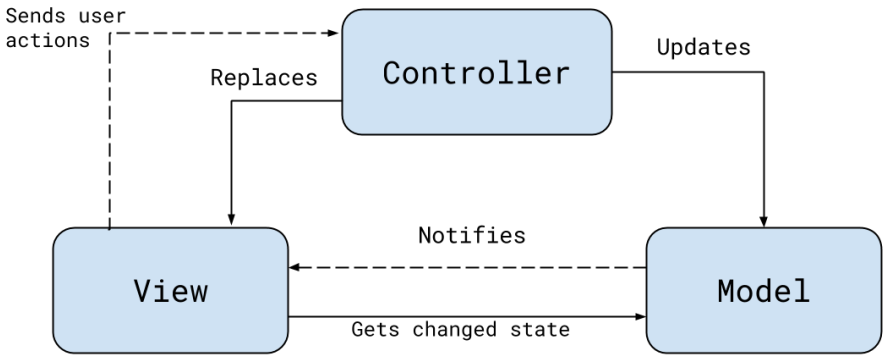

Figura 2.1: Model View Controller original. (Fuente: Las imágenes de esta sección están basadas en una adaptación de [1].)

En el patrón Model View Controller original, abreviado como MVC, la vista carece de

estado y es actualizada por el controlador cada vez que se recibe un cambio, por lo que los componentes del modelo de arquitectura están estrechamente unidos, ya que cada una de ellos está relacionado con los demás, por lo que la reutilización de cada uno de ellos se ve drásticamente reducida, debido a su dependencia, por lo que Apple propone una modificación del MVC original.

2.2. Model View Controller propuesto por Apple

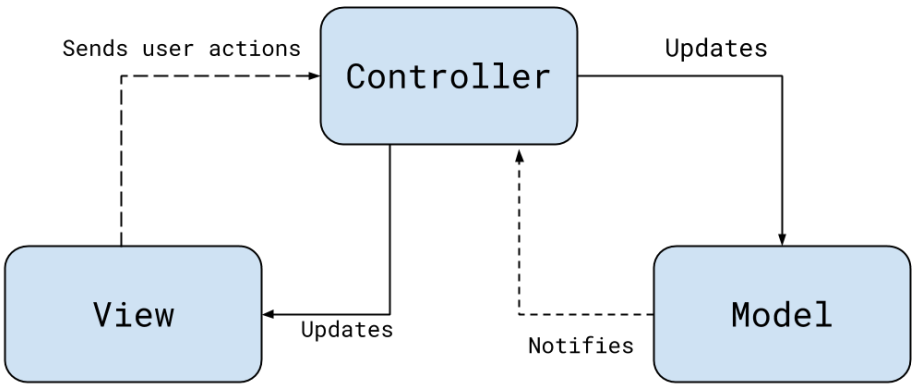

Figura 2.2: Model View Controller propuesto por Apple

Apple propone un modelo de arquitectura en el cual el controlador es un mediador entre la Vista y el Modelo, deshaciendo la relación de dependencia que existía entre estos en el patrón MVC Original.

El problema de este patrón reside en el controlador, ya que, pese a que se consigue independencia entre elementos, el controlador aglomera toda la lógica de negocio, y está muy relacionado con la vista, debido a la implementación de la clase nativa de Apple, el UIViewController [2]. Todo ello, genera un problema que la comunidad de desarrolladores ha denominado Massive View Controller [3], es decir, aglomeración masiva de lógica de negocio en el ViewController, haciendo inviable la aplicación de Tests, como se comenta más adelante.

Por ello, puestos a analizar el modelo de arquitectura propuesto por Apple, hagámoslo respecto al verdadero modelo de arquitectura, teniendo en cuenta el acoplamiento existente en iOS entre la Vista y el Controlador.

2.3. Model View Controller realmente propuesto por Apple

En iOS, la clase principal sobre la que se sustenta la interacción entre el usuario y la aplicación es la clase nativa UIViewController, la cual se relaciona directamente con la vista, siendo esta un fichero .xib [4].

La clase UIViewController es la encargada de capturar los eventos de la Vista y manejarlos, por lo que no existe una separación lógica entre UIView y UIViewController, quedando el modelo de arquitectura tal y como se puede observar en el diagrama de la figura 2.3.

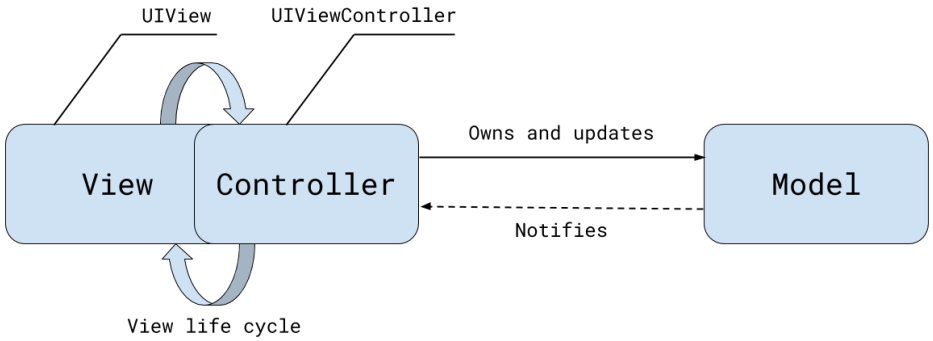

Figura 2.3: Model View Controller realmente propuesto por Apple

Es difícil decir que realmente la Vista y el Controlador sean dos objetos separados, ya que el Controlador está tan ligado al ciclo de vida de la vista, que acaba siendo más bien un delegado o una fuente de datos de la misma, aglomerando toda la lógica de negocio, incluyendo la interacción con el modelo de datos o la solicitud de datos de un servidor externo a la aplicación.

Además, en la implementación de este proceso es frecuente que la vista reciba una instancia de una entidad de modelo como parámetro con la cual configurarse, violando completamente lo definido en el patrón MVC donde se especifica claramente que el Modelo y la Vista no deben tener ningún tipo de dependencia entre ellos, pero de no hacerlo así, de nuevo se sobrecarga el controlador.

Pese a todo, el problema se hace muchísimo más evidente cuando se deciden aplicar tests unitarios, ya que, debido a la estrecha unión entre el ViewController y la Vista, la lógica de negocio suele estar ligada al ciclo de vida de la propia vista, por lo que no suele existir una separación entre la lógica de diseño, configuración vista y la lógica de negocio.

Como conclusión, el modelo de arquitectura MVC propuesto por Apple es un modelo sencillo, que puede ser factible en desarrollos simples, pero a medida que un proyecto crece, provocará un coste elevado en cuanto a mantenimiento y ampliación de la funcio-

nalidad, por lo que resulta evidente que se deben buscar alternativas a este modelo de arquitectura.

2.4. Model View ViewModel

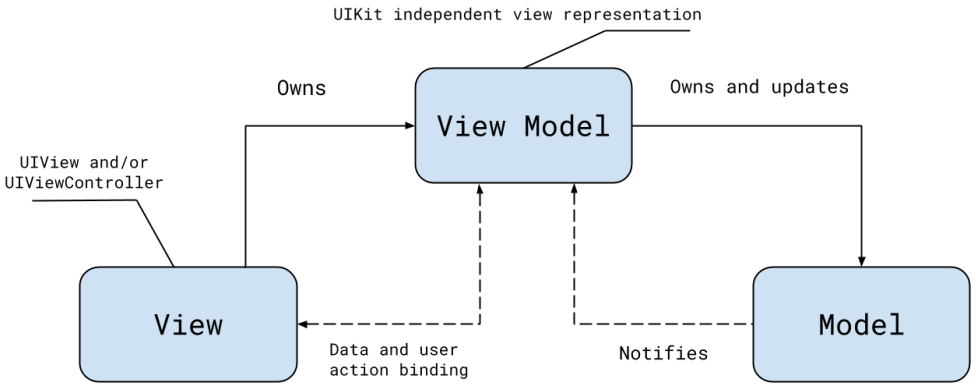

Figura 2.4: Model View ViewModel

El Model View ViewModel, abreviado MVVM, es el modelo de arquitectura más completo de todos los que definen en su implementación los componentes Modelo y Vista. Además, añade un componente nuevo, llamado ViewModel o modelo de vista, el cual hace de mediador entre ambos componentes.

Este modelo de arquitectura trata al ViewController como si de la vista se tratase, y el componente ViewModel es el encargado de contener toda la lógica de negocio que afecta a la vista, desacoplando de la lógica propia de la vista, que estará alojada en el Controller.

El propio ViewModel es el encargado de obtener los cambios que se producen en el Modelo, y proveer a la Vista de los mismos con el fin de que esta se actualice.

Esta implementación es extremadamente útil enfocada a la programación reactiva, ya que la actualización de los cambios del ViewModel que afectan a la Vista se hacen de forma automática y transparente de cara al desarrollador.

Como conclusión, este modelo de arquitectura puede ser una alternativa factible para desarrollos de envergadura media, no obstante, no acaba de separar toda la lógica de negocio, y tiende a crear ViewModels exageradamente grandes, por lo que se acaba trasladando el problema del Massive View Controller al ViewModel.

2.5. Arquitectura tradicional de capas

Lamentablemente, los modelos de arquitectura analizados hasta ahora, no aportan ninguna definición más allá de las áreas relacionadas con la Interfaz de Usuario o Presentación, por lo que en el caso de que se decidiera realizar una implementación de un software basado en dichos modelos de arquitectura, existirían lagunas a la hora de definir las dos áreas restantes mencionadas.

La arquitectura de capas tradicional [5], define tres capas que estructuran la lógica del software, siendo la primera la contenedora de la lógica de User interface o interfaz de usuario, la cual se relaciona directamente con la capa contenedora de la Business Logic o lógica de negocio, y esta a su vez se relaciona con la capa llamada Data o contenedora de datos. Además de estas relaciones en cascada, las tres capas tienen una relación directa con el área de infraestructura, es decir Base de datos o Servidor, tal y como se muestra a continuación en la figura 2.5.

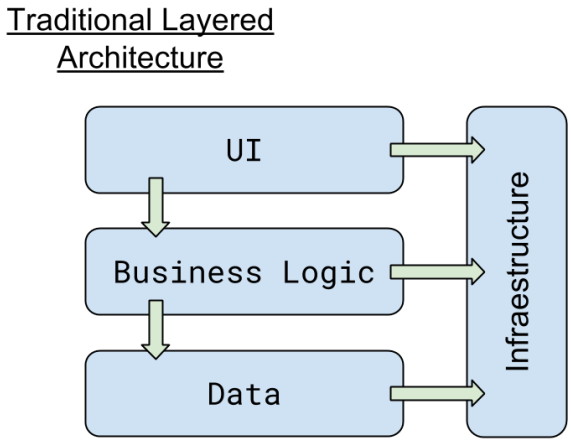

Figura 2.5: Arquitectura tradicional de capas

Pese a que este modelo de arquitectura define separación entre las tres capas principales del software, contiene un acoplamiento total entre la infraestructura y cada una de estas capas, dando pie a violaciones de los principios SOLID, ya que más de un componente de las diferentes áreas está relacionado con el área de la infraestructura.

2.6. Onion Architecture

Con el fin de solventarlo, Jeffrey Palermo propuso la denominada Onion Architecture [6] o arquitectura en forma de cebolla, la cual reestructura el modelo de arquitectura en círculos, siendo estos círculos las áreas del modelo de arquitectura.

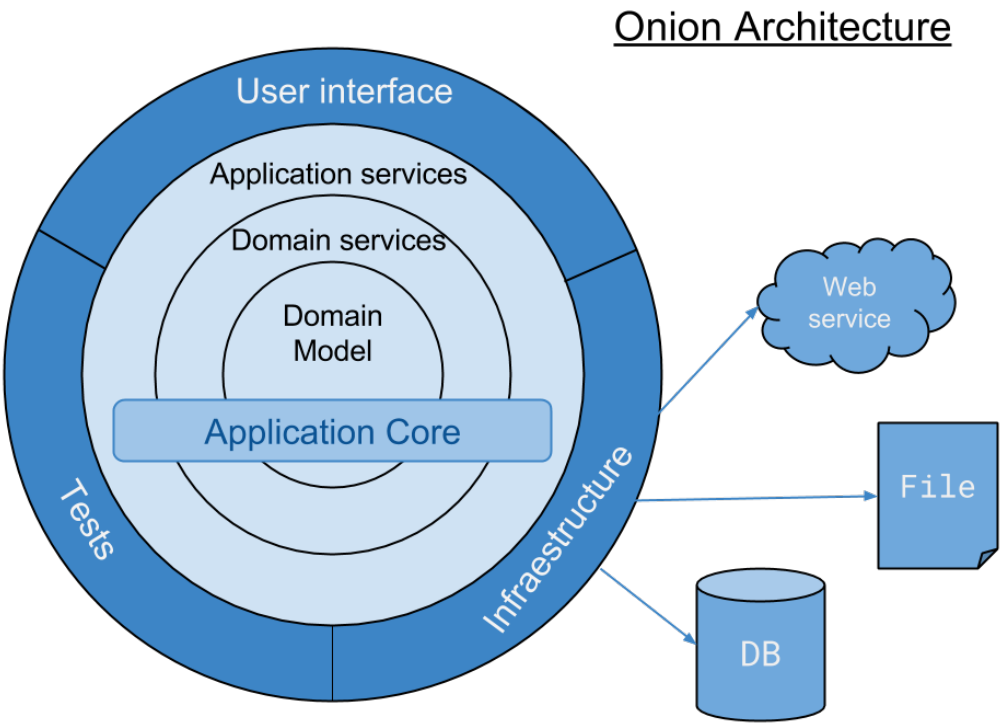

Figura 2.6: Onion Architecture

Respecto a la Arquitectura de Capas tradicional la infraestructura pasa a estar en el círculo más externo, por lo que la relación de dependencia ya no es múltiple.

2.7. La idea Clean Architecture

Clean Architecture [7] es una idea de modelo de arquitectura que propone el desarrollador Robert C. Martin, el mismo que propuso aplicar los conceptos SOLID al desarrollo de software.

En el mundo del software las tendencias acaban marcando un antes y un después en maneras de hacer las cosas, y Clean Architecture es uno de estos casos. La idea tuvo una buena acogida en el sector, pero tras analizarla con detenimiento se puede observar es una agrupación de ideas que ya existían, evolucionando de la Onion Architecture, tal y como se puede observar a continuación.

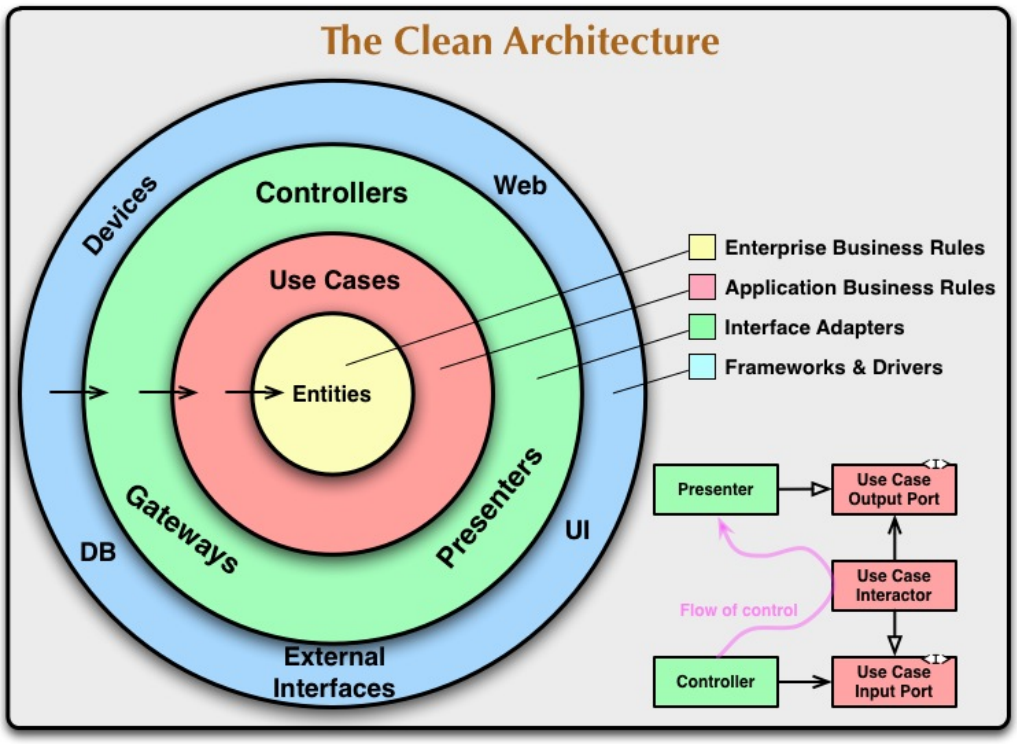

Figura 2.7: The Clean Architecture

En la imagen superior se puede observar la esfera de Clean Architecture, la cual describe las diferentes capas, las cuales más que componentes agrupan ideas o conceptos relacionados, de más a menos externo, a lo que denominaremos áreas. Las flechas indican la regla de la dependencia, desde fuera hacia adentro.

Desde el centro hacia afuera encontraríamos los objetos de modelo de nuestro desarrollo, seguidos de la lógica de negocio, justo después los manejadores de datos y finalmente las partes más externas de la infraestructura, que sería hacia dónde dirigir los datos o eventos, es decir UI o BBDD.

Clean Architecture define sus cuatro esferas como Entities, Use Cases, Interface Adapters y Frameworks and External Interfaces. A continuación se define cada una en profundidad.

Las **Entities** o Entidades, son los objetos de negocio los cuales encapsulan las reglas de negocio más generales y de alto nivel. En su implementación, serán los denominados Business Objects [8], y puesto que estarían situados en el círculo más interno, deberían ser los menos expuestos a cambios influidos por capas exteriores.

Los **Casos de uso**, contienen las reglas de negocio específicas del desarrollo, por lo que encapsulará todos los casos de uso de la aplicación, los cuales pueden estar expuestos a cambios ya que la funcionalidad del software siempre es ampliable. Los casos de uso serán los encargados de manejar el flujo de datos, y utilizará las Entities para alcanzar los objetivos del caso de uso y propagará los resultados a las capas más externas.

Los **Adaptadores de Interficie** serán los encargados de transformar los datos que se reciben de los Casos de uso para enviarlos hacia la capa más externa, donde reside la UI, la BBDD o bien un Servidor externo, y viceversa.
Robert propone el uso del patrón MVC para la implementación de esta capa, pero ya se ha comentado previamente que el uso de este patrón estaba descartado, por lo que realizaremos una implementación diferente a la propuesta por Robert.

Los **Frameworks e Interficies Externas** serán los que formen la capa más externa, generalmente compuesta por frameworks y herramientas como la Base de datos, la UI o el Servidor. Según Robert en esta capa debe escribirse únicamente código de comunicación hacia la capa de Casos de Uso, ya que los frameworks y herramientas se consideran detalles externos que no deben afectar a las capas internas.

2.7.1. Regla de la dependencia

Esta arquitectura basa su estructura en el **Dependency Inversion Pattern** [9] o patrón de inversión de dependencias y en la denominada regla de la dependencia.

Esta regla indica que una capa sólo puede conocer a otra capa que esté en un nivel más céntrico que él, jamás uno más externo.

Dicho de otro modo, la dependencia desde el exterior hacia el interior, por lo que la comunicación entre módulos se basa en implementar interfaces que permiten comunicar un módulo superior con uno inferior.

Esto permitirá que las capas más externas están más expuestas a extensiones (recordemos que jamás deberían ser modificaciones), mientras que las capas más internas deberían ser más estáticas a lo largo del desarrollo.

Características que garantiza la Regla de la dependencia

Con esta regla se pretende que el modelo de arquitectura cumpla los siguientes puntos:

Independiente de los Frameworks: La arquitectura no depende de la existencia de alguna biblioteca de software cargado de características. Esto le permite utilizar frameworks como herramientas, en lugar de tener que acoplar su sistema a sus restricciones, limitándolo.

Testable. Las reglas de negocio pueden probarse sin la interfaz de usuario, la base de datos, el servidor Web o cualquier otro elemento externo.

Independiente de Interfaz de Usuario. La interfaz de usuario puede cambiar fácilmente, sin cambiar el resto del sistema. Una interfaz de usuario Web se puede sustituir por una interfaz de usuario de consola, por ejemplo, sin cambiar las reglas de negocio.

Independiente de la base de datos. Puede intercambiar el framework utilizado para gestionar la base de datos. Sus reglas de negocio no están vinculadas a la base de datos ni a su framework.

Independiente de cualquier agencia externa. De hecho, sus reglas de negocio simplemente no saben nada en absoluto sobre el mundo exterior.

2.7.2. VIPER

A diferencia de una propuesta de modelo de arquitectura, Clean Architecture es más bien un concepto. Si bien es cierto que especifica las áreas de las cuales debe estar compuesto el desarrollo, así como la relación entre ellas, no es un modelo de arquitectura cerrado y tampoco define realmente unas reglas de implementación, sino que debe ser diseñado específicamente para un desarrollo concreto, por lo que en este trabajo se diseñará, documentará e implementará para una aplicación en iOS utilizando el lenguaje de programación de Apple llamado Swift con las ampliaciones, adaptaciones y consideraciones pertinentes.

No obstante, el punto de partida será el concepto VIPER, que no es más que una aplicación parcial de Clean Architecture para iOS.

VIPER es el acrónimo de las palabras View, Interactor, Presenter, Entity y Router, las cuales vienen a ser las diferentes áreas de responsabilidad que constituyen una aplicación móvil para el sistema operativo iOS. [10]

Figura 2.8: Componentes de VIPER, Fuente: [10]

Componentes de VIPER

A continuación se describen brevemente cada uno de los componentes de VIPER.

- La **View**, o Vista muestra todo cuanto le dice el Presenter, y envía la interacción del usuario hacia el Presenter.
- El **Interactor**, contiene la lógica de negocio específica de un caso de uso.
- El **Presenter** contiene la lógica para preparar el contenido recibido por el Interactor para ser mostrado en la Vista.
- Las **Entities** contienen modelos básicos de datos usados por el Interactor.
- El **Router** contiene la lógica de navegación y animación.

2.7.3. Mejoras respecto al resto de modelos de arquitectura documentados

VIPER separa la interfaz de usuario de la lógica de negocio, ya que la existencia de componentes dedicados a una sola responsabilidad lo permite.

Por ahora, VIPER es el único modelo que dedica un componente a la lógica de navegación y animación entre pantallas, algo que es muy frecuente en el desarrollo móvil, incluido el desarrollo para iOS, por lo que el hecho de desacoplar esa lógica permite al equipo de desarrollo focalizarse en algo tan importante como es la User Experience, o UX, cosa que en el resto de modelos de arquitectura estaba completamente acoplada a la lógica de negocio y a las vistas, haciendo difícil su desarrollo.

2.7.4. Carencias de VIPER

Lamentablemente, pese a que el aporte de mejoras respecto a los modelos de arquitectura descritos hasta ahora en este trabajo es significativo, la aplicación de Clean Architecture denominada VIPER no cumple realmente los siguientes puntos que debería garantizar la regla de la dependencia:

No es **Independiente de los Frameworks, Independiente de la base de datos ni Independiente de cualquier agencia externa**, debido a que en VIPER no se especifica una forma de acceder a datos externos, por lo que se tiende a implementar en el Interactor, dejando la lógica de negocio y la lógica de obtención de datos completamente ligada.

No es del todo **Testable** debido a que VIPER especifica en su implementación que todas las relaciones entre componentes dependan de sus interfaces, y por lo tanto se produce acoplamiento entre componentes dificultando la realización de tests.

Pese a que VIPER no describe cómo debe aplicar la arquitectura a la parte de obtención de recursos externos, así como conexiones a servidores remotos u otras fuentes de datos, ni tampoco cómo debe establecerse realmente la conexión entre los diferentes componentes ni la retención en memoria de los mismos, será el punto de partida de la denominación de los componentes, debido al impacto que tuvo en su aparición en el desarrollo para iOS, por lo que podría entenderse la arquitectura planteada en este trabajo como una evolución del planteamiento VIPER, con las correcciones y ampliaciones convenientes para solventar dichas carencias.

Capítulo 3

Arquitectura de software

¿Que es una arquitectura de software?

La Arquitectura del Software, resumiendo [11], es la temática de la Ciencias de la Computación que indica y define la **estructura**, **funcionamiento** e **interacción** entre las partes del software, siendo el diseño a más alto nivel de la estructura de un sistema.

Una arquitectura se selecciona, si ya existiera, o bien se diseña expresamente en base a los requerimientos y las restricciones del problema a solventar mediante la misma.

Además, debe definir, de manera abstracta, los componentes o elementos que forman parte de la arquitectura, sus interfaces y cómo se comunican entre ellos.

Puesto que la arquitectura debe describir diversos aspectos del software, estos se describen de la forma más comprensible empleando diferentes modelos, que sumados juntan la descripción completa de la arquitectura de entre los cuales se destacan los tres más fundamentales:

El **modelo estático**, describe qué componentes contiene la arquitectura, haciendo uso de diagramas de clases y diagramas de objetos, los cuales definen las clases y sus relaciones entre ellas.

El **modelo dinámico**, describe el comportamiento de los componentes a lo largo del tiempo y la interacción entre los mismos, haciendo uso de diagramas de estados y diagramas de suceso.

El **modelo funcional**, describe la funcionalidad de cada componente, haciendo uso del diagrama de funciones, el cual define los actores que perpetúan los procesos y su flujo.

Los modelos pueden expresarse mediante diferentes lenguajes, ya sea el lenguaje natural, o mediante diagramas de flujo de datos, diagramas de estado, etc...

Cada forma de expresión, puede ser apropiada únicamente para un determinado modelo de la arquitectura, por lo que actualmente existe el consenso de adoptar **UML** [12] (Unified Modeling Language, es decir lenguaje unificado de modelado) como único lenguaje para todos los modelos. No obstante, siempre puede darse el caso en el que se requiera de otro tipo de expresión para describir de forma más clarificadora alguna especificación determinada, por lo que no es una forma de expresión cerrada o estricta, ya que lo que se busca es la comprensión de la expresión.

Puesto que en este trabajo, el objetivo es realizar una propuesta de modelo de arquitectura aplicable a cualquier tipo de aplicación iOS que requiera las funcionalidades más frecuentes en este tipo de desarrollo, se procederá a desarrollar únicamente el **modelo estático** de la arquitectura de forma abstracta y se añadirán anotaciones relacionadas con el **modelo dinámico** y el **modelo funcional**.

3.1. Ciclo de desarrollo de una arquitectura

El desarrollo de la arquitectura del Software precede a la construcción del propio Software, por lo que, siendo también un desarrollo, tiene sus propias etapas [13].

Puesto que el objetivo de este trabajo es conseguir aplicar una arquitectura de software a un problema concreto, empezaremos por describir las etapas en el desarrollo de una arquitectura.

La primera etapa es la de definición de **requerimientos** que influenciarán a la arquitectura, donde se enumeran y documentan.

La segunda etapa es la de **diseño** de la arquitectura. Es la etapa más compleja, ya que se deben definir las estructuras que componen la arquitectura, así como todos sus elementos y componentes. El diseño se basa en el uso de patrones y técnicas de diseño, así como elecciones tecnológicas con el fin de satisfacer los requerimientos.

Tras el diseño, está la etapa de **documentación**, la cual sirve para comunicar a cualquiera que esté involucrado en el desarrollo, incluyendo la representación de las estructuras, elementos y componentes de forma adecuada, utilizando los diferentes modelos nombrados previamente en función de la necesidad, con el fin de clarificar al máximo el diseño de la propia arquitectura.

Finalmente está la etapa de **evaluación**, donde se analiza la arquitectura diseñada antes de aplicarla al desarrollo, con el fin de identificar posibles problemas y riesgos y en caso de encontrarlos analizarlos de manera temprana y con un bajo coste en comparación con la corrección de dichos problemas en una arquitectura llevada ya a cabo mediante software.

Pese a que tal y como se ha descrito previamente, el ciclo de desarrollo de una ar-

quitectura está formado por cuatro partes, y puesto que en este trabajo el objetivo es documentar la evolución que se ha seguido hasta conseguir la propuesta del modelo de arquitectura se adaptarán los cuatro apartados, de forma que se dedicará la primera parte de esta memoria a la definición de requerimientos, donde se mentarán tanto los requerimientos funcionales que deberá implementar la aplicación iOS como los requerimientos arquitectónicos que deberá cumplir la arquitectura propuesta y sus componentes.

Debido a que la parte de diseño de la arquitectura propuesta ha sufrido cambios durante toda la elaboración de este trabajo, se describe de forma conjunta la evolución del diseño y la documentación del mismo, con el fin de clarificar al máximo el resultado.

Capítulo 4

Definición de requisitos

Los requisitos del modelo de arquitectura a diseñar deben estar ligados tanto a buenas prácticas del desarrollo de software, como a patrones de diseño, y por supuesto relacionados directamente con el framework nativo de iOS.

4.1. Requisitos de la estructura

Para el diseño de la **estructura** del modelo de arquitectura, se tendrán en cuenta las características que pretendía garantizar la regla de la dependencia descrita en el apartado de Clean Architecture, los cuales se recuerdan a continuación.

Independiente de los Frameworks: La arquitectura no depende de la existencia de alguna biblioteca de software cargado de características. Esto le permite utilizar frameworks como herramientas, en lugar de tener que acoplar su sistema a sus restricciones, limitándolo.

Testable. Las reglas de negocio pueden probarse sin la interfaz de usuario, la base de datos, el servidor Web o cualquier otro elemento externo.

Independiente de Interfaz de Usuario. La interfaz de usuario puede cambiar fácilmente, sin cambiar el resto del sistema. Una interfaz de usuario Web se puede sustituir por una interfaz de usuario de consola, por ejemplo, sin cambiar las reglas de negocio.

Independiente de la base de datos. Puede intercambiar el framework utilizado para gestionar la base de datos. Sus reglas de negocio no están vinculadas a la base de datos ni a su framework.

Independiente de cualquier agencia externa. De hecho, sus reglas de negocio simplemente no saben nada en absoluto sobre el mundo exterior.

Cumpliendo estos requisitos descritos, conseguimos una arquitectura de calidad en

cuanto a las relaciones entre sus componentes, ya que todos estos atributos están directamente relacionados con cómo se relacionan los componentes.

4.2. Requisitos del funcionamiento

Para el diseño del **funcionamiento** de la arquitectura y sus componentes, se tendrán en cuenta los principios **SOLID** [14], acrónimo para cinco principios que introdujo Robert C. Martin referentes a la programación orientada a objetos, y al diseño de esta, y se describen a continuación.

El primero, **Single responsibility principle** o principio de responsabilidad única, especifica que un objeto debe tener tan solo una responsabilidad, es decir, no debería abarcar más de una responsabilidad, separando las responsabilidades en objetivos y manteniendo una estructura clara y sencilla.

El segundo, **Open/Closed principle** o principio de abierto/cerrado, específica que una entidad de software debe estar siempre abierta a poder ser extendida, y cerrada a cualquier modificación, siendo este un principio que puede marcar la escalabilidad del desarrollo.

El tercero, **Liskov substitution principle** o principio de sustitución de Liskov, especifica que, en un programa, los objetos se pueden reemplazar por otros que hereden de éstos sin que el funcionamiento se vea afectado.

El cuarto, **Interface Segregation principle** o principio de segregación de la interfaz, específica que es preferible una segregación de interfaces específicas a una única interfaz de propósito general.

El quinto, **Dependency inversion principle** o principio de inversión de la dependencia, especifica que el software debe depender de abstracciones y no de implementaciones.

4.3. Requisitos de la interacción

Para el diseño de la **interacción** de la arquitectura y sus componentes, se tendrán en cuenta los principios **GRASP** [15], acrónimo de General Responsibility Assignment Software Patterns, los cuales son patrones generales para la asignación de responsabilidades, y se describen a continuación.

El patrón **Experto en información**, relacionado con el patrón **Creador**, es el principio básico de asignación de responsabilidades, el cual describe que la responsabilidad de la creación de un objeto o la implementación de un método debe recaer sobre la clase que conoce toda la información necesaria para crearlo, obteniendo de esta forma un diseño con mayor cohesión y encapsulando la información.

El patrón **Controlador** sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, dando pie a separar la lógica de negocio de la lógica de presentación, facilitando la reutilización de código y disminuyendo el acoplamiento.

El patrón de **Poliformismo**, especifica que siempre que sea necesario llevar a cabo una responsabilidad que dependa del tipo, se haga uso del polimorfismo de clases o métodos.

El patrón de **Variaciones Protegidas** es el principio fundamental de protegerse del cambio, de tal forma que todo lo que prevemos en un análisis previo que es susceptible de modificaciones, lo envolvemos en una interfaz, utilizando el **polimorfismo** para crear varias implementaciones y posibilitar implementaciones futuras, de manera que quede lo menos ligado posible a nuestro sistema, de forma que cuando se produzca la variación, nos repercuta lo mínimo. Forma parte de los patrones Grasp avanzados.

Los patrones de **Alta cohesión** y **Bajo acoplamiento** definen que la cohesión del código y el acoplamiento del mismo deben ser inversamente proporcionales. La cohesión define que la información que almacena una clase debe ser coherente y estar relacionada con la misma, mientras que el acoplamiento especifica la dependencia entre clases.

El patrón de **Fabricación pura** especifica que, dado el caso en el cual se necesite disminuir el acoplamiento entre clases, existe la posibilidad de crear clases que no representen un ente u objeto real del dominio, pero que facilitan la cohesión y potencien la reutilización de código.

El patrón de **Indirección** permite mantener el bajo acoplamiento entre dos clases asignando la responsabilidad de la mediación entre ellos a un tercer elemento intermedio.

Capítulo 5

Diseño y documentación del Modelo de arquitectura

En primer lugar, el diseño se centrará en la estructura de la arquitectura, donde se separarán los componentes en función de su afinidad según el tipo de funcionalidad con la que estén relacionados.

Esta separación se formará en tres áreas de software ya mencionadas al tratar **Clean Architecture**, las cuales serán el ámbito de **Business Logic**, el ámbito de **Interface Adapters** y el ámbito de **Frameworks y Drivers**.

A continuación se muestra un diagrama de estructura con las áreas mencionadas, los componentes, sus relaciones, el paso de mensajes entre ellos y los tipos de objetos que utilizan.

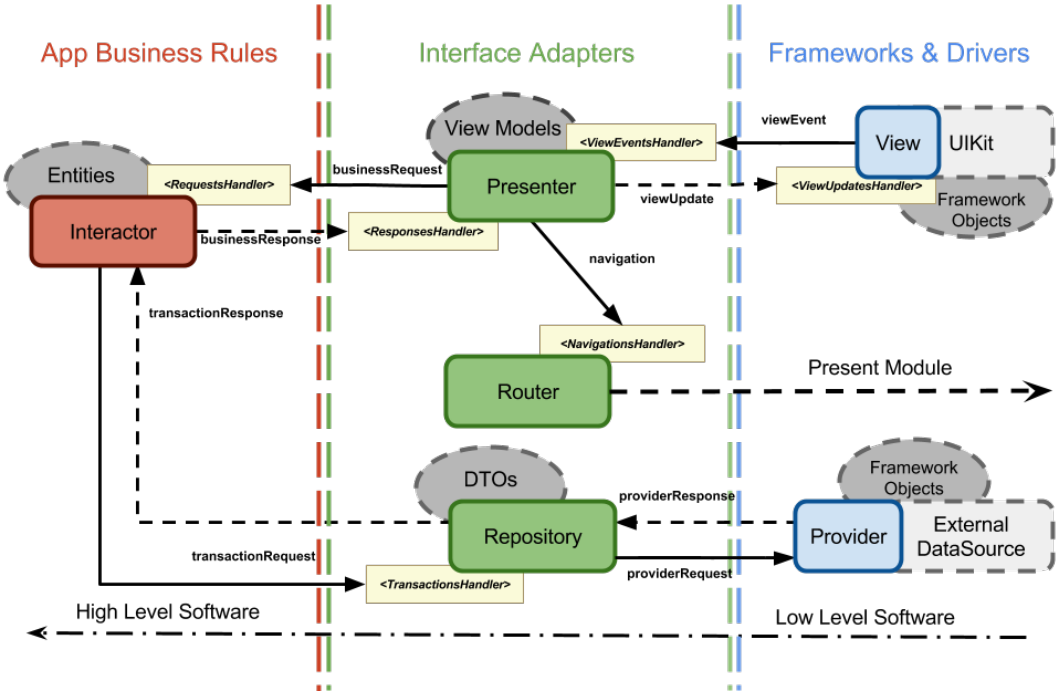

Figura 5.1: Visión general del modelo de arquitectura

Tal y como se puede observar en el diagrama superior, la estructura de la arquitectura está basada en la agrupación de sus componentes en función del nivel del software al que corresponden, en función de si están más o menos alejados de la comprensión humana. A continuación se describen las tres áreas.

5.1. Áreas del modelo de arquitectura

5.1.1. Business Rules

Es el área más cercana al entendimiento y comprensión humana, por lo que alojará toda la lógica de negocio y normas que definen la funcionalidad del software.

En esta área reside el componente **Interactor**, encargado de la lógica de negocio.

5.1.2. Interface Adapters

Es el área intermedia, la cual realiza la función de puente entre el área de **Business Rules** y el área de **Drivers y Frameworks**.

En esta área residen los componentes **Presenter**, **Router** y **Repository**, encargados de servir como puente entre componentes de otras áreas.

5.1.3. Drivers y Frameworks

Es el área más alejada de la lógica de negocio, en la cual residen los componentes que interactúan directamente con los frameworks utilizados en el software.

En esta área residen los componentes **View** y **Provider**, los cuales son intérpretes de los frameworks que utilizan para ser implementadas sus funcionalidades.

No obstante, puesto que resultaría complicado describir la evolución del diseño de la arquitectura y sus relaciones entre componentes si partimos de la distribución de estas áreas, se realizará una descripción en función del flujo de datos iniciado en una **View** hasta llegar a un **DataSource** Externo.

5.2. Conceptos del modelo de arquitectura

Tal y como se puede ver en el diagrama del apartado previo, el modelo de arquitectura diseñado se compone de elementos que se referenciarán en este trabajo como componentes, los cuales son los ya nombrados previamente: **View, Presenter, Router, Interactor, Repository y Provider**.

5.2.1. Tipos de Objetos

Cada componente pertenece a una de las tres áreas descritas en el apartado previo, y utiliza un tipo de objeto para la gestión interna de su lógica, los cuales son:

- Objetos propios del Framework o **Framework Objects** para los componentes **View** y **Provider**.
- **ViewModels** para el componente **Presenter**.
- **Entities** para el componente **Interactor**.
- **DTO** o Data Transfer Object para el componente **Repository**.

El objetivo de estos objetos, es que cada componente realice su lógica únicamente con su tipo de objeto asociado, pudiendo ser receptor de algún otro tipo de objeto, pero únicamente como transportador de datos desde otra capa, y nunca deberá aplicar lógica usando ese objeto, sino transformándolo previamente a su tipo de objeto asociado. De esta forma, se evita que un componente pueda realizar lógica que debería realizarse en otro componente.

5.2.2. Relación entre Componentes

La relación de **agregación**, se indica en el diagrama con una flecha continua desde el componente que es poseedor la referencia, hasta el componente referenciado.

La relación de **asociación**, se indica en el diagrama con una flecha discontinua desde el componente que es poseedor la referencia, hasta el componente referenciado.

5.2.3. Comunicación entre Componentes

La comunicación entre dos componentes relacionados entre sí, será interpretada como si se comunicaran mediante el envío de mensajes. El envío de un mensaje por parte del componente A hacia el componente B, será equivalente a que el componente A, tenga como referencia el componente B, e invoque un método del componente B.

Los posibles mensajes a enviar, es decir los métodos que implemente un componente estarán definidos en las interfaces implementadas por ese componente, tal y como se explicará más detalladamente en la documentación de cada uno de los componentes.

Una vez concretados estos detalles como rasgos generales de la arquitectura, se inicia la documentación de la misma por el elemento **View**.

5.3. View

La **View** pertenece al **área de Frameworks y Drivers**, ya que su implementación está completamente ligada al framework que utiliza para visualizarla en pantalla, en este caso el framework nativo de iOS para vistas, denominado **UIKit**. [16]

La **responsabilidad** de la **View** consiste en el manejo de datos relacionados con la representación gráfica de los datos y su interacción con el usuario, cumpliendo con el **Single responsibility principle**.

Las **relaciones** de la **View** serán tanto de **entrada** como de **salida**, tal y como se muestra en el diagrama a continuación.

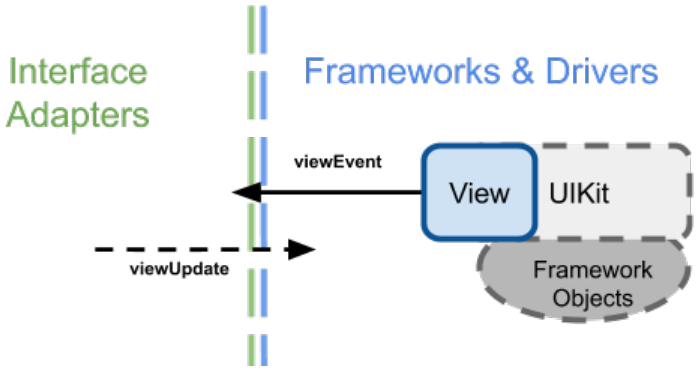

Figura 5.2: Entradas y salidas de la **View**

En el diagrama se puede observar que la relación de entrada es en forma de mensajes, denominado **viewUpdate** el mensaje de entrada, mientras que el de salida será denominado **viewEvent**.

5.3.1. Salida de ViewEvents

Los mensajes de tipo **viewEvent** serán la representación de un evento realizado por el usuario, junto a los valores asociados a ese evento, como podría ser pulsar un botón tras haber introducido un texto.

5.3.2. ViewEventsHandler Interface

Para la correcta interpretación de un **viewEvent**, se define una interfaz denominada ***ViewEventsHandler*** la cual será implementada por el componente receptor del evento, que en esta arquitectura será el **Presenter**, quedando la arquitectura tal y como se muestra en el diagrama.

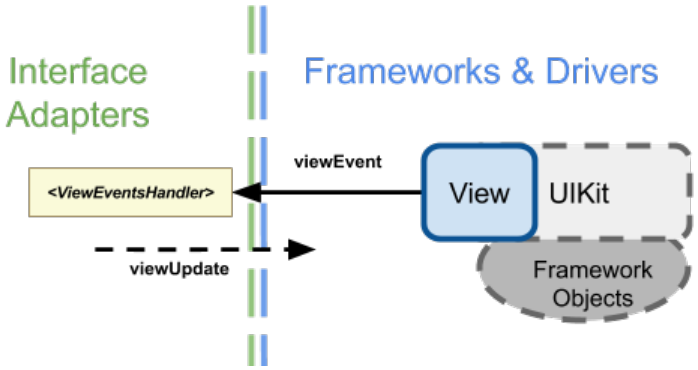

Figura 5.3: ***ViewEventsHandler*** Interface

5.3.3. Entrada de ViewUpdates

Los mensajes de tipo **viewUpdate** serán la representación del envío de parámetros y acciones de actualización de la **View** por parte del componente **Presenter**, como podría ser el nombre y la imagen de perfil de un usuario, o la acción de mostrar un indicador de progreso en la **View**.

5.3.4. ViewUpdatesHandler Interface

Para la correcta interpretación de un update, se define una interfaz denominada ***ViewUpdatesHandler*** y será implementada por la **View**, por lo que la **View** cumplirá con el **Open/Closed Principle**, ya que su funcionalidad se extenderá al incorporar la implementación de la interfaz, pero su funcionalidad cómo **View** no será modificada, así como

también cumplirá con el **Dependency inversion principle**, ya que la funcionalidad dependerá de la interfaz y no de la implementación de la **View**, quedando la arquitectura tal y como se muestra en el diagrama.

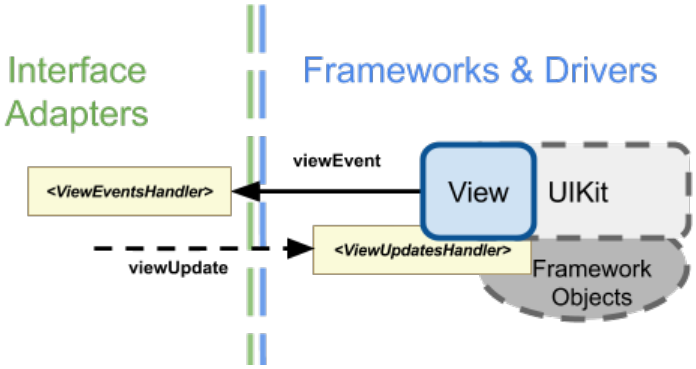

Figura 5.4: ***ViewUpdatesHandler*** Interface

Puesto que el modelo de arquitectura aplica los patrones de **Alta cohesión** y **Bajo acoplamiento**, y que el objetivo es desacoplar la detección de eventos del tratamiento de los mismos, aparece el elemento en la arquitectura ya citado, denominado **Presenter**.

5.4. Presenter

El **Presenter** pertenece al **área de Interface adapters**, ya que es un componente intermedio entre el área de **Frameworks y Drivers** y el área de **Business Rules**, respetando el patrón de **Indirección**.

La **responsabilidad** del **Presenter** consiste en interpretar un event enviado por la **View** y decidir si debe transmitirse hacia la lógica de negocio o bien hacia la lógica de navegación, cumpliendo con el **Single responsibility principle**.

Las **relaciones** del **Presenter** serán tanto de **entrada** como de **salida**, tal y como se muestra en el diagrama a continuación.

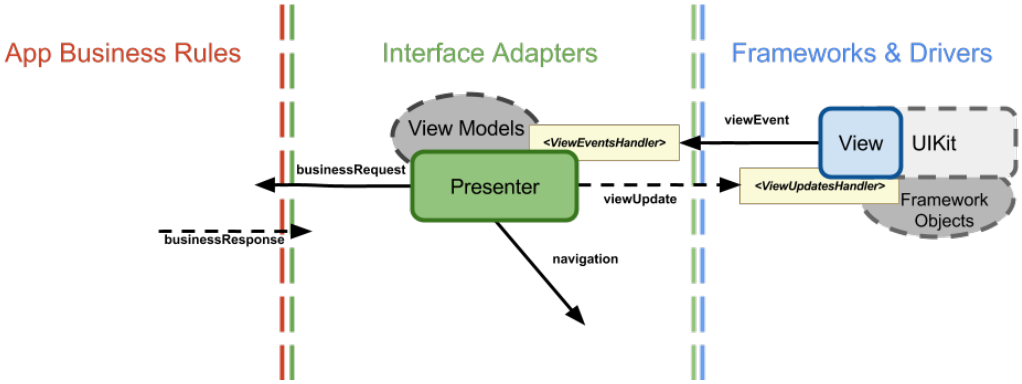

Figura 5.5: **Presenter**, entradas y salidas

En el diagrama se puede observar que existen dos relaciones de entrada, en forma de mensajes denominados **viewEvent** y **businessReponse** y tres de salida, denominados **viewUpdate**, **navigation** y **businessRequest**.

5.4.1. Objetos de tipo ViewModel

Un objeto **View Model** es un tipo de objeto que contendrá los datos planos necesarios sin lógica alguna para poder ser enviado desde un componente **Presenter** hacia un componente **View** y su implementación suele ser una estructura o clase. Este tipo de objeto puede permitir el uso de programación reactiva, y actualizar los datos del componente **View** directamente al editar los valores del objeto **ViewModel** en el componente **Presenter**, pero en este Trabajo no se ha realizado esa implementación por falta de tiempo, por lo que se comenta brevemente en las conclusiones del trabajo.

5.4.2. Entrada de ViewEvents

Los mensajes de tipo **viewEvent recibidos** provenientes de la **View** ya se detallaron en la documentación de la **View** por lo que ya son conocidos, y el elemento **Presenter** implementará la interfaz ***EventHandler*** descrita previamente en la **View.**
Además está relacionada con los mensajes de salida **businessRequest** que van dirigidos hacia el área de **Business Rules**, y a los mensajes **navigation**, que van dirigidos hacia otro componente del área de **Interface Adapters**.

5.4.3. Salida de ViewNavigations

Los mensajes de tipo **navigation enviados** serán la representación de una acción de navegación hacia un nuevo caso de uso junto a los parámetros necesarios para realizar la correcta presentación, como podría ser la acción de presentar una pantalla de detalle del perfil del usuario junto a los datos necesarios para indicar de que usuario se trata.

5.4.4. NavigationsHandler Interface

Para la correcta interpretación de un mensaje de tipo navigation, se define una interfaz denominada ***NavigationsHandler*** y será implementada por el componente receptor del mensaje navigation, que en esta arquitectura será el **Router**, quedando la arquitectura tal y como se muestra en el diagrama.

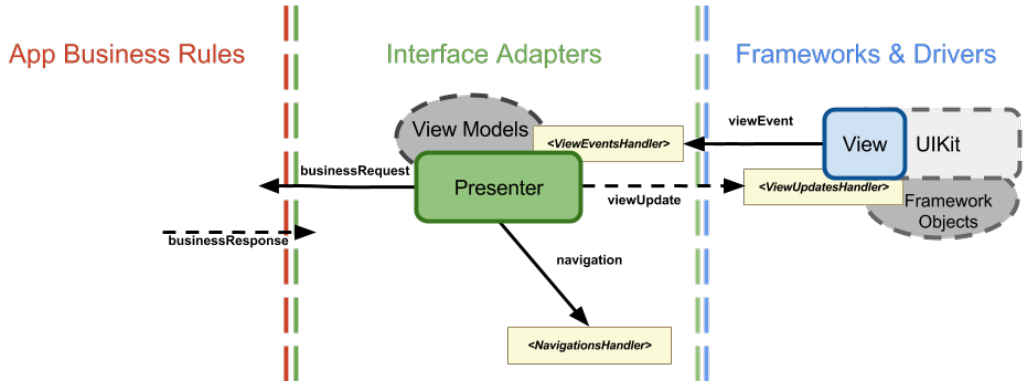

Figura 5.6: ***NavigationsHandler*** Interface

5.4.5. Salida de BusinessRequests

Los mensajes de tipo **businessRequest enviados** serán la representación de la solicitud de datos y de parámetros realizada por el **Presenter** hacia la lógica de negocio, como podría ser los datos del perfil de un usuario, o la verificación de si ya existe un usuario registrado en la aplicación.

5.4.6. RequestsHandler Interface

Para la correcta interpretación de una request, se define una interfaz denominada ***RequestsHandler*** y será implementada por el componente receptor de la request, que en esta arquitectura será el **Interactor**, quedando la arquitectura tal y como se muestra en el diagrama.

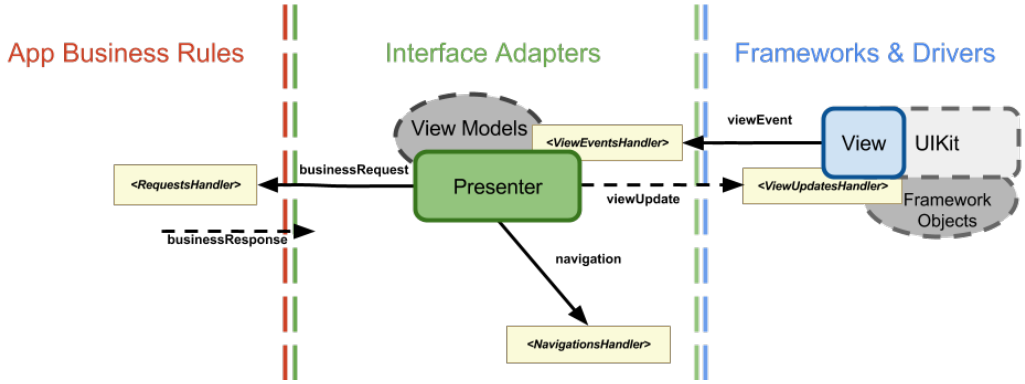

Figura 5.7: ***RequestsHandler*** Interface

5.4.7. Salida de ViewNavigations

Los mensajes recibidos de tipo **businessResponse** serán la representación del envío de parámetros y notificaciones del estado de una solicitud de datos realizada previamente por el **Presenter** a la lógica de negocio, como podría ser el nombre y la imagen de perfil de un usuario, o la acción de mostrar un indicador de progreso en la **View**.

5.4.8. ResponsesHandler Interface

Para la correcta interpretación de una response, se define una interfaz denominada ***ResponsesHandler*** y será implementada por el **Presenter**, por lo que el Presenter cumplirá con el **Open/Closed Principle**, ya que su funcionalidad se extenderá al incorporar la implementación de la interfaz, pero su funcionalidad cómo **Presenter** no será modificada, así como también cumplirá con el **Dependency inversion principle**, ya que la funcionalidad dependerá de la interfaz y no de la implementación del **Presenter**, quedando la arquitectura tal y como se muestra en el diagrama.

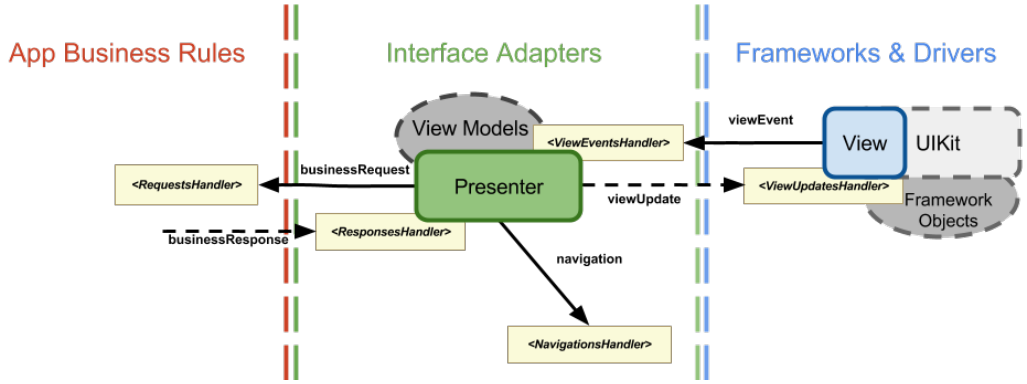

Figura 5.8: ***ResponsesHandler*** Interface

5.4.9. Salida de viewUpdates

Finalmente los mensajes de tipo **viewUpdate** enviados hacia la **View** ya se detallaron en la documentación de la **View**, y el elemento **Presenter** será el responsable de enviar dichos updates.

Continuando con el uso de los patrones de **Alta cohesión** y **Bajo acoplamiento**, con el fin de desacoplar la interpretación de los events recibidos desde la **View** de la lógica a aplicar respecto a ellos, aparecen los dos elementos ya citados, denominados **Interactor** y **Router**.

5.5. Router

El **Router** pertenece al **área de Interface adapters**, ya que es un componente intermedio entre dos casos de uso a visualizar, respetando el patrón de **Indirección**.

La **responsabilidad** del **Router** consiste en realizar la lógica de navegación pertinente, así como la lógica de animación si fuera requerida, para presentar un nuevo caso de uso cuando se reciba un evento de navegación por parte del **Presenter**.

Las **relaciones** del **Router** serán tan solo de **entrada**, tal y como se muestra en el diagrama a continuación.

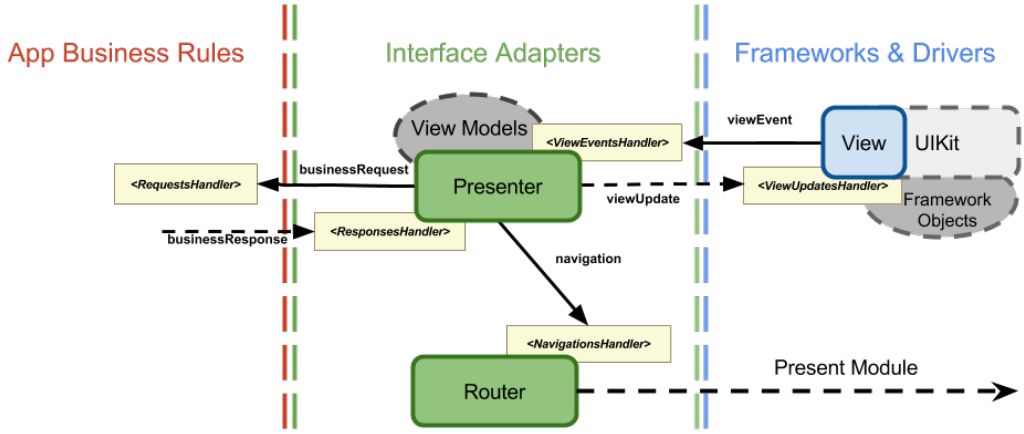

Figura 5.9: **Router**

5.5.1. Entrada de viewNavigations

Los mensajes recibidos denominados **navigation** ya se detallaron en la documentación del **Presenter**, por lo que el elemento **Router** implementará la interfaz ***NavigationsHandler*** descrita previamente en el **Presenter**.

Veamos ahora el otro componente con el que comunicará el **Presenter**, residente en el área de **Business Rules**, denominado **Interactor**.

5.6. Interactor

El **Interactor** pertenece al **área de Business Rules**, ya que su implementación está completamente ligada a la lógica de negocio y debe ser independiente del resto de áreas, respetando el patrón de **Indirección**.

La **responsabilidad** del **Interactor** consiste en realizar la lógica de negocio requerida por el **Presenter** y responder con los resultados al mismo.

Las **relaciones** del **Interactor** serán tanto de **entrada** como de **salida**, tal y como se muestra en el diagrama a continuación.

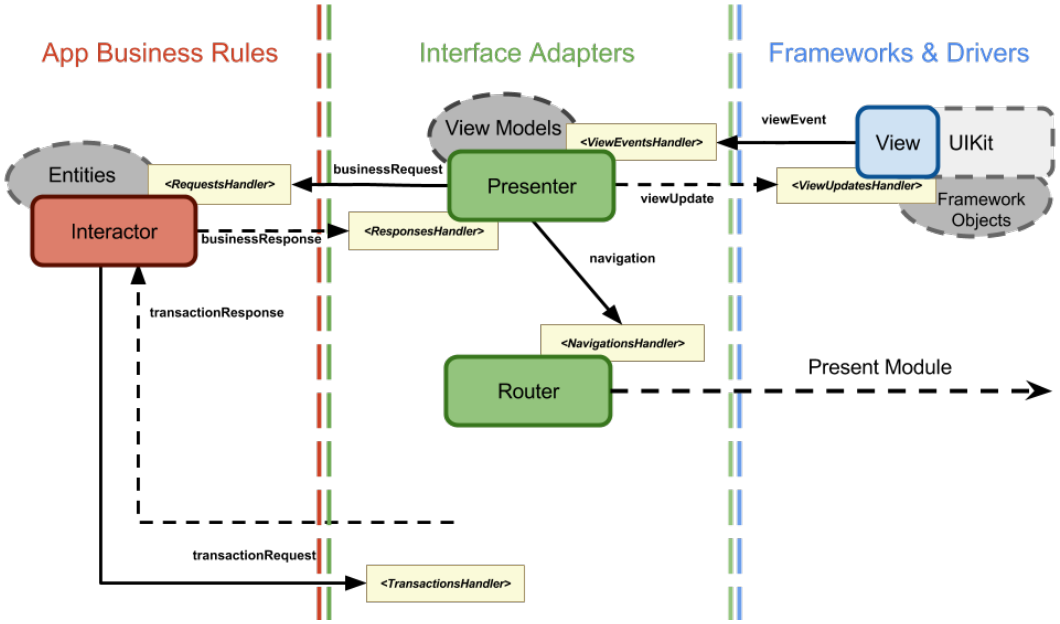

Figura 5.10: **Interactor**

En el diagrama se puede observar que existen dos relaciones de entrada, en forma de mensajes denominados **businessRequest** y **transactionResponse** y dos de salida, denominados **businessResponse** y **transactionRequest**.

5.6.1. Entrada de businessRequests

Los mensajes de tipo **businessRequest recibidos** provenientes del **Presenter** ya se detallaron en la documentación del presenter por lo que ya son conocidos, y el elemento Interactor implementará la interfaz **RequestsHandler** descrita previamente en el presenter.

Además está relacionada con los mensajes de salida **businessResponse** que van dirigidos hacia el **presenter**, y a los mensajes **transactionRequest**, que van dirigidos hacia otro componente del área de **Interface Adapters** denominado **Repository**.

5.6.2. Objetos de tipo Entity

El **Interactor** hará uso de los objetos denominados **Entities**, descritos en el apartado de **Clean Architecture** con los cuales podrá realizar el envío de los mensajes response con los datos apropiados. Los objetos de tipo **Entity** son la representación de las entidades de negocio y para crearlos es posible que sea necesario aplicar lógica de negocio, por lo que su creación únicamente es responsabilidad del **Interactor**.

5.6.3. Salida de businessResponses

Los mensajes de tipo **businessResponse** enviados hacia el **Presenter** ya se detallaron en la documentación de la **Presenter**, y el elemento **Interactor** será el responsable de enviar dichas responses cuando disponga de los datos solicitados por una **request**.

5.6.4. Salida de transactionRequest

Los mensajes de tipo **transactionRequest enviados** serán la representación de la solicitud de datos y de parámetros realizada por el **Interactor** hacia un componente residente en el área de **Interface adapters**, denominado **Repository**, como podría ser una query a base de datos, una petición de red, u obtener el valor asociado a un iBeacon.

Para la correcta interpretación de una **transactionRequest**, se define una interfaz denominada ***TransactionsHandler***, y será implementada por el componente receptor del mensaje transaction, que en esta arquitectura será el **Repository**, descrito más adelante.

5.6.5. Entrada de transactionResponse

Finalmente, los mensajes de tipo ***transactionResponse recibidos***, serán la representación de los datos proporcionados por un componente externo al **Interactor**, que en esta arquitectura será el **Repository**.

Estos **transactionResponse** serán objetos de tipo **DTO**, por lo que el **Interactor** deberá transformarlos a objetos de tipo **Entity** aplicando la lógica de negocio pertinente para ello.

Continuando con el uso de los patrones de **Alta cohesión** y **Bajo acoplamiento**, con el fin de desacoplar la interpretación de las transactions recibidos desde el **Interactor** se requiere del componente denominado **Repository**.

5.7. Repository

El **Repository** pertenece al **área de Interface Adapters**, ya que es un componente intermedio entre el área de **Business Rules** y **Frameworks y Drivers**, respetando el patrón de **Indirección**.

La **responsabilidad** del **Repository** consiste en completar una **transaction** solicitada por el **Interactor** contactando con el elemento **Provider** para recuperar los datos solicitados de una fuente externa ajena a la implementación, cumpliendo con el **Single responsibility principle**.

Las **relaciones** del **Repository** serán tanto de **entrada** como de **salida**, tal y como se muestra en el diagrama a continuación.

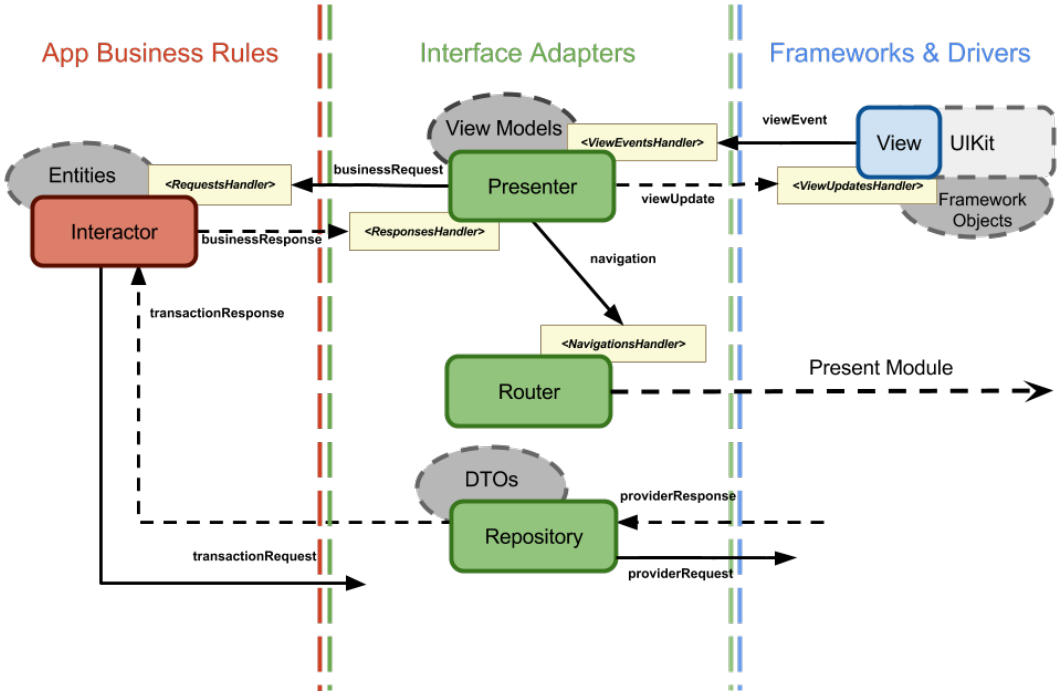

Figura 5.11: **Repository**

En el diagrama se puede observar que existen dos relaciones de entrada, en forma de mensajes denominados **transactionRequest** y **providerResponse** y dos de salida, denominados **transactionResponse** y **providerRequest**.

5.7.1. Objetos de tipo DTO

Los **objetos DTO** [17] no tienen más comportamiento que almacenar y entregar sus propios datos, siendo estos objetos simples que no deben contener lógica de negocio, y serán creados por el **Repository** cuando este interprete los datos recibidos desde un **Provider**.

5.7.2. Entrada de transactionRequest

Los mensajes de tipo **transactionRequest recibidos** provenientes del **Interactor** ya se detallaron en la documentación del **Interactor** por lo que ya son conocidos, y el elemento **Repository** implementará la interfaz ***TransactionsHandler*** descrita previamente en el **Interactor**.

Por lo que el **Repository** cumplirá con el **Open/Closed Principle**, ya que su funcionalidad se extenderá al incorporar la implementación de la interfaz, pero su funcionalidad cómo **Repository** no será modificada, así como también cumplirá con el **Dependency**

inversion principle, ya que la funcionalidad dependerá de la interfaz y no de la implementación del propio **Repository**, quedando la arquitectura tal y como se muestra en el diagrama.

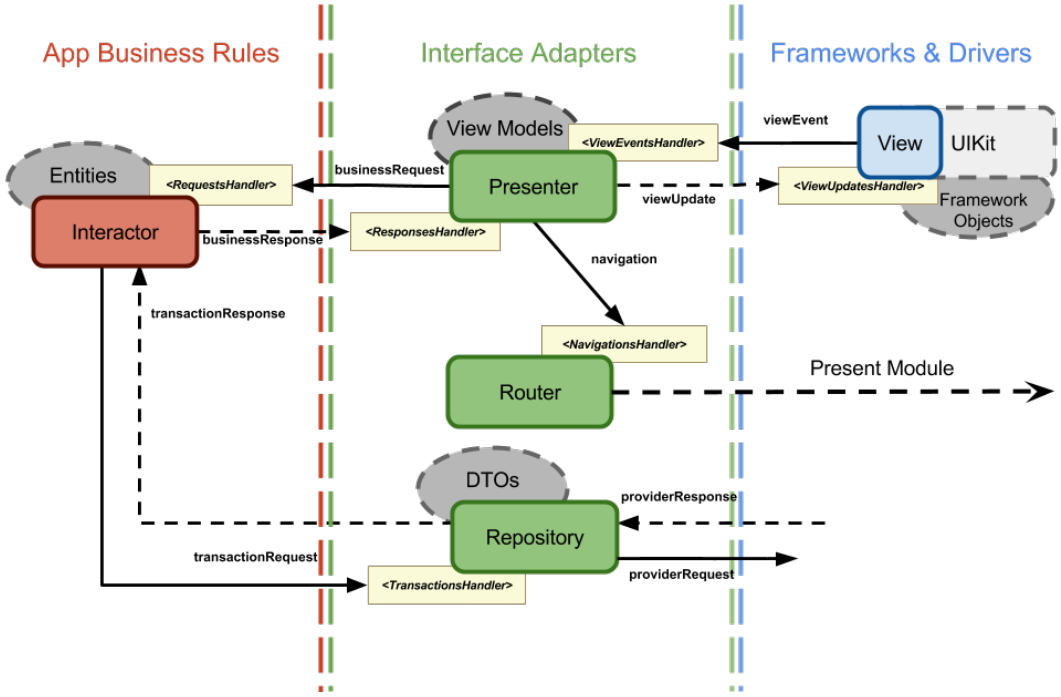

Figura 5.12: **Interfaz *TransactionsHandler***

Además está relacionada con los mensajes de salida **providerRequest** que van dirigidos hacia el área de **Frameworks y Drivers**.

5.7.3. Entrada de providerRequest

Los mensajes de tipo **providerRequest enviados** serán la representación de los datos asociados a una **transactionRequest** a enviar por el **Repository** hacia un componente residente en el área de **Frameworks y Drivers**, denominado **Provider** que se encargará de interpretarlos para realizar la obtención de los datos requeridos.

5.7.4. Mensajes entrantes de tipo providerResponse

Los mensajes recibidos de tipo **providerResponse** serán la representación de los datos recibidos desde el componente **Provider** después de realizar una **providerRequest**.

5.7.5. Entrada de transactionResponse

Los mensajes enviados de tipo **transactionResponse** serán la representación de los datos que el **repository** recibirá de una **providerResponse** asociada a una **transactionRequest realizada** previamente por el **Interactor**.

Continuando con el uso de los patrones de **Alta cohesión y Bajo acoplamiento**, con el fin de desacoplar la interpretación de las transactions recibidos desde el **interactor**, aparece el último componente de la arquitectura, denominado **Provider**.

5.8. Provider

El **Provider** pertenece al área de **Frameworks y Drivers**, ya que su implementación está completamente ligada al framework que utiliza **DataSource** asociado a él para servir como fuente de datos.

La responsabilidad del **Provider** consiste en la facilitación de datos y la persistencia de los mismos a partir de un **DataSource** externo al cual el propio **Provider** accede, cumpliendo con el **Single responsibility principle**.

Las relaciones del **Provider** serán tanto de entrada como de salida, tal y como se muestra en el diagrama a continuación.

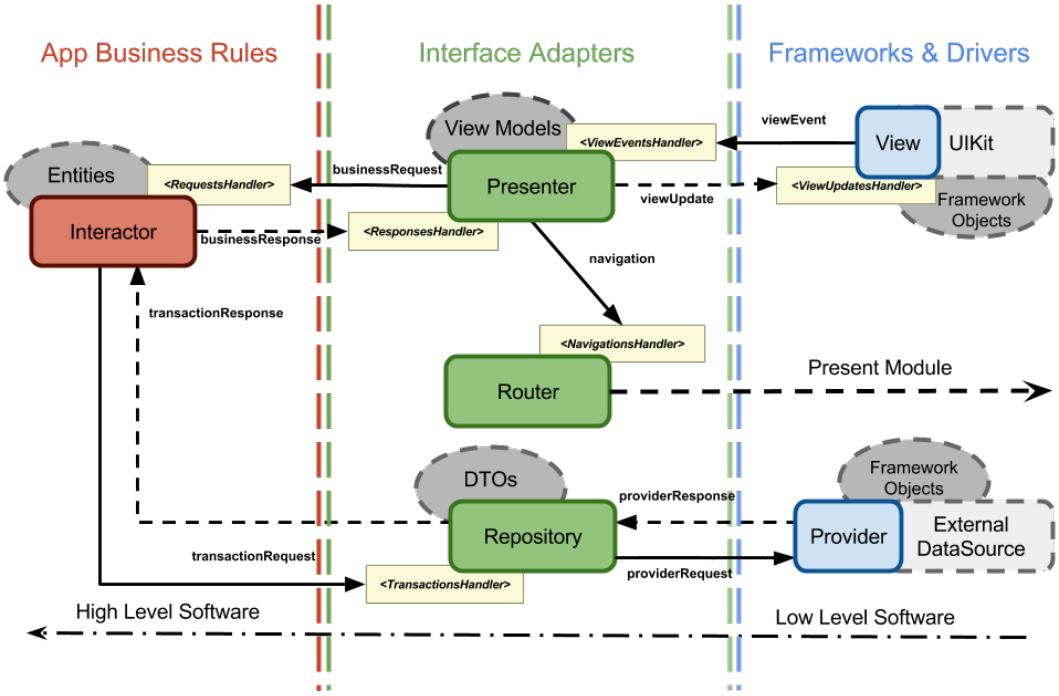

Figura 5.13: **Provider**

En el diagrama se puede observar que existe una relación de entrada, en forma de mensaje denominado **providerRequest** y uno de salida denominado **providerResponse**.

5.8.1. Mensajes entrantes de tipo providerRequest

Los mensajes de tipo **providerRequest recibidos** serán la representación de los datos asociados a una **transactionRequest** enviada por el **Repository** hacia el **Provider**, el cual se encargará de interpretarlos para realizar una correcta transformación directamente relacionada con el framework del **Datasource** que este pueda interpretar el contenido a recuperar o guardar.

5.8.2. Entrada de providerResponse

Los mensajes enviados de tipo **providerResponse** serán la representación de los datos recibidos desde el componente externo **DataSource** después de realizar una **providerRequest**, recibirlos en la implementación que proporcione el framework del DataSource y transformarlos debidamente para poder ser interpretados por el **Repository receptor**.

5.9. External DataSource

El **DataSource** pertenece al área de **Frameworks y Drivers**, ya que su implementación está completamente ligada al framework que utiliza para ser implementado.

La responsabilidad de la **DataSource** consiste en la facilitación de datos y la persistencia de los mismos directamente relacionada con el framework con el cual esté implementado, cumpliendo con el **Single responsibility principle**.

Pese a que las relaciones de entrada y salida no están plasmadas en el diagrama final de la arquitectura, su envío y recepción de mensajes son los mismos que los del componente **Provider**, ya que este último no es más que un configurador del propio **DataSource** para poder ser utilizado en el proyecto, así como un intérprete del tipo de datos utilizado por el **DataSource** ligado al **Framework**.

Capítulo 6

Aplicación del Modelo de arquitectura

6.1. Proyecto desarrollado

El proyecto de ejemplo que se ha desarrollado aplicando la arquitectura documentada en este trabajo pretende verificar la utilidad de la misma, por lo que se han implementado tres casos de uso que dan pie a las situaciones del desarrollo más interesantes de cara a la aplicación de la propia arquitectura.

Los **casos de uso desarrollados** son tres, los cuales se enumeran y describen brevemente a continuación:

- Realizar un **login mediante LinkedIn** [18], en el cual el usuario podrá loguearse accediendo a su cuenta de linkedIn, con el fin de recuperar su usuario, obteniendo así los skills de interés para la aplicación.
- Mostrar **ofertas de la API de infoJobs** [19], relacionadas con el perfil del usuario obtenido.
- Configurar **iBeacons** [20] de forma que sea posible monitorizar su localización mediante Bluetooth con el fin de recibir notificaciones asociadas a dichos iBeacons.

El desarrollo se ha llevado a cabo en el lenguaje de programación **Swift 3.0**, y utilizando el IDE de Apple para desarrollo llamado **Xcode**.

6.2. El concepto Módulo

En este trabajo, se referencia como caso de uso a todo lo relacionado con una pantalla de la aplicación, junto a su funcionalidad, sus componentes y sus dependencias.

Para la implementación de un caso de uso, se ha definido el concepto Módulo, que no será más que la agrupación de todos los componentes de código del proyecto de software

relacionados con un caso de uso.

Por ello, un Módulo contendrá los componentes, View, Presenter, Router e Interactor, los cuales han sido descritos en el apartado de diseño y documentación del modelo de arquitectura de este trabajo.

Para su creación se han implementado unos templates que facilitan la creación de un módulo, creando las clases equivalentes a los componentes de la arquitectura mencionados, así como sus relaciones entre ellos.

Para establecer las relaciones entre ellos, se ha aplicado los patrones de diseño Experto en información junto al patrón Creador, dando pie a la aparición de un Componente a nivel de desarrollo denominado Builder, que será el encargado de instanciar todos los componentes de un Módulo y actualizar sus relaciones.

6.3. Templates para la automatización de la creación un Módulo

6.3.1. Builder

```
1
2  class __IDENTIFIER__Builder
3  {
4      static func build() -> UIViewController {
5          let viewController = __IDENTIFIER__ViewController(nibName:String.init(
       describing: __IDENTIFIER__ViewController.self), bundle: nil)
6          let presenter = __IDENTIFIER__Presenter()
7          let interactor = __IDENTIFIER__Interactor()
8          let wireframe = __IDENTIFIER__Wireframe()
9
10         viewController.presenter = presenter
11         presenter.viewController = viewController
12         presenter.interactor = interactor
13         presenter.wireframe = wireframe
14         interactor.presenter = presenter
15         wireframe.viewController = viewController
16
17         _ = viewController.view //force loading the view to load the outlets
18         return viewController
19     }
20 }
```

6.3.2. View

```
1
2  // MARK: - Protocol to be defined at ViewController
3  protocol __IDENTIFIER__ViewUpdatesHandler: class
4  {
5      //That part should be implemented with RxSwift.
6      //func updateSomeView()
7  }
8
9  // MARK: - ViewController Class must implement ViewModelHandler Protocol to handle
       ViewModel from Presenter
10 class __IDENTIFIER__ViewController: UIViewController,
       __IDENTIFIER__ViewUpdatesHandler
11 {
12     //MARK: relationships
13     var presenter: __IDENTIFIER__EventHandler!
14
15     var viewModel : __IDENTIFIER__ViewModel {
16         return presenter.viewModel
17     }
18
19     //MARK: View Lifecycle
20     override func viewDidLoad() {
21         super.viewDidLoad()
22         configureBindings()
23     }
24
25     func configureBindings() {
26         //Add the ViewModel bindings here ...
27     }
28
29     override func viewWillAppear(_ animated: Bool) {
30         super.viewWillAppear(animated)
31         presenter.handleViewWillAppear()
32     }
33
34     override func viewWillDisappear(_ animated: Bool) {
35         super.viewWillDisappear(animated)
36         presenter.handleViewWillDisappear()
37     }
38 }
```

6.3.3. Presenter

```
1
2  // MARK: - Protocol to be defined at Presenter
3  protocol __IDENTIFIER__EventHandler: class
4  {
5      var viewModel : __IDENTIFIER__ViewModel { get }
6
7      func handleViewWillAppear()
8      func handleViewWillDisappear()
9  }
10
11 // MARK: - Protocol to be defined at Presenter
12 protocol __IDENTIFIER__ResponseHandler: class
13 {
14     // func somethingRequestWillStart()
15     // func somethingRequestDidStart()
16     // func somethingRequestWillProgress()
17     // func somethingRequestDidProgress()
18     // func somethingRequestWillFinish()
19     // func somethingRequestDidFinish()
20 }
21
22 // MARK: - Presenter Class must implement Protocols to handle ViewController
       Events and Interactor Responses
23
24 class __IDENTIFIER__Presenter: __IDENTIFIER__EventHandler,
       __IDENTIFIER__ResponseHandler {
25
26     //MARK: relationships
27     weak var viewController : __IDENTIFIER__ViewUpdatesHandler?
28     var interactor : __IDENTIFIER__RequestHandler!
29     var wireframe : __IDENTIFIER__NavigationHandler!
30
31     var viewModel = __IDENTIFIER__ViewModel()
32
33     //MARK: EventsHandler Protocol
34     func handleViewWillAppear() {
35         //TODO:
36     }
37
38     func handleViewWillDisappear() {
39         //TODO:
40     }
41
42     //MARK: ResponseHandler Protocol
43
44     // func somethingRequestWillStart(){}
45     // func somethingRequestDidStart(){}
46     // func somethingRequestWillProgress(){}
47     // func somethingRequestDidProgress(){}
48     // func somethingRequestWillFinish(){}
49     // func somethingRequestDidFinish(){}
50
51 }
```

6.3.4. Router

```
1
2  // MARK: - Protocol to be defined at Wireframe
3  protocol __IDENTIFIER__NavigationHandler: class
4  {
5      // Include methods to present or dismiss
6  }
7
8  // MARK: - Wireframe Class must implement NavigationHandler Protocol to handle
       Presenter Navigation calls
9  class __IDENTIFIER__Wireframe: __IDENTIFIER__NavigationHandler
10 {
11     weak var viewController : __IDENTIFIER__ViewController?
12 }
```

6.3.5. Interactor

```
1
2  // MARK: - Protocol to be defined at Interactor
3  protocol __IDENTIFIER__RequestHandler: class
4  {
5      // func requestSomething()
6      // func requestUser(id: String)
7  }
8
9
10 // MARK: - Presenter Class must implement RequestHandler Protocol to handle
       Presenter Requests
11 class __IDENTIFIER__Interactor: __IDENTIFIER__RequestHandler
12 {
13     //MARK: Relationships
14     weak var presenter : __IDENTIFIER__ResponseHandler?
15
16     //MARK: RequestHandler Protocol
17     //func requestSomething(){}
18 }
```

6.4. Módulos implementados

6.4.1. LaunchScreen Module

LaunchScreen Module es el punto de entrada a la app. En este módulo se comprueba si el usuario había iniciado sesión en LinkedIn anteriormente para de esta manera, presentarle una sección de la app u otra.

Esto se consigue fácilmente gracias a la arquitectura diseñada, con el **Interactor** trabajando en tándem con el **Presenter** y **Wireframe**. En este primer caso de uso vemos un sencillo ejemplo de la interacción entre ellos.

Como podemos ver en el código que se detalla a continuación, cuando acaba de cargar la app el **Interactor** comprueba si el usuario había iniciado sesión previamente:

```
1
2    func requestLoginStatus() -> Void {
3        if((UserDefaults.standard.string(forKey: "profileName")) != nil){
4            presenter?.loginStatusRequestDidFinish(isUserLogged: true)
5        }else{
6            presenter?.loginStatusRequestDidFinish(isUserLogged: false)
7        }
8    }
```

De esta forma, envía al Presenter el mensaje ***loginStatusRequestDidFinish(isUserLogged: Bool)***, que a su vez, mediante el módulo **Wireframe**, presenta al usuario la sección adecuada: una pantalla para hacer un login o la pantalla de bienvenida:

```
1
2    func loginStatusRequestDidFinish(isUserLogged: Bool) -> Void{
3        if(!isUserLogged){
4            self.wireframe.presentSignInModule()
5        }else{
6            self.wireframe.presentWelcomeModule()
7        }
8    }
```

6.4.2. SignIn Module

En el módulo SiginInModule se implementa el caso de uso mediante el cual un usuario inicia sesión mediante LinkedIn, para ello se le presenta una interfaz como se puede ver en la siguiente captura de pantalla:

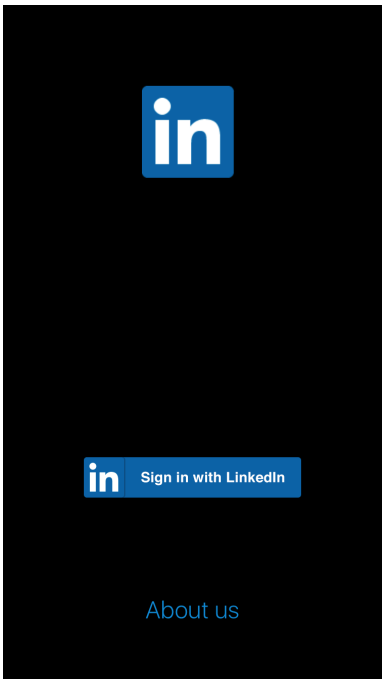

Figura 6.1: SignIn Module

Los detalles de implementación de este módulo se resumen en la interacción del usuario con la app mediante el botón 'Login with LinkedIn'. Al pulsar dicho botón, se dispara el evento en la view, que mediante el protocolo ***handleSignInButtonPressedEvent()***, envía al presenter para gestionar la acción, lo que desencadena la presentación de un nuevo módulo a través del **wireframe**: 'LoginWithLinkedIn' descrito en el apartado siguiente.

6.4.3. SignInWithLinkedIn Module

Este módulo interacciona con el servicio de LinkedIn a través de peticiones REST, las cuales se gestionan con la librería Moya. Esta librería, actuando como **Provider**, se encargaría de interactuar con los datos externos de la app.

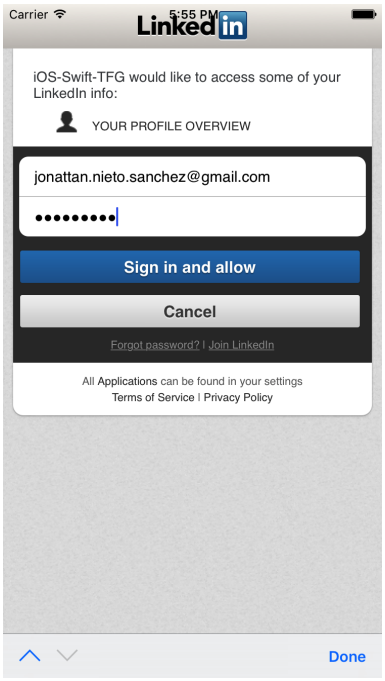

Figura 6.2: SignInWithLinkedIn Module

Este caso de uso es más complejo que el anterior ya que se deben realizar dos peticiones de red para realizar el login; primero se obtiene un authorizationCode, y con este código, se recupera el accessToken del usuario mediante OAUTH2. Este accessToken es necesario para, posteriormente, acceder a la información del usuario tales como su correo, nombre, skills, ...

Para ello, se ha definido los métodos adecuados en el **Interactor**: ***requestAuthorizationCodeURLRequest()*** y ***requestAccessTokenSending(authorizationCode:String)***. Por su parte la clase **Presenter**, cuando la view aparece en pantalla, desencadena la acción hacia el **Interactor**:

```
1    func handleViewWillAppear() {
2    // When the ViewController will appears, it sends a event to the Presenter,
3    // and the Presenter request the AuthorizationCodeURLRequest to the Interactor
4        self.interactor.requestAuthorizationCodeURLRequest()
5    }
```

y, por otra parte, cuando ha terminado la obtención del código, almacena el token en el dispositivo e invoca al **wireframe** para que presente el siguiente módulo.

```
1    func tokenRequestDidFinish(accessToken: String) {
2        UserDefaults.standard.set(accessToken, forKey: "LIAccessToken")
3        UserDefaults.standard.synchronize()
4
5        self.wireframe.presentWelcomeModule()
6    }
```

6.4.4. Welcome Module

En este módulo aparece la información del usuario y se le da la opción de buscar las ofertas de acuerdo a sus intereses:

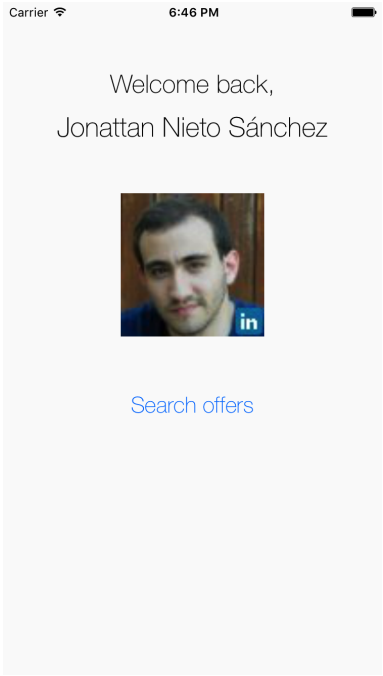

Figura 6.3: Welcome Module

De nuevo la librería Moya actúa como **Provider**, y se obtiene la información del servidor LinkedIn a través del **Interactor**:

```
func requestProfile(){

        if let accessToken = UserDefaults.standard.object(forKey: "LIAccessToken")
    {

            self.repository.fetchProfile(accessToken: accessToken as! String,
    entityReceivedClosure: { (profileResponseDTO) in
                switch(profileResponseDTO) {
                case let .success(profileDTO: ProfileDTO):
                    self.presenter?.profileRequestDidFinish(profileDTO: ProfileDTO
    )
                    break
                case .fail(profileFailDTO: _):

                    break

                }
            })
        }
    }
```

```
18
```

En el código anterior vemos como el **Interactor**, a través del protocolo ***requestProfile()***, obtiene el token de acceso del usuario. En caso de existir, pide al **Repository**, que en este caso estaría implementado mediante el **NetworkProvider**, los datos de usuario. La respuesta es, como hemos visto en el desarrollo de esta memoria, un objeto **DTO** que contiene los datos del usuario. Realmente, este DTO se ha definido a dos niveles: por una parte, tenemos el **ProfileResponseDTO** y por otra el **ProfileDTO**:

```
1
2 enum  ProfileResponseDTO {
3     case success(profileDTO: ProfileDTO)
4     case fail(profileFailDTO: ProfileFailDTO)
5 }
6
7
```

Como podemos ver en el fragmento de código superior, el **ProfileResponseDTO** es un objeto que encapsula dos DTOs a su vez: uno que contiene el profile del usuario en caso de que los datos se hayan podido recuperar correctamente y otro DTO que describe un fallo en caso de que haya algún problema para recuperar los datos.

Finalmente, aquí tenemos el ejemplo de **ProfileDTO** el cual incluye los mappings para construir el objeto

```
1  class ProfileDTO: Mappable {
2      var firstName: String?
3      var lastName: String?
4      var pictureURL: String?
5
6
7      required init?(map: Map) {
8
9      }
10
11     func mapping(map: Map) {
12         firstName <- map["firstName"]
13         lastName <- map["lastName"]
14         pictureURL <- map["pictureUrl"]
15     }
16 }
17
18
```

6.4.5. Offers Module

La sección Offers muestra el listado de ofertas relevantes para el usuario mediante una tabla.

Figura 6.4: Offers Module

Estos datos ya no se obtienen desde LinkedIn sino desde Infojobs, usando los skills que el usuario tuviera en su perfil de LinkedIn.

Gracias al diseño de la arquitectura es muy sencillo añadir una nueva API, simplemente se define un nuevo protocolo y target en el **NetworkRepository**. En el código a continuación vemos los dos primeros protocolos atacan a la API de LinkedIn y el ultimo a la de Infojobs.

```
1
2  protocol AccessTokenTransactions {
3      func fetchAccesTokenSending(authorizationCode: String,
       handleAccessTokenResponseDTO: @escaping (AccessTokenResponseDTO) -> () )
4  }
5
6  protocol ProfileRepository {
7      func fetchProfile(accessToken: String, entityReceivedClosure: @escaping (
       ProfileResponseDTO) -> () )
8  }
9
10 protocol OffersRepository {
11     func fetchOffers(query: String, province:String, handleOffersResponseDTO:
       @escaping (OffersResponseDTO) -> () )
12 }
```

6.4.6. BeaconsCoordinator

Por último, nos encontramos con la clase **BeaconsCoordinator**. Esta clase es una representación de un **Repository**; pero a diferencia del **NetworkRepository** que se encargaba de obtener los datos procedentes de servidores Web, el **BeaconsCoordinator** nos facilitará la tarea de gestión de los iBeacons.

Los iBeacons son unos sencillos dispositivos que emiten una señal Bluetooth. El caso de uso habitual es un colocar un iBeacon en un punto estratégico (una tienda, un museo, etc...), de tal manera que, cuando un usuario se aproxime a él y la app detecte su señal, puede reaccionar ante tal evento.
En el caso que nos ocupa, se ha decidido que cada beacon representará una oferta y el usuario recibirá una notificación en su móvil con los detalles de la misma al acercarse a ella. Para esto no es necesario que la app este ejecutándose, puede darse el caso de que se encuentre en background o completamente cerrada y el sistema operativo la ejecutará indicando el evento que la ha lanzado.

De tal manera que, al pulsar sobre la notificación, se nos llevará directamente a la pantalla de la descripción de la oferta:

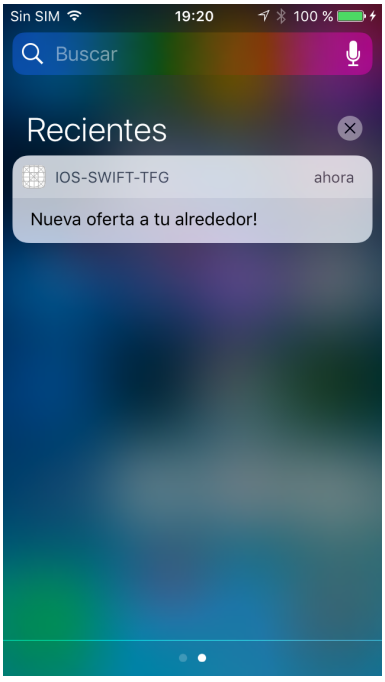

Figura 6.5: El usuario recibe una notificación en el móvil cuando está próximo a una oferta de trabajo, gracias al **BeaconsCoordinator**

Entrando en los detalles de su implementación, se ha definido el nuevo **repository** y se ha añadido en el **Interactor**:

```
var repository = NetworkRepository()
var beaconRepository: BeaconsCoordinator?
```

y también se ha definido un nuevo protocolo, que será el que invoque el **BeaconsCoordinator** cuando éste detecte un nuevo beacon en su rango:

```
protocol BeaconRepositoryHandler:class
{
    func handleBeaconRange(beaconID: String)
}
```

Cuando el **Interactor** reciba el evento ***handleBeaconRange***, delegará en el presenter la tarea de mostrar la información al usuario:

```
func handleBeaconRange(beaconID: String) {
        self.presenter?.beaconRepositoryDidRangeBeacon(beaconID: beaconID)
    }

```

Finalmente el **presenter** se encargará de gestionar toda la logística involucrada en la navegación de la app, ya que comentábamos anteriormente, la app podría encontrarse completamente cerrada o en background, en cuyo caso no partiría de cero sino que, una vez vista la oferta, continuaría en la sección en la que se encontraba navegando antes de cambiar de app.

Por ejemplo, en el siguiente fragmento de código de la clase **Presenter** vemos qué ocurre si el usuario se encontraba en una sección determinada cuando llega la notificación:

```
    func showOffer() {
        self.wireframe.presentOffer()
    }

```

6.5. Estructura Lógica del Proyecto

Una parte importante en el desarrollo de un proyecto de software es la estructura lógica de ficheros que se emplea, ya que aporta claridad a la hora de visualizar los ficheros del proyecto.

Es aconsejable que la estructura lógica del proyecto esté directamente relacionada con la estructura del modelo de arquitectura que se aplica al mismo.

Lamentablemente, **Clean Architecture** no define una estructura para organizar los ficheros, ya que de seguir la distribución de áreas, se darían casos extraños, como por

ejemplo que la BBDD y la UI estarían en el mismo grupo o carpeta lógica.

Por lo que en este trabajo se ha optado por una distribución en función de los casos de uso directamente relacionada con los diferentes componentes descritos en la documentación del modelo de arquitectura, como se puede observar a continuación.

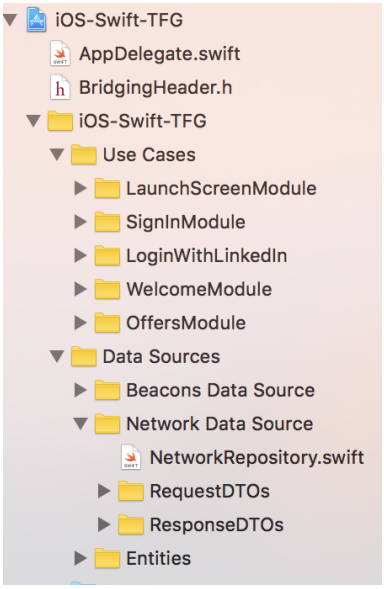

Figura 6.6: Estructura lógica de los ficheros del proyecto.

Capítulo 7

Conclusiones

En este trabajo de fin de grado se han alcanzado los objetivos propuestos, empezando por realizar un estudio del estado del arte de los modelos de arquitectura aplicados a iOS.

Se han recopilado, documentado y aplicado los patrones de diseño necesarios para garantizar la calidad del modelo de arquitectura propuesto.

Se ha diseñado y documentado un modelo de arquitectura para el desarrollo de software en iOS, y tras ello se ha aplicado en un proyecto de software garantizando la simplificación en el desarrollo, mantenimiento y ampliación del mismo.

Se han diseñado y desarrollado las plantillas que generan de forma automática el código equivalente a los componentes descritos en el modelo de arquitectura propuesto.

Se ha comprobado que el uso del modelo de arquitectura resultaba óptimo en cuanto al tiempo dedicado en el desarrollo y a la refactorización del código existente.

Como conclusión inicial respecto al objetivo principal del diseño de la arquitectura, he mejorado mi capacidad de análisis de diferentes modelos de arquitectura, he aprendido a ser crítico con las fuentes y he conseguido adaptar todo el conjunto de información relacionada con modelos de arquitectura ya existentes en el estado del arte de este trabajo. y arquitectura de software obtenidos durante estos años cursando el grado de Ingeniería informática, aplicando conceptos tales como el lenguaje UML para el correcto entendimiento de los modelos de arquitectura consultados.

Además he comprendido lo costoso que resulta documentar de forma técnica y a la vez comprensible para el entendimiento humano un conjunto de componentes interactuando entre sí, por lo que creo haber conseguido o como mínimo haber intentado plasmar todo mi conocimiento en la documentación del modelo de arquitectura, pero sé que queda abierto a posibles modificaciones, las cuales estaré encantado de recibir.

Finalmente me he iniciado definitivamente en la programación en el lenguaje Swift 3.0

ya que hasta la fecha no había dedicado mi vida profesional a este lenguaje, por lo que considero la experiencia de realizar este TFG muy positiva para mi persona.

Capítulo 8

Perspectivas de futuro

En cuanto al diseño de la arquitectura, se podrían integrar algunas ampliaciones en la gestión de la parte de User Interface como ya se comenta de pasada en la documentación del componente Presenter, cuando se especifica el uso de objetos ViewModels, y sería aplicando los paradigmas de la **programación reactiva** [21], y su aplicación práctica seria utilizar **RxSwift** [22] para la implementación de dicha parte en el proyecto de software.

También se podrían integrar ampliaciones en la gestión de dependencias de los componentes Repository, utilizando la librería **Swinject** [23] y aplicando el patrón de diseño de **Dependency Injection** [24].

Por otra parte, una ampliación del trabajo tanto a nivel de diseño y documentación como de aplicación, seria aplicar el modelo de arquitectura a la **metodología Test Driven Development** [25], para utilizar sus ciclos de desarrollo y la implementación de Tests, lo cual podría realizarse perfectamente en un Trabajo de Fin de Grado, ya que no es un concepto fácil ni trivial.

Finalmente, también se podría documentar casos más específicos relacionados con la implementación de ciertos casos de uso en los cuales se requiera la aplicación de algún componente especifico del framework de iOS y ver como encajan en el modelo de arquitectura, así como profundizar en la parte de user experience para ver cómo facilita la separación de la lógica de navegación en un único componente, que es el Router.

Índice de figuras

2.1. Model View Controller original. (Fuente: Las imágenes de esta sección están basadas en una adaptación de [1].) . . . 5
2.2. Model View Controller propuesto por Apple . . . 6
2.3. Model View Controller realmente propuesto por Apple . . . 7
2.4. Model View ViewModel . . . 8
2.5. Arquitectura tradicional de capas . . . 9
2.6. Onion Architecture . . . 10
2.7. The Clean Architecture . . . 11
2.8. Componentes de VIPER, Fuente: [10] . . . 14

5.1. Visión general del modelo de arquitectura . . . 26
5.2. Entradas y salidas de la **View** . . . 28
5.3. ***ViewEventsHandler*** Interface . . . 29
5.4. ***ViewUpdatesHandler*** Interface . . . 30
5.5. **Presenter**, entradas y salidas . . . 31
5.6. ***NavigationsHandler*** Interface . . . 32
5.7. ***RequestsHandler*** Interface . . . 33
5.8. ***ResponsesHandler*** Interface . . . 34
5.9. **Router** . . . 35
5.10. **Interactor** . . . 36
5.11. **Repository** . . . 38
5.12. **Interfaz *TransactionsHandler*** . . . 39
5.13. **Provider** . . . 41

6.1. SignIn Module . . . 49
6.2. SignInWithLinkedIn Module . . . 50
6.3. Welcome Module . . . 51
6.4. Offers Module . . . 53
6.5. El usuario recibe una notificación en el móvil cuando está próximo a una oferta de trabajo, gracias al **BeaconsCoordinator** . . . 54
6.6. Estructura lógica de los ficheros del proyecto. . . . 56

Bibliografía

[1] https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52

[2] https://developer.apple.com/documentation/uikit/uiviewcontroller

[3] https://www.raywenderlich.com/132662/mvc-in-ios-a-modern-approach

[4] https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/LoadingResources/CocoaNibs/CocoaNibs.html

[5] http://jeffreypalermo.com/blog/the-onion-architecture-part-1/

[6] http://jeffreypalermo.com/blog/the-onion-architecture-part-1/

[7] https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

[8] https://fr.wikipedia.org/wiki/BusinessObjects

[9] https://en.wikipedia.org/wiki/Dependency_inversion_principle

[10] https://www.objc.io/issues/13-architecture/viper

[11] https://es.wikipedia.org/wiki/Arquitectura_de_software

[12] https://es.wikipedia.org/wiki/Lenguaje_unificado_de_modelado

[13] https://sg.com.mx/revista/27/arquitectura-software#.WUq7RxPyi8U

[14] https://es.wikipedia.org/wiki/SOLID

[15] https://es.wikipedia.org/wiki/GRASP

[16] https://developer.apple.com/documentation/uikit

[17] https://es.wikipedia.org/wiki/Objeto_de_Transferencia_de_Datos_(DTO)

[18] https://developer.linkedin.com/docs/rest-api#

[19] https://developer.infojobs.net/documentation/quick-start/index.xhtml

[20] https://developer.apple.com/ibeacon

[21] https://en.wikipedia.org/wiki/Reactive_programming

[22] https://github.com/ReactiveX/RxSwift

[23] https://github.com/Swinject/Swinject

[24] https://en.wikipedia.org/wiki/Dependency_injection

[25] https://en.wikipedia.org/wiki/Test-driven_development