Universidad Nacional de Córdoba

Práctica Professional Supervisada de Ingeniería en Computación

Estudio, Selección e Implementación de un Patrón de Arquitectura de Software para el Desarrollo de Aplicaciones Android

Autor:
Esteban Andrés Morales
*Matrícula:*35.104.714

Tutor Docente:
Mg.Ing. Miguel Solinas
Supervisor:
Ing. Santiago Salamandri

Facultad de Ciencias Exáctas, Físicas y Naturales
Laboratorio de Arquitectura de Redes y Computadoras.

17 de octubre de 2018

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación y Objetivos

Como primera parte de la práctica profesional se propuso el estudio y el análisis de arquitecturas y patrones de diseño empleados en el desarrollo de sistemas de software orientados a multiplataformas (o sistemas operativos) y con una fuerte influencia de frameworks de desarrollo. Con esta tarea en mente se procedió a cumplir con las siguientes metas:

- Entnender el concepto de patron de arquitectura de software
- Entender el concepto de patrones de diseño de software.
- Investigar las arquitecturas más comunmente usadas en el desarrollo de aplicaciones nativas para dispositivos móviles.
- Entender los detalles de las opciones y tomar en consideración las dificultades más importantes de cada alternativa.
- Elegir una opción y justificar tal elección
- Introducir una posible alterativa de implementación de tal arquitectura y la correspondiente descripción de componentes.

En una segunda parte se analizará una implementación real del diseño expuesto.

1.2. Sobre los patrones en la etapa de diseño

El empleo de patrones de arquitectura así como patrones de diseño en el proceso de desarrollo de software permite estandarizar la abstracción del problema y modularizar la solución en componentes conocidos y de responsabilidad limitada. Definir una cantidad y variedad limitada de componentes favorece la reutilización de estructuras y tanto la implementación de nuevas funcionalidades así como la corrección de errores implica una ordenada cantidad de tareas que facilitan al mismo tiempo su apropiada planificación. Tanto la manipulación de datos como la interacción con sistemas vecinos puede incorporarse como partes del diseño contemplado siempre y cuando se respete el esquema de responsabilidades y las políticas de dependencias. Por lo general la utilización de patrones en la etapa de diseño ayuda a definir con precisión la distribución de responsabilidades entre distintas clases, incrementa la cohesión entre objetos, el encapsulamiento y la definición de contratos mediante la abstracción de interfaces. Adicionalmente la inclusión de diseño como la inversión de control deberían facilitar en gran medida el planteo, la implementación y la ejecución de pruebas unitarias, funcionales y de interfaz de usuario.

1.3. Porqué invertir en una arquitectura sólida

Una de las más comunes e incorrectas prácticas en el desarrollo de software es optar por utilizar un enfoque NAIVE (ingénuo) que permite ciertamente conseguir prototipos más veloces ya que subestima el esfuerzo de la etapa de diseño al inicio del proyecto. Sin embargo este tipo de prácticas, tarde o temprano terminará mostrando síntomas de su ejecición tales como demasiados objetos omnipotentes y omnipresentes (god objects) que resultan sumamente inflexibles, al mismo tiempo el arbol de dependencias presenta un nivel de complejidad tal que hace al proyecto propenso a errores ante el más mínimo refactoreo. En fin, a pesar de la efectividad del enfoque en conseguir resultados al corto plazo, en algún momento se convertirá en un caos total y la adición de nueva funcionalidad será muy difícil requiriendo de una necesaria una refactorización importante. Por lo antes expuesto nunca debe subestimarse la complejidad de una aplicación, sobre todo cuando la etapa de especificación fue realizada con poca rigurosidad. En ocaciones es bastante fácil comenzar rápidamente a hackear una aplicación preexistente, pero este

enfoque no escala con facilidad. La opción que suele no funcionar bien para proyectos duraderos es aquel donde primero se construye un producto mínimo viable y, a continuación, se agrega incrementalmente una gran cantidad de características adicionales al mismo. Justamente cuantas más funcionalides se agregan al sistema más tiempo lleva tal incorporación por lo que este enfoque ciertamente no es apto para el desarrollo de productos en el mundo ágil de desarrollo de software.

Exploraremos los aspectos comunes de la utilización, implementación y configuración del framework de desarrollo Android cuyo uso no es opcional al momento de crear aplicaciones para tales dispositivos. Probablemente no tiene nada que ver en particular con la lógica de negocios principal de la aplicación o proyecto, pero es la estructura necesaria que hará que el software pueda ser ejecutado.

- Backwards compatibility
- Cambios de configuración (orientación), estado de interfaz de usuario y subprocesos
- Almacenamiento local, sincronización de datos con API remota
- Entorno frágil (conexión a internet irregular, y otras APIs)
- Uso eficiente de recursos restringidos (memoria, ancho de banda, CPU)
- Interfaz de usuario atractiva, moderna y con material design validado con el departamento de UX y que tenga buen rendimiento

Administrar las consignas anteriores no es trivial. El propio SDK tampoco es evidente, así que no debe subestimarse. Hay diversos componentes, patrones de interacción con el SO y funcionalidades enlatadas, algunos de los cuales operan sólo con otros componentes particulares.

- Actividades, servicios, receptores de difusión (Broadcast Receivers), proveedores de contenido, paquetes, intents,etc.
- Fragmentos, vistas, notificaciones, recursos
- Bases de datos, IPC, Hilos, Almacenamiento

- Miles de APIs relacionadas con servicios y características específicas de hardware de dispositivos y periféricos
- Librerías de soporte y otras de terceros comunmente usadas

Y toda esta complejidad está ahí, incluso antes de empezar a implementar las funcionalidades que ésta aplicación en particular debe ofrecer. Así que el verdadero desafío consiste en averiguar dónde agregar el código que define lo que va a hacer nuestra aplicación.

- Ciclo de vida, estados guardados y restaurados (Bundles, Parcelables)
- Múltiples puntos de entrada (Intents, Notifications deep links)
- Navegación no trivial (back, up, backstack, procesos , banderas de intención)
- Vistas y Fragmentos y su intercomunicación (interfaces, callbacks, transacciones de fragmentos, inflación)
- Transiciones, elementos de héroe compartidos, diseños y recursos alternativos para dispositivos, orientación, etc.

Todas estas tareas implican un montón de código ya existente, que maneja un montón de asuntos de framework que nada tienen que ver con la lógica del negocio. La programación de toda la lógica de negocios mezclada con las tareas antes listadas hará que el código sea realmente ilegible, y probablemente dificultará en gran medida el seguimiento de lo que está pasando, debido a un conjunto pequeño de clases repletas de funcionalidades no categorizadas. Necesitamos una mejor separación de responsabilidades (separation of concerns) y un conjunto de patrones comunes, lo que impondrá algún orden en nuestro código. También debe proporcionar flexibilidad suficiente, hacer que el código sea facilmente verificable y reducir la duplicación de código y todas las otras malas prácticas de desarrollo de software, que normalmente brotan cuando uno escribe código apurado para conseguir un prototipo en cuestion de semanas.

Capítulo 2

Análisis de arquitecturas Clean, VIPER, Hexagonal

El objetivo principal de emplear una estructura fija para la implementación del proyecto es utilizar un único "lenguaje arquitectónico"familiar, tanto para el desarrollo de la aplicación en android, iOS o cualquier otra plataforma que pueda aparecer en el futuro, más bien durante la vida útil del producto. De esta manera no es necesario pagar un costo demasiado alto al incluir una implementación del mismo sistema para una plataforma distinta. Los desarrolladores de ambas plataformas podrán discutir aspectos de diseño, validar reglas de negocio y evacuar dudas sin tener en cuenta detalles de las platadormas, así mismo será más fácil conservar coherencia y mostrar armonía entre las implementaciones nativas para las plataformas.

2.1. Uncle Bob Clean Architecture

También conocida como arquitectura de capas (onion architecture). El punto principal de este enfoque es que la lógica de negocio, también conocida como dominio, está en el centro del universo (Al medio entre las entradas del sistema y las salidas).

2.2. Dominio Transparente

Cuando se listen los directorios de un proyecto que cumple con los lineamiento de esta arquitectura, con tan solo leer el nombre de las carpetas debería ser posible casi de inmediato tener una idea de qué se trata esta aplicación, independientemente de la tecnología. Todo lo demás es un *detalle de implementación.*

Por ejemplo, la persistencia es un detalle. Definir una interfaz con el objetivo de definir la responsabilidad con un contrato (Contrato de Persistencia), de esta forma uno podría implementar de una manera rápida e inefieciente una estrategia de persistencia en memeoria RAM y no pensar en ello demsiadpo sino hasta que la lógica de negocio esté completamente definida. Una vez definidos los requerimeintos de persistencia y verificados con el cliente, se puede proceder a tomar una decisión definitiva de **cómo** deberán persistirse los datos.

Almacenamiento en una base de datos local, comunicación remota con un servicio REST remoto, almacenamiento sobre el sistema de archivos, ante este panorama incluso sería razonable pensar el planteo de la creación de un esquema de cache, y sin embargo no es poco frecuente que luego de la elicitación de requerimientos resulte que el sistema no tiene que persistir ningun resultado en absoluto.

En una frase: *las capas internas contienen lógica de negocios, las capas externas contienen detalles de implementación.* Adicionalmente esta arquitectura debe cumplir con un conjunto de características:

- Regla de dependencia
- Abstracción
- Comunicación entre capas

2.2.1. Regla de Dependencias

La regla de dependencia se puede explicar mediante el siguiente diagrama:

Las capas externas deben depender de las capas internas. Esas tres flechas en el cuadro rojo representan dependencias. En lugar de "depende", tal vez sea mejor usar términos como "ve", "conoce" o "está consciente de...". En estos términos, las capas externas ven,

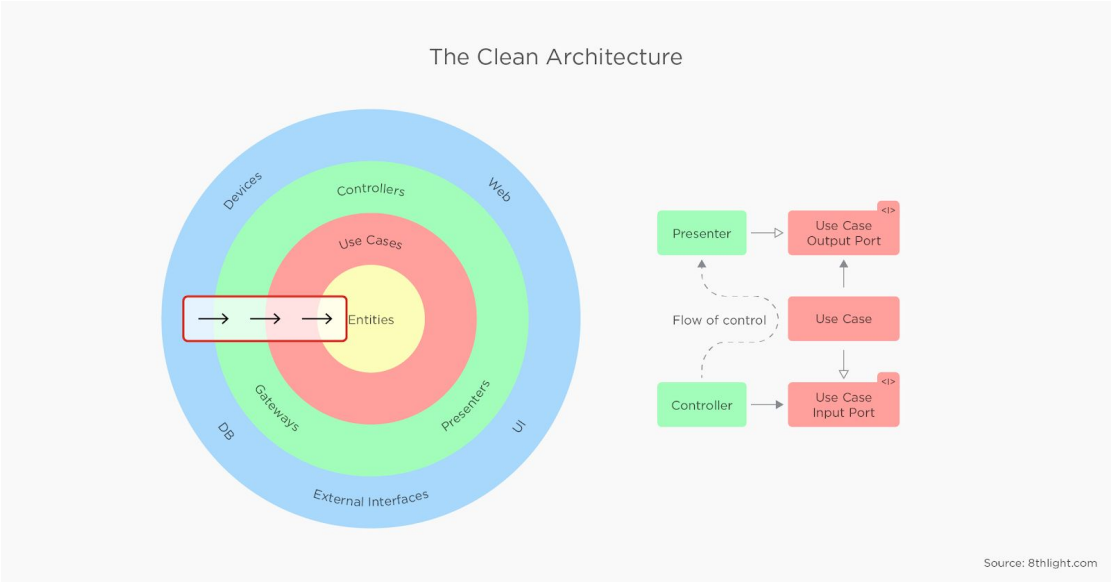

FIGURA 2.1: Esquema de dependencias para una arquitectura en capas.

conocen y son conscientes de las capas internas, pero las capas internas no ven ni conocen, ni son conscientes de, las capas externas. Como dijimos anteriormente, las capas internas contienen lógica de negocios y las capas externas contienen detalles de implementación. Combinado con la regla de dependencia, se deduce que la lógica de negocio no ve, ni conoce, detalles de implementación. Y eso es exactamente lo que estamos tratando de lograr.

No existe una única forma de implementar esta regla dependerá del encargado del proyecto. Una estrategia consiste en colocar las clases de cada capa en paquetes diferentes, poniendo especial cuidado en no importar paquetes "externos" en paquetes "internos". Sin embargo, si algún programador del equipo no es consciente del principio de dependencias, nada les impediría romperlo. Un mejor enfoque sería separar las capas en diferentes módulos de Android, por ejemplo, y ajustar las dependencias en el archivo de construcción para que la capa interna simplemente no pueda utilizar la capa externa, sin embargo este enfoque implica un exaustivo conocimiento de la herramineta de construccion de la plataforma para la que se está desarrollando.

2.2.2. Principio de Abstracción

El principio de la abstracción ya se ha insinuado antes. Dice que, a medida que se están moviendo hacia el centro del diagrama, las cosas se vuelven más abstractas. Eso tiene

sentido: como lo repetimos anteriormente el círculo interno contiene lógica de negocios y el círculo exterior contiene detalles de implementación.

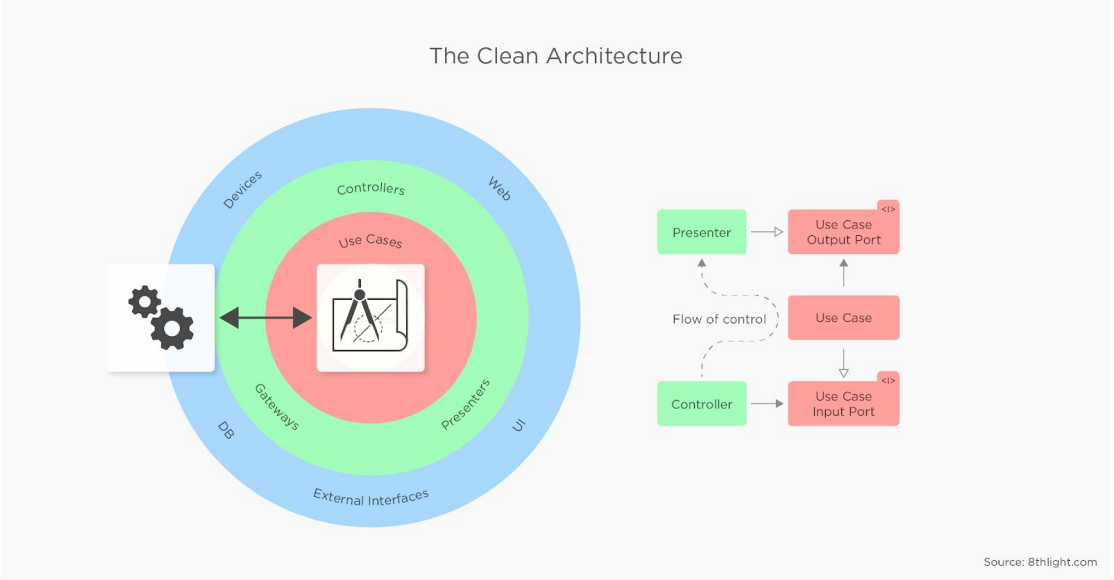

FIGURA 2.2: Principio de Abstracción en una arquitectura por capas.

Incluso puede plantearse el mismo componente lógico dividido entre varias capas, como se muestra en el diagrama. La parte más abstracta se puede definir en la capa interna, y la parte más concreta en la capa externa.

De esta manera, la lógica de negocios podría producir como un efecto secundario que se muestren notificaciones del sistema por ejemplo, pero no sabe nada acerca de los detalles de la implementación (cómo se implementan las notificaciones para una plataforma dada) . Además, la lógica empresarial ni siquiera sabe que existen detalles de implementación. Por lo tanto la regla de las dependencias se conserva.

2.2.3. Comunicación entre Capas

La lógica del negocio está en el medio y debe mediar entre los sitemas externos de salida y los sistemas externos de entrada como la interfaz de usuario, pero ni siquiera sabe que esos dos tipos existen. Esta es una cuestión de comunicación y flujo de datos. Necesitamos que los datos sean capaces de fluir de las capas externas a las internas y viceversa, pero la regla de dependencia no lo permite.

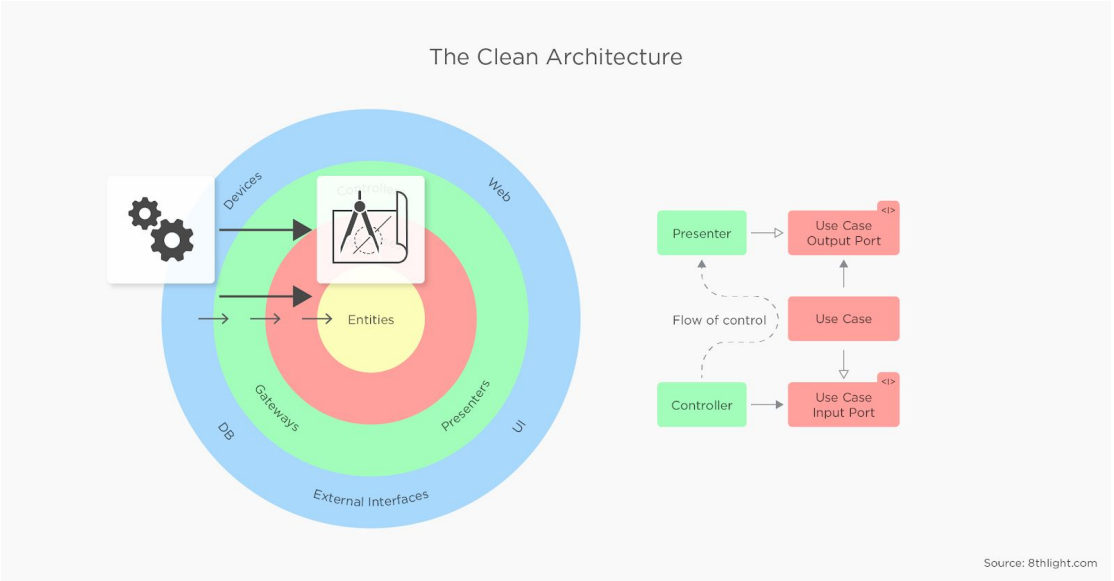

FIGURA 2.3: Principio de Abstracción en una arquitectura por capas.

Sólo tenemos dos capas, la verde y la roja. El verde es exterior y sabe sobre el rojo, y el rojo es interior y sólo se conoce a sí mismo. Necesitamos que los datos fluyan desde el verde al rojo y viceversa. La solución ya se ha insinuado antes y se muestra en el siguiente diagrama:

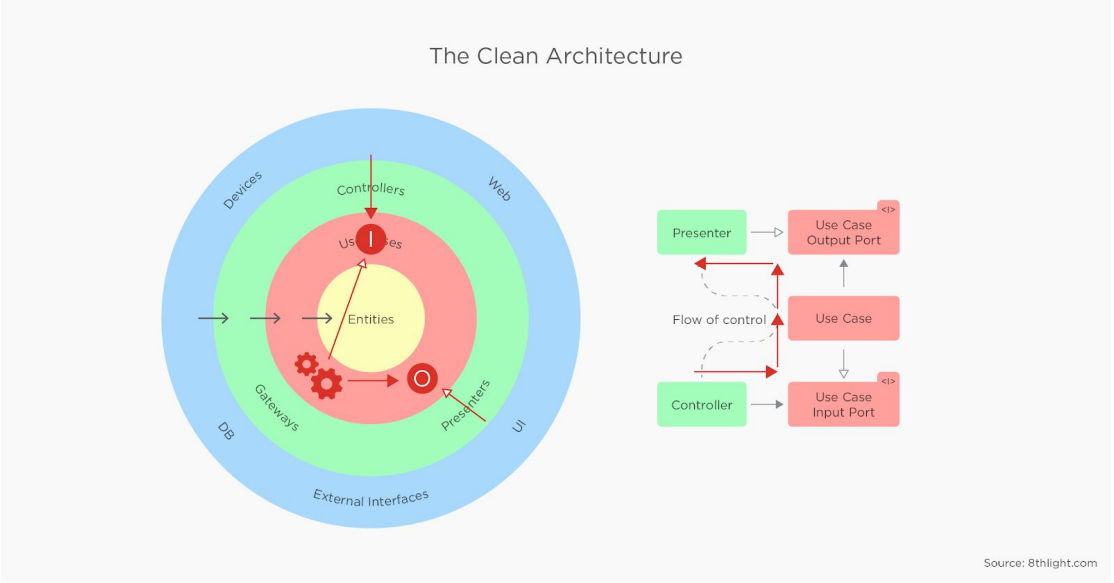

FIGURA 2.4: Comunicación entre capas.

La parte del diagrama en la parte inferior derecha muestra el flujo de datos. Los datos van desde el controlador, a través del puerto de entrada del caso de uso (o reemplazar

el caso de uso con el componente de su elección), luego a través del propio caso de uso y después a través del puerto de salida del caso de uso al presentador.

El controlador tiene un puerto de entrada, literalmente tiene una referencia a él. Llama a un método en él, de modo que los datos van del controlador al puerto de entrada. Pero el puerto de entrada es una interfaz, y la implementación real es el caso de uso: por lo que ha llamado un método en un caso de uso y los flujos de datos al caso de uso. El caso de uso hace algo y quiere enviar los datos de vuelta. Tiene una referencia al puerto de salida, ya que el puerto de salida está definido en la misma capa, por lo que puede llamar al método en él. Por lo tanto, los datos van al puerto de salida. Y finalmente, el presentador es, o implementa, el puerto de salida.

2.3. Arquitectura Hexagonal

La arquitectura hexagonal comparte el enfoque de división de capas y utiliza los principios que la anterior sin embargo coloca a la DB en el centro del control del flujo de datos por lo que la diferencia real entre ambas es meramente de implementación y consideración.

2.4. Arquitectura VIPER para iOS

VIPER significa Views, Interactors, Presenters, Entities and Routing. La combinación de todos estos componentes vive dentro del llamado Módulo. La principal motivación detrás de esta arquitectura es proporcionar una solución a un problema en iOS conocido como Massive View Controllers. La arquitectura MVC tradicional utilizada para desarrollar la aplicación iOS simplemente no proporciona suficiente modularidad y separación de responsabilidades. Las partes View y Model permanecen más o menos limpias, pero toda la complejidad termina en ViewControllers, que tienen que manejar demasiadas cosas. Más de 1000 líneas de código no es poco común para un ViewController relativamente rico en funciones.

Nuevamente se trata de una arquitectura separada por capas con separación de responsabilidades. En el caso de esta arquitectura en particular el único aspecto que cabe la pena hacer énfasis es en el esquema de comunicación que tiene lugar entre el Presentador

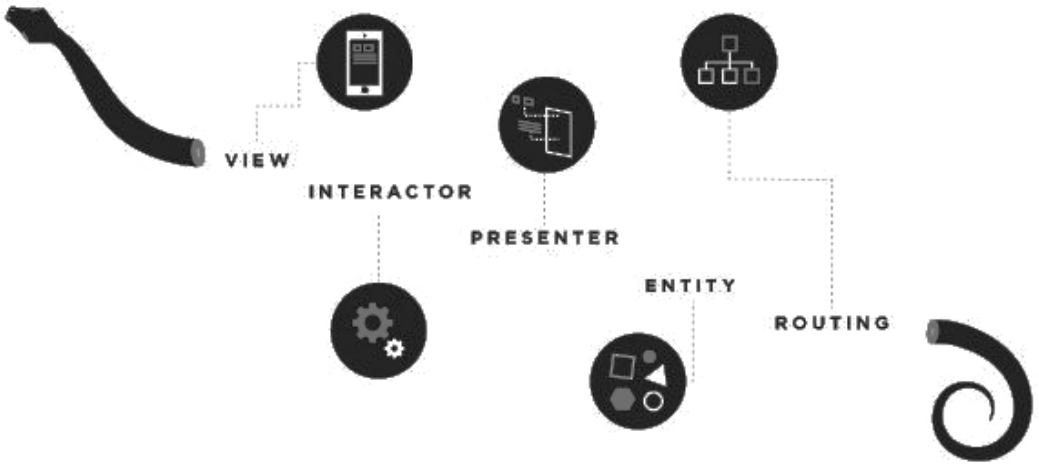

FIGURA 2.5: Esquema arquitectura VIPER.

y el Interactor. El Interactor trabaja con las Entidades y nunca las pasa al Presentador. En su lugar, los llamados objetos DisplayData se construyen a partir de las Entidades. Estos objetos contienen sólo los datos requeridos por el presentador que se mostrarán en la vista.

Hay un par de aspectos a los que uno debe prestar especial cuidado al utilizar la arquitectura VIPER en iOS. Lo primero es que uno tiene que instanciar todos los componentes con sus dependencias y hacer todo el cableado entre ellos en un módulo. Esto puede dar la impresión de que se está escribiendo demasiado del mismo código, cada vez que se crea un nuevo módulo, pero no hay forma de evitarlo. Es posible automatizar un poco este proceso escribiendo fragmentos de código personalizado y scripts de generación de código, pero muchas veces los resultados son demasiado generales para el módulo específico que estamos construyendo actualmente.

Otra aspecto difícil de manejar viene como resultado del cableado manual mencionado anteriormente - referencias cíclicas fuertes. Puesto que algunos de los componentes necesitan sus referencias cíclicas para trabajar, por ejemplo Presenter ¡-¿Interactor, es posible generar memory leaks con gran facilidad si no se presta atención al momento de comunicar los objetos, cuáles referencias pueden ser fuertes y cuáles deben ser débiles. Esta tarea se vuelve más complicada cuanto más componentes y módulos hay en la aplicación. Es posible que se produzcan fugas adicionales de memoria ocultas cuando se usen características de lenguaje como los closures de Swift, así que uno debe ser muy

cuidadoso al llamar a los métodos de Presenter desde un closure localizado dentro del Interactor.

Por último, esta estricta separación de las responsabilidades a veces conduce a tener Presenters muy simples, que sólo reorientar algunos flujos de datos de la Vista a los Interactores. Esto sucede cuando la arquitectura se aplica a unas pantallas relativamente simples, que no tienen muchas cosas que el Presentador pueda hacer.

2.4.1. VIPER sobre Android

La implementación de VIPER sobre android es impráctica y desbeneficiosa.

Hay algunas diferencias significativas en la forma en que funcionan los SDK en iOS y Android. Android tiene Actividades, que son más o menos el equivalente de ViewControllers en iOS. Una gran diferencia es que uno no puede instanciar estos objetos. Uno puede pedirle al framework para iniciar una Actividad y luego puede anular sus métodos de ciclo de vida para hacer cualquier trabajo necesario. Por lo tanto, al implementar módulos VIPER, uno no puede realizar todo el cableado en una clase Wireframe cuando se instancia la actividad. En su lugar, todo el cableado del módulo VIPER inicial tiene que tener lugar cuando una Actividad es inicializada por el propio Framework. El SDK de Android también proporciona Fragmentos, que de hecho pueden ser instanciados por el programador, pero hay tanta magia relacionada con la creación de Fragmentos que el framework hace automáticamente, que hacen muy dificil controlar el ciclo de vida de tales Fragmentos. Por lo tanto, No considero el uso de Fragmentos como las vistas de VIPER e instanciándolas explícitamente en un Wireframe como una solución estable. Luchar contra el framework es siempre una mala idea.

Otra cosa específica en Android son los cambios de configuración, que normalmente destruyen toda la interfaz de usuario y la recrean. Eso incluye Actividades, Fragmentos y Vistas. Eso puede resultar catastrófico, porque todo el Módulo VIPER será eliminado por el recolector de basura (garbage colector) y recreado de nuevo. Así que uno tiene que asegurarse de que el estado de los componentes del módulo se guarda de alguna manera. Además, hay que tener cuidado con los objetos de larga vida y las operaciones de mantenimiento de las referencias a los componentes del módulo, ya que podría causar grandes fugas de memoria. Un ejemplo típico es una operación de petición de red, que

tiene una referencia fuerte al Interactor del Módulo como su devolución de llamada. Dado que el Interactor tiene referencia al presentador y mantiene referencia a la vista, que contiene referencia a todas las cosas de la interfaz de usuario, esto podría dar lugar a una pérdida de memoria grande. Así que uno tiene que cuidar de limpiar tales referencias cuando un Módulo entero está siendo destruido y recreado.

Por estas razones se decidió descartar el uso de VIPER en el desarrollo de las apps como conseción y optar por implementar el enfoque de la Clean Architecture.

Capítulo 3

Clean Architecture

3.1. Modelos de Implementación Clean Architecture

Los mejores intentos de implementación de esta arquitectura vienen de la mano de un desarrollador argentino Fernando Cejas y los ejemplos de los Blueprints de Google.

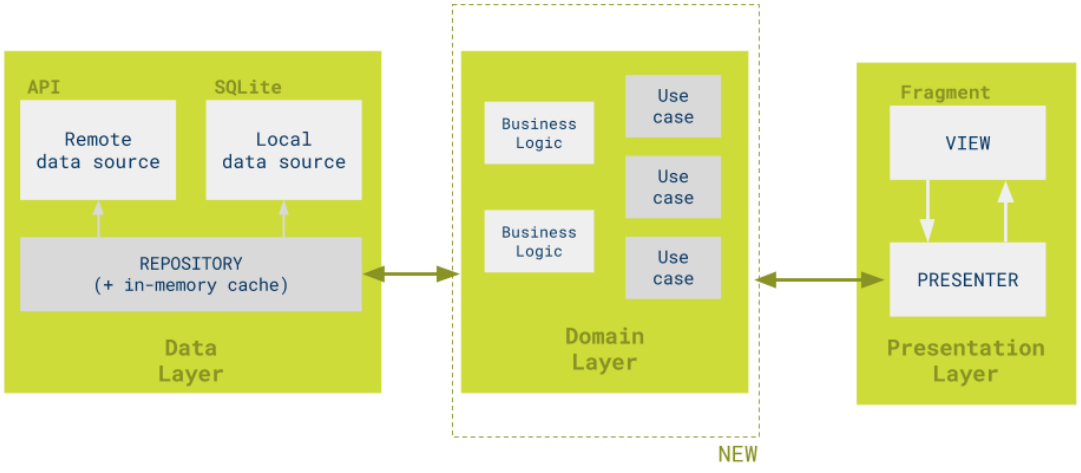

FIGURA 3.1: Esquema de dependencias para una arquitectura en capas.

Ambas implementaciones están compuestas de tres capas distintas:

- Presentation Layer: Esta capa se encarga de interactuar con la UI. Implementa un patrón de diseño conocido como **MVP (Model View Controller)**.
- Domain Layer: Esta capa contiene toda la lógica de negocio. La capa de dominio comienza con las clases denominadas casos de uso o interactores según la literatura,

utilizados por los presentadores de la aplicación. Estos casos de uso representan todas las acciones posibles que un desarrollador puede realizar desde la capa de presentación. Los casos de uso se implementaron utilizando el patrón de diseño conocido como **Commander**.

- Data Layer: Esta capa administra la adquisición de datos y es capaz de utilizar diferentes fuentes de datos. Esta capa se suele implementar utilizando el patrón de diseño conocido como **Repository**.

3.2. Presentation Layer: MVP

El patrón de arquitectura que se utiliza en la capa de presentación de ambas implementaciones se conoce como Modelo-Vista-Presentador. La idea detrás del patrón es concentrar la lógica de la interacción con el usuario en una entidad conocida como presentador, las operaciones directamente relacionadas con la manipulación de objetos gráficos y la captura de acciones de usuario están delegadas a la entidad Vista, finalmente la adquisición de datos y la ejecución de los algoritmos que encapsulan la lógica de negocio forman parte de las entidades modelo en el patrón. El diagrama de componentes que describe la relación entre las partes principales se puede observar a continuación

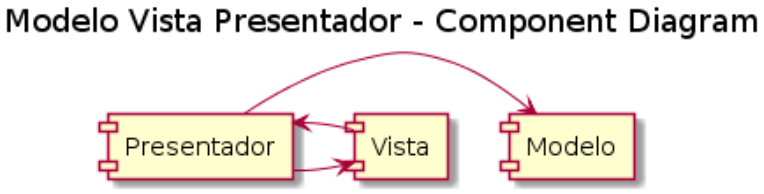

FIGURA 3.2: Diagrama de componentes del patrón.

Es posible deducir el esquema de comunicación entre los componentes a partir del diagrama. La vista se comunica de manera bidireccional con el presentador y cuando es necesario el presentador se comunica de manera unidireccional con el modelo.

Una convención para la implementación del patrón es tratar de generar vistas completamente ajenas de cualquier lógica operativa y agnósticas del estado de la aplicación. Esto las convierte en un mero instrumento de interfaz entre lo que percibe el usuario y sus reacciones.

Otra de las convenciones sugiere utilizar objetos modelo-vista en la comunicación entre el presentador y la vista para estandarizar el tipo de mensaje y el proceso de actualización de la vista.

En el caso de las implementaciones antes mencionadas la interface con el modelo es satisfecha mediante el uso de objetos casos de uso ó interactores, ambos términos suelen utilizarse de manera intercambiable.

3.2.1. Mapeo con Implementaciones

La capa de presentación tienen una Una Actividad Android es ejecutada por el proceso principal de la aplicación. La tarea principal de la actividad es inicializar los objetos del patrón: la instancia del fragmento que cumple con el contrato de la vista, la instancia del objeto presentador y las instancias de los casos de uso que serán ejecutados por el presentador.

- Vista → Fragmento:
- Presentador → Presenter
- Modelo → Casos de Uso

El proceso de creación de los objetos y la interacción básica inicial se ilustra en la siguiente diagrama:

3.3. Domain Layer: Commander Pattern

El patrón de diseño conocido como Commander se utiliza para abstraer la ejecución de procedimientos mediante la implementación de entidades comando. Estos objetos ejecutan un único algoritmo y encapsulan la lógica de negocio de la aplicación o sistema. Originalmente el diseño contempla 4 entidades principales que se pueden apreciar en el siguiente diagrama de clases:

1. Cliente: Este componente se encarga de crear las instancias de cada comando y distribuirlas entre los correspondientes invocadores.

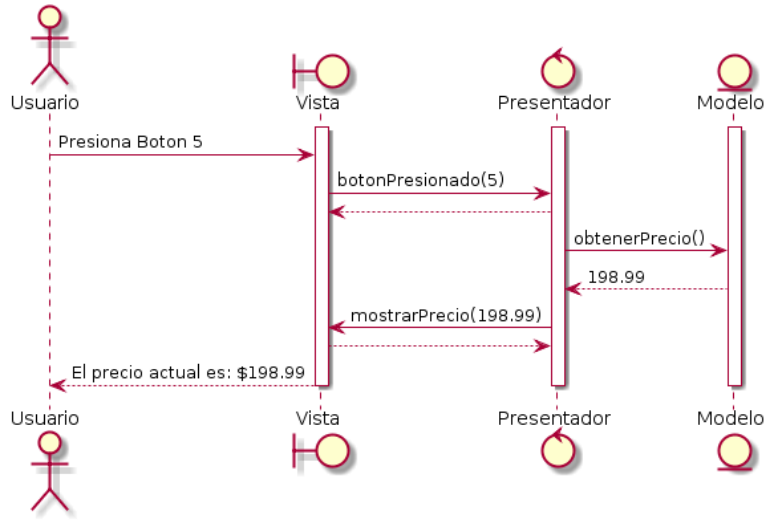

FIGURA 3.3: Diagrama de secuencia para una interacción con el usuario.

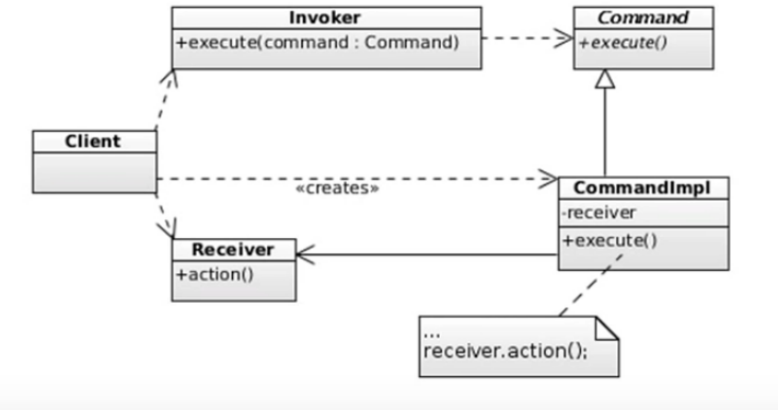

FIGURA 3.4: Diagrama de clases para el planteo inicial del patrón Commander.

2. Receptor: Es la entidad que se ve afectada por la ejecución de un comando. Puede ser compartida por varios comandos o bien un único comando puede interactuar con varios receptores en su ejecución.

3. Comando: Esta entidad contiene la implementación del algoritmo o lógica de ejecución.

4. Invocador: Se encarga de ejecutar instancias de comandos.

El enfoque inicial sugiere la implementación de un comando por cada una de las operaciones soportadas por el sistema o aplicación. Sin embargo en sistemas suficientemente

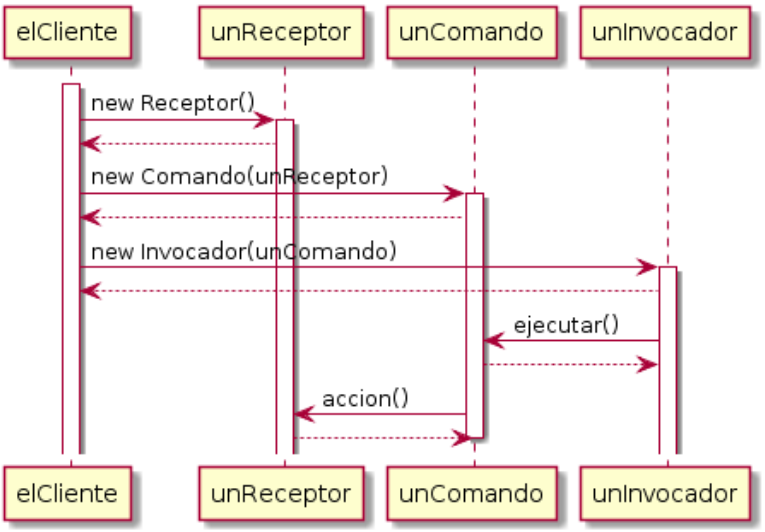

FIGURA 3.5: Diagrama de secuencia para el patrón Commander.

grandes la diversidad de funcionalidades soportadas es tan grande que el diseño propuesto se vuelve impráctico. Para mitigar este problema se suele implementar de manera adicional una modificación que permite la ejecución paramétrica de los comandos para reducir al máximo la cantidad de comandos implementados. Esta modificación permite diversas alternativas de implementación pero la más utilizada es incorporar conceptos del patrón Request-Response dónde el caso de uso se trata como una entidad de caja negra que admite Solicitudes y emite Respuestas estandarizadas para cada caso.

- Solicitud (Request): Un objeto que contiene el conjunto de parametros de entrada que deben ser satisfechos para poder realizar la ejecución de la rutina del comando.
- Respuesta (Response): Un objeto que contiene los valores que se obtuvieron de la ejecución del algoritmo del comando.

Por lo tanto puede inferirse el flujo de operación y ejecución de los comandos.

1. El cliente crea instancias de comandos y sus correspondientes invocadores.
2. El invocador crea e inicializa los objeto solicitud necesarios para ejecutar cada comando.
3. El invocado ejecuta los comandos llamando al método ”ejecutar” implementado por cada comando pasando como parámetro la solicitud previamente creada.

4. El invocador observa los resultados en espera activa implementando el patrón Observer o mediante algún esquema de callback.

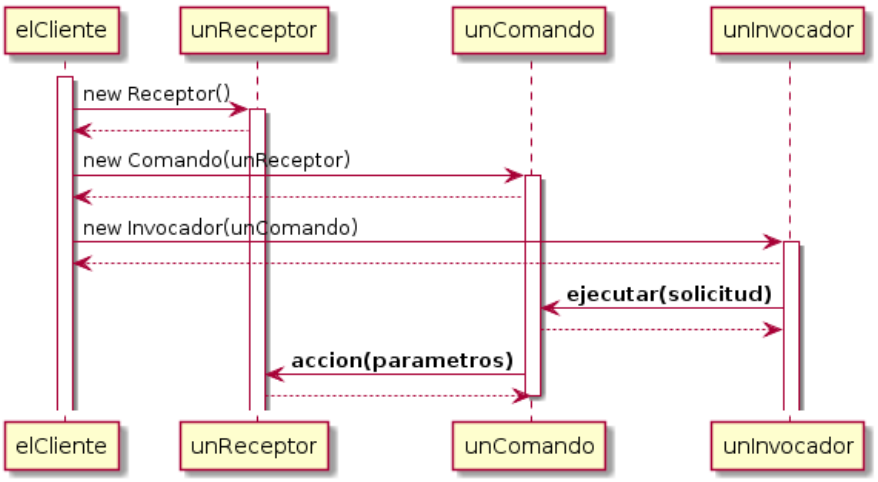

FIGURA 3.6: Diagrama de secuencia para el diseño revisado.

Siguiendo los lineamientos de la arquitectura propuesta los autores denominan a los comandos: Casos de Uso, ó Interactores.

En las implementaciones estudiadas se encontraron las correspondencias como se listan a continuación:

- Cliente → Actividad
- Receptor → Repositorio
- Comando → CasoDeUso
- Invocador → Presentador

Como una nota relevante de implementación se recomienda ejecutar las rutinas de los comandos en un hilo/proceso separado para para evitar bloquear el proceso principal de la apliación.

3.4. Data Layer: Repository Pattern

En la capa de datos se propone la implenentación de un patrón de diseño conocido como Repository(Repositorio). Originalmente se concive a este diseño como una forma

de estandarizar la implementación y el uso de los objetos DAO (Data Access Object) comunmente utilizados para mapear objetos entidad con las persistencias en la base de datos. Adicionalmente este patrón encapsula en la clase repositorio todos los metodos particulares de filtrado, procesamiento calculado y ordenamiento de entidades. Sin embargo se define una interfaz generica que deberá ser respetada por todas las implementaciones de repositorios para todo el sistema independientemente de la entidad que atienda.

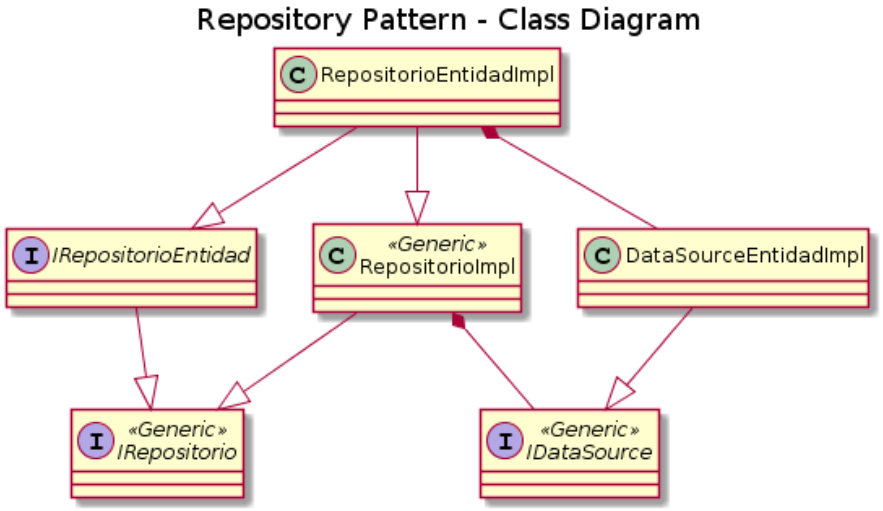

FIGURA 3.7: Diagrama de clases del patrón Repository.

Como puede apreciarse en el diagrama se definen:

- IRepositorio: es una interfaz genérica que establece el contrato básico que deben respetar todas las implementaciones de repositorios.
- RepositorioImpl: es una clase generica que establece la interacción con una fuente de datos genérica.
- IRepositorioEntidad: es la interfaz que *Especifica* la interfaz generica de repositorio y establece el contrato o métodos particulares que deberá cumplir la implementacion concreta de repositorio para esta Entidad en particular.
- RepositorioEntidadImpl: es la clase que *Especifica* la implementacion generica de repositorio e implementa los métodos particulares para esta Entidad en particular.

En una descripción más detallada del diagrama puede observarse que existen dos interfaces genéricas para acceso de datos IRepository y IDataSource, esto podría generar confusión y ser de alguna forma repetitivo.

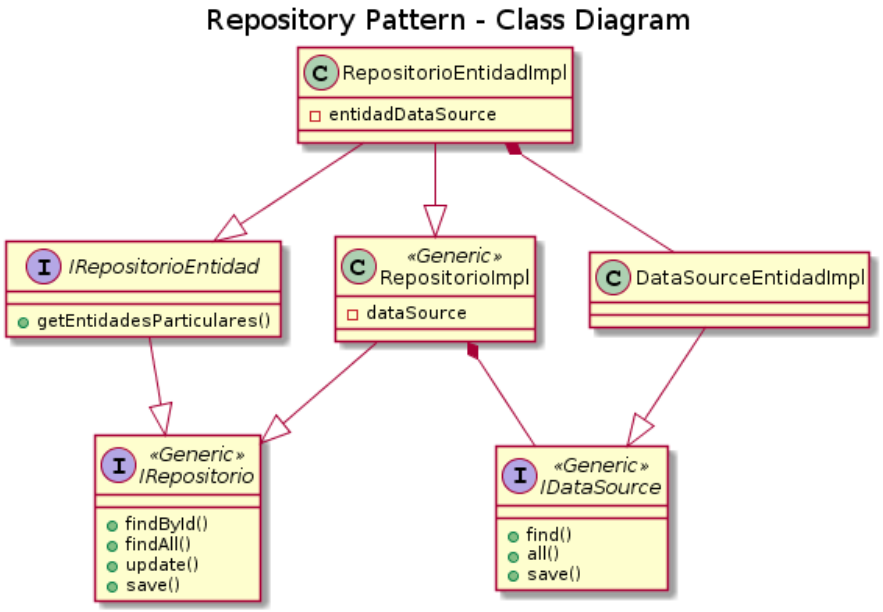

FIGURA 3.8: Diagrama de clases detallado del patrón Repository.

Por lo antes expuesto parece razonable plantear una fusión entre el concepto de repositorio y fuente de datos. Coloquialmente es trivial ya que un repositorio definitivamente es una fuente de datos. Si además se quita la estandarización por genéricos se consigue un diseño más flexible.

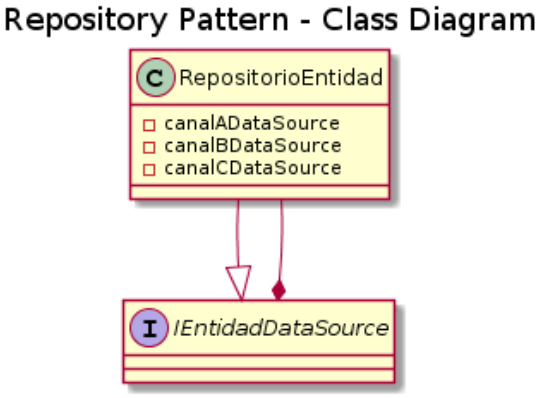

FIGURA 3.9: Diagrama de clases del patrón Repository modificado.

Principalmente orientado a encapsular la manipulación, selección, priorización y mantenimiento de diversas fuentes u orígenes de datos, este esquema de repositorios modificado

permite que el peticionario se comunique con una única interfaz para solicitar operaciones sobre datos permaneciendo agnóstico del orígen de datos sobre el cual tendrán impacto. Implementar una política de caching local se convienrte en una tarea secilla de implementar y mantener. Esta es la implementación del patrón que se observó en los codigos estudiados.

Capítulo 4

Conclusiones

Durante el desarrollo de la presente práctica se entendió las ventajas de implementar un sistema de software utilizando un patrón de arquitectura. La inspección de código escrito por profesionales del área introdujo conceptos de programación Java acanzados tales como el uso de clases anónimas y la implementación de clases y métodos genericos. Se pudo observar las teécnicas empleadas para realizar las pruebas sobre el código implementado y fue necesario invertir tiempo en la investigacion de las librerias y framework para pruebas tales como Mockito y Espress.

De manera indirecta se pudo observar cómo la definición de una arquitectura de estas caracterísitcas no solo tiene impacto en la organización y testabilidad del código sino también introduce un procedimeinto de trabajo tanto para la adición de nuevas funcionalides sino para la remoción de errores y la insepcción del código en general.

Como una desventaja notoria se menciona la empinada curva de aprendizaje para la inlcusión de nuevos miembros en un hipotético equipo de desarrollo. Así mismo se hizo evidente que todos los conceptos de abstracción que fueron introducidos se traducen en un aumento notable en la cantidad de lineas de código meramente dedicadas a mantener la estructura del diseño pero que no proveen una funcionalidad concreta al sistema.

Finalmente, se observaron inconsistencias entre el planteo teórico de la arquitectura y la implementación real del software mayormente por la dificultad técnica y conseciones en la que incurrieron los programadores para disminuir la verbosidad de algunos componentes o interacciones.

Bibliografía

[1] •. . [Accedido el].

[2] MS Windows NT kernel description. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: 2010-09-30.

[3] MultiMedia LLC. MS Windows NT kernel description, 1999. URL http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm.

[4] Nick Jenkins. *A Software Testing Primer: An Introduction to Software Testing.* 2008.

[5] PhD Doug Dahlby. *Applying Agile Methods to Embedded Systems Development.* http://web.archive.org/web/20080207010024, 2008. Accedido: 07/03/2016.