Monet mobile

18 de julio de 2011

Resumen

Desarrollo del entorno de movilidad de Monet, una plataforma para el desarrollo de sistemas de información basado en el desarrollo dirigido por modelos.

Autor: Rayco José Araña Rodríguez

Tutores: Mario Hernández Tejera y Jose Juan Hernández Cabrera

Índice

Parte I

Introducción

En la actualidad, la computación en movilidad ha experimentado un crecimiento exponencial. Los teléfonos inteligentes o Smartphone están tomando el protagonismo como medio de acceso a la sociedad de la información. Dispositivos cada vez más accesibles económicamente, tarifas de acceso a internet cada vez más baratas y unos dispositivos cada vez más potentes, orientan claramente este tipo de dispositivos como el futuro de la sociedad de la información y por tanto, el presente de cualquier organización que pretenda competir en el mercado de hoy en día.

Dada la cantidad de ecosistemas y dispositivos existentes en el mercado, los costes de desarrollo asociados a extender los sistemas de información de las organizaciones para soportar estos dispositivos pueden llegar a ser prohibitivos para cualquier organización de tamaño pequeño o mediano. Se hace necesario minimizar la cantidad de desarrollo necesario para dar soporte a estos dispositivos, si bien también existe la necesidad de minimizar el riesgo de apostar por un ecosistema concreto para tratar de minimizar costes, dado que el mundo de los teléfonos inteligentes es muy nuevo y cambiante.

La ingeniería dirigida por modelos, cuyo principio es el uso de modelos del dominio de la organización como base de construcción del software que de soporte al sistema de información, es la línea de investigación en la que se basa este trabajo. Partiendo de un software que siguiendo los principios de la ingeniería dirigida por modelos, permite la construcción de software de apoyo al sistema de información de una organización, se pretende iniciar el estudio de la adaptación de este tipo de soluciones a los teléfonos inteligentes de forma que con un mismo desarrollo, la organización tenga cubierto todas sus necesidades en el tratamiento de la información.

Este trabajo se realizará utilizando Monet, una plataforma que basada en el principio de la ingeniería dirigida por modelos, permite el desarrollo de modelos de sistemas de información y su ejecución en entornos web, tratando que sin necesidad de cambios o realizando cambios mínimos en un modelo que se ejecuta sobre Monet, sea posible su acceso a través de teléfonos inteligentes.

Parte II

Antecedentes y estado del arte

1. Ingeniería dirigida por modelos

La calidad en la gestión que las organizaciones de hoy necesitan para desarrollar o apoyar la competitividad y el desarrollo de los negocios, es el reto más importante que hoy enfrentan los responsables de implantar las tecnologías de la información en las empresas. Los modelos de gestión empresarial tradicionales son dinámicos y es por esto que los enfoques arquitectónicos que se seleccionen deben apoyar los procesos de negocio y no ser obstáculo.

La Ingeniería Dirigida por Modelos ofrece un prometedor enfoque para superar la incapacidad de los lenguajes de tercera generación y para gestionar de forma flexible el software que da soporte a los sistemas de información en las organizaciones. El objetivo que se persigue es reducir los desvíos que actualmente se observan entre las necesidades de gestión de la información en las organizaciones y los Sistemas de Información que realizan dicha gestión.

Tanto la creación como la gestión de un sistema de información son procesos complejos en los que habitualmente se presentan problemas de diversa naturaleza: desviaciones con respecto a la planificación y el presupuesto; falta de comprensión de los requerimientos del usuario; elección de tecnologías no adecuadas. Estos problemas surgen de la dificultad que supone gestionar correctamente proyectos de sistemas de información en los que hay que saber llegar a soluciones tanto relacionadas con la organización como con la tecnología.

La ingeniería dirigida por modelos (MDE) es una metodología de desarrollo de software que se centra en la creación y explotación de modelos de dominio en vez de en conceptos de programación[14]. Un modelo del dominio es una representación de conceptos u objetos en una situación real del dominio del problema. La aproximación de la ingeniería dirigida por modelos pretende incrementar la productividad mediante la reutilización de modelos estandarizados, simplificar el proceso de diseño mediante el uso de modelos que responden a patrones de

diseño y promocionando la comunicación entre individuos y equipos de trabajo en el sistema mediante la estandarización de la terminología y el uso de las mejores prácticas usadas en el dominio de aplicación.

El modelado de un sistema es considerado efectivo dentro de MDE si todos los modelos tienen sentido desde el punto de vista del usuario que es familiar con el dominio de aplicación y si ellos pueden servir como base de implementación del sistema. Los modelos se desarrollan a través de una comunicación extensiva con todos los implicados, desde jefes de producto, diseñadores y desarrolladores hasta los usuarios que manejan el dominio de aplicación.

1.1. Model-Driven Architecture (MDA)

Desarrollado dentro del Object Management Group (OMG), es un marco de trabajo para el desarrollo del software que define una arquitectura orientada a modelos y unas guías de transformación entre los mismos para recoger las especificaciones de un sistema, donde los productos que se generan son modelos formales.

MDA define una aproximación a la especificación de sistemas de tecnologías de la información que separa la especificación de la funcionalidad del sistema de la especificación de la implementación de dicha funcionalidad en una plataforma tecnológica específica. Para tal fin, MDA define una arquitectura para modelos que proporcionan un conjunto de pautas para estructurar las especificaciones expresándolas como modelos[12]. MDA separa la lógica del negocio de la lógica de la aplicación de la plataforma tecnológica y representa esta lógica con modelos semánticos muy precisos. MDA, al ser un marco arquitectónico, perfecciona la estrategia arquitectónica y mejora la eficiencia del grupo de desarrollo.

El enfoque MDA y los estándares que lo soportan permiten que la misma funcionalidad del sistema especificada como modelo pueda explotarse en múltiples plataformas, y permite a diferentes aplicaciones integrarse mediante la comunicación de sus modelos, habilitando la integración y la interoperabilidad[2, 3, 5]. Por tanto, proporciona una solución para los cambios de negocio y de tecnología, permitiendo construir aplicaciones independientes de la plataforma e implementarlas en plataformas como CORBA, JEE o Servicios Web.

El proceso de desarrollo en MDA está basado en un esquema de compilaciones sucesivas por el que, partiendo de un nivel de abstracción muy alto, se va transformando los modelos iniciales en modelos más detallados. Este esquema de compilaciones sucesivas permite que se pueda introducir lógica del negocio en los productos intermedios. Si el modelo de negocio inicial no describe completamente el negocio, este esquema de compilaciones permite al programador introducir código donde lo necesite. En cierto modo, se asemeja bastante a lo que es un generador de código, que a partir de un lenguaje de 4ª generación (4GL) generan código en lenguajes de 3ª generación (3GL).

2. Monet

Monet es una plataforma que pretende implementar la metodología de la ingeniería dirigida por modelos, si bien se diferencia de la línea seguida por MDA en que el sistema hace la interpretación de los modelos para generar un sistema plenamente operativo en lugar de la traducción o compilación de los modelos para generar un programa compilado.

En el ámbito de las organizaciones son más importantes atributos como la flexibilidad a los cambios y la agilidad en el desarrollo frente a la eficiencia computacional del código. Un programa compilado se ejecutará más deprisa que uno interpretado, pues el compilador ha producido previamente todo el código máquina necesario, y normalmente ha llevado a cabo un proceso de optimización. No obstante, la compilación tiene sentido cuando es necesario cuidar la eficiencia del código porque estamos tratando con sistemas que presentan una demanda significativa de recursos computacionales[7].

En el ámbito de las organizaciones, los recursos computacionales no son los críticos; es mucho más crítico que el sistema sea flexible para admitir cambios o que el desarrollo sea ágil.

2.1. Unidad de negocio y modelo de negocio

Una unidad de negocio en la arquitectura de la información propuesta por Monet es una entidad atómica dentro de la organización que cumple una determinada

función o competencia, ofreciendo al exterior una serie de servicios cuyos clientes serán clientes finales u otras unidades de negocio. Para la consecución de sus objetivos, la unidad de negocio se apoyará en recursos humanos que colaboran en una estructura horizontal. Cada persona tendrá una función y esta podría ser sustituida por un agente virtual, que delegaría esa función en otra unidad de negocio.

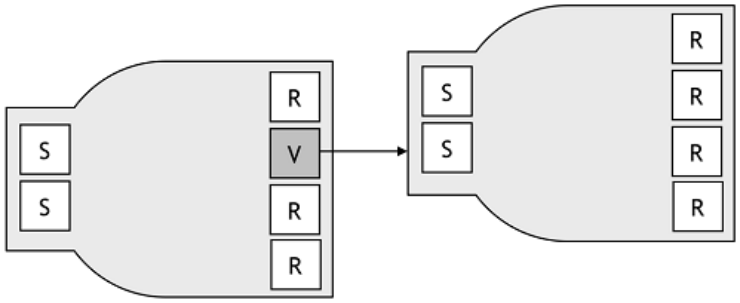

Figura 1: Unidades de negocio dependientes

El modelo de negocio es el modelo que interpretará Monet y que describe completamente el funcionamiento de la unidad de negocio. Un modelo de negocio en Monet es un conjunto de definiciones, representados en XML, que definen cada uno de los aspectos estáticos de la unidad de negocio. Las definiciones representan entidades concretas del sistema de información como servicios, formularios, documentos, tareas, colecciones, catálogos o informes. Las definiciones forman parte de una arquitectura similar a la orientación a objetos donde cada definición puede tener una definición padre de la que hereda sus características. También existe la figura de definición abstracta, que impone además la restricción de que debe ser heredada por otra definición ya que no es posible la creación de un objeto a partir de ella.

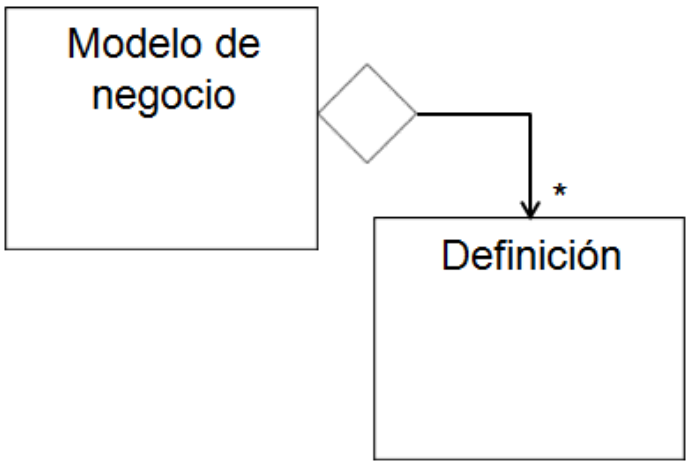

Figura 2: Relación entre el Modelo de negocio y las definiciones

Muchas de las entidades del sistema de información no son estáticas y tienen un cierto comportamiento. En Monet, existe un conjunto de definiciones a las que se les puede agregar un comportamiento dinámico, que permitirá describir la parte dinámica del sistema de información.

Las definiciones están a su vez supeditadas a un Meta-Modelo, donde se define la estructura de las definiciones en Monet, el cual a su vez está descrito mediante un Meta-Meta-Modelo que en este caso es el descrito por el lenguaje XML.

El modelo de negocio, una vez completamente definido, es proporcionado al llamado Espacio de negocio, que es el lugar creado por Monet a partir de las definiciones existentes en el modelo y donde convivirán todos los objetos de negocio creados a partir de las definiciones del modelo de negocio.

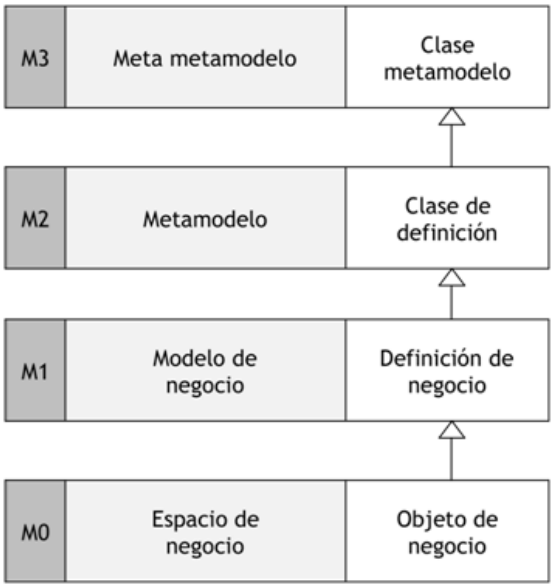

Figura 3: Diferentes niveles de lenguaje en Monet

A la hora de ver un modelo de negocio, este es posible verlo desde tres dimensiones diferentes, la organizacional, la operacional y la tecnológica. Cada una de ellas describe aspectos de la unidad de negocio a un nivel de abstracción diferente.

- Organizacional. Representa la visión del negocio. En ella se definen los servicios que presta la unidad de negocio, la relación con el cliente y el cuadro de mando desde el que se podrá tener un seguimiento de la marcha del negocio.
- Operacional. Representa la arquitectura de la información y los procesos necesarios para proporcionar los servicios a los clientes. En esta dimensión se definen los objetos de información, las tareas a realizar y las reglas de asociación y/o recomendación de estas tareas a los trabajadores.
- Tecnológica. Representa cómo se gestiona la información tanto para facilitar la comprensión por parte de los usuarios como permitir la interoperabilidad entre sistemas. En esta dimensión se definen los comportamientos del sistema de información, las plantillas de documentos y los modos de visualización y representación de la información.

En el momento de comenzar el diseño y desarrollo de un modelo de negocio, es importante que este siga cada una de las dimensiones en el orden indicado, siendo este el más lógico desde un punto de vista externo a la unidad de negocio.

Lo importante en una unidad de negocio y lo primero que debe estar definido son los servicios que presta la unidad de negocio y que aspectos de la actividad regular de la unidad de negocio son claves para el éxito de la misma desde un punto de vista empresarial (económico en el ámbito privado o de satisfacción del ciudadano en el ámbito de la administración pública).

A partir de estos aspectos, se diseña la dimensión operacional de la unidad de negocio para dar soporte a la prestación de los servicios y para registrar/tracear la actividad llevada a cabo para dicha prestación mediante los objetos de información. En este sentido se definirán los objetos de información que se manejarán de forma interna a la unidad de negocio, los flujos de trabajo que se siguen para completar las tareas que forman en forma conjunta la respuesta a un servicio, etc.

Por último, se definen los aspectos más técnicos y de "bajo nivel", esto es, la dimensión tecnológica, donde se estipula la forma en la que los usuarios interactuarán con estos objetos de información, las reglas de validación de estos, plantillas de documentos que se manejan, etc.

El error muchas veces cometido a la hora de plantear un sistema de información está en no seguir esta forma de trabajo "arriba-abajo", sino al revés. Esto se hace generalmente porque el cliente presiona para tener resultados rápidos, comenzando a definir la interfaz gráfica, los objetos de información de bajo nivel, etc. El problema viene cuando una vez todo construido, se va a comenzar a definir la capa de abstracción superior, esto es, la visión del negocio con el cuadro de mando. En este momento comienzan a salir datos que son necesarios para la construcción del cuadro de mando que no se han tenido en cuenta en las capas inferiores, cuando se tienen también muchos datos registrados que no se usan para nada, por lo que es necesario volver a cambiar todas las capas inferiores. Esto empeora cuando se usa una visión tradicional de construcción de software y estos cambios implican cambios en toda la aplicación, desde controles en interfaz gráfica, lógica de negocio, almacenamiento en base de datos, etc.

2.2. Arquitectura de la información

La arquitectura de la información en Monet se representa a partir de una serie de objetos de información que pueden usarse y que representan distintas relaciones entre los datos que almacenan. Podemos distinguir un gran grupo dentro de los objetos de información, que son todos los llamados nodos de información, que son los encargados de almacenar los datos del sistema de información. Luego existe otros tipos de objeto que complementan y/o enriquecen a estos objetos de tipo nodo.

Los diferentes tipos de nodo existentes en Monet son:

2.2.1. Nodo Formulario

Los nodos de tipo formulario son estructuras de información organizadas en campos, que pueden ser simples o múltiples. Existe un amplio abanico de tipos de campos que puede albergar un formulario:

- Campos básicos de texto, número, patrón de texto o código de texto autogenerado.
- Campos de selección, que no permiten una edición libre, sino que se debe seleccionar un valor de entre los posibles. Estos posibles valores se seleccionan a partir de un tesauro, otro objeto dentro de la arquitectura de la información que se describirá más adelante.
- Campos de tipo vínculo que permite establecer un enlace con otro nodo (siendo sus ciclos de vida independientes), así como campos de tipo contenedor, que permiten almacenar un nodo dentro del propio formulario (estando sus ciclos de vida enlazados, se crean y se destruyen a la vez que el formulario que los contiene).
- Campos avanzados como imagen, fichero o posición geográfica.

2.2.2. Nodo Colección

Los nodos de tipo colección permiten almacenar un conjunto de nodos, que si bien pueden ser de distinto tipo, todos ellos deben tener la característica común de implementar la misma referencia. La referencia es una definición que establece un conjunto de atributos que debe aportar un nodo, como si de una interfaz se tratase (en el contexto de la orientación a objetos).

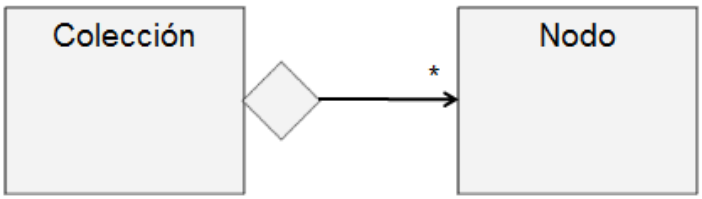

Figura 4: Relación entre una colección y los nodos contenidos

Este tipo de nodo representa se caracteriza por representar una relación de agregación de uno a muchos. El tiempo de ida de los objetos que contiene no está determinado por el tiempo de vida de la colección en la creación. Cuando se crea la colección, esta está vacía y no contiene ningún elemento, estos se van agregando a posteriori. Sin embargo, cuando la colección se elimina, todos los nodos contenidos son eliminados también. Los nodos se visualizan en un listado, en donde se muestran los datos de la referencia. Este listado se puede clasificar, ordenar y filtrar por cualquiera de los atributos definidos en la referencia.

2.2.3. Nodo Contenedor

Los nodos de tipo contenedor se basan en la relación de composición, es decir, están compuestos por uno o varios nodos de distinto tipo, que nada tienen en común. El tiempo de vida de sus componentes está determinado por el tiempo de vida del contendor que nos contiene, es decir, cuando se crea el contenedor, se crean sus componentes y cuando se elimina el contenedor, sus componentes también se eliminan.

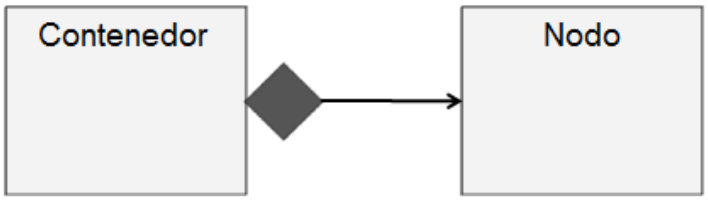

Figura 5: Relación de composición entre un contenedor y el nodo que contiene

Otra de las características de este tipo de nodos es que no implementan directamente una referencia, sino que por defecto implementan todas las referencias de los nodos que contiene, por lo que puede ser agregado en cualquier nodo de tipo colección que requiera una referencia que implementa cualquiera de los nodos que contiene.

2.2.4. Nodo Catálogo

Los nodos de tipo catálogo tienen como uso principalmente el hacer de envoltorio a una determinada colección, a la que se le aplican una serie de filtros, ordenación y/o agrupación por defecto. Estos catálogos se pueden usar luego para que los campos de tipo vínculo puedan mostrar al usuario los nodos que puede vincular con ese campo en particular.

2.2.5. Nodo Documento

Los nodos de tipo documento son, como su propio nombre indica, los que representan un documento. Estos nodos tienen un estado, en el que se indica si el documento es modificable o está consolidado. En función de este estado se pueden realizar operaciones diferentes. Si está en estado modificable, el documento puede recalcularse, es decir, el usuario no puede editar directamente el documento sino que este se rellena a partir de los datos de diferentes formularios, contenedores o colecciones del sistema de información. Una vez consolidado el documento, este ya no puede cambiar y a partir de ese momento se pueden realizar otro tipo de operaciones como su firma digital o su distribución a otras unidades de negocio. La interoperabilidad entre unidades de negocio se basa en el intercambio de documentos producidos por este tipo de nodos.

2.2.6. Nodo Escritorio

Los nodos de tipo escritorio pretenden ser un nodo especial donde el usuario pueda encontrar todo aquello que utiliza en su trabajo diario a mano, además de herramientas de groupware como videoconferencia o telefonía VOIP, notificaciones, notas, etc. Es un concepto que aún no está desarrollado en todo su potencial en Monet actualmente, estando en un estado aún muy primario.

2.3. Mapas de trabajo (Workmaps)

Los mapas de trabajo en Monet son una forma dinámica de plantear el trabajo colaborativo en una organización. Su fundamento es que los flujos de trabajo en la vida real no son estáticos y no es abordable desde el punto de vista práctico generar un sistema que tenga en cuenta todos los posibles casos que se pueden llegar a dar en la realidad. No es abordable no solo porque la explosión de posibles combinaciones sea casi infinita, sino también porque no es posible siquiera conocer todos los posibles casos que se pueden dar.

La visión tradicional de los flujos de trabajo (WorkFlows), tienen el problema de una falta de dinámica, donde los usuarios puedan indicar un cambio en el flujo de trabajo en un momento determinado, debido a circunstancias no previstas en el momento de diseño del mismo[7].

En los mapas de trabajo se diseñan bajo la premisa de que los usuarios son responsables del trabajo que están haciendo y son ellos los que deben velar por que el trabajo se realice como se debe y no que sea el ordenador el que les ordene como deben trabajar, que pueden hacer o que cosas del mundo real pueden o no pueden pasar, limitándose el usuario a simplemente darle al botón “siguiente”. En los mapas de trabajo, lo que se asegura es una trazabilidad de todo el trabajo realizado y todas las decisiones tomadas, de forma que siempre quede constancia de “quién hizo qué”.

El paradigma en el que se basan los mapas de trabajo es un mapa de carreteras, donde tenemos una serie de lugares y una serie de carreteras que conectan cada uno de los lugares del mapa. En este mapa existe una serie de líneas de autobús que cubren todos los lugares, pero que pasan por todas las carreteras para

ello. La equivalencia al caso de los mapas de trabajo es la siguiente. Los lugares son los llamados WorkPlaces y son como estados por los que pasa la tarea que realizamos. Las carreteras son los llamados WorkLines y representan un trabajo determinado que culmina en un determinado estado. Una vez se comienza un trabajo (subimos a una línea de autobús para recorrer una carretera), este puede finalizar en una serie de estados en función de cómo ha terminado este (lugar en el que nos bajamos del autobús). Sin embargo, como hemos comentado anteriormente no todas las carreteras están cubiertas por autobuses, por lo que no todos las posibles trabajos (carreteras) están representados en el mapa de trabajo, como ocurre en la realidad, no es posible tener en cuenta todas las situaciones que se pueden dar.

Puede ocurrir que estando en un determinado lugar, queremos llegar a otro, pero en ese momento determinado no podemos/queremos tomar una línea de autobús y tomamos una forma alternativa de transporte por otra carretera. Esto es lo que simboliza en los mapas de trabajo una situación excepcional no contemplada por la que podemos pasar de un lugar a otro sin pasar por ningún WorkLine.

Definido lo que es un mapa de trabajo, a continuación vamos a ver un ejemplo acompañado de la representación gráfica con la que se analizan y diseñan los mapas de trabajo. En esta representación los lugares (WorkPlaces) se representan en columnas y las líneas de trabajo (WorkLines) en horizontal. Cuando se cruzan, si existe una parada en ese lugar de esa línea existe lo que denominamos un WorkStop.

Por último queda definir el comportamiento del motor de mapas de trabajo. El motor tratará de completar el mapa de trabajo siguiendo los caminos disponibles a partir de un lugar inicial. Cuando se llega a una línea que tiene más de una parada, es necesario indicarle al motor (el conductor del autobús) en que parada nos queremos bajar. Para tomar esta decisión necesitaremos información que puede estar ya disponible en el sistema de información, información de otra unidad de negocio, información por parte del usuario, información determinada por otra tarea o información que estará disponible pasado un determinado tiempo. Todas estas situaciones pueden no poderse concretar en el momento en el que estamos en la línea y será necesario bloquear la línea hasta que tenemos esos datos. Este concepto es el que denominamos WorkLock. Su función es bloquear el mapa de trabajo hasta que se obtenga el dato necesario. Pueden existir varios

bloqueos en la línea y solo se podrá avanzar cuando alguno de ellos se libere. Existen los siguientes tipos de bloqueos:

- DesicionLock. Se le pide al usuario que tome de forma directa que parada debe tomarse de las disponibles en la línea.
- SyncLock. Bloquea la línea hasta que una determinada tarea termina su trabajo.
- BranchLock. Se bloquea dando el control al comportamiento definido por el analista, el cual de forma automática y en base a datos existentes en el sistema de información designará la parada a tomar.
- ServiceLock. Realiza una petición de servicio a otra unidad de negocio y espera a que esta responda. Una vez recibida la respuesta, se continua con el mapa.
- FormLock. Se muestra un formulario al usuario, el cual debe rellenar para poder continuar con el trabajo.
- TimerLock. Se desbloquea la línea hacia una determinada parada una vez ha pasado un determinado espacio de tiempo desde que se alcanzó el bloqueo.

Ejemplo de Mapa de trabajo y diagrama de representación Para ejemplificar el funcionamiento y representación de un mapa de trabajo, vamos a plantear un caso sencillo, la resolución de incidencias relacionadas con un programa informático. Para simplificar el ejemplo, vamos a suponer que no existen más casos ante una incidencia sino que esta sea resuelta o que no proceda.

En primer lugar, se notifica una incidencia y con ello se crea una tarea en Monet que tiene un mapa de trabajo definido como se muestra en la Ilustración 6, donde vemos que el mapa de trabajo está compuesto por dos líneas de trabajo (WorkLines), cuatro lugares (WorkPlaces), tres paradas (WorkStops) y dos bloqueos (WorkLocks).

En primer lugar vamos a describir los lugares:

- Notificado. La incidencia acaba de ser notificada. Este lugar es de tipo "event", lugar donde comienza el mapa de trabajo.
- Pendiente operación. El fallo indicado en la incidencia existe y está pendiente de ser corregido.
- Terminado. La incidencia se ha resuelto, estando el fallo notificado corregido. Este lugar es de tipo "goal", indicando que al llegar a este lugar se termina de forma satisfactoria el mapa de trabajo.
- Cancelado. El fallo al que hace referencia la incidencia no existe o no se ha podido reproducir, por lo que no se ha corregido nada. Esta es un lugar de tipo "dead-end", indicando que si se llega a este lugar, se aborta el mapa de trabajo.

Cuando el mapa de trabajo comienza a ejecutarse, se encuentra en el lugar inicial de "Notificado". Se avanza por la línea de trabajo disponible: "Clasificando incidencia" y nos encontramos con un bloqueo de tipo "DecisionLock", en el que el usuario debe decidir en qué parada nos vamos a bajar de la línea de trabajo, existiendo dos paradas posibles: "Hay que corregirlo" y "No procedente".

Si el usuario escoge "No procedente", se abandona la línea por esa parada, encontrándonos en el lugar "Cancelado". Dado que este lugar es final de tipo "dead-end", aquí se aborta el mapa de trabajo.

Si el usuario escoge "Hay que corregirlo", se abandona la línea por esa parada, encontrándonos en el lugar "Pendiente operación". En ese lugar comienza otra línea, "Ejecutando", por la que sigue la ejecución del mapa de trabajo, encontrándose con otro bloqueo, también de tipo "DecisionLock". En esta ocasión, solo existe una posible opción, la parada "Corregido". Esto servirá para que el usuario pueda indicar el momento en el que la incidencia ha sido corregida. Una vez llegado a esa parada, se abandona la línea, quedándose en el lugar "Terminado", que al ser un lugar de tipo "goal", hace que el mapa de trabajo termine.

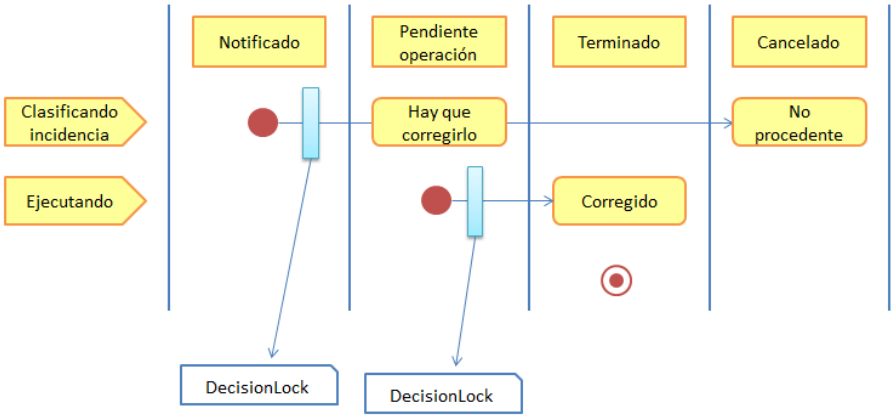

Figura 6: Ejemplo de representación de un Mapa de trabajo

Una vez hemos visto cómo funcionan los mapas de trabajo, imaginemos que nos encontramos durante la ejecución de una resolución de una incidencia y cuando vamos a corregirla vemos que el fallo ya ha sido corregido de forma indirecta, ya que se debía a otro error. El problema es que esta incidencia realmente no procedía, ya que era la misma que otra incidencia ya indicada y resuelta anteriormente, pero por alguna razón se clasificó como que había que corregirla. El usuario ahora se vería obligado a responder que está corregida, ya que es la única opción disponible, cuando realmente no es lo que ha ocurrido, ya que no ha habido ningún tipo de acción asociado a esta incidencia que haya realizado.

Aquí es donde radica la potencia de los mapas de trabajo, el usuario podrá hacer un cambio de lugar, haciendo que el mapa de trabajo se sitúe en el lugar "Cancelado", abortando el mapa de trabajo, quedando registrado que esta incidencia no se resolvió debido a que ya estaba resuelta anteriormente.

Parte III

Fundamentos Teóricos del Trabajo/Desarrollo

Expuesto los conceptos base sobre los que se fundamenta Monet y las características de esta plataforma, el objetivo será el estudio de en qué forma pueden estos modelos de ser interpretados por una plataforma móvil que se ejecuta en dispositivos móviles. Si bien esta interpretación de los modelos no puede ser completa, tal y como lo hace Monet, ya que es necesario que una entidad mantenga la integridad de la información y esta responsabilidad recae en el núcleo de Monet.

3. Herramientas y plataforma de trabajo

La primera decisión a tomar para la realización de este trabajo es en que plataforma se va a trabajar y que herramientas se utilizarán. En el entorno de los Smartphones, existen distintas plataformas, siendo las más destacadas por volumen de ventas y cuota de mercado Android (desarrollado por Google), iPhone (desarrollado por Apple) y Blackberry (desarrollado por RIM). Para este trabajo se ha tomado la decisión de desarrollar Monet Mobile para la plataforma Android. Las razones que nos llevan a tomar este camino son:

- Plataforma abierta. Android es una plataforma de software libre y apoyado por muchos de los grandes fabricantes de teléfonos del mercado como Samsung, HTC, Motorola o LG. Este apoyo se materializa en un enorme abanico de modelos de teléfono de distintas gamas, calidades, características y precios, haciendo de la plataforma la más fácil de adoptar por costes y dado el amplio apoyo de la industria, la que menos riesgos implica a la hora de apostar por una de ellas.
- Diversidad. Android no restringe la forma en que los fabricantes deben construir los dispositivos, encontrándonos con un enorme abanico de posibilidades. Desde dispositivos de gama alta, con actividades táctiles y una

gran variedad de sensores de todo tipo a dispositivos de gama empresarial, que disponen de teclados físicos, actividades más pequeñas o carcasas diseñadas para una alta durabilidad en entornos hostiles donde las caídas, el polvo o las altas temperaturas hace que la mayoría de teléfonos inteligentes no pueda operar normalmente.

- Herramientas de desarrollo. Android se apoya en el lenguaje de programación Java, así como en el entorno Eclipse, para el que todas las herramientas de desarrollo están disponibles en forma de plug-ins. Esto permite frente a otras plataformas una ventaja en costes, además de ser un lenguaje y unas herramientas muy experimentadas, productivas y una gran comunidad de desarrolladores detrás.

- Cuota de mercado. Android está experimentando un crecimiento espectacular desde su nacimiento y es ya el sistema operativo para Smarthphones más usado en EEUU y, según las cifras de crecimiento que está experimentando, pronto lo será también en Europa[10].

Para el desarrollo del trabajo se usarán las herramientas de desarrollo recomendadas, tal y como se ha en el punto anterior, estas son el entorno de programación Eclipse y el lenguaje de programación Java.

3.1. Arquitectura de una aplicación Android

Antes de comenzar con la exposición del trabajo realizado cabe destacar por último la arquitectura de aplicaciones que propone Android, ya que es un elemento muy importante a la hora de comprender como se estructuran y desarrollan las aplicaciones para esta plataforma y cómo afecta ello a la arquitectura de Monet Mobile.

Una aplicación en Android se divide en los que se llaman Actividades. Una actividad no es más que una tarea concreta que desea realizar el usuario. Ejemplos de una actividad pueden ser la actividad que permite al usuario visualizar su bandeja de correos electrónicos, otra la que le permite visualizar el contenido de un correo electrónico o una que le permita seleccionar un contacto a partir de una lista de contactos.

En estos ejemplos podemos ver que en todos los casos siempre existe una acción que el usuario desea realizar. Esto es lo que se identifica en Android como intención (Intent) y es lo que identifica ante el sistema que acción permite realizar una actividad. Cuando una aplicación se instala en un dispositivo con Android, el sistema operativo solicita a la aplicación su manifiesto donde esta declara todas sus actividades y a que intenciones de realizar una acción por parte del usuario, es capaz de responder cada una de estas actividades. De esta forma, cuando una aplicación recibe por parte del usuario su deseo de realizar una acción, como por ejemplo, seleccionar un contacto al que enviarle un correo electrónico; esta notifica al sistema operativo indicándole que tiene la necesidad de mostrar una actividad que sea capaz de seleccionar un contacto mediante una intención[13, 11].

El sistema operativo consulta su base de datos donde tiene registradas todas las actividades de todas las aplicaciones instaladas en el sistema y selecciona de ellas las que puedan satisfacer esa intención solicitada. Así por ejemplo, a esta intención es posible que responda una actividad del propio sistema que permitirá seleccionar un contacto desde la agenda. Pero también es posible que existan más de una actividad, por ejemplo imaginemos una actividad que permite seleccionar el contacto desde una red social como Facebook o Twitter.

Si existe solo una actividad candidata a responder a esa intención, el sistema la lanzará automáticamente. Si por el contrario existe más de una que pueda responder a esa intención, el sistema informará al usuario de todas las actividades que pueden satisfacerla y será este quien seleccione con que actividad quiere resolver la intención que se ha lanzado.

De esta forma, la experiencia de usuario será que las aplicaciones que tiene instaladas en su dispositivo colaboran para ayudarle a realizar la tarea que quiere realizar de forma natural, sin que estas tengan que integrarse unas con otras de forma explícita.

Desde el punto de vista del desarrollador este esquema permite eliminar por completo el acoplamiento en la aplicación entre sus distintas partes, pudiendo cambiar entre ellas sin necesidad de un cambio de código. Es más puede ver como su aplicación es extendida por otros para añadirle nuevas características.

4. Prototipado

Una vez estudiado la arquitectura de la plataforma de desarrollo que vamos a utilizar y teniendo claro los tipos de objeto que se pueden utilizar (descritos anteriormente), pasamos a la fase de prototipado. En esta fase del trabajo el objetivo ha sido tratar de ver el resultado final de la aplicación, ya no solo desde el punto de vista del diseño gráfico, sino que la colocación de los elementos dentro de la aplicación, así como poder adelantar posibles requisitos que no se habían descubierto hasta el momento. Este paso es especialmente importante en el caso de una aplicación para dispositivos móviles, donde el tamaño de la actividad es muy limitado y por tanto la cantidad de información que es posible mostrar en cada momento a de estudiarse para aprovecharlo al máximo. [1]

4.1. Hub de Unidades de negocio

La primera actividad que se le presentará al usuario cuando este abra la aplicación principal de Monet Mobile será el llamado "Hub" de conexiones con las unidades de negocio. Aquí se mostrará un listado de todas las unidades de negocio añadidas por el usuario para su acceso. El listado se conforma de un icono, un titulo principal y una descripción de la unidad de negocio. Todos estos datos serán recogidos por la aplicación en el momento de la "instalación" del modelo de negocio en el teléfono. Para esta configuración el usuario introducirá la dirección URL donde se encuentra el entorno de ejecución, así como sus credenciales.

Una vez finalizado el proceso de instalación, el usuario podrá acceder a la unidad de negocio a través del Hub o a través de un icono específico que se añadirá a uno de los escritorios de su teléfono, para poder acceder de forma directa. Aquí por tanto aprovecharemos una de las características de la arquitectura de la aplicación y es el poder acceder directamente a contenidos concretos dentro de la misma, lanzando la intención adecuada para cada unidad de negocio.

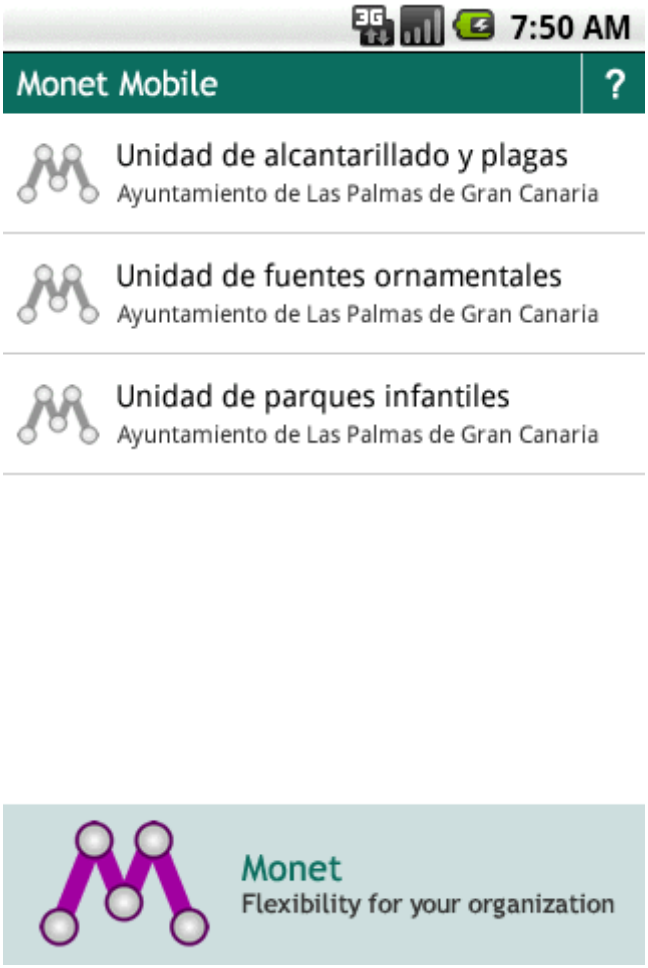

Figura 7: Hub de conexión con distintas unidades de negocio

4.2. Nodo de tipo escritorio o desktop

Los nodos de tipo escritorio o tipo desktop están en el momento de desarrollo actual de la plataforma Monet en un estado primitivo de lo que se espera que ser en un futuro. Actualmente lo que permite es mostrar enlaces a otros nodos simpre que estos sean singleton. La idea futura tras este tipo de nodos es que sirvan de espacio donde el usuario pueda hacer sus cosas cotidianas, como acceder a notas personales, acceder al sistema de información, ver trabajo pendiente, hacer reuniones virtuales, videoconferencia, etc.

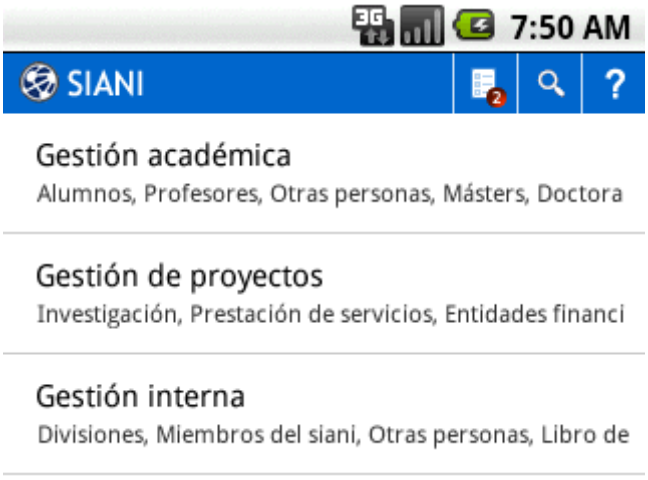

Figura 8: Nodo de tipo escritorio o desktop

Por lo tanto, la interfaz para este tipo de nodos será simplemente un listado de cada uno de los nodos a los que se enlaza, permitiendo al usuario el acceso a las distintas partes que conforman el sistema de información.

Una característica común que tendrán todas las actividades que muestran objetos dentro de una unidad de negocio es la barra superior, donde encontraremos, además de la etiqueta del objeto que actualmente estamos visualizando, los botones de tareas, búsqueda y ayuda.

- Tareas. Permite visualizar la cantidad de tareas pendientes que esperan

respuesta y en la que podemos participar, además de poder acceder al listado de tareas.

- Búsqueda. Permite realizar una búsqueda dentro del sistema de información a partir del objeto actual en profundidad. Esta funcionalidad estará integrada con el botón de búsqueda que tienen algunos teléfonos Android.
- Ayuda. Permite visualizar la ayuda del objeto que estamos visualizando y que el analista habrá definido en el modelo.

4.3. Nodo de tipo contenedor

Para visualizar los nodos de tipo contenedor, se diseña una interfaz inicialmente donde cada uno de los notos contenidos se muestra en forma de lista, de forma muy similar a como se visualizan los nodos de tipo escritorio, si bien durante la implementación de la aplicación se opto por cambiar esta visión por una en la que se usan pestañas, a través de las cuales los usuarios pueden ver las vistas embebidas de cada uno de los nodos contenidos. Esa nueva visión se extrae a partir de la buena experiencia obtenida en la versión de la aplicación web, donde una vez introducido este concepto, se revolucionó todo lo relacionado con las vistas de los contenedores y su contenido.

4.4. Nodo de tipo colección

Los nodos de tipo colección se visualizan listando cada uno de sus elementos en forma de scroll infinito. Este tipo de scroll consiste en que cuando el usuario alcanza el final del bloque de elementos que visualiza, el sistema recupera el siguiente bloque de elementos, mostrándolos a continuación, teniendo el usuario la sensación de que la lista crece continuamente en cuanto se acerca al final. Este patrón de diseño de interfaces evita que el usuario tenga que estar navegando usando la paginación numerada o con los tediosos “anterior” y “siguiente”.

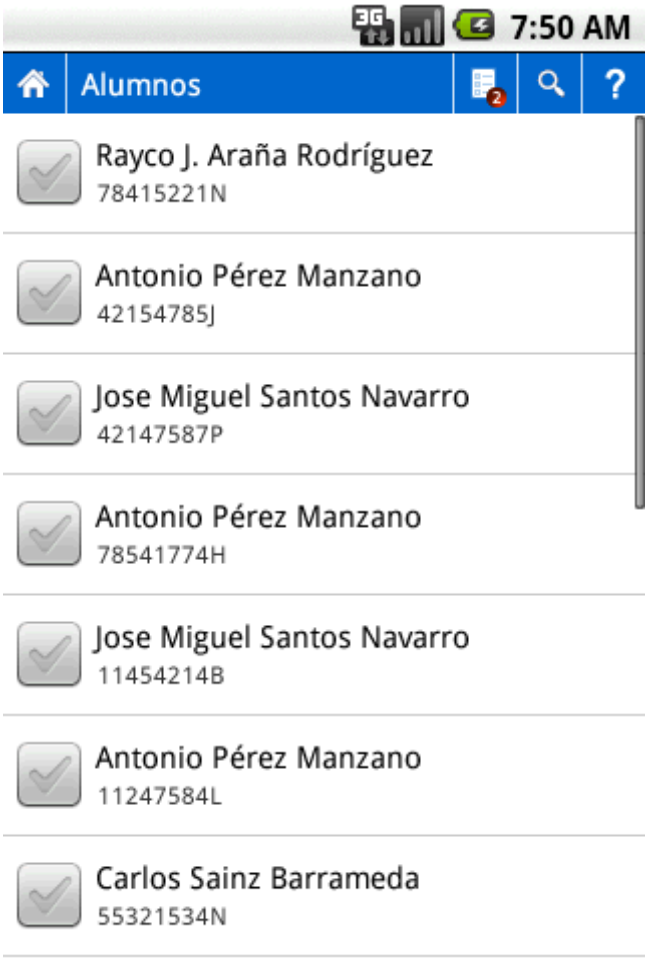

Figura 9: Nodo de tipo colección

Para la visualización de cada uno de los elementos se utilizará de forma destacada la etiqueta del elemento y en segunda línea una ristra con todos los valores de la referencia calculada de ese nodo.

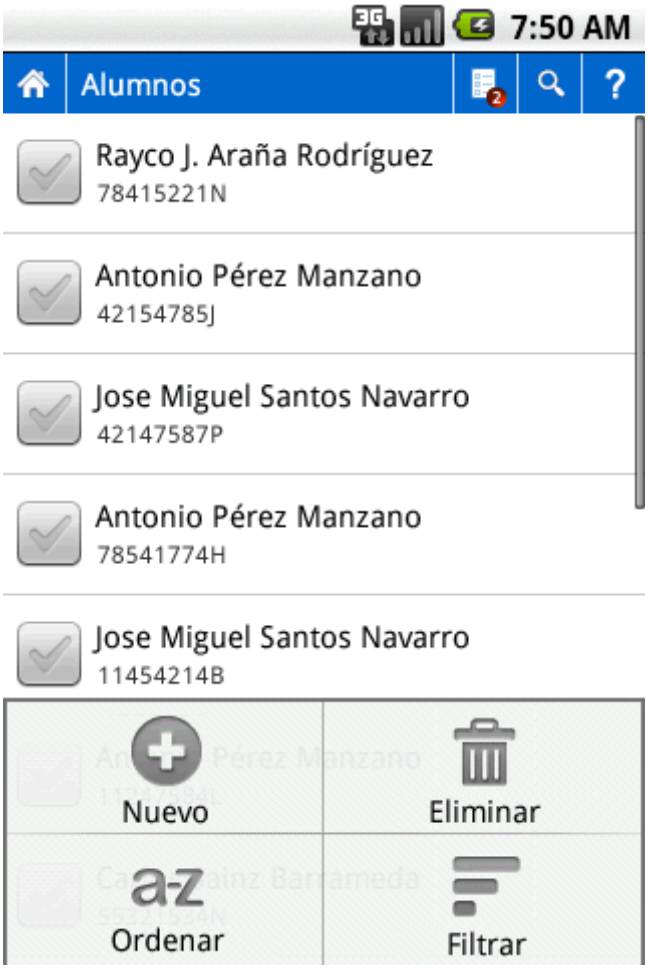

Figura 10: Menú de opciones en las colecciones

En el menú de esta actividad, que se despliega con el botón hardware dedicado en los teléfonos Android, el usuario podrá acceder a las siguientes funciones:

- Nuevo. Permite añadir un nuevo objeto a la colección (mostrándose una lista de los tipos de objetos que se pueden añadir a dicha colección).
- Eliminar. Permite eliminar los elementos seleccionados.
- Ordenar. Permite establecer el criterio de ordenación.
- Filtrar. Permite establecer un filtro.

Figura 11: Diálogo de selección del nuevo tipo de objeto a añadir

4.5. Nodo de tipo formulario

Los nodos de tipo formulario mostrarán cada uno de sus campos, tanto en modo visualización como en modo edición en forma de lista. En el caso de las secciones de un formulario, estas se resuelven en otra actividad, lo que mejora la experiencia de edición al evitar el usuario tener que tener que hacer un scroll muy grande para rellenar los datos.

Al igual que en el caso de los contenedores, es posible que se definan distintas

vistas en forma de pestaña, cada una de las cuales mostrarán subconjuntos de campos del formulario, si bien no estaba previsto en el momento del prototipado este comportamiento.

Los campos sobre los que se aplica algún tipo de restricción, como por ejemplo que el campo debe ser obligatorio, se mostrará en el modo de edición con una señal a continuación de la etiqueta del campo.

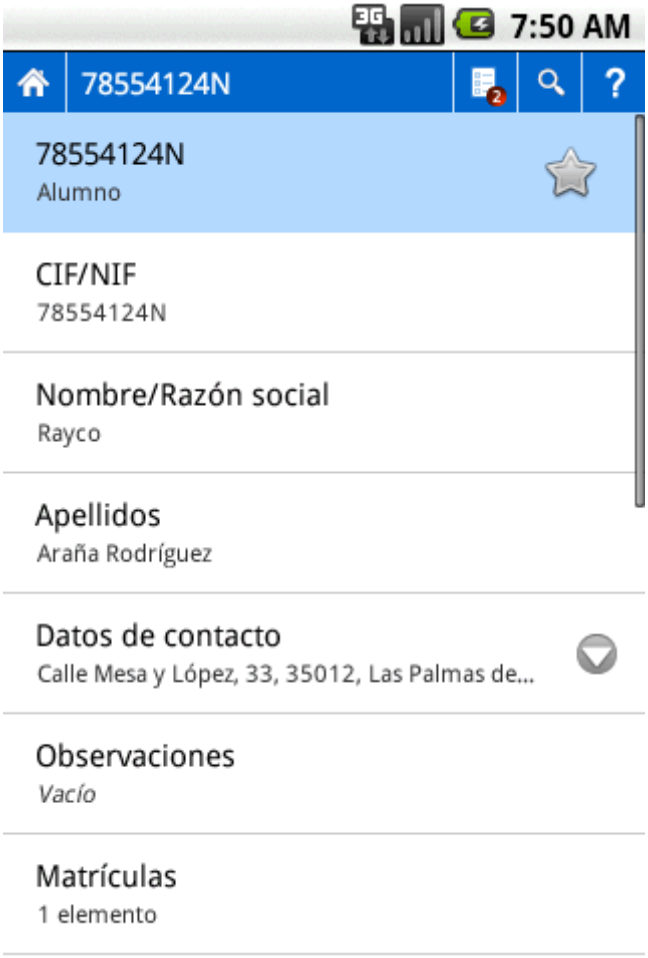

Figura 12: Nodo de tipo formulario

Figura 13: Nodo de tipo formulario en modo edición

Una de las características que se quiere potenciar es la posibilidad de rellenar el formulario por medio de la voz, gracias al reconocimiento de voz disponible en Android y evitar así el tener que teclear cada uno de los datos. Esto permitiría una edición rápida en el campo de trabajo y luego a posteriori y ya en un entorno más favorable como es el ordenador de escritorio, una edición a conciencia del formulario.

5. Arquitectura

Una vez hemos concluido la fase de prototipado, lo siguiente es plantear la arquitectura de la aplicación, tanto la aplicación cliente que se ejecutará en el dispositivo móvil, como la arquitectura de la aplicación web que será necesaria desarrollar para dar soporte a los servicios requeridos por la aplicación móvil.

En primer lugar, plantearemos la arquitectura de la aplicación para el dispositivo móvil. Esta arquitectura tendrá que encajar en el modelo de aplicaciones de Android. Si bien esto no es obligatorio, ya que es posible crear una aplicación que se ejecute por completo dentro de una sola actividad y por tanto es posible obviar toda la arquitectura base que propone Android. Sin embargo, creando la aplicación de esta forma no aprovechamos algunas de las características que hace de la arquitectura de las aplicaciones Android uno de sus mejores argumentos para su uso.

Una de las características que se quiere aprovechar y potenciar en la arquitectura de la aplicación es la que tiene que ver el punto de entrada de la aplicación y la integración con otras aplicaciones. Si el planteamiento que realizamos de la aplicación es de una actividad por cada tipo de objeto de información de que se dispone en Monet y permitimos mediante el manifiesto de la aplicación que otras aplicaciones puedan llamar directamente a estas actividades, conseguiremos un sistema con una alta cohesión y múltiples puntos de entrada. Si bien el punto de entrada principal será Monet Mobile y el primer nodo de información que verá el usuario será aquel que se designe en el modelo de negocio, es posible generar aplicaciones externas a Monet que se integren fácilmente con Monet Mobile.

Los ejemplos de tipos de integraciones que se pueden dar son infinitos, como infinitas son las posibilidades. Por ejemplo se puede permitir que una aplicación permita realizar mediante una fotografía a un código 2D, traducir la información a un enlace a un objeto de información en Monet, el cual al ser lanzado, produce que el usuario visualice el nodo de información asociado a esa etiqueta. Esto puede ser muy útil para un modelo de negocio que gestione stocks en un almacén.

Otro posible uso con esta misma característica es la posibilidad que ofrece en el trabajo de campo, es decir, el trabajo que se realiza en lugares externos a la organización. Por ejemplo, un técnico que vaya a revisar una incidencia

en un parque, donde se encuentra que un determinado elemento del parque está estropeado, por el simple hecho de estar en ese lugar, es posible crear una aplicación accesoria que permita a partir del lugar donde se encuentra el técnico abrir el nodo de información en el sistema de información que representa ese parque o abrir un catálogo donde se muestren todas las incidencias abiertas que tienen origen en ese parque.

Esta arquitectura basada en actividades sin embargo presenta un desafío en el desarrollo del resto de elementos presentes en la aplicación, que deben estar diseñados para permitir un arranque rápido y no dependiente del punto de entrada usado para abrir la aplicación. Esto significa que la aplicación no va a tener un punto de inicialización central o una pantalla de carga, por lo que hay que optimizar la aplicación y su arquitectura de forma que el arranque sea rápido y escalonado en función de los elementos requeridos por cada actividad.

Otro requisito importante a la hora de crear la arquitectura de la aplicación es que esta debe favorecer en la medida de lo posible su prueba a través de test unitarios. Los test unitarios en un entorno móvil cobran mayor importancia, ya que bien usados pueden evitar tener que probar la aplicación continuamente con el emulador del dispositivo, lo que puede reducir el tiempo de desarrollo, además del resto de virtudes del desarrollo orientado a la prueba.

Para cubrir estos requisitos, el planteamiento principal de la arquitectura se basa en el uso del patrón de diseño Model-View-Presenter.

5.1. Model-View-Presenter (MVP)

El patrón de diseño Model-View-Presenter está basado en el patrón de diseño Model-View-Controller, si bien pretende optimizar este para favorecer escenarios fundamentalmente de construcción de interfaces gráficas. Aquí el Presenter toma toda la responsabilidad de control acerca de cómo se maneja la vista. Todos los manejadores de eventos producidos en la interfaz gráfica, si bien se implementan en la vista, esta lo único que realiza es una simple conversión de los objetos a objetos de la aplicación y lanza a su vez un evento hacia su presenter. El presenter, que únicamente ve a la vista como una intefaz, no accediendo directamente a la clase particular que la implementa, recibe un evento no dependiente de

ningún framework gráfico particular y realiza las acciones convenientes sobre el resto de la interfaz gráfica, a través de la interfaz de vista. La vista únicamente tendrá que traducir esos comandos de bajo nivel de interfaz gráfica a la API de interfaz gráfica particular en la que se basa su implementación. Esto permite que al basarse todo en una API abnostica en cuanto a la API gráfica y usar interfaces, toda la lógica asociada a la interfaz gráfica puede ser probada con test unitarios en un porcentaje mucho mayor al código que es posible probar en una arquitectura basada en Model-View-Controller[9, 6].

Por aprovechamiento del código suele ser habitual ver este tipo de arquitectura con una serie de capas de control transversales a todos los Presenters, que ofrecen servicios comunes como comunicaciones, persistencia, etc., que usarán varios Presenters. Así aparece el concepto de Controller, si bien no tendrá el mismo significado que el Controller del patrón Model-View-Controller. En este caso el Controller tendrá como responsabilidad la de ofrecer una capa de servicios de alto nivel, asociados al dominio de la aplicación y lejos por completo de las particularidades de la vista.

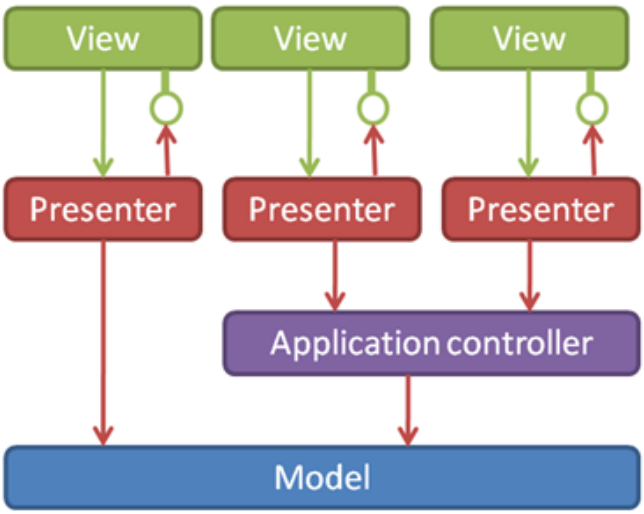

Figura 14: Arquitectura Model-View-Presenter

Este patrón suele presentar un problema con el uso del modelo de dominio de la aplicación (Cabe destacar aquí para no confundir al lector, que nos referimos al

modelo particular de una aplicación, no del modelo de negocio del que hablamos en el desarrollo dirigido por modelos en Monet. En el caso de Monet Mobile el modelo de dominio son los objetos de información, es decir; formularios, contenedores, colecciones, tareas). El problema reside en la desincronización que existe entre los modelos y los datos requeridos por una vista. En la práctica, una vista usará más de un modelo de dominio, haciendo uso de un subconjunto de datos de estos. El problema de esta visión es por un lado el uso de información extra innecesaria que debe leerse desde la fuente de persistencia de datos, teniendo por tanto un problema de rendimiento.

Este no es el único problema asociado, ya que existe un acoplamiento no deseado entre el modelo y las vistas, que obliga a cambiar las vistas cuando cambia el modelo. Si tenemos varias implementaciones de una misma vista, esto obligará cambiar el código en cada una de ellas. Para resolver este problema, uno de los patrones de diseño que más se están usando en la actualidad es el llamado Model-View-ViewModel-Controller y su equivalente Model-View-ViewModel-Presenter.

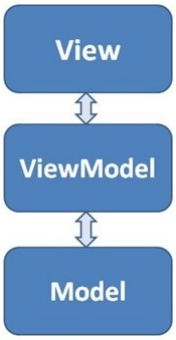

Figura 15: Relaciones entre la vista, el modelo de la vista y el modelo del dominio

En este patrón de diseño se introduce el denominado ViewModel o modelo de la vista. Este es un modelo que no existe en el modelo de dominio tal cual, sino que está ajustado a lo que la vista necesita mostrar. De esta forma, las vistas de la aplicación solo entienden de su modelo particular.

Una vista solo recibe un modelo, el cual muestran al usuario. Este modelo se lo provee el Presenter, el cual será el encargado de "traducir" desde los distintos

modelos del dominio al modelo de la vista en particular. Con esto se evita que la vista pueda por error hacer uso de datos que no le competen, que este acoplada a los modelos de dominio, teniendo únicamente que cambiar los Presenters para acomodar la "traducción". Otra ventaja de esta forma de trabajo es que los modelos de vista pueden ser más fácilmente transportados por un canal de comunicaciones, al ser más ligeros y no tener dependencias con otros modelos. Por último, las vista solo tendrán un método para asignarles su modelo, en vez de tener que realizar N número de llamadas, evitando errores debidos a no llamar a todos los métodos necesarios para inicializar la vista con todos los datos necesarios.

En la aplicación particular de este patrón de diseño a Monet Mobile, vamos a hacer un uso particular de este patrón de diseño. Una de las cuestiones a destacar es que Monet Mobile no es sino un visualizador de lo que está ocurriendo en el espacio de negocio, controlado por el núcleo de Monet. Debido a la importancia de la concurrencia en la ejecución del comportamiento de los modelos de negocio, así como la validación de reglas y a la imposibilidad de que el comportamiento sea ejecutado en los clientes debido al potencial uso de cualquier información existente en el sistema de información, no es posible que el comportamiento del modelo de negocio se ejecute fuera del núcleo de Monet.

Esto unido a la importancia de un uso óptimo del canal de comunicación, nos hace plantear que todo el modelo usado por Monet Mobile sean ViewModels. Será Back Mobile, una aplicación que se ejecuta en el espacio de negocio, accediendo directamente al núcleo de Monet, la que ofrezca los servicios a Monet Mobile y será esta la encargada de hacer la traducción de los objetos de información de Monet a los ViewModels usados por Monet Mobile.

Esto nos permitirá que con una única llamada realizada por el Presenter correspondiente a una determinada vista, nos traigamos del servidor todos los datos necesitados por la vista, minimizando los datos a transferir, los tiempos de proceso y la memoria usada por los datos en el dispositivo móvil.

5.2. Eficiencia vs Acoplamiento

Los patrones de diseño que usan para crear la aplicación recomiendan el uso de interfaces o clases abstractas para evitar el acoplamiento o las referencias

circulares entre clases. En una aplicación de escritorio para PC o una aplicación web, lo más aconsejable es su debido, ya que el coste computacional es residual, respecto a los beneficios que aporta.

En el desarrollo para móviles y más concretamente el desarrollo de aplicaciones para Android, al estar basado en Java, hay que tener en cuenta las implicaciones que tiene el uso de interfaces o clases abstractas con métodos virtuales. Debido a la implementación que tiene Java para permitir esos mecanismos, se produce una resolución del método que se debe llamar en tiempo de ejecución, es decir, una llamada a un método de una clase no es un puntero directo al código de esa clase. Antes se produce una resolución mediante reflexión, buscando en las tablas que definen las clases, para saber a qué método se ha de llamar realmente. Esto tiene un coste computacional que en un dispositivo móvil es importante, lo que resta velocidad y aumenta el consumo de energía.

En el desarrollo para móviles y más concretamente el desarrollo de aplicaciones para Android, al estar basado en Java, hay que tener en cuenta las implicaciones que tiene el uso de interfaces o clases abstractas con métodos virtuales. Debido a la implementación que tiene Java para permitir esos mecanismos, se produce una resolución del método que se debe llamar en tiempo de ejecución, es decir, una llamada a un método de una clase no es un puntero directo al código de esa clase. Antes se produce una resolución mediante reflexión, buscando en las tablas que definen las clases, para saber a qué método se ha de llamar realmente. Esto tiene un coste computacional que en un dispositivo móvil es importante, lo que resta velocidad y aumenta el consumo de energía.

5.3. Identificación de las actividades

Una vez definida la arquitectura base que se usará para implementar Monet Mobile, en primer lugar será necesario identificar que actividades o vistas serán necesarias, así como que Presenters y controladores serán necesarios para la aplicación.

De esta forma podemos identificar tres controladores especializados en un subconjunto de modelos del dominio, teniendo así:

- HubPortLayer. Encargado de las conexiones con las unidades de negocio.

- NodeLayer. Encargado de las operaciones con nodos.
- TaskLayer. Encargado de las operaciones con tareas.

A continuación, las actividades o vistas con sus respectivos Presenters, agrupados por el subconjunto de modelos de dominio que usan principalmente, si bien esto no significa que los Presenters hagan uso de otros controladores.

- HubPorts
  - Lista de Hubports
  - HubPort
  - Creación\edición de un HubPort
- Nodos
  - Escritorio
  - Contenedor
  - Colección
  - Formulario
  - Documento
- Tareas
  - Lista de tareas
  - Tarea
  - Historial
  - Estado

Con todo ello, la arquitectura resultante queda como se puede observar en la Figura 16, donde se refleja toda la arquitectura. Si bien las capas superiores de actividades y Presenters sobre mayor explicación acerca de sus responsabilidades, ya que no van más allá de lo descrito de los patrones de diseño en los que se basan su diseño, el resto de la arquitectura será descrita a continuación.

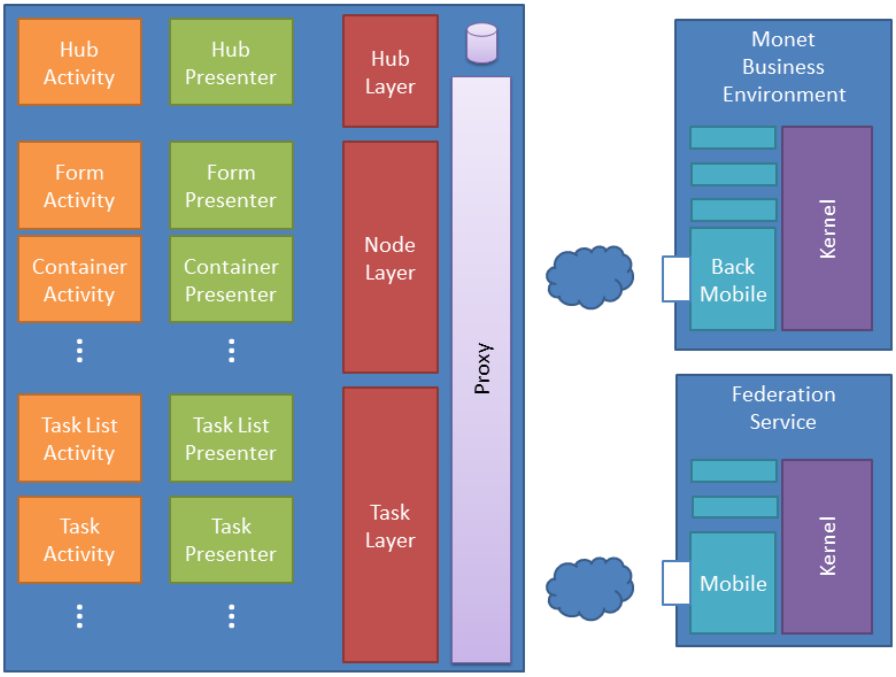

Figura 16: Arquitectura del sistema

5.4. Capa de lógica de negocio

La capa de lógica de negocio, representadas en la ilustración Ilustración 16 por los rectángulos de color rojo, está compuesta por una serie de sub capa por cada tipo de objeto que se maneja. Así podemos ver que existen el HubLayer, NodeLayer y el TaskLayer. Cada una de estas capas está encargada de realizar la función de cacheo de datos, petición de los datos a la capa Proxy o a la base de datos local disponible en el dispositivo. Para optimizar el uso de los canales de comunicación, la capa de negocio cachea las peticiones de datos realizadas por los presenters, así como adelantan y mantienen en memoria datos relacionados con los pedidos que si bien no se devuelven en esa misma llamada, si se solicitarán en una llamada próxima. Por ejemplo cuando se solicita un nodo, la capa almacena no solo ese nodo en cache una vez lo ha solicitado, sino que mantiene datos acerca del contenido y los nodos relacionados con este mismo.

5.5. Proxy

Es una capa que sirve de sustrato de bajo nivel a las capas superiores de la aplicación y se ocupa de todas las comunicaciones con los diferentes servicios que se ofrecen desde la aplicación web.

Esta capa permite de forma transparente a las capas superiores transferir objetos desde la aplicación móvil hasta la aplicación web y traer de vuelta los objetos que los servicios que ofrece Monet. De forma interna, esta interacción se hace a través de la serialización de los objetos a XML para realizar las peticiones, así como la deserialización del XML a objetos de la aplicación. El mecanismo luego se basa en peticiones RESTfull hace el servidor. Por qué usar RESTfull y no usar los estándares del W3C para servicios web, los llamados WS*, es porque RESTfull permite clientes más ligeros que WS*.

Los mensajes SOAP usados por los servicios WS* son muy pesados e introducen mucho overhead al paso de mensajes, además que añaden inflexibilidad para el envío o recepción de datos binarios, complicando para ello aún más los mensajes SOAP y aumentando su peso sustancialmente. Todo sin añadir el soporte para la autenticación de WS-Security y otros estándares de seguridad. Para una plataforma móvil, como se ha venido comentando a lo largo de este trabajo, es poco recomendable todo aquello que produzca un coste computacional relativamente alto, debido a la capacidad de proceso de estos dispositivos, el ancho de banda disponible (y el posible coste asociado a este si no existe una tarifa plana asociada), además del quizás más importante cada vez coste en el consumo de la batería del dispositivo (ya que las redes y los dispositivos son cada vez más potentes, pero no así las baterías).

La arquitectura de la capa Proxy permite tener todas las ventajas de los servicios WS* desde el punto de vista de mantenimiento del código, ya que la cada automáticamente serializa/deserializa los datos entre cliente y servidor. En el caso de requerir una nueva operación, un nuevo parámetro o cambiar un objeto de información de la aplicación, solo será necesario hacer el citado cambio en la clase java correspondiente. Las clases que se usan por tanto para la transferencia de datos entre la aplicación móvil y la aplicación web son compartidas por ambas, para facilitar el mantenimiento. A nivel de organización de código, todas estas clases residen en un proyecto Java de librería (Shared Mobile Model), al cual referencian tanto el proyecto de la aplicación móvil, como la aplicación web.

La arquitectura de la capa Proxy permite tener todas las ventajas de los servicios WS* desde el punto de vista de mantenimiento del código, ya que la cada automáticamente serializa/deserializa los datos entre cliente y servidor. En el caso de requerir una nueva operación, un nuevo parámetro o cambiar un objeto de información de la aplicación, solo será necesario hacer el citado cambio en la clase java correspondiente. Las clases que se usan por tanto para la transferencia de datos entre la aplicación móvil y la aplicación web son compartidas por ambas, para facilitar el mantenimiento. A nivel de organización de código, todas estas clases residen en un proyecto Java de librería (Shared Mobile Model), al cual referencian tanto el proyecto de la aplicación móvil, como la aplicación web.

5.6. Back Mobile

Back Mobile es el soporte en la aplicación web de Monet que ofrece acceso a las aplicaciones móviles a la funcionalidad del Kernel de Monet. Back Mobile ofrece los servicios de autenticación y seguridad, así como la capa de serialización/deserialización y transformación de los datos que son enviados por la aplicación móvil.

El FrontController es la capa encargada de implementar esta funcionalidad, basándose en el patrón de diseño de mismo nombre, usado para controlar el punto de entrada a los servicios ofrecidos por BackMobile. La ventaja de uso de este patrón es que permite un control en un único punto de la seguridad, así como los ya citados servicios de bajo nivel de deserialización de los objetos asociados a las peticiones y la serialización de los objetos que se envían como respuesta, facilitando el mantenimiento de la aplicación. Una vez esta capa deserializa y autentica correctamente la petición, esta se pasa a la acción que corresponde.

Las acciones realizarán la función particular, para ello en primer lugar hacen una adaptación desde los objetos del Shared Mobile Model a las clases que maneja el Kernel de Monet. Realiza las llamadas necesarias al Kernel de Monet y por último realiza una nueva traducción, pero esta vez en dirección inversa, para generar los objetos de Shared Mobile Model que espera recibir el cliente móvil.

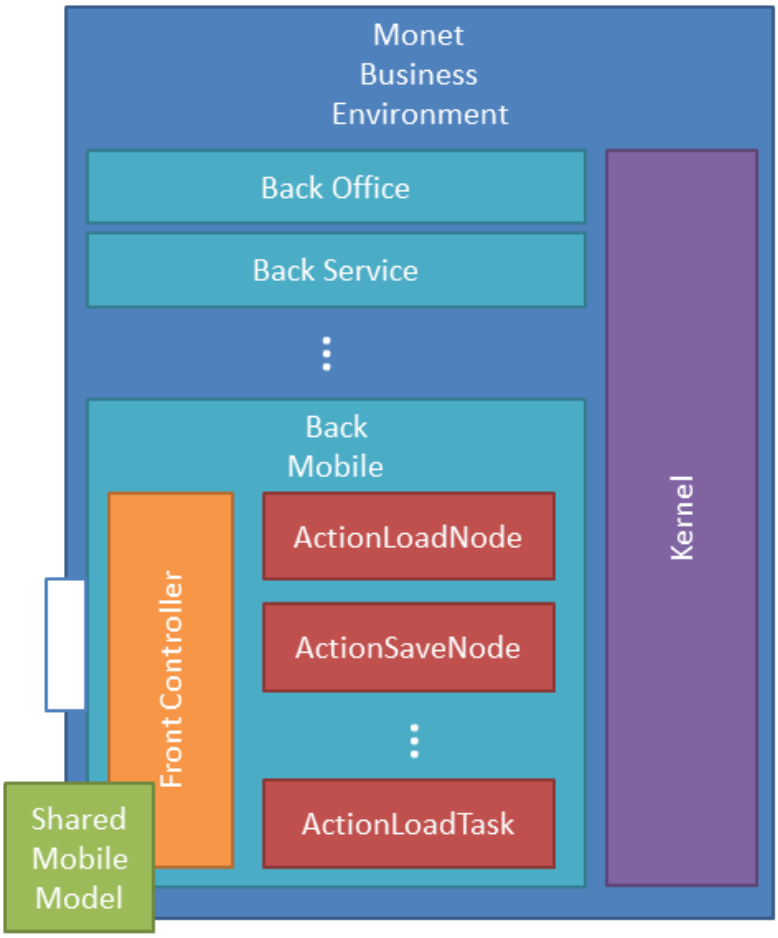

Figura 17: Arquitectura de Back Mobile

6. Diseño de pruebas

Como último punto para describir el trabajo realizado y la arquitectura plateada, vamos a detallar como se han diseñado las pruebas unitarias para la aplicación, así como justificar las decisiones tomadas a lo largo del desarrollo del trabajo.

Antes de comenzar a describir la solución, es necesario aclarar ciertos conceptos

típicos que se manejan en el dominio de las pruebas unitarias. Lo primero es describir que es una prueba unitaria, que no es más que una prueba que se le realiza a un determinado método de una clase, para probar un cierto comportamiento de esta. La prueba debe estar aislada del resto del sistema, siendo ideal que lo único necesario para ejecutar dicha prueba sea únicamente la clase a probar. El objetivo de este requisito es que se puede comenzar a implementar pruebas antes de tener todo el software construido. Además, esto permite asegurar que si una prueba falla, este fallo se encuentra en la clase objetivo de la prueba y no en cualquier otra clase y/o librería.

Para poder realizar pruebas unitarias necesitaremos por tanto que las clases que desarrollamos no obtengan instancias de objetos de terceras clases directamente, ya que esto rompe por completo con el requisito comentado anteriormente, ya que será imposible probar esa clase sin tener la clase dependiente ya desarrollada. Es por tanto importante que los objetos requeridos por una clase le sean proporcionados desde el exterior.

La inyección de dependencia es una tendencia actual que pretende dar solución a este tipo de aproximaciones, donde es un ente exterior, a modo de dios, quién conoce todos los objetos y clases existentes, proporcionándole todos los objetos necesarios para que una clase funcione[8, 9]. De esta forma, es posible inyectarle a la clase que vamos a probar implementaciones artificiales, lo que se conoce como objetos Mock[4]. Así a una clase, se le inyectarán todas sus dependencias como interfaces. Cuando el software está operando normalmente, los objetos inyectados serán las implementaciones de las interfaces reales que realizan su cometido de forma normal. Cuando estamos realizando una prueba unitaria, los objetos que inyectados serán objetos Mock, con implementaciones simples que simulan de forma artificial a la clase real, por ejemplo, devolviendo un valor concreto siempre.

Tanto para la inyección de dependencia como para la generación de los objetos Mock, existen Frameworks, que permiten su implementación de forma sencilla.

En el caso de los Frameworks de objetos Mock, usamos el framework de Mockito (http://mockito.googlecode.com). Este potente framework de creación de objetos Mock, permite la creación de objetos Mock en tiempo de ejecución, donde podemos codificar que respuestas deben dar los objetos Mock ante determinadas llamas a alguno de sus métodos. El framework tiene la gran capacidad de

poder generar estos objetos Mock basándose directamente en una clase normal, no siendo necesario que nos basemos en una interfaz. Esto es un requisito que como comentamos en la sección anterior, es de vital importancia para un buen desempeño de la aplicación en un dispositivo móvil, al no obligarnos acceder a los objetos mediante interfaces, lo que es más costoso computacionalmente.

Sin embargo, a la hora de implantar la otra pieza necesaria para una correcta implementación de las pruebas unitarias, la inyección de dependencia, si que presenta este problema, ya que todos los Frameworks de inyección de dependencia requieren el uso intensivo de reflexión en tiempo de ejecución para obtener datos de los objetos Java y poder así inyectarles convenientemente las implementaciones. Además, lo ideal en el uso de este tipo de Frameworks es el acceso a la implementación a través de una interfaz, cosa que queremos evitar por cuestiones de rendimiento.

La solución alternativa al uso de un Framework de inyección de dependencia es que todas las clases siempre ofrezcan constructores alternativos donde se les pueda proporcionar los objetos Mock. En el constructor por defecto, el objeto obtendrá las instancias por su cuenta. El principal problema de esta aproximación es por un lado la disciplina que hay que mantener a la hora de desarrollar, cuestión abordable al tratarse de una aplicación para móvil, que no tendrá un tamaño muy grande. Por otro lado, es necesario tener código que no va a ser usado nunca en el compilado de la aplicación, lo que perjudica la ejecución del código al consumir más memoria, ser más lentas las cargas de la clase, etc. Para solucionar este inconveniente, hemos usado el producto ProGuard.

ProGuard es una herramienta que entre sus características está la capacidad de ofuscar el código Java de forma que sea difícil decompilarlo para extraer código Java a partir del bytecode. Adicionalmente a esta característica, esta herramienta tiene la capacidad de eliminar clases, métodos o atributos no usados, haciendo un análisis de todo el código una vez compilado. Haciendo uso de esta característica, podemos eliminar los constructores y/o métodos usados para inyectar los objetos Mock en las pruebas de todas las clases, teniendo un código óptimo para la puesta en producción de la aplicación.

Una vez hemos planteado lo que son las pruebas unitarias y el planteamiento general que seguiremos al ahora de implementar las pruebas de la aplicación, vamos a mostrar a como de ejemplo como se estructuran las pruebas. Estas

pruebas nos permitirán probar la mayor parte de la aplicación sin la necesidad de lanzar la aplicación mediante un emulador.

En la Figura 18 se muestra el diagrama de clases necesario para implementar las pruebas unitarias de un Presenter, en el caso de ejemplo el Presenter de formularios. La clase de pruebas unitarias "FormPresenterTest", instancia la clase "FormPresenter". Para ello, necesitará sendas instancias de las clases "FormActivity" y "NodeLayer", esto es, la capa superior y la capa inferior. Para ello, creará objetos Mock para cada clase, condicionando la respuesta a determinados métodos en función de la prueba concreta a realizar. Con ello podremos probar casi toda la lógica asociada a la vista, sin necesidad de su prueba con el emulador, así como también podemos evitar tener las capas inferiores y por tanto un servidor con Monet en ejecución.

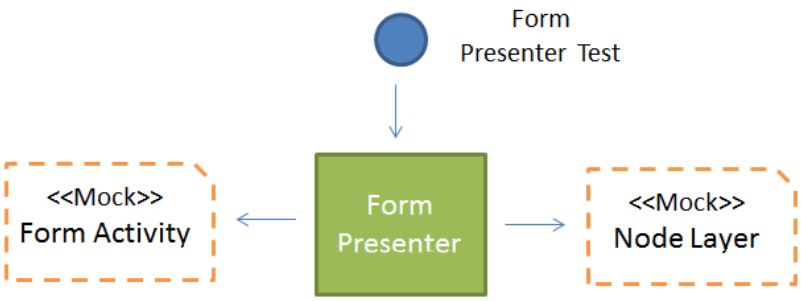

Figura 18: Test unitario al Presenter de formularios

En la Figura 19, se muestra de forma similar el test unitario a la capa de lógica de negocio de nodos, donde en este caso la clase que es necesaria simular mediante objeto Mock es la capa inferior Proxy. En este caso nos evitará tener que disponer del servidor web para poder realizar la prueba de la capa de lógica de negocio.

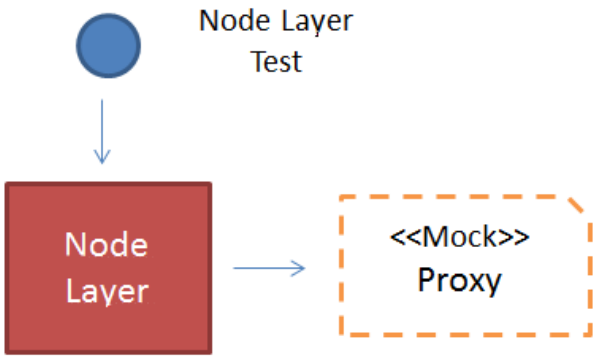

Figura 19: Test unitario a la capa de lógica de negocio de nodos

Los diagramas mostrados no se muestran todas las clases que son necesarias simular mediante objetos Mock por claridad. Lógicamente será necesario instanciar objetos Mock de otras clases del sistema Android, como son aquellos que gestionan la base de datos interna o clases de utilidad varias para obtener recuersos como cadenas o datos de configuración. Estas son necesarias ya que el objetivo es que los test unitarios se puedan ejecutar fuera del emulador. Si bien existe soporte para test unitarios en el paquete de herramientas que acompaña al kit de desarrollo de Android, la velocidad de ejecución de estos es muy lenta, requiriendo el despliegue del código en el emulador y su ejecución. Creando objetos Mock, podemos evitar esto ejecutando las pruebas unitarias con JUnit en nuestra máquina de desarrollo.

Parte IV

Resultados

Como resultados de este trabajo, se ha obtenido una arquitectura para el desarrollo de aplicaciones ligeras en dispositivos móviles, aplicable a multitud de aplicaciones diferentes. La ventaja de esta arquitectura radica en la facilidad de adaptar la parte cliente, al ser muy ligera y no concentrar el núcleo de la lógica de negocio en la aplicación móvil para luego sincronizar los datos.

Como consecuencia de esta arquitectura, se ha desarrollado una aplicación que haciendo uso de esta arquitectura, implementa un cliente móvil para la plataforma Monet. La aplicación, si bien no está desarrollada por completo (para evitar una dedicación excesiva en tiempo a este trabajo), al igual que la plataforma Monet que se encuentra en constante evolución, se encuentra en un punto de desarrollo bastante maduro. La aplicación debería poder ser adaptada a otras plataformas móviles de forma sencilla, gracias a las características de su arquitectura y el uso del patrón de diseño Model-View-ViewModel-Presenter, o la comunicación basada en XML y servicios Restfull, por el que los modelos de vista se hacen llegar desde la aplicación web Back Mobile hasta el cliente móvil.

Con este trabajo se ha conseguido depurar y evolucionar el concepto de Mapas de Trabajo, que cuando se inició este trabajo se encontraba en un estado bastante primitivo. Si bien la idea germen ya estaba planteada desde un inicio, surgieron distintos caminos posibles a la hora de conseguir llegar a implementar esa idea. Después de muchas iteraciones y pruebas de prototipos, se llegó al concepto de bloqueos y sus distintas variantes, las cuales no son sino el inicio de las posibilidades de esta tecnología.

Por último, si bien este trabajo no ha tenido una alta carga de investigación en su desarrollo, más allá del desarrollo de los mapas de trabajo, sí que es un paso importante para el desarrollo de nuevas líneas de investigación. Los teléfonos inteligentes tienen hoy en día una característica muy importante y es que se han convertido en una auténtica sonda del medio en el que se encuentra. Su cada vez mayor cantidad de distintos tipos de sensores como sensores de luz, de presencia, temperatura, posición, aceleración, orientación, localización, imagen,

sondo y un largo etc.; ofrecen una enorme cantidad de posibilidades que se han de estudiar para poder sacar partido a toda esta información e incorporarla al sistema de información.

Parte V

Conclusiones y Líneas Futuras

Las conclusiones que puedo extraer del desarrollo de este trabajo son varias. La de mayor importancia es que gracias a este trabajo he podido comprender y compartir la visión del desarrollo dirigido por modelos, como una metodología de desarrollo muy potente. En mi corta experiencia en el desarrollo de software a medida, he podido comprobar que incluso en el espacio de tiempo que transcurre desde que se realiza el análisis del proyecto hasta que su desarrollo se termina, los requisitos cambian con mucha facilidad. Más aún cuando los usuarios empiezan a experimentar con la aplicación, así como futuros evolutivos de la aplicación. Estos cambios suelen conllevar muchos quebraderos de cabeza, ya que los cambios a realizar son muchos en muchos sitios, lo que introduce errores en la aplicación. Se puede llegar incluso a problemas de arquitectura, en los que se encuentra que la arquitectura de la aplicación no encaja con los nuevos requisitos. Esto suele ser la principal causa de retrasos en los proyectos o que los costes de mantenimiento se disparen. Con el desarrollo dirigido por modelos, al trabajar a un nivel de abstracción mayor, los cambios se pueden realizar con una facilidad sorprendente, reduciendo los costes del mantenimiento de forma considerable.

Este campo del desarrollo dirigido por modelos es sin duda un campo por el que se habrán infinidad de líneas de investigación, desde la interoperabilidad con sistemas existentes, los mapas de trabajo o la automatización de ciertas tareas haciendo uso de agentes inteligentes que puedan realizar ciertas tareas dentro del sistema de información.

Los mapas de trabajo son una buena prueba de las posibilidades que están por explorar referentes al desarrollo dirigido por modeos, ofreciendo una alternativa a la visión de los flujos de trabajo "tradicionales" (workflows), permitiendo abordar situaciones inesperadas gracias a su alta flexibilidad. Esta es una de las líneas futuras de trabajo que se abre y que habrá que estudiar su comportamiento en situaciones reales para verificar y evolucionar el concepto.

Desde un punto de vista de la plataforma móvil, caben líneas futuras de trabajo como la adaptación de la Monet Mobile a distintos sistemas operativos.

Estudiando esta línea de trabajo es posible incluso el desarrollo de una plataforma para el desarrollo de aplicaciones para dispositivos móviles basado en el desarrollo dirigido por modelos sin que esta necesite un apoyo de un servidor externo.

Para llegar a este punto, sería necesario desarrollar aspectos que no se han podido desarrollar por cuestiones de tiempo en este trabajo como por ejemplo el trabajo fuera de línea. En el actual estado, el trabajo fuera de línea presenta la complicación de la ejecución del comportamiento asociado a las acciones del usuario. Este comportamiento es necesario que se ejecute en el servidor dado que a nivel operativo, ese comportamiento para su ejecución puede depender del acceso a otros objetos dentro del sistema de información, los cuales pueden no estar disponibles en la cache del dispositivo, siendo imposible su ejecución.

Una posible vía de investigación es la posibilidad de que existan pequeños modelos que se ejecuten por completo sobre el dispositivo móvil, como si de una pequeña unidad de negocio se tratara. Luego cuando esta mini-unidad de negocio quiere interactuar con la unidad de negocio maestra para interactuar con la unidad de negocio maestra, podría utilizar los canales de interoperabilidad. Esto tiene como inconveniente que el modelo de la organización quizás tenga que dividirse de forma un tanto artificial o quizás no. Una ventaja de hacer esta división es que es posible acercar el lenguaje del metamodelo de Monet para Monet Mobile y dar soporte a los sensores como parte activa del sistema de información, la cual hace sus aportes de información al sistema de forma automática o semi-automática.

Esta aproximación nos acercaría a tener una plataforma que también nos sirva para el desarrollo de aplicaciones para dispositivos móviles, independiente del sistema operativo y que podría funcionar de forma autónoma, con posibilidades de consumir información de fuentes externas como pueden ser otras aplicaciones o fuente de información en la red.

Índice de figuras

Referencias

[1] J. Andrzejewski. Designing a user interface for smartphones. a balance between the pragmatic and the hedonic dimension of usability–a case study. 2004.

[2] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[3] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *Computer*, 39(2):33–40, 2006.

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

[5] S. Beydeda, M. Book, and V. Gruhn. *Model-driven software development*. Springer Verlag, 2005.

[6] J.P. Boodhoo. Design patterns-model view presenter. *MSDN Magazine-Louisville*, pages 91–100, 2006.

[7] H. Cabrera and J. Juan. Implantación del ebussiness en pequeñas organizaciones con una orientación al modelado y la interoperabilidad. 2011.

[8] M. Fowler. Inversion of control containers and the dependency injection pattern. *Actualizado el*, 23.

[9] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Professional, 2003.

[10] B. Gadhavi. *Analysis of the Emerging Android Market*. PhD thesis, San José State University, 2010.

[11] S.Y. Hashimi, S. Komatineni, and D. MacLean. *Pro Android 2*. Springer, 2010.

[12] J. Miller, J. Mukerji, et al. Model driven architecture (mda). *Object Management Group, Draft Specification ormsc/2001-07-01*, 2001.

[13] R. Rogers. *Android application development*. O'Reilly, 2009.

[14] D.C. Schmidt. Model-driven engineering. *IEEE computer*, 39(2):25–31, 2006.