INTERACCION
Revista Digital de AIPO
Asociación Interacción Persona-Ordenador
Vol. 1, No 1 (2020)
ISSN 2695-6578

UML2App: Avanzando en la generación automática de interfaces de usuario para dispositivos móviles

UML2App: Towards the automatic generation of user interfaces for mobile devices

Víctor López-Jaquero
LoUISE Research Group
Univ. de Castilla-La Mancha
02071 Albacete, España
VictorManuel.Lopez@uclm.es

Pascual González
LoUISE Research Group
Univ. de Castilla-La Mancha
02071 Albacete, España
Pascual.Gonzalez@uclm.es

Francisco Montero
LoUISE Research Group
Univ. de Castilla-La Mancha
02071 Albacete, España
Francisco.MSimarro@uclm.es

José Pascual Molina
LoUISE Research Group
Univ. de Castilla-La Mancha
02071 Albacete, España
JosePascual.Molina@uclm.es

Recibido: 19.09.2019 | Aceptado: 15.12.2019
DOI: https://doi.org/10.65234/interaccion.3

Palabras Clave

Diseño de interfaces de usuario basado en modelos
Desarrollo dirigido por modelos
Aplicaciones móviles
Perfil UML
Android

Resumen

El desarrollo de aplicaciones para dispositivos móviles es cada vez más imprescindible para cualquier empresa de software. Sin embargo, dicho desarrollo presenta retos adicionales frente al desarrollo tradicional de aplicaciones de escritorio. Las características y capacidades de interacción difieren de las interfaces de escritorio, lo cual plantea la necesidad de incorporar otras maneras distintas de interactuar en los dispositivos móviles. Otro aspecto relevante en el desarrollo de dichas aplicaciones es la fragmentación existente, donde tanto la variabilidad en los sistemas operativos que usan, y por lo tanto en las guías de diseño para dichos sistemas, como las propias características físicas de dichos dispositivos hacen que mantener desarrollos que estén dirigidos a distintas plataformas móviles sea complejo. En este trabajo se presenta una aproximación dirigida por modelos orientada a la generación de aplicaciones móviles que trata de aliviar los problemas anteriormente identificados, a través de una generación automática basada en un perfil UML y modelos habitualmente usados en el diseño de interfaces de usuario basadas en modelos. De esta manera, se persigue impulsar la reutilización de la mayor parte de los modelos creados en la generación para distintas plataformas. Actualmente, la implementación en los casos de estudio se ha centrado en el marco de trabajo de Android.

Keywords

Model-Based User Interface Design
Model-Driven Development
Mobile Apps
UML Profile
Android

Abstract

The development of apps for mobile devices is becoming a must for every software company. However, this development poses additional challenges compared to the traditional desktop application development. The characteristics and interaction capabilities are different from the ones found in desktop user interfaces. Therefore, different ways of interacting must be considered to provide a natural way to interact with the mobile device. Another relevant issue in the development of such apps is the existing fragmentation, where the variability in the operating systems they use, and therefore the variability in the design guidelines, as well as their different hardware characteristics, makes it hard to maintain developments targeting several platforms. In this work, a model-driven approach is presented for the generation of mobile apps relying on a UML profile and in some models widely used in model-based user interface development. Thus, we aim at fostering reusability of most parts of the models designed for several platforms. Currently, the implementation targets Android framework.

1. Introducción

El uso de dispositivos móviles se ha disparado a nivel mundial. En concreto, España lidera las estadísticas en cuanto al uso de móviles con un 88% de usuarios únicos, frente a la media mundial global del 66% % (Ditrendia, 2019). Este imparable ascenso en el uso de los dispositivos está provocando que el foco del desarrollo de aplicaciones se mueva cada vez más hacia el mercado de dispositivos móviles. Sin embargo, el desarrollo de aplicaciones para móviles supone un reto en múltiples sentidos (Joorabchi, Mesbah, & Kruchten, 2013). En primer lugar, las plataformas más populares difieren de manera amplia tanto en aspectos hardware como software. Al mismo tiempo, la creciente demanda requiere aplicaciones que funcionen en diferentes plataformas, al menos en las más populares (Android, iOS y Windows Phone). Sin embargo, las herramientas que permiten crear aplicaciones multiplataforma no son capaces de aprovechar al máximo los recursos específicos ofrecidos por cada plataforma. Todo ello abre las puertas a una estrategia de desarrollo diferente como es el Desarrollo Dirigido por Modelos (MDD). En este contexto, este trabajo presenta una aproximación dirigida por modelos para el desarrollo de apps para dispositivos móviles llamada UML2App, que trata de avanzar en la búsqueda de soluciones que permitan generar aplicaciones usables para distintas plataformas de dispositivos móviles de una manera, además, eficiente.

El resto del documento se estructura del siguiente modo: en la Sección 2 se tratarán las distintas aproximaciones existentes para el desarrollo de apps para dispositivos móviles, incluyendo aquellas dirigidas por modelos. En la Sección 3 se presenta UML2App, primero describiendo su arquitectura general y luego cada uno de sus componentes. La Sección 4 ilustra un caso de estudio que permite entender de manera más precisa cómo funciona UML2App. Las lecciones aprendidas durante el desarrollo de la herramienta se presentan en la Sección 5, finalizando el artículo con algunas conclusiones y trabajo futuro.

2. Desarrollo de aplicaciones móviles

Como plantean Biørn-Hansen et al. (Biørn-Hansen, Grønli, Ghinea, & Alouneh, 2019) tradicionalmente el desarrollo de aplicaciones para móviles se ha realizado utilizando herramientas diseñadas específicamente para cada una de las plataformas existentes (iOS para Apple; Android para Google o Windows 10 Mobile para Microsoft). Este enfoque, aunque hace difícil la portabilidad de aplicaciones a otras plataformas, permite aprovechar al máximo las peculiaridades que cada entorno ofrece, obteniendo mejores resultados en cuanto a rendimiento (Willocx, Vossaert, & Naessens, 2015) o experiencia de usuario (Dalmasso, Datta, Bonnet, & Nikaein, 2013). En todo caso, la dificultad de creación de aplicaciones multiplataforma abre nuevos escenarios para otras aproximaciones. Una alternativa es el uso de las los denominados *cross-platform mobile development frameworks*, los cuales permiten la reutilización de código entre diferentes plataformas, facilitando de este modo la portabilidad entre distintas plataformas y, con ello, una gran mejora en coste-efectividad. Como comentan Biørn-Hansen et al. (Biørn-Hansen, Grønli, & Ghinea, 2018), dentro de este ámbito existe un amplio abanico de propuestas que hacen difícil seleccionar la más adecuada. Para facilitar su elección dentro de su estudio, plantean una taxonomía y estado de investigación de las propuestas *cross-platform development* que permite clasificar las propuestas existentes en cinco grandes grupos: híbridas; interpretadas; compilación cruzada; apps web progresivas; y dirigidas por modelos. Las propuestas híbridas (Hybrid) se apoyan en el uso de tecnologías web convencionales, incluyendo HTML, CSS y JavaScript. Para ello utilizan internamente un componente WebView que hace la función de un navegador web incrustado. Un ejemplo de este tipo de frameworks es PhoneGap o Ionic Framework. Las propuestas clasificadas dentro de interpretadas (Interpreted) permiten renderizar directamente componentes de la interfaz nativa en la pantalla. Un ejemplo de aplicaciones de este tipo son React Native o Adobe AIR. En el siguiente grupo de frameworks, de compilación cruzada o Cross-Compiled, las aplicaciones desarrolladas no dependen de los componentes de WebView ni de intérpretes en el dispositivo para la representación de la interfaz de usuario o la comunicación con las características de la plataforma y del dispositivo. En su lugar, utilizan un lenguaje, como C#, para su desarrollo accediendo a las funcionalidades ofrecidas por el SDK de las diferentes plataformas. Por lo tanto, no hacen uso de la capa puente como en las aproximaciones anteriores. La siguiente aproximación, denominada apps web progresivas *(Progressive Web Apps)*, es en esencia una aplicación Web con capacidades mejoradas, intentando que las aplicaciones Web tengan un aspecto similar (*look & feel*) a las aplicaciones nativas utilizando para ello HTML y CSS. A su vez, esta aproximación permite realizar una instalación del sitio Web en el dispositivo de tal modo que permite su uso sin conexión u offline. Por último, la aproximación dirigida por modelos se apoya en el uso de modelos independientes de la plataforma definidos a diferente nivel de abstracción que finalmente pueden convertirse en código ejecutable para una determinada plataforma. En todo caso, esta aproximación, aunque tiene una notable presencia en el ámbito de la investigación, su adopción no parece tener la misma penetración en el ámbito de la industria (Biørn-Hansen et al., 2019). A pesar de esto, también es relevante resaltar que dentro de las herramientas analizadas en el informe Gartner

sobre "Magic Quadrant for Mobile App Development Platforms" (Wong, Baker, Leow, & Resnick, 2018) encontramos en algunas de las propuestas (Appian, GeneXus o Mendix) referencias al uso de Model-Driven Development (MDD).

2.1 Desarrollo dirigido por modelos de interfaces de usuario

El Desarrollo Dirigido por Modelos (MDD) se basa en el uso de modelos para el desarrollo de sistemas, en vez del más tradicional uso de un lenguaje de programación. Se trata de llevar los modelos a un primer plano, para conseguir beneficios como la generación del código de la aplicación (Selic, 2003).

Más concretamente, en el mundo de las interfaces de usuario, se lleva usando desde principios de los 90 con el nombre de Diseño de Interfaces de Usuario Basado en Modelos (MB-UIDE) (Paternò, 1999). Aunque no existe un estándar sobre qué modelos usar para describir una interfaz de usuario, sí parece que se ha adoptado como estándar de facto el marco de trabajo de *Cameleon* (Calvary et al., 2003) (véase Figura 1). Dentro de este marco se proponen cuatro niveles de abstracción. El más abstracto es Tareas & Conceptos (*Tasks & Concepts*). En este nivel se recogen dos de los modelos más ampliamente usados en la generación de interfaces de usuarios: el de tareas y el de dominio (conceptos). El modelo de tareas describe cuáles son las tareas que puede realizar el usuario del sistema a través de la interfaz de usuario, así como las relaciones temporales existentes entre dichas tareas. Por ejemplo, podríamos especificar que la tarea de conectarse al sistema consiste en tres tareas (introducir nombre de usuario, introducir contraseña y validar) y que dichas tareas deben realizarse de manera secuencial, en este caso. Por otro lado, el modelo de dominio describe qué información debe manejar la interfaz de usuario para ser capaz de realizar las tareas especificadas en el modelo anterior. Distintas representaciones de este modelo son posibles, siendo la más habitual una notación basada en los modelos de clases de UML. Siguiendo con el ejemplo anterior, necesitaríamos una clase *Usuario* que recoja los datos del usuario que se quiere conectar al sistema. En nuestro pequeño ejemplo, serían necesarios dos atributos: uno para representar el nombre de usuario y otro para la contraseña, así como un método que lanzará la validación.

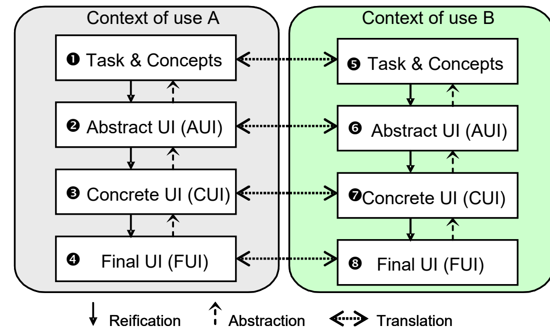


Figura 1. Marco de referencia de Cameleon (Calvary et al., 2003).

El siguiente nivel es la Interfaz de Usuario Abstracta (AUI). En este nivel se describe cómo será la presentación de la futura interfaz de usuario, usando para ello Objetos de Interacción Abstractos (AIO). Esta representación de la presentación de la interfaz de usuario presenta dos aspectos fundamentales: por un lado, es una representación independiente de la plataforma destino para la que estamos diseñando, y por otro lado es independiente de la modalidad de interacción destino, es decir, no importa si se está describiendo una interfaz vocal o gráfica, por ejemplo. El siguiente nivel de Interfaz de Usuario Concreta (CUI) también representa la presentación de la interfaz de usuario, pero esta vez usando como elementos fundamentales los Objetos de Interacción Concretos (CIO). De manera similar al AIU, este nivel también es independiente de la plataforma destino a la que va dirigida. Sin embargo, en este caso no es independiente de la modalidad o modalidades para las que se está diseñando. Para completar los niveles de abstracción del marco de trabajo, la Interfaz de Usuario Final (FUI) representa el propio código del programa, ya sea interpretado (p.ej. HTML) o compilado (Java, C#, etc).

Adicionalmente, en la Figura 1 aparece el concepto de Contexto de Uso. Dicho concepto nos permite describir en qué condiciones se usará la aplicación que estamos diseñando. Habitualmente, el contexto de uso se considera que engloba la plataforma (tanto software como hardware) donde se ejecuta la aplicación, el entorno dentro del cual se ejecuta (en casa, en la oficina, en la calle, etc.) y las características del usuario que hará uso de la aplicación. El Contexto de Uso nos facilita información que permite diseñar una aplicación mejor adaptada a las condiciones en las que será usada.

El marco también plantea tres tipos de transformaciones entre dichos niveles de abstracción: reificación, abstracción y translación. Reificación se da cuando se pasa de un nivel a otro nivel menos abstracto. Por ejemplo, a partir del nivel de Tareas & Conceptos se puede generar el AUI. Abstracción es el camino opuesto y, finalmente, translación es la transformación para adecuar un modelo de un nivel diseñado para un contexto de uso A al mismo nivel en un contexto de uso B. Por ejemplo,

podríamos transformar un CUI para un contexto de uso donde se usa la modalidad gráfica en otro CUI para un contexto de uso donde se usa la modalidad vocal.

El presente trabajo se basa en el primer nivel de dicho marco para la descripción de la aplicación a generar, como veremos más adelante. Es decir, se parte de los modelos de tareas y dominio para producir la interfaz de usuario de una manera automática mediante transformación aplicando operaciones de reificación.

Para la representación de los modelos requeridos existen multitud de metamodelos (Guerrero, González-Calleros, Vanderdonckt, & Muñoz Arteaga, 2009), habitualmente basados en XML, siendo UsiXML (Limbourg, Vanderdonckt, Michotte, Bouillon, & López-Jaquero, 2005) uno de los más extendidos. A continuación, se mostrará una revisión de aquellas aproximaciones dirigidas por modelos que se han propuesto para la generación de aplicaciones móviles que se han considerado más relevantes para el trabajo presentado.

2.2 Desarrollo de aplicaciones móviles dirigido por modelos

En el estudio de Biørn-Hansen et al. (Biørn-Hansen et al., 2018) podemos encontrar un conjunto de propuestas que se basan en cierta medida en la propuesta de desarrollo de aplicaciones dirigido por modelos. Como comentan Umuhoza y Brambilla (Umuhoza & Brambilla, 2016) estas propuestas pueden clasificarse en dos grandes grupos: Propuestas de investigación; y Soluciones comerciales. Dentro de las primeras podemos encontrar propuestas que todavía no han dado el salto al ámbito comercial. Dentro de este grupo podemos encontrar propuestas como MD^2 (Heitkötter, Majchrzak, & Kuchen, 2013), Applause (Behrens, 2010) o MAG (Usman, Iqbal, & Khan, 2014). La primera de ellas, MD^2 se apoya en el patrón MVC *(Model-View-Controller)* para estructurar su propuesta de desarrollo dirigido por modelos para aplicaciones de negocio. Para ello, define un lenguaje específico de dominio (DSL) que representa una descripción independiente de la plataforma, la cual será posteriormente transformada en código nativo para Android o iOS. La segunda, Applause (Behrens, 2010), se basa en la generación de aplicaciones centradas en datos y, al igual que la anterior, se apoya en la propuesta de un DSL para describir aplicaciones para móviles, y una serie de generadores de código que utilizarán estas descripciones para generar aplicaciones nativas para las principales plataformas móviles (iOS, Android, Windows Phone). En este caso, el comportamiento casi no es modelado y, sin embargo, las interfaces se modelan con gran precisión.

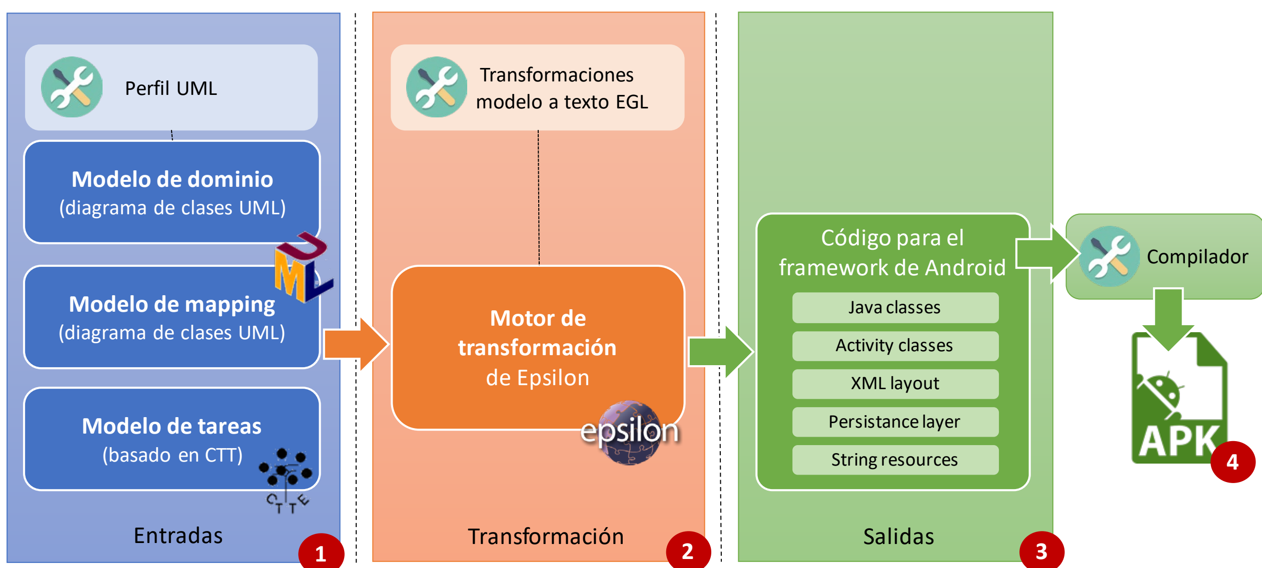


Figura 2. Arquitectura general para la generación de apps móviles.

Finamente, la propuesta MAG (Usman et al., 2014) no define su propio lenguaje específico de dominio (DSL) sino que se apoya en el uso de un conjunto de modelos de UML y un perfil de UML específicamente creado para modelar conceptos específicos de dominio de las aplicaciones móviles. En (Vaupel, Taentzer, Gerlach, & Guckert, 2018) se presenta una aproximación dirigida por modelos para la generación de aplicaciones para iOS y Android.

En este caso, como metamodelo se utiliza eCore para el modelo de dominio, y una serie de DSLs para el resto. Esta aproximación no se basa en la generación de la interfaz de usuario a partir del modelo de dominio y de tareas, sino que es el diseñador el que especifica la interfaz de usuario y su comportamiento directamente.

Para la generación para distintas plataformas se apoya en el concepto de variabilidad habitualmente usado en las aproximaciones dirigidas por modelos de línea de producto.
RAPPT ((Barnett, Avazpour, Vasa, & Grundy, 2019)) propone la generación rápida dirigida por modelos de apps para Android, usando para ello un lenguaje visual específico del dominio (DSVL) y un lenguaje de transformación específico para el dominio (DSTL). El objetivo de esta aproximación es crear prototipos, no apps finales. Finalmente, dentro de este ámbito también encontramos propuestas como LIZARD (Ruiz, Serral, & Snoeck, 2018) que se apoyan en la extensión de *frameworks* de tipo general, como CAMALEON (Calvary et al., 2003), para adaptarlos al dominio del desarrollo de aplicaciones para móviles. En este caso, ofrecen una propuesta que enriquece la transformación que se puede realizar entre la interfaz de usuario concretos (CUI) y la interfaz de usuario final (FUI). Para ello simplifican el framework CAMALEON agrupando el modelo de dominio, de tareas y CUI en un nuevo modelo denominado Interfaz de usuario de alto nivel (HLUI) y eliminando, a su vez, la interfaz de usuario abstracta (AUI) propuesta en CAMALEON.

En el otro ámbito, el de soluciones comerciales, Umuhoza y Brambilla (Umuhoza & Brambilla, 2016) citan, entre otras, propuestas como Mendix o Appian, dos de las que se incluyen en el "Magic Quadrant for Mobile App Development Platforms" (Wong et al., 2018) de Gartner. La primera de las propuestas (Mendix, 2019) se apoya en el uso de herramienta de modelado visual para la definición del modelo de dominio, las interacciones del usuario y la lógica del negocio, los cuales son posteriormente ejecutados en tiempo real. La particularidad de estos modelos es que sirven como medio de comunicación entre los responsables del negocio y los desarrolladores en un proceso de co-diseño. Por otra parte, Appian se ofrece como una herramienta para la gestión de procesos de negocio a través de BPM. Para ello, emplea el marco de trabajo *Appian Self-Assembling Interface Layer* (SAIL) para renderizar aplicaciones y se basa en React Native para generar aplicaciones para móviles y React (JavaScript) para el desarrollo Web. La plataforma funciona en la Appian Cloud y es compatible con Docker y Kubernetes.

3. UML2App: Hacia el desarrollo de aplicaciones móviles dirigido por modelos

Como se ha planteado en las secciones anteriores, el desarrollo de aplicaciones móviles plantea retos adicionales a los que se encuentran habitualmente en el desarrollo de aplicaciones de escritorio o web. Las diferencias, por ejemplo, entre las guías de estilo para el desarrollo de aplicaciones para Android e iOS hace que el uso de una aplicación no nativa para ambas plataformas no sea una solución óptima, especialmente cuando se trata de aplicaciones con un alto grado de interacción.

Una de las principales ventajas de la utilización de una aproximación dirigida por modelos para el desarrollo de una app es la capacidad de compartir la mayoría de los modelos creados para la generación del código de las distintas plataformas soportadas, mejorando con ello la mantenibilidad del producto. Adicionalmente, frente otras aproximaciones basadas en frameworks como Cordova (The Apache Software Foundation, 2019), que usan una misma aplicación HTML en distintas plataformas, con la aproximación dirigida por modelos que se presenta se generan aplicaciones nativas para cada plataforma, lo que permite tener apps que cumplen con las guías de diseño de cada una de las plataformas soportadas. Como principal desventaja está el menor control sobre la interfaz de usuario generada, a pesar de que la interfaz de usuario generada podría ser modificada a posteriori.
Para afrontar este reto, el desarrollo dirigido por modelos puede proporcionar un marco de trabajo común que permite generar las aplicaciones para distintas plataformas, a la vez que contribuye a la aceleración de su proceso de desarrollo.
En este trabajo se presenta un marco de trabajo dirigido por modelos y orientado a la generación de aplicaciones móviles, que actualmente está siendo probada en el desarrollo de aplicaciones para dispositivos basados en Android. A continuación, se describe la arquitectura general usada, cuyos componentes serán detallados en mayor profundidad en sucesivas secciones.

La arquitectura que se presenta (véase la Figura 2) incluye 4 pasos. En el primero se especifica el sistema usando una serie de modelos. En el segundo, dichos modelos se transforman mediante un motor de transformación concreto. A continuación, dichos modelos se integran en un proyecto de Android Studio para ser finalmente compilados en el paso 4.

Veamos estos pasos de una manera más detallada. El primer paso toma como base los modelos de tareas y de dominio (que coincide con el primer nivel de *Cameleon Framework*). El modelo de tareas permite especificar qué tareas podrá el usuario realizar a través del sistema, incluyendo en qué orden se realizan y otras relaciones temporales adicionales, como qué tareas son opcionales, tareas alternativas a otras, etc. Un modelo de tareas basado en CTT (Paternò, 1999) y representado en XML se usa para este propósito. Adicionalmente, el modelo de dominio permite describir qué información se requiere para llevar a cabo las tareas especificadas en el modelo anterior. En este caso, un diagrama de clases de UML se usará para representarlo. Dicho diagrama de clases se complementa aplicándole un perfil UML (Fuentes & Vallecillo, 2004) que permite enriquecer el diagrama de clases con información adicional que ayuda, y en algunos casos resulta imprescindible, durante el proceso de transformación requerido para generar la aplicación móvil.

Aunque existen distintos motores de transformación, se ha optado por Epsilon (Eclipse Foundation, 2019a), por contar con un framework completo y un lenguaje de transformación actualizado y potente. El motor de transformación toma como entrada los dos modelos descritos anteriormente y el conjunto de transformaciones que se desea aplicarles. Como salida producirá el código necesario para la aplicación móvil.
En la siguiente sección, se describe el perfil que se ha desarrollado para la generación de las aplicaciones móviles.

3.1 Desarrollo dirigido por modelos basado en perfiles UML

Un perfil UML permite especificar una serie de estereotipos (véase la Figura 3) que se pueden aplicar a distintos elementos del diagrama. En nuestro caso se aplicarán a distintos elementos de un diagrama de clases de UML. A qué elementos del diagrama de clases es aplicable un estereotipo es una de las características que se especifican durante la creación del perfil, y ayuda a evitar que el estereotipo se aplique de manera errónea a elementos para los que no tiene sentido. Por ejemplo, en la Figura 3 podemos ver una serie de estereotipos que extienden la metaclase *Class*, que representa el concepto de clase en UML. De esta manera, dichos estereotipos sólo podrán ser aplicados a clases en el diagrama, pero nunca a relaciones, por ejemplo. Adicionalmente, los estereotipos pueden incluir atributos, donde se puede especificar información adicional, y que podrá ser añadida para cada uno de los elementos del diagrama de clases sobre los que se aplique el estereotipo. Por ejemplo, el estereotipo *Task* en la Figura 3 incluye un atributo llamado *type* de tipo *TaskType*, en el que para cada clase a la que se le aplique el estereotipo se podrá especificar el tipo de tarea.

Durante el proceso de generación, las transformaciones accederán a los estereotipos y sus atributos para decidir cómo realizar las transformaciones, y tomarán la información que requieran de los atributos que incluyan dichos estereotipos.

Otro de los aspectos clave que nos facilita el uso de perfiles es la validación. Mediante el uso de restricciones escritas, usando el lenguaje OCL *(Object Constraint Language)*, se pueden especificar validaciones que eviten que se puedan dar situaciones anómalas en el diagrama de clases sobre el que se ha aplicado en perfil. Por ejemplo, en el perfil desarrollado, una de las validaciones que existen es que sólo puede existir un elemento al que esté aplicado el estereotipo *AndroidConfiguration*, ya que no tiene sentido que existan dos configuraciones para Android contradictorias. Estas validaciones son importantes, ya que permiten evitar numerosos errores durante el proceso de modelado, que conducirían a una generación de código errónea. Seguidamente, se describe el conjunto de estereotipos desarrollados dentro del perfil UML creado.

3.1.1 Un perfil UML para la generación de apps

El perfil está formado por distintos estereotipos con distintos propósitos. Por un lado tenemos la capa de persistencia que nos permitirá describir qué información queremos guardar en la base de datos, por otro tenemos algunos perfiles destinados a describir la plataforma destino, es decir, formarían parte de lo que anteriormente hemos denominado modelo de contexto de uso, y finalmente existen una serie de estereotipos encaminados a interrelacionar los modelos de tareas y de dominio, realizando lo que se denomina *mapping* entre modelos (Montero et al., 2006a).

Los estereotipos *PersistentDomainClass* (véase Figura 3), *PersistentRelationship* y *PersistentAttribute* (véase Figura 4) están relacionados con la capa de persistencia. *PersistentDomainClass* debe ser aplicado a todas aquellas clases del modelo de dominio que deseamos que sean incluidas en la base de datos, es decir, en la capa de persistencia. Todas las clases con este estereotipo tendrán su correspondiente representación en la base de datos y se generará el código necesario para poder hacer consultas o manipular los datos que dicha clase representa. El código de la capa de persistencia para el caso de Android está basado en *Room* (Google Inc., 2019e). *Room* proporciona una capa de abstracción sobre un gestor de bases de datos relacional local basado en *SQLite*, la cual es especificada añadiendo una serie de anotaciones sobre clases Java donde se especifican los datos, sus tipos de datos y las relaciones entre las distintas

clases. Además, todos aquellos atributos de las clases a las que se les ha aplicado el estereotipo *PersistentDomainClass*, que queramos que sean considerados durante la generación de la base de datos, se les debe aplicar a su vez el estereotipo *PersistentAttribute*, si no el atributo será ignorado durante la generación de la base de datos, aunque no será ignorado para la generación de la clase correspondiente.
Todas las anotaciones Java necesarias son generadas por el motor de transformación. Obviamente, la estructura de la base de datos no se genera sólo a partir de clases individuales, sino que también es necesario representar las relaciones entre dichas clases, de manera que dependiendo

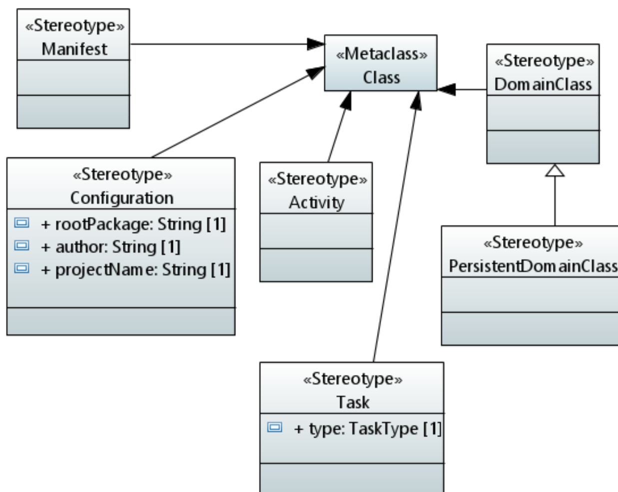
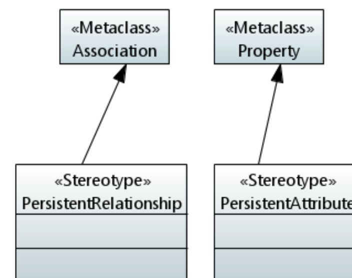


Figura 3. Estereotipos aplicables a clases UML en el perfil.

de las relaciones especificadas tendremos relaciones 1:1, 1:N, o N:M. *PersistentRelationship* (véase Figura 4) se añade a todas aquellas relaciones entre clases que deseamos que sean consideradas en la generación de la capa de persistencia.

Dichas relaciones darán lugar a claves foráneas en la base de datos, o tablas adicionales en el caso de relaciones muchos a muchos. La información que deben contener las tablas generadas es derivada de las relaciones y de los atributos con el estereotipo *PersistentAttribute* de las clases que incluyen el perfil *PersistentDomainClass*.

Además de los estereotipos genéricos para especificar la capa de persistencia, existen una serie de estereotipos adicionales dedicados a detallar la generación de la app para distintas plataformas. En este caso, los estereotipos *AndroidActivity*, *AndroidManifest* y *AndroidConfiguration* (véase Figura 3) están dedicados a la generación para la plataforma Android.



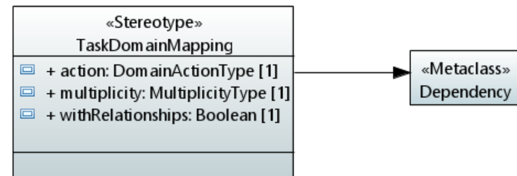


Figura 4. Relaciones usadas en el perfil.

AndroidManifest permite especificar metadatos sobre la aplicación que deseamos que sean incluidos en la app. Por ejemplo, en el caso de Android se especifica cuál es la versión de Android mínima (*minSDKversion*), o qué versión se quiere usar para compilar (*targetSDKversion*). *AndroidConfiguration* permite especificar quién es el autor, el nombre del proyecto o el paquete Java que servirá de base para el código Java generado. Es importante destacar, que este tipo de perfiles deberían ser especificados para las distintas plataformas destino para poder conseguir una generación de código de una calidad suficiente para hacerla realmente usable.

Finalmente, el perfil también incluye algunos estereotipos ligados al modelo de tareas. Estos estereotipos son necesarios para establecer las relaciones entre el modelo de tareas y el modelo de dominio (*mappings*) (Montero et al., 2006b). Esta relación es necesaria durante la generación, por ejemplo, para elegir qué widget es más adecuado para realizar una tarea específica. Si tenemos una tarea de entrada, el widget usado dependerá del tipo de datos del elemento que queremos introducir, pues no es lo mismo introducir un número que una cadena de caracteres, por ejemplo. Para ello, existe el estereotipo *Task*, que es aplicable a cualquier tarea que se haya especificado en el modelo de tareas. La tarea además incluye un atributo *type* que permite describir qué tipo de tarea se está realizando. Este atributo *type* es uno de los elementos clave en la elección de la interfaz de usuario que se generará para dicha tarea.

El conjunto de tipos de tareas (véase *TaskType* en Figura 5) hace referencia a los tipos de tarea establecidos en (Gonzalez-Calleros, Guerrero-Garcia, Vanderdonckt, & Munoz-Arteaga, 2009). Este estereotipo es complementado con el estereotipo *TaskDomainMapping*, que permite relacionar una tarea con el

elemento del dominio que manipula. Esta relación puede ser detallada especificando el tipo de acción que realiza la tarea sobre el elemento del modelo de dominio (crear, leer, actualizar o borrar), la cardinalidad (véase el tipo *MultiplicityType* en la Figura 4) y *withRelationships*, que indica que además se incluye a los elementos con los que está relacionado esa entidad. Por ejemplo, si una tarea indica que lee (READ) un cliente, podría querer también sus pedidos (que están relacionados con el cliente a través de una relación en el diagrama de clases).

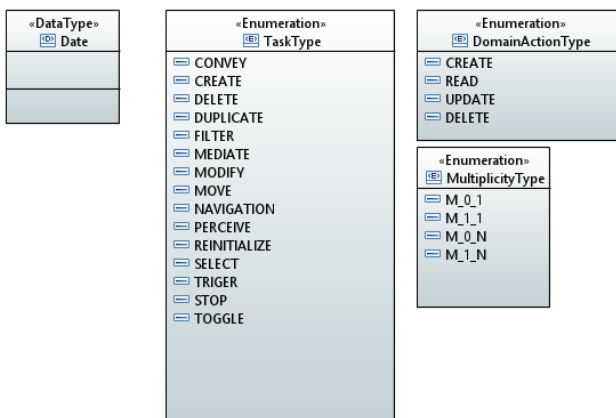


Figura 5. Tipos de datos usados en el perfil.

3.2 Generación de apps mediante transformación

Una vez se ha creado el modelo de tareas, y el modelo de dominio ha sido enriquecido con los estereotipos del perfil anteriormente descrito, llega el momento de la generación de la app. Como puede apreciarse en la Figura 2, se parte de un conjunto de modelos (tareas, dominio y *mapping*), los cuales serán la entrada del módulo de transformación de modelos, a partir del cual se genera el código para el framework de Android. A continuación, se describirá cómo se ha realizado el proceso de generación en el caso de UML2App, centrándonos en la fase de transformación y generación del código para Android.

3.2.1 Generación basada en transformación modelo a texto de apps

El proceso de transformación de una serie de modelos en código se realiza a partir de una transformación modelo a texto (M2T). Existen distintos marcos de trabajo que permiten dicho tipo de transformaciones, siendo algunos de los más populares Acceleo, Xpand o Epsilon. En este caso se ha optado por el lenguaje EGL *(Eclipse Generation Language)* (Eclipse Foundation, 2019b), un lenguaje basado en plantillas con una sintaxis cercana a OCL *(Object Constraint Language)*, el cual está incluido dentro del framework de Epsilon.

Una de las ventajas del marco de trabajo Epsilon es que las transformaciones escritas en EGL pueden ser orquestadas de manera sencilla usando el lenguaje EGX, el cual permite modularizar fácilmente la generación, algo imprescindible si no queremos crear un código de transformación imposible de mantener.

Durante el proceso de transformación descrito en EGL se irán procesando los elementos de los modelos de entrada (tanto de tareas como de dominio), leyendo la información que se presenta en los distintos modelos, e incluyendo los estereotipos que les hayan sido aplicados para afinar el proceso de generación. Por ejemplo, para la generación en Android para cada clase se genera una clase en Java. Si, además, dicha clase contiene el estereotipo *PersistentDomainClass*, se le añadirán todas las anotaciones necesarias para que dicha clase se guarde en la base de datos usando el marco de trabajo *Room* (Google Inc., 2019e) de Android. Adicionalmente, las clases extra necesarias para su uso en la app son generadas, como la propia clase de la base datos o los DAO *(Data Access Object)*, donde se describen las operaciones básicas como crear, borrar o modificar aplicables a cada una de las entidades almacenadas en la base de datos. Posteriormente, a partir de las relaciones del diagrama de clases con el estereotipo *TaskDomainMapping*, se crearán las consultas necesarias a la base datos para devolver, crear o modificar los datos requeridos para que se pueda llevar a cabo cada tarea concreta.

3.2.2 Generación de la interfaz de usuario

La capa de persistencia no tiene mucha utilidad si no podemos manejarla a través de la interfaz de usuario. El proceso de generación de la interfaz de usuario se basa en la identificación, definición y especificación de las tareas que el usuario puede hacer en el sistema y las relaciones temporales entre ellas principalmente, pero también es de gran importancia saber qué datos manipula cada tarea para decidir cuál es la manera más apropiada de interaccionar con esos datos, con el fin de realizar esas tareas de la manera más usable posible.

El proceso comienza con la generación de las ventanas que tendrá la app (*Activity* en la terminología de Android). Así, para cada tarea abstracta del modelo de tareas crearemos una nueva ventana, excepto para aquellas que no incluyan sólo una tarea no abstracta, es decir, que contengan una tarea atómica que no se puede descomponer, ya que no será necesario al poder incluirla en la tarea padre que las contiene. Dependiendo de la plataforma destino también debemos considerar las peculiaridades de sus guías de estilo. Por ejemplo, en el caso de Android no se debe incluir el elemento

de interacción para salir o volver atrás, ya que para ello se usa el botón atrás del dispositivo (ya sea este virtual o físico). Dentro de cada ventana, y de manera recursiva, se van creando los elementos de interacción (widgets) apropiados para cada tipo de tarea (Bodart & Vanderdonckt, 1994), considerando también en su elección aspectos como el tipo de datos de la información que manipula la tarea o la cardinalidad, es decir, no es lo mismo manejar un solo elemento que dos, o que una colección. Por ejemplo, si tenemos una tarea de selección (SELECT) que está relacionada con un atributo de tipo *Color* en el modelo de dominio, podríamos usar un *ColorPicker* para permitir que el usuario pueda elegir el color de una manera usable, y consistente con cómo se suele hacer habitualmente en esta plataforma. Por otro lado, si la tarea de selección está relacionada con el nombre de un país (de tipo *String*) con una relación 1:N, usaremos una lista desplegable para que el usuario seleccione el país de entre esos N elementos. Durante el proceso de transformación se van generando las interfaces de usuario XML de las distintas pantallas, ya que en Android existe una separación entre la especificación de la interfaz de usuario y la lógica que la maneja, así como los recursos necesarios para dichas interfaces. Es necesario crear todas las cadenas de texto, colores, dimensiones u otros recursos necesarios para que las actividades generadas funcionen correctamente.

4. Un caso de estudio

Como ejemplo se ha escogido una sencilla aplicación de alquiler de vehículos. La Figura 6, muestra el modelo de dominio para dicha aplicación, donde se puede ver que tenemos alquileres (*Rental*) que incluyen vehículos (*Vehicle*), los cuales son de un tipo determinado (*VehicleType*).
Adicionalmente, se representa el país donde se ha realizado el alquiler y el método de pago usado. Las tareas que puede realizar el usuario a través de la aplicación se han especificado mediante un modelo de tareas expresado usando la notación CTT (véase la Figura 7). Dicha notación no sólo permite la especificación de las tareas que podrá realizar el usuario, si no también cuáles son las relaciones temporales que se pueden establecer entre ellas. Por ejemplo, se establece que, como es lógico, antes de pagar el alquiler hay que seleccionar los coches. Durante el proceso de transformación se usará el modelo de tareas representado en formato XML, añadiendo el tipo interacción de la tarea (véase la Figura 5) a cada una de las tareas que sean de tipo Interacción (Mori, Paterno, & Santoro, 2002).

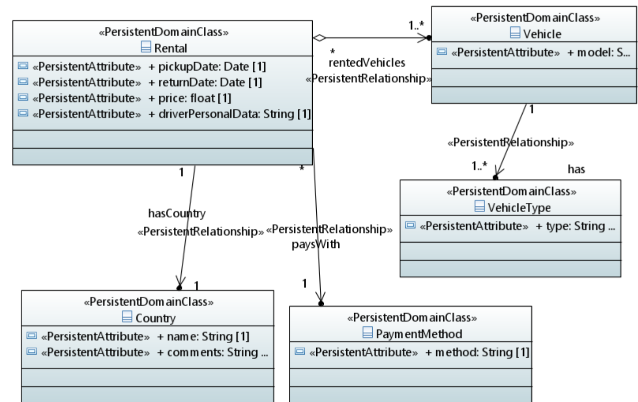


Figura 6. Modelo dominio del caso de estudio.

Finalmente, se establecen las correspondencias (*mappings*) entre el modelo de tareas y el de dominio (véase la Figura 8). En dicho modelo puede observarse cómo las correspondencias entre las tareas y los elementos del modelo de dominio se pueden crear tanto entre tareas y clases como entre tareas y atributos de las clases, o incluso con métodos para especificar qué método se ejecuta al realizar la tarea. Esto es necesario, ya que muchas veces sólo queremos manipular un atributo concreto de una clase, cuya especificación nos dará valiosa información sobre qué widgets escoger para realizar la tarea en liza.

Para ilustrarlo veamos un ejemplo. Como se puede ver en el modelo de tareas de la Figura 7, para alquilar un vehículo, una de las tareas que hay que hacer es especificar la fecha de recogida. Esta tarea está incluida dentro de una tarea abstracta denominada AlquilarVehiculo. Por lo tanto, durante la generación se ha creado una nueva ventana que contendrá todas las tareas que dicha tarea abstracta incluye. Para determinar qué widget se debería usar para realizar la tarea FechaRecogida necesitamos conocer qué tipo de tarea es y qué tipo de información manipula. En este caso es una tarea para realizar una operación de selección. Este tipo de operación es especificado en el modelo de tareas, como se describió anteriormente.

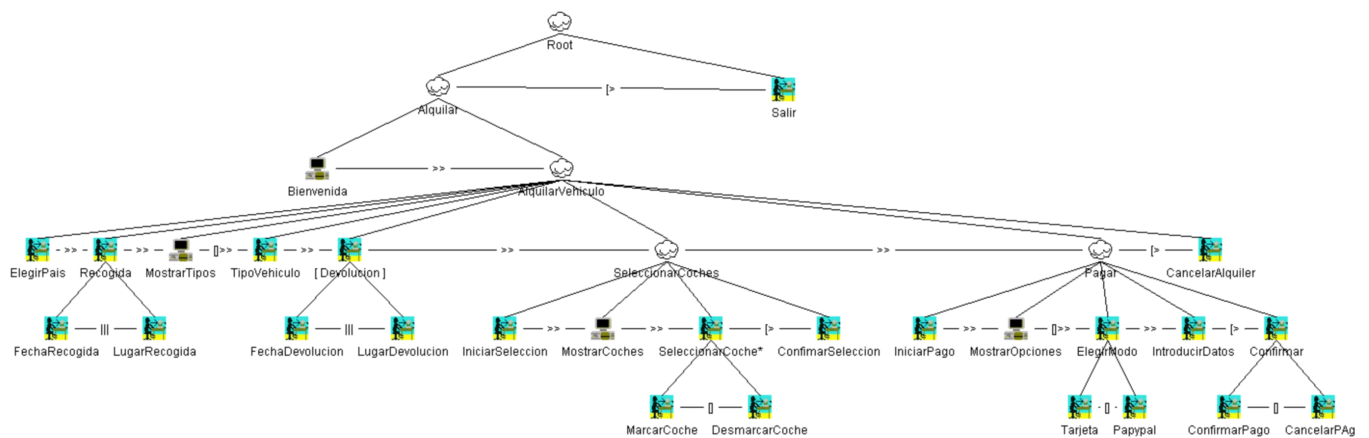


Figura 7. Modelo de tareas del caso de estudio.

Para saber qué información selecciona tendremos que acudir al modelo de dominio y de *mapping*. El modelo dentro de la entidad Rental (véase la Figura 6), incluye qué información necesita la aplicación, pero no qué información concreta manipula cada tarea. Para conocer qué información concreta manipula cada tarea tenemos que acudir al modelo de *mapping*. La Figura 8 describe un fragmento de dicho modelo para el caso de uso en curso. En este caso nos interesa la parte inferior de dicha figura, donde podemos observar cómo se especifica que la tarea FechaRecogida usa la información incluida en el atributo pickupDate, que como se puede ver en el modelo de dominio en la clase Rental es de tipo Date. En la figura no está recogido, pero entre los atributos especificados para la relación pickupdate se ha descrito que dicha relación es M_1_1, es decir, 1:1. Por lo tanto, ya sabemos que necesitamos un widget capaz de permitir la selección una sola fecha. Por esta razón, para la interfaz generada se escoge un DatePicker que un widget para Android especializado en la selección de fechas, y que cualquier persona familiarizada con Android encontrará intuitivo.

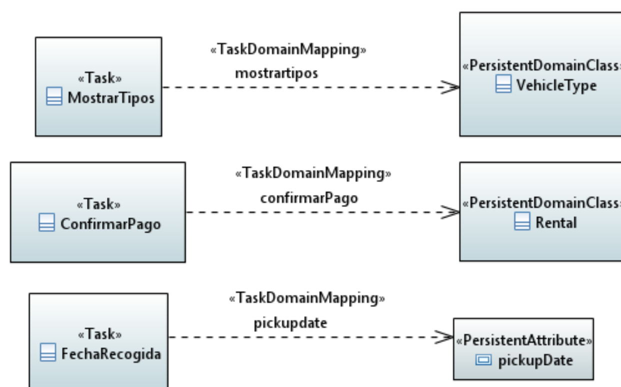


Figura 8. Correspondencias entre los modelos de tareas y dominio (fragmento).

La Figura 8 muestra los archivos fuente generados a partir de los tres modelos anteriormente descritos. La generación del código se ha realizado de acuerdo a la arquitectura recomendada para Android (Google Inc., 2019c). En ellos se puede apreciar que se han creado 4 actividades y la capa de persistencia, así como las capas intermedias que permiten comunicar la interfaz de usuario, la lógica y persistencia. Obsérvese que la capa de persistencia (incluida en la carpeta *model*) contiene las entidades que se almacenarán (carpeta *entity*), los objetos de acceso a los datos (carpeta *dao*) que actuarán como interfaz para manipular las entidades almacenadas, los conversores de tipos de datos del modelo de dominio a la base de datos para aquellos tipos de datos no soportando directamente por el gestor de la base de datos (en este caso para almacenar/leer las fechas), la propia base de datos, la vista (*viewmodel*), que será lo que realmente manipule la interfaz de usuario, y finalmente el repositorio, de acuerdo a las buenas prácticas para la creación de aplicación en Android anteriormente citadas. Adicionalmente, se han generado los recursos necesarios en la carpeta *res*: la interfaz de usuario de las actividades en XML (carpeta *layout*), los menús para dichas actividades, así como las cadenas de texto, colores o estilos, además de la información sobre la app que debe ser incluida en el manifiesto para su correcto funcionamiento y compilación.

Para compilar la aplicación se debe hacer uso el entorno oficial para el desarrollo de aplicaciones para Android: Android Studio. Para ello habrá que crear un nuevo proyecto vacío y copiar las carpetas dentro del propio proyecto. Las instrucciones sobre cómo hacerlo exactamente también se generan dentro del fichero readme.txt que aparece en la Figura 9.

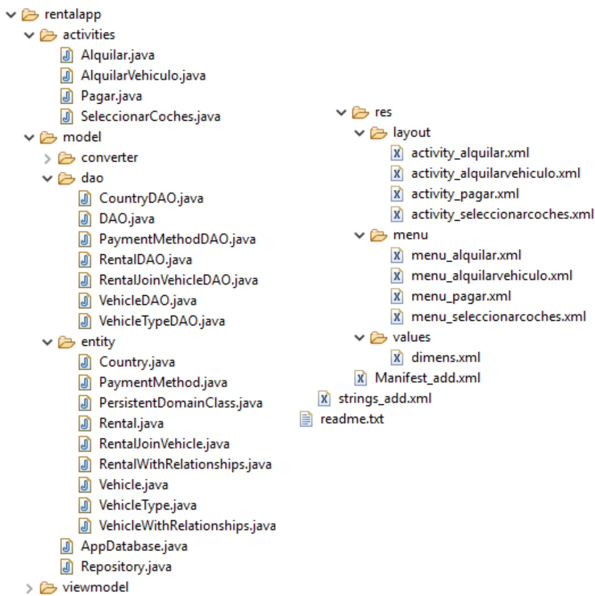


Figura 9. Código y recursos generados a partir de los modelos.

La Figura 10 muestra una de las pantallas generadas para Android. En este caso la pantalla del inicio del alquiler con la información básica. Uno de los aspectos que se puede apreciar es que no se ha cuidado el diseño gráfico. Aunque inicialmente sí que se consideró, finalmente se descartó tras realizar evaluaciones informales con desarrolladores en empresas. Las razones de esta decisión serán descritas en la siguiente sección.

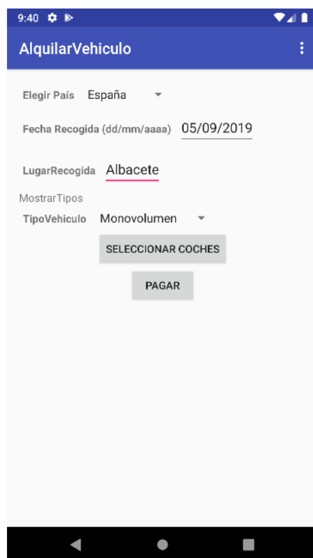


Figura 10. Pantalla de alquiler de vehículos de la app generada.

5. Lecciones aprendidas

Una de las primeras lecciones aprendidas, durante el desarrollo de esta herramienta de generación de aplicaciones para dispositivos móviles, es la rapidez con la que evolucionan los propios entornos oficiales de desarrollo para dichas plataformas. Originalmente, se generaba todo el proyecto para Android Studio, incluyendo toda la configuración del propio proyecto. Sin embargo, esa idea se desechó, debido a los constantes cambios en dichos ficheros que Google realizaba, lo cual dificultaba mantener la herramienta correctamente.

Después de distintos casos de estudio, nos dimos cuenta de que la generación de aplicaciones para estos dispositivos que no tengan una interacción estándar (juegos, por ejemplo) es dificultosa, ya que requieren patrones de generación y especificación de modelos bastante distintos a los habituales. Esto hace necesaria la creación de herramientas específicas para el diseño e implementación de dicho tipo de aplicaciones.

Un aspecto que no se debe olvidar, es que los desarrolladores que hacen uso UML2App necesitan un cierto entrenamiento. Esto se debe a que necesitan aprender cómo funciona el generador. Especialmente relevante en el modelado de tareas, donde debido al enorme número de combinaciones posibles, no siempre todas las combinaciones producen interfaces de usuario usables.

Desde el punto de vista del esfuerzo de modelado necesario, como se ha podido apreciar en los modelos del caso de estudio presentado, se requiere la creación de un solo modelo de dominio (véase Figura 6), la creación del modelo de tareas correspondiente a la interacción del usuario que se desea soportar (véase Figura 7), así como de los enlaces entre ambos modelos necesarios para saber qué información maneja cada tarea o qué métodos dispara (Figura 8).

La introducción de un modelo para representar el aspecto deseado de la interfaz de usuario (*Look & Feel*) sería sencillo de introducir en el caso de Android, ya que en las directrices de Google para el desarrollo de apps para Android (Google Inc., 2019d) está detallado. Sin embargo, aunque inicialmente se comenzó a incluirlo, se desechó. La razón principal es que, al hablar con desarrolladores en empresas, nos pedían que esa parte pudiera ser creada directamente por los diseñadores gráficos para darle un toque más atractivo a la interfaz. Aplicar plantillas a app creadas es la solución más interesante, ya que se consigue independizar el aspecto de la interfaz de la generación en gran medida, permitiendo un mantenimiento y actualización de la app sencillo.

Aunque la generación actual se ha hecho para Android, actualmente ya existen marcos de trabajo que permiten generar código para distintas plataformas como Flutter (Google Inc., 2019b). De esta manera en vez de generar código directamente para Android e iOS, se podría generar para

Flutter y así mejorar la mantenibilidad de la herramienta, dejando que sea Flutter quien genera el código final.

6. Conclusiones y trabajo futuro

En este trabajo se ha presentado una aproximación dirigida por modelos para el desarrollo de aplicaciones móviles llamada UML2App. El objetivo final que se persigue es crear una herramienta capaz de generar la aplicación para distintos dispositivos, incluso que usen distintos sistemas operativos.

Para ello, se han usado como base los modelos de tareas y de dominio. Dichos modelos son anotados a través de una serie de estereotipos especificados en un perfil UML. Mediante dichos estereotipos el diseñador es capaz de especificar cómo debe ser la generación de la aplicación. La propia generación de la aplicación se basa en el marco de trabajo de Epsilon, el cual garantiza que las propias transformaciones que producen la aplicación automáticamente se puedan ir actualizando de manera sencilla, al no estar directamente incluidas en el código del generador. La herramienta, denominada UML2App, soporta actualmente la generación para el marco de trabajo de Android, el cual ha sido elegido para ser el primero en ser soportado debido a su amplia expansión y a no requerir hardware específico para su desarrollo.

Actualmente, la herramienta presenta algunas limitaciones que se están abordando. En este sentido, el proceso de generación sería más preciso si se especificara un modelo de plataforma donde se describieran las capacidades de la propia plataforma destino. Este es un aspecto cuya importancia, sin embargo, se ha ido difuminando con la evolución de los propios marcos de trabajo de desarrollo para los dispositivos móviles. Por ejemplo, en Android la disposición de los componentes se realiza actualmente usando ConstraintLayout (Google Inc., 2019a), que precisamente persigue que las interfaces de usuario descritas usando este elemento sean capaces de adaptarse a distintos tamaños y formas de pantallas *(responsive)*.

Otro aspecto que se está abordando es mejorar la personalización de las aplicaciones generadas, para dar más flexibilidad al diseñador de la app. Por ejemplo, si prefiere una aplicación basada en pestañas (*tabs*) o no, qué elementos desea tener en la barra de acción (*actionBar*), etc. Estos aspectos, sin duda, contribuirían a mejorar la usabilidad de las aplicaciones generadas.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el Ministerio de Economía, Industria y Competitividad de España, la Agencia Estatal de Investigación (AEI) / Fondos Europeos para el Desarrollo Regional (FEDER, UE) mediante Vi-SMARt (TIN2016-79100-R) y el proyecto regional NeUX (SBPLY/17/180501/000192). Dicho proyecto está financiado por la Unión Europea, el fondo Europeo de Desarrollo Regional (FEDER) y la Junta de Comunidades de Castilla-La Mancha.

Referencias

Barnett, S., Avazpour, I., Vasa, R., & Grundy, J. (2019). Supporting multi-view development for mobile applications. Journal of Computer Languages, 51, 88–96. https://doi.org/10.1016/j.cola.2019.02.001

Behrens, H. (2010). MDSD for the iPhone. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10 (p. 123). New York, New York, USA: ACM Press. https://doi.org/10.1145/1869542.1869562

Biørn-Hansen, A., Grønli, T.-M., & Ghinea, G. (2018). A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development. ACM Computing Surveys, 51(5), 1–34. https://doi.org/10.1145/3241739

Biørn-Hansen, A., Grønli, T.-M., Ghinea, G., & Alouneh, S. (2019). An Empirical Study of Cross-Platform Mobile Development in Industry. Wireless Communications and Mobile Computing, 2019, 1–12. https://doi.org/10.1155/2019/5743892

Bodart, F., & Vanderdonckt, J. (1994). On the Problem of Selecting Interaction Objects, (August), 23–26.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., & Vanderdonckt, J. (2003). A Unifying Reference Framework for multi-target user interfaces. Interacting with Computers, 15(3), 289–308. https://doi.org/10.1016/S0953-5438(03)00010-9

Dalmasso, I., Datta, S. K., Bonnet, C., & Nikaein, N. (2013). Survey, comparison and evaluation of cross platform mobile application development tools. In 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC) (pp. 323–328). IEEE. https://doi.org/10.1109/IWCMC.2013.6583580

Ditrendia. (2019). Informe Ditrendia Mobile en España y el Mundo 2018. Retrieved March 8, 2019, from http://mktefa.ditrendia.es/blog/todas-las-estadísticas-sobre-móviles-que-deberías-conocer-mwc18

Eclipse Foundation. (2019a). Epsilon. Retrieved March 12, 2019, from https://www.eclipse.org/epsilon/

Eclipse Foundation. (2019b). Epsilon Generation Language. Retrieved March 14, 2019, from https://www.eclipse.org/epsilon/doc/egl/

Fuentes, L., & Vallecillo, A. (2004). An introduction to UML profiles. UML and Model Engineering, 2, 6–13.

Gonzalez-Calleros, J. M., Guerrero-Garcia, J., Vanderdonckt, J., & Munoz-Arteaga, J. (2009). Towards Canonical Task Types for User Interface Design. In 2009 Latin American Web Congress (pp. 63–70). IEEE. https://doi.org/10.1109/LA-WEB.2009.33

Google Inc. (2019a). Build a Responsive UI with ConstraintLayout. Retrieved March 14, 2019, from https://developer.android.com/training/constraint-layout

Google Inc. (2019b). Flutter. Retrieved September 12, 2019, from https://flutter.dev/

Google Inc. (2019c). Guide to app architecture. Retrieved September 5, 2019, from https://developer.android.com/jetpack/docs/guide

Google Inc. (2019d). Material Design. Retrieved March 14, 2019, from https://material.io/develop/android/

Google Inc. (2019e). Room Persistence Library. Retrieved March 12, 2019, from https://developer.android.com/topic/libraries/architecture/room

Guerrero, J., González-Calleros, J. M., Vanderdonckt, J., & Muñoz Arteaga, J. (2009). A Theoretical Survey of User Interface Description Languages: Preliminary Results. In Proc. of LA-WEB/CLIHC 2009 (pp. 36–43).

Heitkötter, H., Majchrzak, T. A., & Kuchen, H. (2013). Cross-platform model-driven development of mobile applications with md 2. In Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13 (p. 526). New York, New York, USA: ACM Press. https://doi.org/10.1145/2480362.2480464

Joorabchi, M. E., Mesbah, A., & Kruchten, P. (2013). Real Challenges in Mobile App Development. In 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 15–24). IEEE. https://doi.org/10.1109/ESEM.2013.9

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., & López-Jaquero, V. (2005). Usixml: A language supporting multi-path development of user interfaces. Engineering Human Computer Interaction and Interactive Systems, 200–220. Retrieved from http://www.springerlink.com/index/barbk4qmlx4ybn8a.pdf

Mendix. (2019). Mendix: Low-code Application Development Platform. Retrieved March 14, 2019, from https://www.mendix.com/

Montero, F., López-Jaquero, V., Vanderdonckt, J., González, P., Lozano, M. D., & Limbourg, Q. (2006a). Solving the Mapping Problem in User Interface Design by Seamless Integration in IdealXML. In S. W. Gilroy & M. D. Harrison (Eds.), Interactive Systems (pp. 161–172). Springer. https://doi.org/10.1007/11752707_14

Montero, F., López-Jaquero, V., Vanderdonckt, J., González, P., Lozano, M., & Limbourg, Q. (2006b). Solving the Mapping Problem in User Interface Design by Seamless Integration in IdealXML (pp. 161–172). https://doi.org/10.1007/11752707_14

Mori, G., Paterno, F., & Santoro, C. (2002). CTTE: support for developing and analyzing task models for interactive system design. IEEE Transactions on Software Engineering, 28(8), 797–813. https://doi.org/10.1109/TSE.2002.1027801

Paternò, F. (1999). Model-Based Design and Evaluation of Interactive Applications. Springer-Verlag.

Ruiz, J., Serral, E., & Snoeck, M. (2018). Evaluating user interface generation approaches: model-based versus model-driven development. Software & Systems Modeling. https://doi.org/10.1007/s10270-018-0698-x

Selic, B. (2003). The pragmatics of model-driven development. IEEE Software, 20(5), 19–25. https://doi.org/10.1109/MS.2003.1231146

The Apache Software Foundation. (2019). Apache Cordova. Retrieved November 13, 2019, from https://cordova.apache.org/

Umuhoza, E., & Brambilla, M. (2016). Model Driven Development Approaches for Mobile Applications: A Survey (pp. 93–107). https://doi.org/10.1007/978-3-319-44215-0_8

Usman, M., Iqbal, M. Z., & Khan, M. U. (2014). A Model-Driven Approach to Generate Mobile Applications for Multiple Platforms. In 2014 21st Asia-Pacific Software Engineering Conference (pp. 111–118). IEEE. https://doi.org/10.1109/APSEC.2014.26

Vaupel, S., Taentzer, G., Gerlach, R., & Guckert, M. (2018). Model-driven development of mobile applications for Android and iOS supporting role-based app variability. Software & Systems Modeling, 17(1), 35–63. https://doi.org/10.1007/s10270-016-0559-4

Willocx, M., Vossaert, J., & Naessens, V. (2015). A Quantitative Assessment of Performance in Mobile App Development Tools. In 2015 IEEE International Conference on Mobile Services (pp. 454–461). IEEE. https://doi.org/10.1109/MobServ.2015.68

Wong, J., Baker, V. L., Leow, A., & Resnick, M. (2018). Magic Quadrant for Mobile App Development Platforms