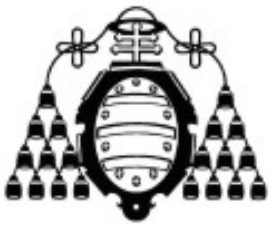
UNIVERSIDAD DE OVIEDO



ESCUELA DE INGENIERÍA INFORMÁTICA

TRABAJO FINAL DE MÁSTER

"GENERACIÓN DE INTERFACES DE USUARIO EN APLICACIONES MÓVILES MULTIPLATAFORMA MEDIANTE TRANSFORMACIÓN DE MODELOS"

AUTOR: Germán Pedrosa Loureiro

DIRECTOR: Francisco Ortín Soler

Vº Bº del Director del Proyecto

Agradecimientos

En primer lugar agradecer a mis padres y hermanos todo el apoyo y ánimos que me han dado durante la realización de este proyecto.

Agradecer a mi director de proyecto, Francisco Ortín, su disponibilidad y el haberme dirigido y aconsejado en todo momento, aportando su amplísima experiencia investigadora.

Agradecer también a mis compañeros de trabajo en Fundación CTIC su ayuda y conocimientos prestados, muy especialmente a Nacho Marín y a Javier Rodríguez con experiencia en esta línea de investigación y cuya ayuda ha sido fundamental.

Por último agradecer a mis amigos el estar siempre ahí. Gracias por todas las veces en las que me habéis hecho olvidar todos los problemas y agobios.

Gracias a todos.

Germán Pedrosa Loureiro, julio de 2012.

Resumen

En los últimos años, el tipo de dispositivos utilizados para acceder a los sistemas de información se ha incrementado notablemente, con diferentes sistemas operativos, tamaños de pantalla, mecanismos de interacción y características software. Esta fragmentación de dispositivos supone un problema importante a la hora de hacer frente al desarrollo de aplicaciones móviles nativas. Para solucionar este problema proponemos la generación de interfaces de usuario nativas basadas en transformaciones de modelos, siguiendo el paradigma Model-Based User Interface (MBUI). Se propone el uso de un modelo intermedio entre los niveles CUI y FUI del framework CAMELEON que facilita la transformación entre esos dos niveles al considerar características específicas de la plataforma. En este trabajo se presenta LIZARD, un framework para crear aplicaciones multiplataforma siguiendo el paradigma MBUI, y una herramienta software que implementa dicho framework. LIZARD proporciona mecanismos para definir las aplicaciones a un nivel alto de abstracción, y aplica transformaciones para obtener una aplicación adaptada a una plataforma específica. LIZARD permite que las aplicaciones generadas sigan los patrones arquitectónicos y de diseño y las guías de diseño de interfaces de usuario especificadas por cada fabricante. El objetivo es que las aplicaciones no se generen de forma genérica siguiendo un mínimo común denominador, sino proporcionando la experiencia del usuario esperada en cada caso. Para mostrar el funcionamiento de LIZARD se ha desarrollado una aplicación de ejemplo. El sistema genera versiones para Windows Phone y Android, dos sistemas basados en diferentes patrones arquitectónicos y diferentes pautas para el diseño de interfaces.

Palabras Clave

MDE

Transformación de modelos

Generación de código

Interfaces gráficas de usuario

Desarrollo multiplataforma

UsiXML

CAMELEON

Abstract

In the last years, the type of devices used to access information systems has been notably increased, using different operating systems, screen sizes, interaction mechanisms and software features. This device fragmentation is an important issue to tackle when developing native mobile service front-end applications. For this purpose, we propose the generation of native user interfaces by means of model transformations, following the Model-Based User Interface (MBUI) paradigm. An intermediate model between the CUI and FUI levels of the CAMELEON framework is proposed to facilitate the transformation between these two levels, considering the platform-specific features. We present LIZARD, a framework to create applications for multiple target platforms following the MBUI paradigm, and a software tool that implements that framework. LIZARD provides mechanisms to define applications at a high level of abstraction, and applies model transformations to obtain an application for a specific platform. LIZARD provides mechanisms that allow the applications generated to follow the architectural and design patterns and user interface design guidelines specified by operating system manufacturers. The objective is that applications are not generated in a generic manner, following a least-common-denominator, but providing the user experience expected by software consumers. We present an example application to show the LIZARD framework. The application is generated to Windows Phone and Android platforms, two example systems based on different software patterns and user interface designs.

Keywords

MDE

Model-to-model transformation

Code generation

Graphical user interfaces

Cross-platform development

UsiXML

CAMELEON

Índice General

Índice de Figuras

Capítulo 1. Introducción

En los últimos años el número de dispositivos que usamos para consumir contenido ha aumentando vertiginosamente: teléfonos móviles, tablets, televisores inteligentes, etc. Uno de los principales objetivos de los desarrolladores de aplicaciones consiste en llegar al mayor número de usuarios posible, sin embargo, la heterogeneidad de dispositivos a nivel de hardware y software imposibilita que el desarrollador pueda crear una única versión de la aplicación que funcione en la mayoría de estos dispositivos, con el consiguiente aumento de costes y esfuerzos para lograr su objetivo. Este problema, conocido como fragmentación de dispositivos, ha propiciado la aparición en los últimos años de un gran número de trabajos de investigación cuyo objetivo es facilitar el desarrollo de aplicaciones multiplataforma. Uno de los aspectos que resultan más difíciles de abordar dentro de la generación de aplicaciones es la creación de las interfaces de usuario debido entre otros a la gran diferencia entre tamaños de pantalla y modos de interacción (gráfico, vocal, táctil, etc.).

Este trabajo sigue las líneas marcadas por la comunidad investigadora, centrándose en el uso de la metodología MDE (Model-driven engineering) y basándose en algunos de los proyectos más relevantes como el framework CAMELEON [1] y el lenguaje UsiXML [2]. El framework CAMELEON sugiere el uso de una serie de modelos definidos a distintos niveles de abstracción y transformaciones para traducir los modelos de un nivel a otro. En sentido descendente en relación al nivel de abstracción el framework propone los niveles Task and Concepts (TC), Abstract User Interface (AUI), Concrete User Interface (CUI) y Final User Interface (FUI). Por su parte UsiXML define una serie de modelos en cada uno de los niveles propuestos por CAMELEON y la semántica de cada uno de ellos. Como resultado de la aplicación de los conceptos teóricos propuestos por CAMELEON y UsiXML se han desarrollado varias herramientas [3][4][5] que permiten la definición de la interfaz de usuario a un nivel alto de abstracción a partir del cual se generan versiones ejecutables en distintas plataformas. Sin embargo ninguna de estas herramientas parece cubrir en profundidad la transformación de CUI a FUI. Generalmente la adaptación a nivel de interfaz se realiza simplemente como una traducción directa, trasladando la misma experiencia de usuario entre todas las plataformas.

El objetivo de este proyecto es pues definir una arquitectura basándose en la propuesta por CAMELEON que permita una mejor solución para la transformación entre los niveles CUI y FUI con el fin de obtener interfaces que tengan en cuenta las guías de estilo de cada plataforma y los patrones de interfaces de usuario y arquitectónicos recomendados para cada dispositivo final.

1.1 Motivación

Las expectativas de los usuarios no sólo incluyen la posibilidad de usar las aplicaciones en cualquier dispositivo, sino que también requieren que éstas mantengan un nivel de usabilidad adecuado y permitan su uso de la forma más adecuada en cada caso [6]. Sin embargo los trabajos llevados a cabo hasta el momento consideran la transformación entre los niveles CUI y FUI como una simple traducción entre los "interactuadores concretos" de CUI (abstracción de los componentes incluidos en los toolkits de programación de UI) y los correspondientes componentes de cada plataforma. Por ello el *look and feel* de las interfaces finales obtenidas para cada plataforma es muy similar entre sí y no tiene en cuenta ni las guías de estilo de cada sistema operativo ni se preocupa de la aplicación de los patrones arquitectónicos y de diseño recomendados en cada caso. Esto supone un problema importante y dificulta la obtención de interfaces de calidad que ofrezcan un nivel aceptable de usabilidad, provocando en muchos casos el rechazo de los usuarios.

El uso de herramientas para la generación multiplataforma facilita en gran medida las tareas de los desarrolladores, permitiéndoles la posibilidad de abarcar un mayor número de dispositivos y por lo tanto incrementando el número de usuarios potenciales. La posibilidad de generar varias versiones de la misma aplicación para distintas plataformas definiendo la aplicación una única vez supondrá grandes ahorros en costes y esfuerzos para el desarrollador y evitará la necesidad del conocimiento de varios lenguajes de programación, APIs y tecnologías. Sin embargo todas estas facilidades que se le ofrecen al desarrollador no deben repercutir en la calidad de los resultados obtenidos. Es evidente por lo tanto la necesidad de que las herramientas de generación multiplataforma tengan en cuenta las particularidades de cada una de las plataformas y tipos de dispositivo para satisfacer las necesidades de los usuarios.

1.2 Finalidad del proyecto

El objetivo principal del proyecto es definir e implementar un nuevo framework basado en los distintos niveles descritos por CAMELEON y en los meta-modelos propuestos por UsiXML que permita mejorar los resultados de las interfaces obtenidas. Se pretende que el sistema sea capaz de generar a partir de una única descripción de alto nivel de todas las características de la aplicación versiones para la combinación de varias plataformas móviles y tipos de dispositivo. Las interfaces generadas deberán estar totalmente adaptadas a la plataforma y tipo de dispositivo, respetando las guías de estilo de cada sistema operativo y aplicando los patrones arquitectónicos y de diseño adecuados en cada caso.

Capítulo 2. Fijación de Objetivos

2.1 Objetivos

A continuación se enumerarán los objetivos propuestos para la creación del framework de desarrollo de aplicaciones móviles multiplataforma mencionado anteriormente.

1. **El sistema a desarrollar facilitará al usuario la definición de aplicaciones.**
   Se pretende que el usuario pueda describir al completo una aplicación sin la necesidad de disponer de unos conocimientos técnicos elevados. Para cada aplicación a generar el sistema ha de permitir al desarrollador describir los siguientes aspectos:
   - **Descripción a alto nivel de las tareas que el usuario puede realizar con la aplicación y su interacción con la interfaz gráfica para llevarlas a cabo.**
     Se debe permitir al desarrollador expresar de forma sencilla las tareas que el usuario podrá llevar a cabo mediante el uso de la aplicación, definiendo el orden en el que éste puede realizar las tareas y las relaciones de dependencia entre estas tareas.
   - **Descripción a alto nivel de la información (datos) que maneja la aplicación.**
     Se realizará una descripción de los datos que podrán intercambiar entre el usuario y la aplicación a través de la interfaz de usuario.
   - **Especificación a alto nivel de las vistas que forman la interfaz de usuario, esta descripción deberá ser independiente del dispositivo.**
     Para cada una de las tareas que el usuario pueda realizar con la aplicación el sistema debe permitir al desarrollador definir a alto nivel la apariencia visual de la interfaz gráfica que permitirá la realización de dicha tarea.
   - **Descripción a alto nivel de plataformas móviles o versiones de éstas.**
     El sistema debe permitir la descripción de plataformas móviles para las que se podrá generar una versión de la aplicación.
   - **Descripción de los dispositivos móviles objetivo.**
     El sistema debe permitir al desarrollador describir el conjunto de dispositivos para los que desea obtener una versión final de la aplicación. Dicha descripción ha de incluir aquellos aspectos que puedan influir en el resultado final, como pueden ser la plataforma (Android, iOS, Windows Phone, etc.), el tipo de dispositivo (móvil o tablet), el tamaño de pantalla, la resolución, etc.
2. **El sistema permitirá obtener una interfaz de usuario final a partir de las descripciones a alto nivel de la aplicación.**
   El sistema debe adaptar la descripción abstracta a particularidades de cada

una de las plataformas objetivo con el fin de generar una versión de la aplicación adecuada en cada caso.

3. **Flexibilidad y extensibilidad del sistema.**
   El sistema creado ha de ser flexible y extensible para que permita de una forma sencilla la incorporación de nuevas plataformas móviles así como nuevas reglas de transformación, por ejemplo para la incorporación de nuevos patrones de diseño de interfaces de usuario.
4. **Generación de aplicaciones nativas para cada plataforma.**
   Las aplicaciones nativas permiten explotar al máximo las capacidades de los dispositivos móviles y ofrecen una serie de ventajas como se verá a lo largo de este documento. El sistema ha de ser capaz de generar aplicaciones nativas para cada una de las plataformas objetivo.
5. **Calidad en los resultados obtenidos, aplicando patrones arquitectónicos, patrones de diseño y siguiendo las guías de estilo de cada sistema operativo.**
   Las aplicaciones generadas por el sistema han de cumplir ciertos requisitos de calidad y adaptarse totalmente a los estándares, patrones arquitectónicos, patrones de diseño de interfaces de usuario y las guías de estilo propuestas para cada sistema operativo. Lograr unos resultados satisfactorios y una aplicación usable es un objetivo fundamental del sistema a desarrollar.

2.2 Posibles ámbitos de Aplicación

El sistema desarrollado supone únicamente un prototipo que permite validar la investigación llevada a cabo y que se centra principalmente en aquellos aspectos considerados más relevantes y que aportan una mayor innovación, como pueden ser las transformaciones de modelos para adaptar los resultados a la plataforma y tipo de dispositivo final. Sin embargo este prototipo cubre todo el proceso de generación de aplicaciones móviles multiplataforma y sienta las bases para la creación de una herramienta de autoría completa.

El principal ámbito de aplicación es por lo tanto la creación de una plataforma que dé soporte al desarrollo de aplicaciones multiplataforma y una herramienta o IDE que permita a los desarrolladores aumentar su productividad reduciendo los tiempos necesarios para la creación de una misma aplicación para distintas plataformas.

Aunque en nuestro caso el trabajo se ha centrado en la generación de aplicaciones para dispositivos móviles en realidad el sistema podría cubrir un abanico de dispositivos mucho mayor, permitiendo la generación de aplicaciones para otros dispositivos como PCs, televisiones inteligentes, etc.

Capítulo 3. Estado Actual de los Conocimientos Científico-Técnicos

Como ya se ha comentado en la introducción este trabajo está relacionado con investigaciones previas llevadas a cabo en el ámbito de la generación de interfaces de usuario multiplataforma. En este apartado veremos algunos de estos trabajos relacionados así como las características y particularidades de cada plataforma que han de ser tenidas en cuenta para el desarrollo del sistema.

3.1 Fragmentación

La fragmentación de dispositivos es la imposibilidad de desarrollar una única versión de una aplicación y que ésta funcione en la totalidad de dispositivos existentes en el mercado. Esta imposibilidad viene dada por la diversidad hardware, como son las diferencias en los parámetros de las pantallas (tamaño, orientación, profundidad de color, etc.), la cantidad de memoria, potencia del procesador, modo de entrada (teclado, pantalla táctil, etc.), opciones de conectividad (Bluetooth, IR, GPRS, WiFi, etc.), y sobre todo por la diversidad software provocada por el gran número de plataformas (Symbian, RIM OS, Windows Mobile, iOS, Android, Windows Phone, etc.).

Sin duda la fragmentación de dispositivos supone el mayor reto para los desarrolladores de aplicaciones para dispositivos móviles e imposibilita seguir la filosofía WORA (Write Once, Run Anywhere, escribir una vez, ejecutar en cualquier sitio). La fragmentación de dispositivos supone que los desarrolladores deban crear una versión de la misma aplicación para cada plataforma. Esto supone un gran problema ya que incrementa considerablemente los costes y esfuerzos además de influir negativamente en la futura mantenibilidad.

En lo que se refiere a las características hardware de los dispositivos sin duda uno de los aspectos que más acentúa la fragmentación de dispositivos y que más incluye en la generación de interfaces gráficas es la diferencia existente entre las pantallas de los dispositivos. Existen grandes diferencias en el tamaño de pantalla de los dispositivos móviles, y estas diferencias han ido aumentando en los últimos años al hacerse populares nuevos dispositivos como las tablets. Sin embargo el tamaño no es el único aspecto a tener en cuenta de las pantallas. Otro aspecto que influye considerablemente es la relación de aspecto, es decir, la proporción existente entre la altura y la anchura de la pantalla. Esta diferencia en la relación de aspecto imposibilita por ejemplo que se pueda utilizar una imagen de fondo en varios dispositivos realizando simplemente un escalado. La profundidad de color, la resolución y los puntos por pulgada son otros aspectos a tener en cuenta en lo que a las pantallas se refiere, lo que nos da una idea de la complejidad del problema y de la cantidad de aspectos que los desarrolladores tienen que tener en cuenta.

Aunque sin duda el mayor problema al que deben enfrentarse los desarrolladores es la cantidad de plataformas móviles existentes en el mercado. La Figura 1 muestra la distribución de las distintas plataformas móviles en el mercado durante el primer trimestre de 2012. Dicha imagen nos indica la dificultad a la que se deben enfrentar los desarrolladores para que sus aplicaciones estén disponibles para un alto porcentaje de usuarios.

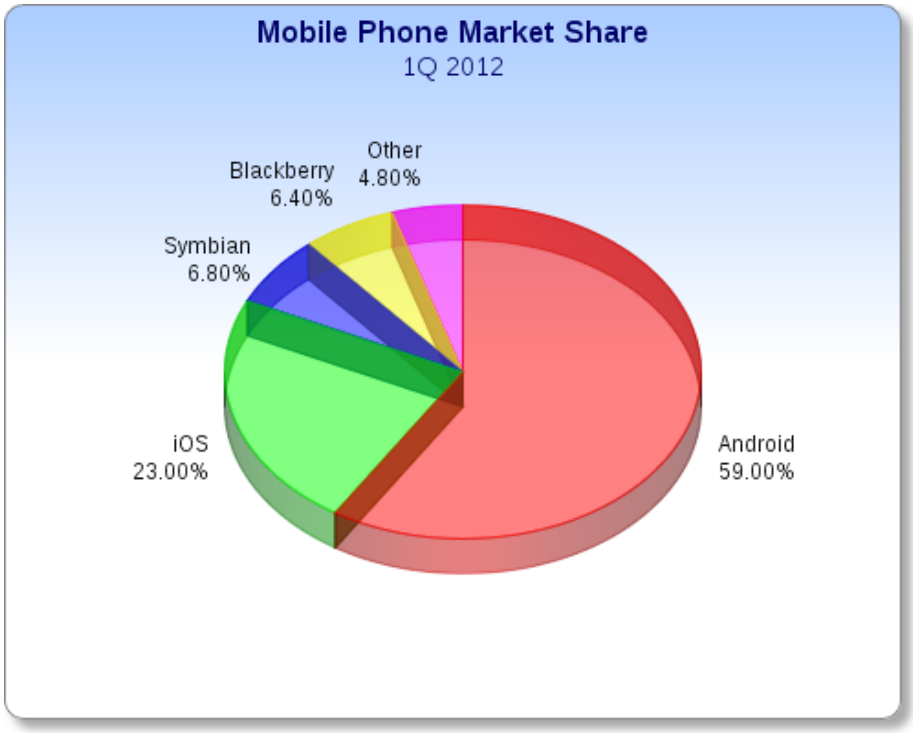


Figura 1. Distribución de las distintas plataformas móviles en el mercado.

3.2 Plataformas móviles

El trabajo relacionado permite la generación de aplicaciones nativas para dos plataformas móviles con una importante participación en el mercado y en gran crecimiento (Android y WP7). La elección de estas dos plataformas para validar los resultados del trabajo desarrollado resulta apropiada ya que permite generar aplicaciones con unas características muy diferentes. No solo se genera código en dos lenguajes de programación distintos (Java en Android y C# para WP7), si no que se hace uso de diferentes patrones arquitectónicos, distintos patrones de diseño de interfaz de usuario adaptados a las guías de estilo de cada plataforma e incluso diferentes dispositivos permitiendo obtener versiones para teléfonos móviles de las dos plataformas y una versión para tablets Android.

En este apartado veremos una descripción breve de estas dos plataformas centrándonos principalmente en la generación de interfaces gráficas y en todas aquellas características que influirán en dicha generación.

3.2.1 Android

Android es un sistema operativo de código abierto para dispositivos móviles basado en Linux utilizado en smartphones, tablets, televisores, etc. Android fue creado por Google y la Open Handset Alliance, una alianza compuesta por fabricantes y desarrolladores de hardware, software y operadores de servicio.

La generación automática de aplicaciones para Android supone un reto en sí mismo ya que incluye un gran número de dispositivos con características muy diversas (móviles y tablets con distintas resoluciones, densidades de píxeles, tamaños de pantalla, etc.). Además desde su nacimiento se han liberado varias versiones que siguen ejecutándose actualmente en gran número de dispositivos Debido a todo esto muchas veces se habla incluso de fragmentación interna de Android. La Figura 2 muestra la distribución de las diferentes versiones de Android que conviven en el mercado y que suponen un reto enorme para los desarrolladores cuyo objetivo es llegar al mayor número de usuarios posible.

Version	Codename	API Level	Distribution
1.5	Cupcake	3	0.3%
1.6	Donut	4	0.6%
2.1	Eclair	7	5.2%
2.2	Froyo	8	19.1%
2.3 - 2.3.2	Gingerbread	9	0.4%
2.3.3 - 2.3.7		10	64.6%
3.1	Honeycomb	12	0.7%
3.2		13	2%
4.0 - 4.0.2	Ice Cream Sandwich	14	0.4%
4.0.3 - 4.0.4		15	6.7%

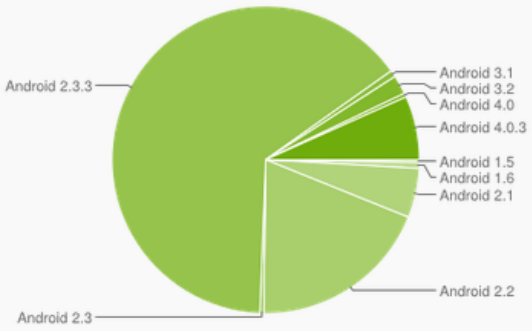


Figura 2. Distribución de las versiones de Android en el mercado. (Junio 2012)

Sin embargo el disponer de una gran diversidad de dispositivos con una gran heterogeneidad de características la convierten en la plataforma ideal para probar el sistema desarrollado en este trabajo [7]. A continuación veremos qué dificultades y soluciones nos presenta Android para el desarrollo de aplicaciones que deban ejecutarse en dispositivos muy diferentes entre sí.

A nivel de desarrollo de interfaz gráfica el mayor problema está en la gran diversidad de tamaños de pantalla y resoluciones que debe tener en cuenta el desarrollador. La Figura 3 muestra la gran cantidad de resoluciones usadas en dispositivos Android, en la que cada punto se corresponde con una resolución. Para adaptar la interfaz al gran rango de dispositivos disponibles Android clasifica los dispositivos en 4 grupos (small, normal, large y extra large screen) y permite definir diferentes versiones de un mismo recurso (layouts, imágenes, etc.) para cada uno de esos grupos.

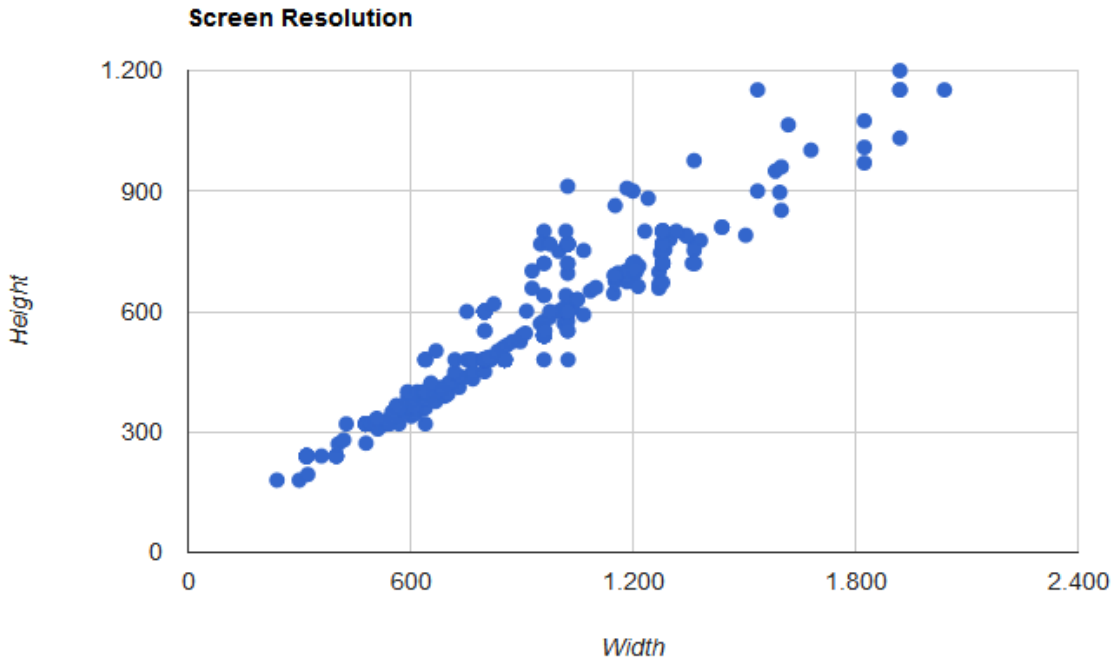


Figura 3. Resoluciones utilizadas en dispositivos Android.

El API de Android ofrece a los desarrolladores una serie de clases de las que pueden heredar para implementar un gran número de tareas. Se utiliza un *Activity* para gestionar el ciclo de vida y hacer las tareas de controlador de cada vista. La navegación se realiza mediante el uso de *Intents*. Para compartir datos entre diferentes vistas se pueden incluir como extras en el *Intent* o compartirlos mediante una clase principal que hereda de *Application* y representa la aplicación.

Para la personalización de la apariencia gráfica de una aplicación Android se usan *estilos* y *temas*. Un estilo en Android es un conjunto de atributos de vista, incluidos en un recurso independiente. Para evitar tener que aplicar el mismo estilo a cada elemento podemos usar un tema. Los temas son estilos, la única diferencia entre ellos es que los temas se aplican a *Activities* o la aplicación entera (al conjunto de todos los *Activities*).

La gran variedad de tamaños de pantalla, resoluciones y densidades de píxeles dificulta el uso de imágenes pudiendo causar que se deformen o pierdan calidad. Para evitar esos problemas y no forzar al desarrollador a proporcionar una imagen independiente para cada configuración Android introduce los *nine-patch drawables*, imágenes redimensionables que Android se encarga de adaptar automáticamente al contenido de la vista en el que se usa como imagen de fondo.

Otro concepto interesante que maneja Android son los *Fragments*, que permiten modularizar el código de una aplicación para que pueda usar layouts significantemente diferentes en los modos vertical y apaisado o en dispositivos muy distintos (smartphone y tablets) sin tener que duplicar código y funcionalidad.

3.2.2 Windows Phone 7

Windows Phone 7 es un sistema operativo móvil desarrollado por Microsoft como sucesor de Windows Mobile y basado en la arquitectura Windows CE.

Con la idea de evitar la fragmentación interna Microsoft fija en un documento de requisitos de certificación muchas de las características hardware a las que deben ceñirse los fabricantes. Por ejemplo en un inició limitó la resolución que debían usar los dispositivos Windows Phone 7 a WVGA (800x480), aunque en futuras versiones (Tango) se permitió también el uso de la resolución HVGA (480x320). El mayor control sobre las características hardware facilita en gran medida el desarrollo de aplicaciones para WP7 y en especial de la interfaz gráfica de usuario.

Aunque no es obligatorio se recomienda el uso del patrón MVVM (Model View ViewModel) para permitir desacoplar la interfaz del resto de lógica de negocio de la aplicación. En el patón MVVM un *ModelView* encapsula el comportamiento y los datos manejados por cada vista. Utiliza *Bindings* desde la vista XAML a propiedades en su *ModelView* y *Commands* para el manejo de los eventos producidos en la interfaz. El patrón MVVM evita en la mayoría de los casos la necesidad de usar *code behind*. En el apartado 3.3.1 veremos en detalle el uso de este patrón.

WP7 usa XAML y Silverlight para definir los controles de la interfaz de usuario y su distribución y C# para la lógica de negocio y acceso a los datos.

Para la definición de estilos se usan Temas. La apariencia de la aplicación se debería adaptar al tema escogido por el usuario en el dispositivo en lugar de dar unos valores concretos al definir la interfaz.

3.3 Patrones arquitectónicos para el desarrollo de interfaces gráficas

Existen una serie de patrones arquitectónicos de uso común en el desarrollo de aplicaciones con interfaces gráficas de usuario, como pueden ser MVC (Model-View-Controller), MVP (Model-View-Presenter), PM (Presentation Model), MVVM (Model-View-ViewModel). El objetivo principal de todos estos patrones es conseguir desacoplar la vista del resto de la aplicación. Esta separación lógica asegura la claridad y mantenibilidad del código ya que la comunicación entre los distintos componentes está definida de forma estricta en cada patrón.

A finales de los años 70 con el crecimiento de las aplicaciones basadas en interfaces gráficas apareció la necesidad de encontrar soluciones que permitiesen separar las responsabilidades de la interfaz de usuario de las del resto de la lógica de la aplicación. En 1979 Trygve Reenskaug describió por primera vez uno de estos patrones (MVC), que posteriormente se haría muy popular tras ser incluido en el libro de patrones de diseño "Design Patterns: Elements of Reusable Object-Oriented Software" conocido frecuentemente como el libro de "Gang of Four" [8]. Unos años más tarde Mike Potel presentó el patrón MVP [9] con el objetivo de solucionar algunos problemas del patrón MVC. En 2004 Martin Fowler analizó esos patrones y propuso su solución al problema describiendo un nuevo patrón, Presentation Model (PM) [10]. Finalmente con la aparición de Windows Presentation Foundation (WPF) entró en escena un nuevo término, Model-View-ViewModel (MVVM).

3.3.1 WP7

MVVM es una especialización del patrón PM y está aceptado actualmente como el más adecuado para el desarrollo de de aplicaciones en Windows Phone 7. Las aplicaciones WP7 generadas por este framework siguen el patrón MVVM.

La arquitectura de MVVM define tres componentes principales:

- **Modelo**: Es la implementación del modelo de dominio de la aplicación, por lo que contiene el modelo de datos y la lógica de negocio y validación que necesita la aplicación. Generalmente incluye una capa de acceso a datos que permite almacenar y actualizar los datos bien en almacenamiento local o remotamente mediante el uso de servicios web.
- **Vista**: Provee la interfaz de usuario de la aplicación, por lo que es responsable de definir la estructura, layout y apariencia de lo que ve el usuario. Idealmente se define completamente en XAML, evitando el uso de code-behind.
- **ViewModel**: Provee una interfaz entre la vista y el modelo, es decir, actúa como intermediario entre las dos y es el responsable de manejar la lógica de la vista. El view model recupera los datos del modelo y los ofrece a la vista adaptándolos en caso de ser necesario. El view model también implementa las acciones (commands) que el usuario puede iniciar desde la vista.

MVVM define la relación entre esos tres componentes y su interacción. La vista conoce al *view model* y el *view model* conoce al modelo, pero el modelo no sabe de la existencia *del view model* ni el *view model* de la vista. Para manejar el enlace entre la vista y el *view model* se usan las capacidades de data binding de *Silverlight*. La Figura 4 muestra los distintos componentes que participan en el patrón y su interacción.

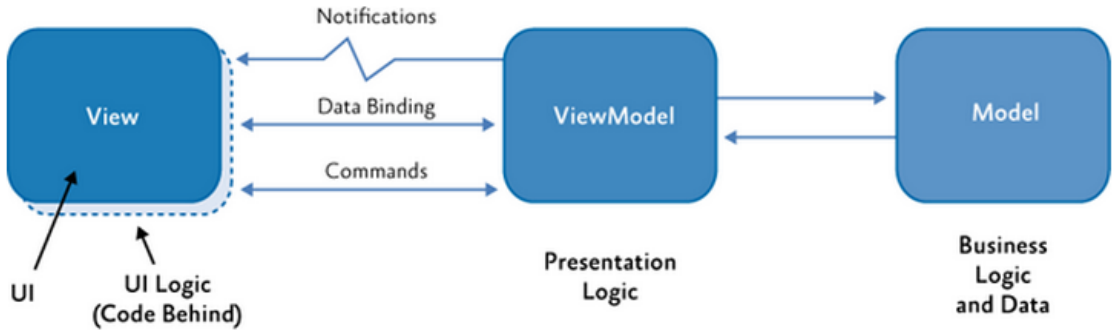


Figura 4. Interacción entre componentes en el patrón MVVM

La clara separación de responsabilidades del patrón MVVM ofrece varios beneficios:

- Permite que durante el proceso de desarrollo los programadores y los diseñadores puedan trabajar de forma concurrente e independiente en sus componentes.
- Los desarrolladores pueden crear pruebas unitarias para sus view model y modelos sin necesidad de utilizar las vistas.
- Facilita reimplementar la vista de la aplicación sin tener que realiza modificaciones en el código ya que la vista está implementada totalmente en XAML.

Existen varias alternativas para asociar la vista y el *view model*, la mayoría de frameworks existentes para implementar el patrón MVVM ofrecen diferentes alternativas basadas en *helper clases* (como *locators*). En el ámbito de nuestro proyecto se usa la librería *Prism* para la implementación del patrón MVVM. Las aplicaciones generadas hacen uso de un *Locator* que utiliza inyección de dependencias (usando *Funq* como contenedor de inyección de dependencias) para resolver instancias de las vistas y *view models*.

3.3.2 Android

En Android el uso de los patrones arquitectónicos comentados anteriormente no encaja tan bien. Aunque es frecuente leer que Android hace uso de MVC en realidad lo hace de una forma poco estricta. En Android las vistas se definen en ficheros XML, que aunque describen los componentes gráficos a usar en pantalla no son equivalentes a las vistas del patrón MVC. Los Activity del API de Android llevan a cabo parte de la responsabilidad de las vistas MVC, como es el enlace entre los datos y los componentes gráficos. Los Activity realizan también tareas propias del controlador. La filosofía del API de Android favorece la herencia frente a la

composición, para definir cada pantalla en Android se ha de heredar de alguno de los tipos de Activity, los cuales ofrecen varios "enganches" (hooks) y métodos de conveniencia.

Por lo tanto en Android aunque es posible no es intuitivo desacoplar la vista y el controlador como se define en el patrón MVC.

El código Android generado por este framework no implementa ninguno de esos patrones, pero sí que sigue las recomendaciones y buenas prácticas del API Android así como de la programación orientada a objetos.

3.4 Patrones de diseño de interfaces de usuario

Existen un gran número de patrones de interfaz de usuario para dispositivos móviles, algunos de ellos propios de una única plataforma o tipo de dispositivo concreto y otros comunes ellos y sólo diferenciados en pequeños matices. La utilización de estos patrones facilita el uso de las aplicaciones por parte de los usuarios. No hay que olvidar que la interfaz gráfica supone el mecanismo de comunicación entre los usuarios y la aplicación, por lo que una interfaz gráfica clara y bien diseñada permitirá al usuario realizar las tareas de una forma más eficiente.

Generalmente los patrones de interfaz de usuario se clasifican por su utilidad o fin, como pueden ser navegabilidad, búsqueda, ordenación, etc. Debido a las diferencias existentes a la hora de interactuar con los distintos dispositivos móviles es lógico que una misma aplicación utilice una serie de patrones de diseño diferentes en cada dispositivo. Es decir, no debe realizarse una traducción directa trasladando la misma experiencia de usuario entre las distintas versiones de una aplicación destinadas a diferentes dispositivos o plataformas, sino que en cada caso debe aplicarse la opción que más facilite el uso de la aplicación.

Existe un gran número de patrones de interfaz de usuario y en los últimos años han aparecido catálogos que permiten ver las ventajas y situaciones en las que se debe utilizar cada uno de ellos [11]. Además los distintos fabricantes de sistemas operativos suelen indicar en sus guías de estilo recomendaciones en cuanto al uso de unos u otros controles.

Uno de los mecanismos utilizados por el proyecto LIZARD para lograr adaptar la interfaz resultante al tipo de dispositivo o plataforma consiste en la aplicación del patrón de diseño de interfaz de usuario más adecuado en cada caso. A continuación veremos algunos de estos patrones, de los que haremos uso en la aplicación de ejemplo presentada en el apartado 6.1.

Dashboard

El patrón de diseño DashBoard permite al usuario el acceso rápido a las tareas que se pueden llevar a cabo con la aplicación y en muchos casos visualizar de forma rápida las novedades o notificaciones en la parte inferior de la pantalla. Muchas aplicaciones están basadas en esté patrón para implementar la pantalla de inicio. La Figura 5 muestra la pantalla inicial de la aplicación de Facebook para iOS basada en este patrón.

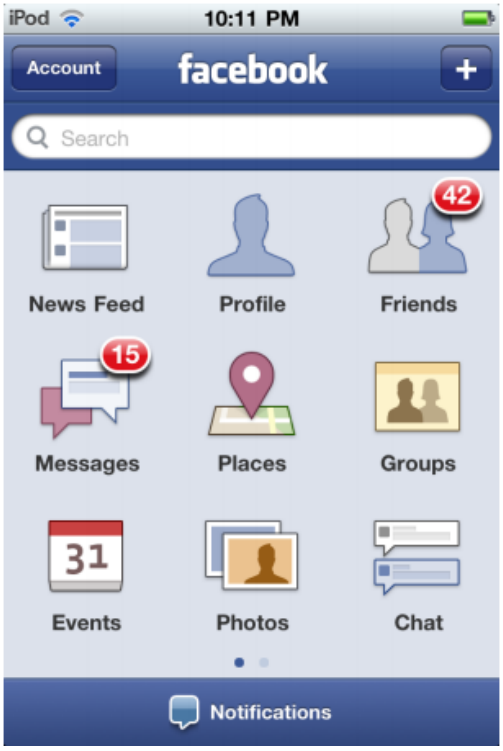


Figura 5. Aplicación de Facebook para iOS basada en el patrón DashBoard.

Action bar

Barra situada en la parte superior de la pantalla para soportar navegación y acceso a las tareas frecuentes. Sustituye a la "title bar".

Es una evolución de los menús, con los que comparte muchos de sus casos de uso pero es más efectivo ya que permanece siempre visible para el usuario.

Se suele dividir en tres partes cada una con un objetivo:

1. App icon: *¿dónde estoy?*
2. View details: *¿Qué puedo ver?*
    a. Simple: No interactivo, título de la aplicación
    b. Rico: Pestañas, drop-down menús, migas de pan, etc.
3. Action buttons: *¿Qué puedo hacer?*

El Action bar permite el uso de pestañas para visualizar distintas secciones de una aplicación (Figura 6).

Figura 6. Ejemplo de action bar con pestañas

El Action bar puede cambiar su apariencia cuando se selecciona ítems para permitir acciones contextuales (Figura 7). Incluye también 3 secciones:

1. Botón "Hecho" que deshace la selección.

2. Detalles de selección.

3. Action buttons. Acciones que podemos realizar con los elementos seleccionados.

Figura 7. Ejemplo de Action bar con acciones contextuales

Multi-pane Layout / Split View

Este patrón se usa para presentar datos maestro-detalle, se conoce como Multi-pane Layout en la plataforma Android y Split View en iOS. Es de gran utilidad en tablets ya que permiten incluir varias pantallas relacionadas en una única ventana compuesta. Generalmente el panel de la derecha muestra el contenido o detalle de los ítems seleccionados en el panel de la izquierda. En Android su implementación se realiza mediante el uso de Fragments. La Figura 8 muestra la representación de este patrón en teléfonos móviles, mientras que en la Figura 9 se muestra su equivalencia en dispositivos de tipo tablet.

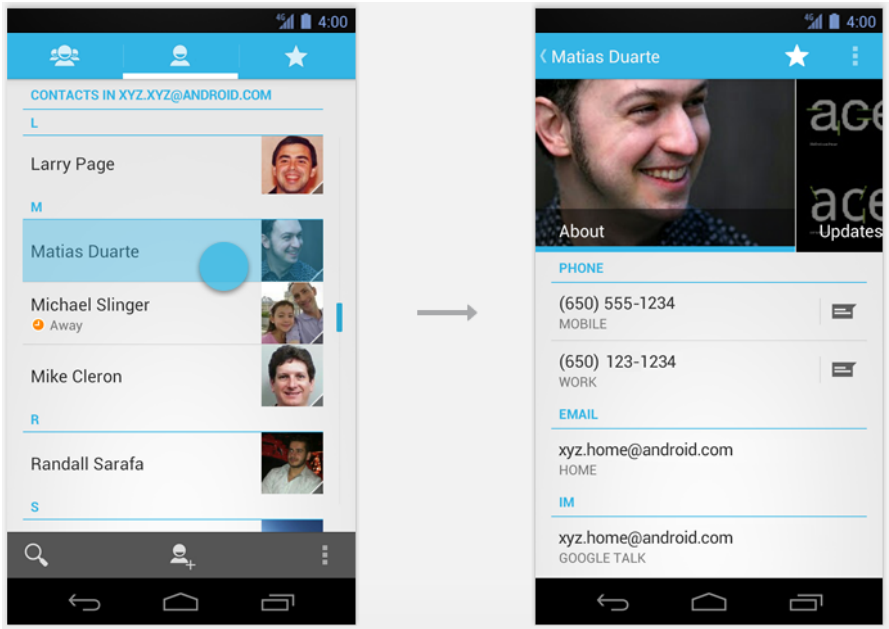


Figura 8. Multi-pane en smartphone. Vista de selección y de detalle en diferentes pantallas.

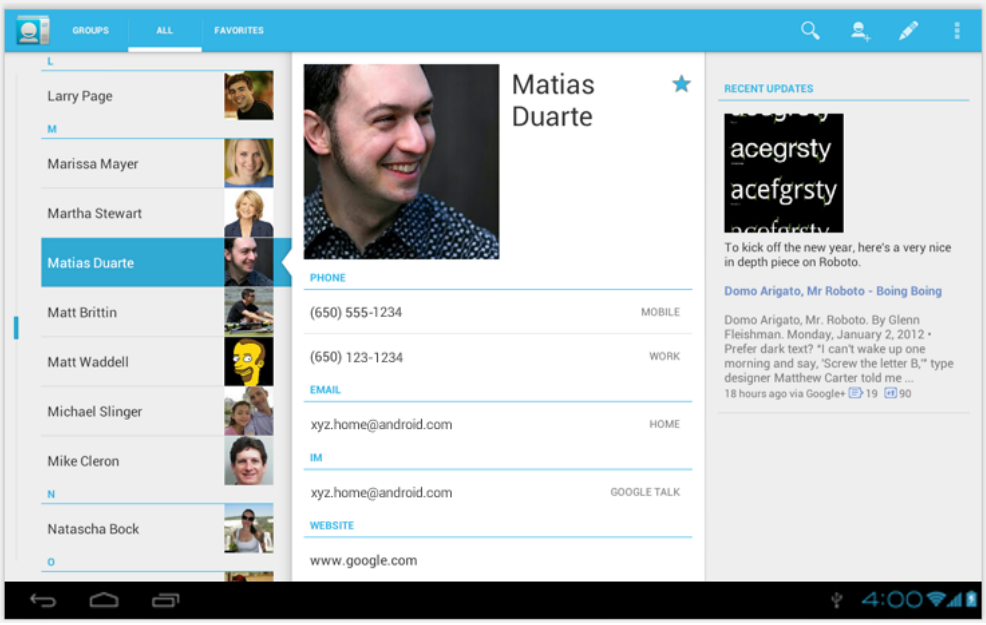


Figura 9. Multi.pane en tablet. Lista en el panel de la izquierda y vista de detalle en el panel de la derecha de la misma pantalla.

3.5 User Interface Description Language (UIDL)

Uno de los aspectos fundamentales de una aplicación es su interfaz de usuario, ya que permite al usuario interactuar con dicha aplicación para llevar a cabo cada una de las tareas que la aplicación ofrece. Por lo tanto para la construcción de un sistema de generación de aplicaciones es vital disponer de un método que permita describir todos los aspectos que intervienen en su creación. El trabajo propuesto permite cubrir el desarrollo completo de aplicaciones para Android y WP7 pero hace hincapié principalmente en la generación de la interfaz gráfica de usuario.

Un User Interface Description Languaje (UIDL) [12] es un lenguaje formal usado en Human-Computer Interaction (HCI) con el fin de permitir describir una interfaz de usuario de forma independiente a su implementación.

Actualmente existen diferentes propuestas para la descripción y generación de interfaces de usuario diferenciadas principalmente por su alcance y los aspectos que describen, clasificándose principalmente en dos grupos:

- Lenguajes destinados a cubrir la mera descripción de interfaces de usuario: UIML, XUL, XAML, Xforms, o AUIML.
- Lenguajes desarrollados para dar soporte al desarrollo basado en modelos, no se limitan a describir la interfaz sino que facilitan mecanismos para describir, dependiendo de los casos, al usuario, sus tareas, su dominio, el contexto, etc.: XIML, UsiXML o Teresa-XML.

Atendiendo únicamente a la interfaz de usuario en sí es necesario seleccionar un UIDL que permita describir los siguientes aspectos:

- La estructura estática (elementos de la UI, y su composición)
- El comportamiento dinámico (eventos, la parte de diálogo, acciones y restricciones de comportamiento)
- Atributos de presentación (look & feel y propiedades de los elementos de la UI)

Aunque disponer de una descripción completa de cada uno de los aspectos vistos anteriormente sería suficiente para la generación de la interfaz no debemos olvidar que en su creación intervienen otros aspectos como la experiencia de sus usuarios, el contexto de uso, las técnicas de interacción, etc.

Los esfuerzos investigadores de la última década se centran en la aplicación de lenguajes basados en modelos y técnicas propias de Model-driven engineering (MDE) destinadas a definir varios meta-modelos con el fin de describir rigurosamente la semántica de los UIDLs.

A continuación describiremos de forma somera las características de alguno de estos lenguajes.

3.5.1 UIML (User Interface Markup Language)

UIML [13] es uno de los lenguajes más utilizados para la descripción de las interfaces de usuario. A pesar de ser independiente del dispositivo específico y del medio utilizado en la presentación, no tiene en cuenta el trabajo de investigación llevado a cabo en la última década sobre enfoques basados en modelos para interfaces de usuario. Por ejemplo, el lenguaje no provee la noción de tarea y principalmente intenta definir una estructura abstracta asociada a la presentación que ofrecerá la interfaz de usuario.

Permite la traducción automática al lenguaje utilizado por el dispositivo final. El proceso de traducción de la interfaz se realiza en el propio dispositivo o en el servidor, dependiendo principalmente de las capacidades del dispositivo del que se trate.

Para describir una interfaz de usuario en UIML se debe realizar por un lado la definición de la interfaz genérica, y por otro un documento UIML que representa el estilo de presentación apropiado para el dispositivo en el cuál la interfaz de usuario se va a ejecutar. De este modo, una aplicación necesitará un único documento UIML de especificación válido para cualquier dispositivo y un documento de estilo propio para cada dispositivo. La flexibilidad para la selección de dispositivos finales es limitada, ya que aunque la parte correspondiente a la interfaz genérica puede mantenerse independientemente a pesar de que aumente el número de dispositivos finales, no sucede lo mismo con la parte que mapea a los dispositivos específicos, que crece con dicho número. Esto incrementa además, evidentemente, el coste de mantenimiento.

Otras de sus limitaciones son que no separa claramente la definición de la interfaz de su renderización y que no está pensado para soportar funciones de gestión del conocimiento. Por ejemplo un fabricante de dispositivos móviles podría usar XIML para almacenar y manejar toda la información relevante a las interfaces de usuario de sus productos, lo cual no es posible mediante el uso de UIML.

3.5.2 XIML (eXtensible Interface Markup Language)

XIML [13] es un lenguaje extensible basado en XML para soportar múltiples modelos de especificación de interfaces de usuario. Una especificación XIML puede conducir a una interpretación en tiempo de ejecución o a una fase de generación de código en tiempo de diseño. XIML surgió como evolución de una propuesta anterior: MIMIC.

El objetivo de este lenguaje es describir la interfaz de usuario de forma abstracta y, posteriormente, obtener una especificación concreta a partir de ella. Se utiliza para ello un desarrollo basado en modelos. Asociado a XIML, se define un modelo de interfaz mediante el uso de cinco modelos adicionales ligados a las tareas, dominio, usuario, diálogo y presentación, junto con el tratamiento de las asociaciones entre modelos, consideradas sólo en sentido descendente. Esto puede ser una limitación frente a UsiXML, que contempla 3 tipos de relaciones entre modelos: Abstracción, cosificación y traducción.

Destaca por su flexibilidad, siendo extensible, lo que permite la definición y declaración de nuevos modelos y relaciones. Aunque parezca una ventaja esto puede convertirse en un problema, ya que si esta definición no es compartida por varias herramientas la definición será local.

Uno de sus principales puntos fuertes es que ofrece una estricta separación entre la definición de la UI y su renderización.

XIML provee un framework para el desarrollo de UI para múltiples dispositivos. Hay una única especificación XIML de la interfaz para los datos a mostrar, la navegación y las tareas del usuario soportadas. Para que la especificación entera sea soportada en múltiples plataformas es necesario definir un componente de presentación para cada dispositivo objetivo. Definir un componente de presentación significa simplemente determinar que widgets, interactuadores y controles serán usados para visualizar cada ítem de datos en cada dispositivo. Tener que crear un componente de presentación para cada dispositivo es un problema, para evitar eso XIML ofrece capacidades adicionales que proveen un alto grado de automatización en el proceso de desarrollo de interfaces multiplataforma. En lugar de crear y manejar un componente de presentación por dispositivo, se trabajaría con un único componente de presentación "intermediario", y XIML se encargaría de definir utilizando relaciones cómo el intermediario mapea los componentes a un widget específico.

En lo relativo a la renderización de la interfaz un dispositivo con capacidad de procesar XML es capaz de procesar directamente la especificación XIML. Si el dispositivo no tiene esa capacidad será necesario usar un convertidor para producir el lenguaje final.

Otra característica que puede resultar interesante es la existencia de herramientas de ingeniería inversa para convertir interfaces desarrolladas en HTML a su equivalente en XIML.

Como aspectos más negativos hay que destacar que, por defecto, no soporta el modelo de contexto, que se centra más en aspectos sintácticos que en semánticos y que las herramientas de soporte no están disponibles de forma pública.

3.5.3 TeresaXML

El lenguaje TeresaXML [4] fue desarrollado para representar las interfaces de usuario diseñadas con la herramienta *Transformation Environment for inteRactivE Systems representAtions* (TERESA). TERESA utiliza dos representaciones XML para representar los modelos de la interfaz de usuario: *ConcurTaskTree* (CTT) y XAUI.

CTT describe los modelos de tareas del usuario mientras XAUI describe el diseño de la interfaz abstracta de usuario. XAUI específica cómo se organizan los AIO (*Abstract Interactor Object*) que componen la interfaz de usuario, junto con la especificación del diálogo de la propia interfaz. XAUI está totalmente separado de los detalles de la plataforma y del dispositivo, y requiere de una transformación para producir una interfaz de usuario concreta.

3.5.4 UsiXML (User Interface eXtensible Markup Language)

El lenguaje UsiXML [2] puede ser usado para especificar interfaces de usuario independientes de la plataforma, del contexto y de la modalidad.

Propone un lenguaje de descripción de interfaces de usuario con el que se asegura poder trabajar a diferentes niveles de abstracción (tareas y conceptos, interfaz abstracta, interfaz concreta, interfaz final), estos cuatro niveles de abstracción fueron definidos en el Cameleon Reference framework con la intención de poder expresar el ciclo de vida del desarrollo de UIs.

UsiXML propone además mecanismos para gestionar las asociaciones que se establecen entre los diferentes modelos contemplados.

En el nivel más alto de abstracción (tareas y conceptos) se tienen en cuenta tres modelos (tareas, dominio y contexto) para llevar a cabo la representación de las tareas suele usarse la notación CTT [15]. En el modelo de dominio se utilizan diagramas de clases UML para modelar los objetos del mundo real y sus relaciones. El modelo de contexto incluye un modelo de usuario que cataloga a los distintos usuarios en perfiles, un modelo de plataforma que captura todos los atributos relevantes de la dupla software-hardware y modelo del entorno que describe cualquier propiedad relevante del entorno físico en el que el usuario utiliza la UI.

En el nivel de interfaz abstracta el modelo AUI representa la interfaz de forma independiente a la modalidad y la plataforma. Un AUI está compuesto por objetos de interacción abstractos (AIO) que representan una abstracción de los widgets encontrados en la mayoría de toolkits tanto para interfaces visuales como vocales pero a un nivel de abstracción elevado en el que sólo se indica la intención del componente (entrada de datos, visualización, etc.).

En el nivel de interfaz concreta el modelo CUI concretiza una AUI para un contexto de uso dado usando CIOs.

En el último nivel, el nivel de interfaz de usuario final, la interfaz de usuario se expresa en el código fuente de la plataforma objetivo.

UsiXML permite definir las relaciones espacio temporales entre AICs a nivel de AUI mientras que los aspectos visuales (look & feel) se definen a nivel de CUI.

Un aspecto interesante para la selección de UsiXML es la gran cantidad de herramientas disponibles para los distintos niveles de abstracción (IdealXML, FlowiXML, TransformiXML, GrafiXML, SketchiXML, ComposiXML, InterpiXML,...). Muchas de ellas de código abierto y disponibles públicamente en el sitio Web de UsiXML [16].

3.5.5 Conclusiones sobre UIDLs

Parece que las aproximaciones basadas en modelos ofrecen una mayor separación de conceptos en cuanto a la descripción de todos los aspectos necesarios para la obtención de una interfaz de usuario. UsiXML parece el mejor candidato ya que cubre la mayoría de objetivo y permite definir la interfaz a diferentes niveles de abstracción. Sin embargo UsiXML utiliza conceptos como el nivel abstracto cuya única aportación es la independencia de la modalidad de uso. En nuestro caso al centrarnos exclusivamente en la modalidad de interfaces gráficas este nivel no tendría sentido. Otro aspecto de UsiXML que no encaja del todo bien en nuestros requisitos es que no ofrece un modelo de diálogo claro que indique cual es el flujo de ejecución de la aplicación ni mecanismos para desacoplar totalmente la interfaz de usuario del resto de la lógica de la aplicación. Por todo lo anterior se ha decidido crear nuevos meta-modelos basados en conceptos de UsiXML pero con ciertas diferencias.

En los siguientes apartados se describirá el framework CAMELEON, como framework de referencia para la clasificación de interfaces de usuario a diferentes niveles de abstracción y contextos de uso y se entrará a ver en mayor detalle las características del lenguaje UsiXML indicándose en que se diferencia la propuesta ofrecida en este trabajo.

3.6 Framework CAMELEON

El framework CAMELEON [1] fue obtenido como resultado del proyecto europeo CAMELEON [17] perteneciente al Quinto Programa Marco. El objetivo del proyecto consistía en la creación de una plataforma para la generación de interfaces de usuario sensibles al contexto de uso.

El framework CAMELEON estructura el ciclo de vida del desarrollo de interfaces de usuario en cuatro niveles de abstracción, desde la especificación de las tareas hasta la interfaz ejecutable final. Los cuatro niveles propuestos son los siguientes:

- **Tareas y conceptos**: se corresponde con el Computational Independent Model (CIM) de MDE y engloba las actividades que el usuario puede lleva a cabo con el uso de la interfaz, generalmente estas actividades se representan jerárquicamente indicándose las relaciones temporales entre ellas.
- **Abstract User Interface (AUI)**: se corresponde con el Platform Independent Model (PIM) de MDE y constituye la descripción de la interfaz en términos de "Presentation Units" (agrupaciones para la ejecución de un conjunto de tareas lógicas conectadas) de forma independiente a la modalidad de interacción (gráfica, vocal, táctil, etc.).
- **Concrete User Interface (CUI)**: se corresponde con el Platform Specific Model (PSM) de MDE y constituye la descripción de la interfaz en términos de "Concrete Interactors" (abstracción de los componentes de los toolkits para el desarrollo de UI) que dependen de la modalidad y de una serie de atributos que definen cómo van a ser percibidos por los usuarios.
- **Final User Interface (FUI)**: se corresponde con el nivel de código de MDE y consiste en el código fuente final en cualquier lenguaje de programación o lenguaje de marcado.

Los niveles descritos anteriormente se estructuran con una relación de *cosificación* que permite ir de un nivel abstracto a uno concreto, una relación de *abstracción* de un nivel concreto a un nivel abstracto y una tercera relación de *traducción* entre dos modelos del mismo nivel de abstracción pero definidos para diferentes contextos de uso.

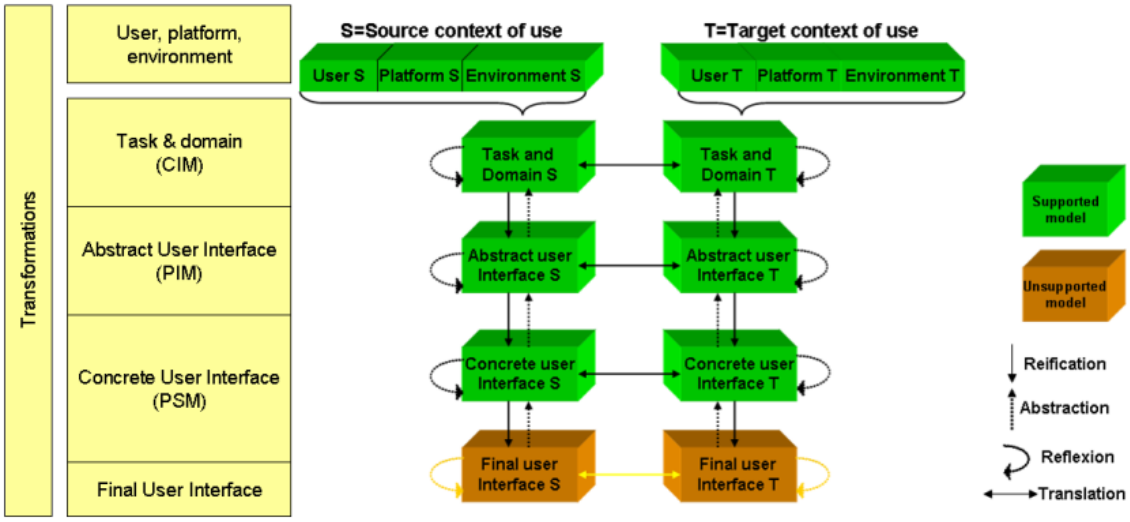


Figura 10. Niveles definidos en CAMELEON y relaciones entre ellos

La Figura 10 muestra los cuatro niveles definidos por el framework CAMELEON y las relaciones entre ellos descritas anteriormente. El lenguaje UsiXML se basa en estos cuatro niveles de abstracción para describir las interfaces de usuario. UsiXML define un conjunto de meta-modelos en cada uno de los niveles de abstracción.

Como explicaremos más detalladamente en apartado 4.1 del siguiente capítulo este trabajo se basa en algunos de estos conceptos, pero en lugar de utilizar los cuatro niveles de abstracción de CAMELEON se utilizan tres niveles. La eliminación de uno de estos niveles está motivada principalmente por el uso de una única modalidad, interfaces de usuario gráficas.

3.7 UsiXML

Anteriormente en el estudio desarrollado sobre UIDLs en este mismo capítulo vimos las características principales de UsiXML y comentamos los motivos por los que lo usaríamos como base para la definición de nuestros meta-modelos. En este apartado veremos brevemente los meta-modelos propuestos por UsiXML que servirán de base para algunos de los meta-modelos propuestos en este trabajo.

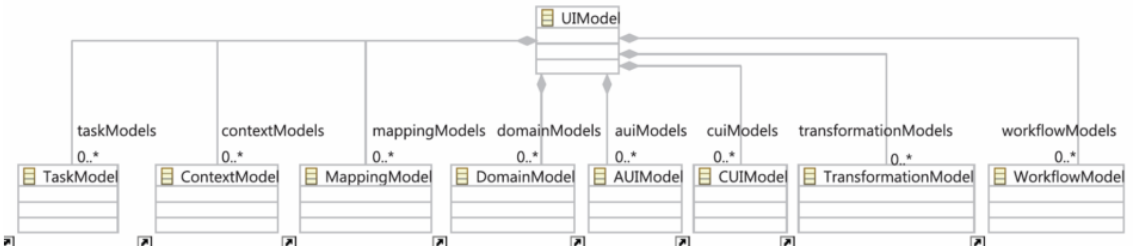


Figura 11. Modelos que componen el meta-modelo propuesto por UsiXML

La Figura 11 muestra el conjunto de modelos definidos en UsiXML para la descripción de las interfaces de usuario. A continuación iremos viendo los modelos de mayor relevancia y en los que nos basaremos para la definición de nuestro sistema.

3.7.1 Modelo de tareas

El modelo de tareas de UsiXML expresa las tareas que puede realizar un usuario de la aplicación a través de la interfaz de usuario, así como una especificación de las relaciones temporales que existen entre dichas tareas. Este modelo permitirá al desarrollador definir una estructura jerárquica de tareas, descomponiéndolas en subtareas y permitiendo expresar tanto las relaciones temporales existentes entre ellas como una serie de atributos que influirán en el resultado final de la interfaz de usuario.

3.7.2 Modelo de dominio

El modelo de dominio de UsiXML describe el conjunto de entidades manipuladas por el usuario al interactuar con el sistema. Generalmente estos conceptos se modelan en la programación orientada a objetos con diagramas de clases de UML, por lo que el modelo de dominio de UsiXML está basado en el modelo de clases de UML.

3.7.3 Modelo de interfaz abstracta

El modelo de interfaz abstracta de UsiXML define la interfaz de usuario en términos de *Presentation Units* independientemente de los elementos de interacción disponibles e incluso de forma independiente a la modalidad de interacción (vocal, táctil, gráfica, etc.).

3.7.4 Modelo de interfaz concreta

El modelo de interfaz concreta de UsiXML permite la especificación de la apariencia y el comportamiento de una interfaz de usuario con elementos que pueden ser percibidos por los usuarios. Concretiza la AUI para un contexto de uso

3.7.5 Modelo de transformación

El modelo de transformación describe el proceso de transformación entre los distintos niveles de abstracción de UsiXML. Contiene un conjunto de reglas que habilitan la transformación de una especificación (en un cierto nivel de abstracción) a otra o la adaptación a un nuevo contexto de uso. Una regla de transformación está compuesta por:

- **LHS (Left-Hand Side)**: modela la parte de la regla que debe casar con el modelo a transformar para que dicha regla se ejecute, es decir, es una precondición.
- **RHS (Right-Hand Side)**: modela la parte de la regla que remplazará sobre la instancia del modelo el grafo que case con la LHS.
- **NAC**: modela una condición de aplicación que tiene que mantenerse falsa para poder casar la LHS con el modelo.
- **AttributeCondition**: expresión indicando una condición de ámbito sobre atributos de los elementos de la parte izquierda de una regla de transformación.
- **RuleMapping**: define los modelos origen y destino de la regla de transformación.

UsiXML contempla tres tipos de transformaciones dependiendo de la dirección en la que se realice la transformación:

- **Cosificación**: transformación de un nivel abstracto a uno concreto.
- **Abstracción**: transformación de un nivel concreto a uno abstracto.
- **Traducción**: transformación entre dos modelos del mismo nivel de abstracción pero definidos para diferentes contextos de uso

3.8 Reglas de transformación

La aproximación a seguir en este trabajo pasa por realizar una descripción de la interfaz de usuario a un nivel alto de abstracción para luego transformar esa representación a un nivel más concreto adaptado a la plataforma y tipo de dispositivo. Es necesario por lo tanto utilizar un conjunto de reglas de transformación que nos permita obtener dicha representación concreta.

En el apartado 3.8.1 veremos un conjunto de sistemas de transformaciones y sus principales características, dichos sistemas de transformación pueden clasificarse según sus formalismos en tres grupos:

- Sistemas basados en la transformación algebraica de grafos.
- Sistemas basados en le transformación de modelos.
- Motores de reglas tradicionales basados en el algoritmo Rete.

Una vez vistos los principales sistemas de transformación en el apartado 3.8.2 veremos un conjunto de propiedades interesantes a tener en cuenta para la elección de uno de estos sistemas. Finalmente en el último apartado de este punto (3.8.4) veremos las conclusiones obtenidas tras el estudio de todos estos sistemas y justificaremos la elección de Henshin como sistema de transformaciones para este trabajo.

3.8.1 Sistemas de transformaciones

3.8.1.1 AGG

AGG [18] es un lenguaje de reglas visual basado en la transformación algebraica de grafos. Su objetivo es la especificación y la implementación de aplicaciones con datos complejos estructurados en forma de grafos. AGG se puede utilizar como motor de transformación de uso general en aplicaciones Java usando métodos de transformación de grafos.

Características:

- Basado en grafos.
- Provee un lenguaje de programación para la especificación de gramáticas de grafos y un intérprete para la transformación de grafos.
- Declarativo.
- Precondiciones: LHS para las condiciones y posibilidad de añadir condiciones adicionales en Java.
- Permite el uso de NAC (Negative Application Conditions) para expresar el requisito de la no existencia de subestructuras.
- Ofrece técnicas de validación para encontrar conflictos entre reglas.
- El motor de transformación de grafos puede ser usado desde Java.
- Sólo permite establecer un orden fijo en la aplicación de reglas, no permite definir un flujo de control, aunque utiliza "capas de reglas". El proceso de interpretación intenta ejecutar primero las reglas de la capa 0, tan pronto como sea posible las de la 1 y así sucesivamente.

3.8.1.2 AToM3

AToM3 [19] es una herramienta para el meta-modelado y la transformación de modelos. El meta-modelado se refiere a la descripción, o la modelización de diferentes tipos de formalismos utilizados para modelar sistemas. La transformación de modelos se refiere al proceso automático de convertir, traducir o modificar un modelo en un formalismo dado, en otro modelo que podría o no estar en el mismo formalismo.

Características:

- Trabaja con meta-modelos.
- Declarativo.
- Precondiciones: LHS para las condiciones y posibilidad de añadir condiciones adicionales en Python.
- Postcondiciones: Código Python, no soporta rollback tras su ejecución.
- Permite ejecutar pre y post acciones antes y después de la ejecución de la regla.
- Soporta prioridad en las reglas.
- Sólo permite establecer un orden fijo en la aplicación de reglas, es decir, no permite definir un flujo de control.

3.8.1.3 VIATRA2

El framework VIATRA (VIsual Automated model TRAnsformations) [20] es el núcleo de un entorno basado en transformaciones, verificación y validación para mejorar la calidad de los sistemas diseñados utilizando el Lenguaje de Modelado Unificado (UML) comprobando de forma automática la consistencia, integridad, y fiabilidad.

Características:

- Trabaja con meta-modelos.
- Declarativo e imperativo.
- Escrito en Java y totalmente integrado en Eclipse.
- Precondiciones: LHS, patrones recursivos de grafos como lenguaje de consulta.
- Postcondiciones: Patrones de grafos con atributos condicionales.
- Para controlar el orden de ejecución de las reglas utiliza máquinas de estado abstractas.
- Permite el uso de funciones nativas (métodos Java que pueden ser invocados desde las transformaciones, estos métodos pueden utilizar cualquier librería Java y acceder al espacio de modelo de VIATRA).

3.8.1.4 VMTS

Visual Modeling and Transformation System (VMTS) es un entorno de propósito general para meta-modelos y transformaciones. VMTS es un entorno altamente configurable que ofrece capacidades para la especificación de lenguajes visuales aplicando técnicas de meta-modelado. VMTS define las restricciones del modelo en OCL [21], mientras que la definición del flujo de control se define mediante diagramas de actividad.

Características:

- Trabaja con meta-modelos.
- Precondiciones: LHS y OCL.
- Postcondiciones: OCL.
- Permite controlar el flujo de ejecución de las reglas.

3.8.1.5 QVT

QVT es un estándar OMG que aborda la necesidad de la normalización de las transformaciones llevadas a cabo entre modelos cuyos lenguajes se definen usando MOF. Además, QVT ofrece una forma estándar de consultar los modelos MOF, y la creación de vista sobre estos modelos. Las consultas en los modelos MOF se requieren tanto para filtrar y seleccionar elementos de un modelo, así como para seleccionar los elementos que son las fuentes para las transformaciones.

La especificación QVT define tres lenguajes, dos declarativos y uno imperativo.

- **Core**: Un meta-modelo y lenguaje definido mediante las extensiones mínimas de EMOF y OCL.

- **Relations**: Un meta modelo de relaciones y un lenguaje que soporta patrones de objetos complejos y la creación de plantillas de objetos. Se crean de forma implícita trazas para grabar lo que sucede durante la ejecución de una transformación. Está relacionado con Core, ya que el lenguaje Relations se transforma en lenguaje Core para su ejecución. El lenguaje Core es semánticamente equivalente al lenguaje Relations pero definido a un nivel de abstracción más bajo. En el lenguaje Relations, una transformación entre modelos candidatos se especifica como un conjunto de relaciones que deben mantenerse para que la transformación sea satisfactoria.
- **Operational**: El lenguaje Operational pretende ofrecer una alternativa imperativa y puede ser invocado desde los lenguajes Relations y Core en forma de "caja negra". OML es un lenguaje procedural basado en Object Constraint Language (OCL) que provee un método de bajo nivel para la definición de transformaciones model-to-model. OCL provee la "Query" en QVT. Con bajo nivel se refiere a la diferencia entre OML y su homónimo de alto nivel, QVT Relations language.

3.8.1.6 ATL

ATL [22] (Atlas Transformation Language) es un lenguaje híbrido (mezcla de imperativo y declarativo) diseñado para expresar transformaciones entre modelos. ATL está descrito por una sintaxis abstracta (MOF meta-modelo), y una sintaxis textual concreta. Un modelo de transformación en ATL se expresa como un conjunto de reglas de transformación. Se recomienda el estilo de programación declarativa. La librería de transformaciones ATL está liberada bajo licencia EPL. Es un lenguaje del estilo QVT pero actualmente no es compatible con los 12 niveles de cumplimiento establecidos en el documento de la OMG. ATL está basado en una máquina virtual de transformación. Su arquitectura es muy modular de forma que puede evolucionar añadiéndole nuevas funcionalidades o mejorar su rendimiento. El código completo de la máquina virtual está disponible y está previsto establecer conexiones entre ATL y otros lenguajes de transformación. ATL forma parte del proyecto M2M de Eclipse [23]

Características:

- Soporta meta-modelos MOF y Ecore
- Como lenguaje de transformación usa OCL con algunas extensiones necesarias para trabajar con meta-modelos no soportados por OCL
- Mayor soporte de herramientas que QVT
- Permite lanzar las transformaciones programáticamente usando la clase AtlLauncher
- No es estándar

3.8.1.7 DROOLS

Drools [24] es un sistema de gestión de reglas de negocio con un motor de reglas basado en el algoritmo Rete [25]. Drools soporta el estándar JSR-94 de Java. La JSR-94, define un API que ofrece un acceso simple a un motor de reglas desde la plataforma Java. Las implementaciones de este estándar deben permitir:

- Registrar y desregistrar reglas.
- Parsear reglas.
- Inspeccionar los metadatos de las reglas.

- Ejecutar reglas.
- Recuperar el resultado de la ejecución de las reglas.
- Filtrar los resultados.

Sin embargo la JSR-94 no estandariza lo siguiente:

- El motor de reglas en sí
- El flujo de ejecución de las reglas
- El lenguaje usado para describir las reglas

El API de Drools está definido en dos partes principales:

- **Rules Administrator API**: Provee clases para la carga de reglas y las acciones asociadas a un conjunto de reglas. Permite la carga de reglas desde recursos externos como URIs, InputStream, XML Element, etc. Provee también métodos para el registro y desregistro de colecciones de reglas. Este paquete puede ser usado también para la definición de permisos y para proveer autorización de acceso a los recursos.
- **Runtime Client API**: Provee clases que los clientes pueden usar para la ejecución de reglas y obtención de resultados. Posibilita a los clientes la adquisición de una sesión y la ejecución de reglas dentro de esa sesión.

Características:

- Permite utilizar POJOs desde las reglas.
- Drools nos permite interactuar con el motor de reglas a través de la interfaz *StatefulKnowledgeSession*.
- Permite la carga de reglas desde el *Classpath*, URL, array de bytes, InputStream, Reader y File. Además RuleAgent permite comprobar cada cierto tiempo si hay nuevas reglas en el repositorio para añadirlas.

Un concepto importante en Drools es el de Agenda. Cuando se añade un objeto a la "knowledge session" Drools intenta casar ese objeto con las reglas. Si se cumplen todas las condiciones de una regla puede ser ejecutada (se activa). Drools añade la regla a la agenda, posteriormente el método *fireAllRules Drools* cogerá una regla de la agenda para su ejecución, lo que puede provocar activaciones y desactivaciones de otras reglas. El proceso continua hasta que la agenda está vacía.

Otro concepto interesante es el de *Ruleflow*, que permite gestionar externamente el orden de ejecución de las reglas. Básicamente consiste en ficheros XML conforme a un XML Schema que permiten definir el flujo de ejecución de las reglas. Como alternativa a *Ruleflow* existen varios métodos para controlar el orden de ejecución:

- **Salience**: valor numérico para indicar la prioridad de ejecución.
- **Activation-group**: cuando varias reglas con el mismo activation-group están en la agenda Drools sólo ejecuta una.
- **Agenda-group**: permite particionar la Agenda en varios grupos que pueden ser ejecutados de forma separada.

3.8.1.8 EMF Tiger

EMF Tiger [26] es un proyecto de código abierto que provee una transformación de modelos basada en modelos de datos estructurados y en conceptos de transformación de grafos.

Las transformaciones se definen y ejecutan directamente sobre los modelos EMF, asegurando seguridad en los tipos y eficiencia.

El lenguaje de transformación basado en reglas de EMF Tiger está inspirado en conceptos de transformación de grafos y combina conceptos tanto declarativos como procedurales. Las reglas de transformación son declarativas (patrones estructurales para la definición de precondiciones de la regla) mientras que los conceptos procedurales (capas, bucles,...) se usan para aplicar un conjunto de reglas de forma controlada.

EMF Tiger está formado por:

- Un editor gráfico para definir de forma visual reglas de transformación de modelos EMF.
- Un compilador para la generación de código Java a partir de las reglas de transformación.
- Un intérprete que traduce las reglas de transformación a AGG, lo que resulta útil para la verificación.

El intérprete que permite el uso de AGG para la verificación supone una ventaja frente a otras aproximaciones parecidas como ATL y QVT.

3.8.1.9 Henshin

El proyecto Henshin [27] ofrece un lenguaje de transformación de modelos *in-place* (trabaja directamente sobre el modelo) para Eclipse Modeling Framework. Henshin es el sucesor de EMF Tiger y al igual que éste está basado también en conceptos de transformación de grafos sin embargo Henshin extiende considerablemente el lenguaje de transformaciones de EMF Tiger.

Características:

- Soporta transformaciones endógenas y exógenas. Ofreciendo una gran eficiencia en las transformaciones endógenas ejecutadas *in-place*.
- Ofrece un lenguaje declarativo de transformación de modelos muy potente. Además de los conceptos básicos de reglas de transformación está enriquecido con potentes condiciones de aplicación y un cálculo de atributos flexible basado en Java o JavaScript.
- Utiliza *transformation units* para la definición de estructuras de control de aplicación de reglas de forma modular. Permite tanto una selección no-determinista de reglas como determinista (asignación de prioridad a las reglas, bucles, etc.).
- En ATL el modelo fuente es de sólo lectura y el modelo objetivo es de sólo escritura (no navegable). Henshin no tiene esa restricción.
- Las transformaciones exógenas requieren un meta-modelo de tracking adicional, no es automático como en el caso de ATL.
- Al estar basado en conceptos de transformación de grafos es posible traducir las reglas a AGG para analizar conflictos y dependencias en la aplicación de reglas y su terminación.
- Ofrece una gran integración con otras herramientas de EMF, lo que puede resultar útil para fases posteriores como la generación de código.

3.8.2 Verificación y validación de transformaciones

Cuando se realizan transformaciones es sumamente importante garantizar ciertas propiedades tanto de las reglas de transformación como en los resultados obtenidos. Por ejemplo, se debería asegurar que varias reglas no resulten incompatibles o que la ejecución de una regla no cause que el sistema se quede colgado o no finalice su ejecución por entrar en un bucle infinito.

En este apartado veremos algunos de los requisitos que sería interesante que cubriesen los sistemas de transformación vistos anteriormente. Posteriormente veremos varias técnicas usadas por los sistemas de transformación para garantizar el cumplimiento de esos requisitos. Finalmente se mostrará cuáles de esas características se implementan en algunos de esos sistemas de transformación.

El sistema implementado en este trabajo es simplemente un prototipo cuya finalidad es cubrir los objetivos reflejados en el apartado 2.1. De momento no realiza una aplicación práctica de estas técnicas formales de validación y verificación, sin embargo, este estudio teórico sí que se ha tenido en cuenta para la elección de uno de estos sistemas. El uso de estas técnicas se considera muy adecuado para la futura extensibilidad del sistema con el fin de garantizar su calidad tras la incorporación de nuevas reglas de transformación o nuevas plataformas.

3.8.2.1 Requisitos

Los requisitos mínimos que debería cumplir un modelo de transformación son sintácticos, sin embargo para garantizar una calidad alta del modelo de transformación se debería verificar también ciertos requisitos semánticos [28][29].

- **Requisitos sintácticos** (estáticos, verificados mediante "*planner algorithms*")
  - *Syntactic correctness*: garantiza que el modelo generado es una instancia sintácticamente bien formada del lenguaje objetivo.
  - *Syntactic completeness*: requisito que comprueba que el lenguaje fuente está completamente cubierto por las reglas de transformación, por ejemplo, probar que existe un elemento correspondiente en el modelo objetivo para cada construcción en el lenguaje fuente.
- **Requisitos semánticos**
  - *Correctness* (dynamic consistency): como las transformaciones de modelos pueden definir proyecciones del lenguaje fuente al lenguaje objetivo la equivalencia semántica no siempre se puede probar. En lugar de eso se definen *correctness properties* (propiedades que deben cumplirse, normalmente específicas de cada transformación).
  - *Termination*: garantiza que una transformación de modelos termina.
  - *Uniqueness* (*confluence*): cuando se realiza una selección no determinista de las reglas debemos garantizar que la transformación obtiene un único resultado.

Resulta muy importante saber si un sistema de transformación de grafos muestra un *comportamiento funcional (*presenta *terminación* y *confluencia)*, por lo que es un tema muy investigado en teoría de sistemas de transformación de grafos, sin

embargo muchos de los resultados obtenidos son negativos. En general no se puede garantizar la terminación de un sistema de transformación de grafos [30], y para conseguirlo se deben diseñar las reglas cuidadosamente. La principal línea de investigación consiste en definir condiciones suficientes de *terminación* y *confluencia*. Por ejemplo *critical pair analysis* es una técnica de análisis estático que detecta reglas conflictivas que pueden violar la *confluencia*.

3.8.2.2 Técnicas de verificación y validación

3.8.2.2.1 Critical pair analysis

El análisis de los *pares críticos* [18] es muy usado en sistemas de reescritura de grafos, generalmente para comprobar si el sistema tiene un comportamiento funcional, es decir, si es *confluente*. Los *pares críticos* formalizan la idea de situación de conflicto en un contexto mínimo. A partir del conjunto de todos los *pares críticos* podemos extraer los objetos y enlaces que causan dependencias o conflictos.

Un *par crítico* es un par de transformaciones que empiezan en un grafo G tales que están en conflicto y G es mínimo de acuerdo con las reglas aplicadas. El grafo G puede ser computado superponiendo la parte LHS de dos reglas de todas las formas posibles, de tal forma que cada una de las intersecciones contenga al menos un elemento que es eliminado o modificado por una de las reglas y ambas reglas sean aplicables a G. El conjunto de los *pares críticos* representa de forma precisa todos los potenciales conflictos, es decir, existe un *par crítico* si y sólo si una regla puede deshabilitar a otra. Hay tres razones por las que la aplicación de reglas puede ser conflictiva, las dos primeras relacionadas con la estructura del grafo y la última con los atributos del grafo:

1. La aplicación de una regla elimina un objeto de un grafo que estaban implicado en la aplicación de otra regla.

2. La aplicación de una regla genera objetos y aparecen estructuras prohibidas por una NAC de otra regla.

3. La aplicación de una regla cambia atributos implicados en la aplicación de otra regla

El análisis de los *pares críticos* se puede usar también para hacer un parseo de los grafos más eficiente, retrasando lo más posible las decisiones sobre las reglas en conflicto, es decir, aplicando primero las reglas no conflictivas y reduciendo el grafo tanto como sea posible.

3.8.2.2.2 Consistency checking

Las *condiciones de consistencia* [31] describen propiedades básicas de los grafos, es decir, la existencia o ausencia de ciertos elementos, independientes de una regla particular. Las condiciones de consistencia son propiedades de los grafos que tienen que conservarse tras la aplicación de las reglas. Para probar que una gramática de grafos satisface una *condición de consistencia* se transforman las condiciones de consistencia en *post application conditions* para cada regla. Una gramática de grafos

es consistente si el grafo inicial satisface las *condiciones de consistencia* y las reglas mantienen esta propiedad.

3.8.3 Aplicación de las técnicas de validación y verificación en los sistemas de transformación

De los sistemas de transformación vistos en el apartado 3.8.1 no todos implementan técnicas formales de verificación y validación. También es cierto que algunos de estos sistemas ofrecen por definición ciertas garantías en cuanto a la corrección. Por ejemplo los sistemas de transformación de modelos garantizan la correcta sintaxis ya que existen meta-modelos contra los que se deben validar los resultados obtenidos.

En este apartado veremos el nivel de implementación de las técnicas de validación y verificación de algunos de los sistemas de transformación.

3.8.3.1 VIATRA

- En VIATRA *syntactic correctness* y *completeness* puede verificarse mediante *planner algorithms* [32].
- El problema de *semantic correctness* se aborda proyectando las reglas de transformación de modelos en sistemas de transición que proveen acceso a servicios automáticos de *model checking* [32].
- VIATRA ejecuta un análisis estático de consistencia sobre cada modelo y lenguajes de modelado, que permite automáticamente detectar (y parcialmente corregir) contradicciones [32].
- VPM (Visual and Precise Meta-model) es un modelo visual pero matemáticamente preciso que resuelve muchos de los problemas que tiene MOF [32].
- Ofrece un rendimiento superior a AGG.

3.8.3.2 AGG

- Los meta-modelos garantizan que la instancia del modelo es sintácticamente correcta en cada paso de la transformación.
- Soporta varios tipos de validación como *graph parsing, consistency checking, conflict and dependency detection* (mediante *critical pair analysis*).
- Para comprobar la *confluencia* del modelo de transformación usa c*ritical pair analysis* [18].
- Si se usan capas para controlar el flujo de ejecución de las reglas (*layered graph grammar)* se puede optimizar el análisis de los *pares críticos* buscándolos sólo para las reglas de la misma capa.
- AGG implementa también un filtrado adicional que permite reducir el número de los *pares críticos*.
- Dispone de un mecanismo que permite comprobar si un grafo satisface ciertas *condiciones de consistencia* especificadas para una gramática de grafos [31].
- AGG es la única herramienta de transformación de grafos que implementa los resultados teóricos disponibles en la transformación algebraica de grafos [33].
- Permite generar *post application conditions* para una regla con restricciones en el grafo, lo que asegura que la restricción se cumple si la regla es aplicada. [34].

- AGG soporta GXL (lenguaje XML para el intercambio de grafos), lo que permite la interoperabilidad con otras herramientas.

3.8.3.3 EMF Tiger

- Como la mayoría de transformaciones realizadas con EMF Tiger pueden ser formalizadas mediante la teoría de transformación algebraica de grafos esas transformaciones se pueden verificar [35].
- El intérprete de EMF Tiger permite el uso de AGG para la verificación, lo que supone una ventaja frente a otras aproximaciones parecidas como ATL y QVT.

3.8.3.4 Henshin

- Henshin incluye una herramienta para generar y analizar el *state space* de las transformaciones de modelos *in-place* que permite verificar la corrección de las transformaciones. Dicha herramienta puede invocarse de forma programática.

- Al estar basado en conceptos de transformación de grafos es posible traducir las reglas a AGG para analizar conflictos y dependencias en la aplicación de reglas y su terminación.

3.8.4 Conclusiones sobre los sistemas de transformación

De los sistemas de reglas anteriormente descritos parece que los que mejor se adaptan a nuestro caso de uso son los basados en transformaciones de grafos y los basados en transformación de modelos.

Algunas de las herramientas disponibles en el ecosistema de UsiXML, como por ejemplo la herramienta *TransformiXML* [35], están basadas en el API que ofrece AGG para realizar la transformación entre los distintos niveles de UsiXML.

Tanto AGG como ATOM3 aunque permiten definir condiciones en Java y Python respectivamente pueden estar algo limitados al ser estrictamente declarativos, además ninguno de los dos permite definir un flujo de control para la ejecución de las reglas, obligando a establecer un orden fijo. VIATRA2 sí que dispone de un lenguaje imperativo y permite controlar el orden de ejecución de las reglas usando *máquinas de estado abstractas* [36].

Las aproximaciones más puramente MDE son QVT y ATL. Las dos parecen ofrecer bastante potencia para realizar transformaciones sobre modelos. QVT es un estándar OMG y ATL no, sin embargo el soporte de herramientas para QVT es muy limitado y el proyecto está en una fase de desarrollo inicial.

Otra alternativa sería usar un motor de reglas tradicional en lugar de herramientas de transformación de modelos. Una de las implementaciones libres más usadas en la actualidad es *JBoss Drools*. Aunque sería posible utilizar Drools como motor de transformaciones esta aproximación no parece encajar del todo bien con los niveles de abstracción y meta-modelos definidos en CAMELEON/UsiXML y que parecen ser también una muy buena alternativa en nuestro trabajo.

Siguiendo una filosofía también MDE y con unas características muy prometedoras están EMF Tiger y principalmente su sucesor Henshin. Henshin parece una muy buena opción como sistema de transformación para nuestro caso de uso ya que no solo cubre todos nuestros requisitos sino que además al utilizar el lenguaje de meta-modelado Ecore de la plataforma Eclipse Modeling Framework (EMF) nos facilitará la integración con un conjunto de herramientas que nos pueden servir de ayuda en otras fases del desarrollo de este trabajo. Como comentaremos más adelante EMF está englobado dentro de Eclipse Modeling Project (EMP). EMP nos facilitará la creación de lenguajes específicos de dominio (DSLs) usando Xtext para definir gramáticas equivalentes a los meta-modelos y permitir a los usuarios la creación de aplicaciones mediante el uso de lenguajes textuales. EMP también provee soluciones para la fase de generación de código, como Xtend. Otro aspecto a resaltar sobre Henshin es que ofrece un plugin para el IDE Eclipse y que las reglas se pueden expresar mediante un lenguaje gráfico basado en colores y anotaciones que facilita en gran medida su comprensión.

En lo que se refiere a las investigaciones llevadas a cabo sobre verificación y validación de transformaciones en general se centran en los sistemas basados en grafos, por lo que la herramienta de referencia para esas comprobaciones es AGG.

El creador de Viatra2 incluye en su tesis [32] un estudio teórico bastante completo sobre verificación y validación de modelos y transformaciones, pero en ocasiones no queda muy claro cuáles de esas técnicas implementa la herramienta.

Henshin también está basado en conceptos de transformación de grafos. Parece una buena alternativa para garantizar la calidad de las transformaciones ya que incluye una herramienta que genera y permite analizar todos los posibles estados de las transformaciones, y además, dispone de un intérprete para traducir las transformaciones a AGG lo cual permitiría realizar un análisis más exhaustivo.

Por todas las ventajas vistas anteriormente se ha decidido el uso de Henshin como sistema de transformaciones para el desarrollo de este proyecto.

3.9 MDE

Model Driven Engineering (MDE) es una metodología para el desarrollo de software que pretende elevar el nivel de abstracción en la especificación de los programas e incrementar la automatización en el proceso de desarrollo del software.

El concepto clave propuesto por MDE es el uso de *modelos* a diferentes niveles de abstracción para el desarrollo de sistemas, lo que posibilita por tanto elevar el nivel de abstracción en la especificación de los programas. Para el incremento de la automatización del proceso propone el uso de transformaciones de modelos ejecutables. Los modelos definidos a más alto nivel se transforman a niveles inferiores hasta alcanza un modelo ejecutable usando *generación de código* o *interpretación de modelos*.

Como en la mayoría de aproximaciones de la ingeniería del software la calidad es un concepto de suma importancia en MDE. Para comprobar y garantizar esta calidad MDE propone tres técnicas diferentes:

- **Model validation**: Model validation o consistency checking permite validar los criterios de calidad semántica y sintáctica, para ello se evalúa un modelo contra su meta-modelo. Además de esta evaluación también se pueden usar *constraints* para la validación de modelo. El lenguaje de validación más usado es *Object Constraint Language* (OCL)
- **Model checking**: Model checking es una técnica de verificación formal basada en la exploración de estados. Dado un sistema de transición de estados y una propiedad, los algoritmos de model checking exploran exhaustivamente el espacio de estados comprobando si el sistema satisface la propiedad.
- **Model-based testing**: Model-based testing permite probar el comportamiento del sistema en tiempo de ejecución mediante el uso de tests generados automáticamente.

La propuesta de MDE pretende incrementar la productividad maximizando la compatibilidad entre sistemas gracias a los modelos estandarizados, simplificando el proceso de diseño mediante el uso de patrones de diseño recurrentes en el dominio de aplicación y mejorando la comunicación entre los equipos de trabajo y los miembros que trabajan en el sistema gracias al uso de buenas prácticas y a la estandarización de la terminología en el dominio de la aplicación.

A menudo se confunde y se utiliza indistintamente los términos MDE y MDA (Model Driven Architecture) pero no son equivalentes. MDA puede ser visto como una visión de MDE propuesta por el OMG (Object Management Group). Otra iniciativa destacada que sigue la filosofía de MDE es Eclipse Modeling Project, que se centra en el uso de tecnologías para el desarrollo basado en modelos dentro de la plataforma Eclipse. Eclipse Modeling Project ofrece frameworks para el modelado y un conjunto de herramientas e implementaciones estándar.

La filosofía definida por MDE encaja perfectamente en las necesidades del proyecto. Siguiendo esta metodología se pretende describir a alto nivel todos los aspectos necesarios para la obtención de una aplicación (mediante el uso de modelos) y utilizar

las transformaciones entre modelos propuestas por MDE para obtener distintas representaciones de la aplicación en función de la plataforma y tipo de dispositivo objetivo. Posteriormente estos modelos se transformarán a código fuente nativo ejecutable en la plataforma de destino.

Para llevar a cabo el desarrollo propuesto se considera que Eclipse Modeling Project ofrece un conjunto de herramientas que cubren todos los pasos necesarios y una gran integración entre dichas herramientas. En los siguientes apartados se describirán brevemente algunas de éstas herramientas y lenguajes utilizados para el desarrollo de este trabajo indicando para cada uno de ellos su rol dentro de la visión global del sistema.

3.10 EMF

Eclipse Modeling Framework (EMF) es un framework de modelado y facilidad de generación de código para la construcción de herramientas y otras aplicaciones basadas en un modelo de datos estructurado.

EMF permite la especificación de modelos a partir de XML Schema, anotaciones Java, especificaciones UML o el lenguaje de modelado propio ECORE, facilitando y suministrando además las bases para la interoperabilidad con otras aplicaciones y herramientas basadas también en EMF.

El framework EMF no sólo permite modelar sistemas sino que además ofrece soporte en tiempo de ejecución y herramientas para la generación de clases Java asociadas al modelo así como facilidades para la visualización y edición de datos mediante la generación de un editor básico.

En el ámbito de este proyecto EMF sienta las bases para la definición del los meta-modelos de descripción de las aplicaciones a generar a un nivel de abstracción elevado, permitiendo modelar conceptos como las tareas que el usuario puede realizar con la aplicación o la apariencia visual de la aplicación. También se hace uso de EMF para modelar mediante el uso del lenguaje Ecore las distintas plataformas objetivo, Android y WP7.

3.11 Xtext

Xtext [38][39] es un framework para el desarrollo de lenguajes de programación y *Domain Specific Launguages* (DSLs) integrado dentro del ecosistema Eclipse. Un DSL puede ser definido como un lenguaje de programación o especificación de lenguaje ejecutable que ofrece, a través de las anotaciones y abstracciones apropiadas, un poder expresivo alto limitándose normalmente a un dominio del problema particular.

Xtext permite definir el DSL usando una gramática EBNF y el generador de Xtext se encarga de crear un parser, un meta-modelo AST implementado en EMF así como un editor textual completo del lenguaje para la plataforma Eclipse.

El uso de Xtext en este trabajo está justificado para ofrecer al desarrollador de aplicaciones facilidades para describir esas aplicaciones a un nivel de abstracción alto

y que sirvan como entrada al sistema para la generación de la aplicación final. A partir de las gramáticas desarrolladas con Xtext el desarrollador podrá usar un editor textual que le ofrece grandes facilidades como sintaxis coloreada, autocompletado, asistente de contenido, posibilidades de renombrado y refactoring, formateo de código, plantillas, etc. Todas esas facilidades repercutirán en la productividad y en la comodidad del usuario del sistema.

3.12 Xtend

Xtend [39][40] es un lenguaje estáticamente tipado integrado con la Java Virtual Machine pero que se traduce a Java en lugar de a bytecode. A pesar de ser un lenguaje de propósito general incluye una serie de características que hacen muy adecuado su uso como lenguaje para la generación de código. Xtend facilita el acceso a las instancias de los modelos Ecore para su recorrido y al sistema de archivos para la generación de los ficheros de salida, ofrece Template expressions con manejo automático del formato del código de salida, multiple dispatch (polymorphic method invocation) que evita la necesidad de crear visitors, extension methods, lambda expressions, etc.

Al integrarse a la perfección con el resto de tecnologías utilizadas en este trabajo el uso de Xtend resulta una buena aproximación para la generación del código fuente final. El módulo de generación de código fuente accederá a las instancias de los modelos resultantes de las transformaciones e irá generando el código fuente de salida así como otros ficheros necesarios (ficheros XML de las vistas, ficheros de propiedades, etc.) para la generación de los proyectos.

Capítulo 4. Descripción del Sistema

4.1 Diferencias con CAMELEON y UsiXML

Como ya hemos comentado algunos de los conceptos aplicados en este trabajo están basados en proyectos de investigación para el desarrollo de interfaces multiplataforma. Durante la fase inicial del proyecto se ha realizado un estudio en profundidad sobre el estado de la técnica en el desarrollo de interfaces multidispositivo, alguno de los trabajos relacionados han resultado de gran interés, destacando principalmente CAMELEON y UsiXML que se describen en los apartados 3.6 y 3.7. A continuación se describirán las principales diferencias con la propuesta de esos trabajos y se presentará la arquitectura propuesta para el sistema desarrollado.

La Figura 12 muestra un diagrama de los diferentes niveles de abstracción propuestos por el framework CAMELEON (figura de la izquierda) frente a la aproximación seguida por nuestra propuesta. El framework teórico presentado y las herramientas software desarrolladas para soportarlo han sido llamados LIZARD (lagartija) por la similitud con CAMELEON y por nuestra visión del sistema como un asistente (wizard) para la generación de aplicaciones multiplataforma [40].

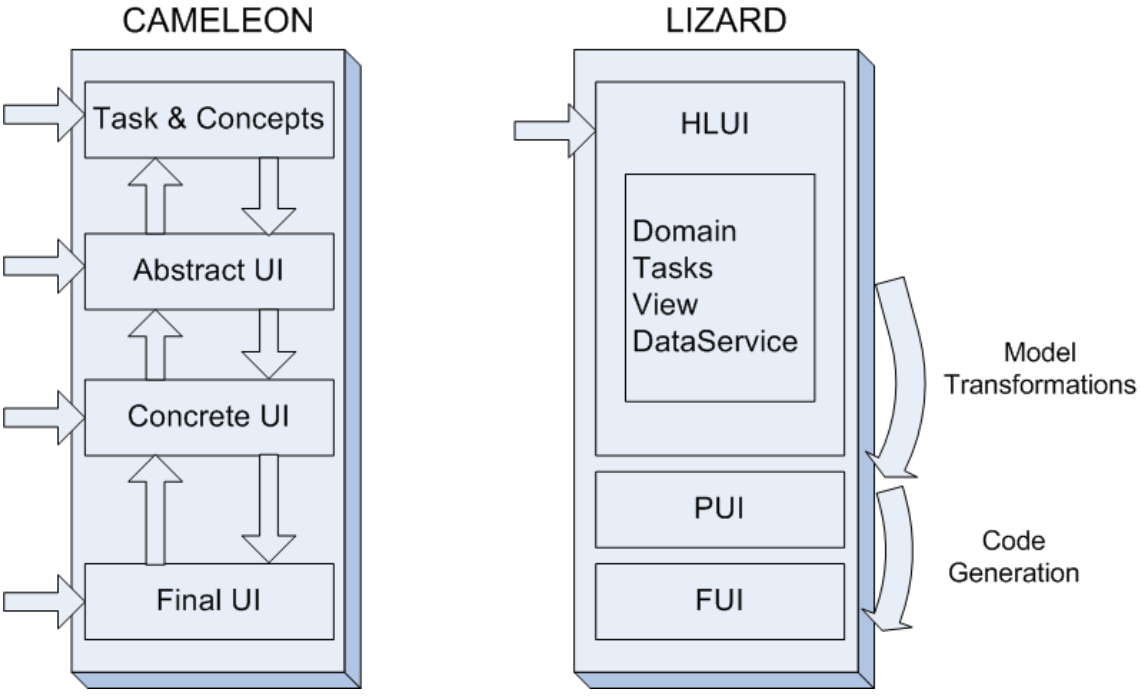


Figura 12. Niveles de abstracción definidos por el framework CAMELEON frente a los propuestos por este trabajo.

En CAMELEON y UsiXML los niveles de abstracción (TC, AUI, CUI, FUI) son independientes y se realizan traducciones de unos a otros. En teoría el usuario podría

definir la interfaz a cualquier nivel de abstracción y obtener su equivalente a cualquier otro nivel mediante el uso de transformaciones. En la práctica obtener una interfaz con un mínimo de calidad partiendo de un nivel de abstracción tan elevado como el nivel de tareas propuesto por CAMELEON es casi imposible, obteniéndose interfaces que no satisfacen al usuario ni cumplen unos mínimos criterios de calidad y usabilidad. Por ello este trabajo propone el uso de tres niveles de abstracción y transformaciones aplicadas en sentido descendente, es decir, partiendo del nivel de abstracción mayor hasta conseguir código fuente ejecutable para la plataforma final. El objetivo principal es conseguir describir la totalidad de la aplicación al mayor nivel de abstracción posible y de forma independiente a la plataforma, para posteriormente mediante el uso de transformaciones y apoyados en modelos que describen cada plataforma en detalle obtener una representación conceptualmente próxima a la plataforma objetivo. El uso de esa descripción intermedia nos proporcionará una semántica adecuada para la obtención de una interfaz de mayor calidad y que tenga en cuenta tanto los patrones de diseño como las guías de estilo recomendadas por cada fabricante para sus sistemas operativos.

Los tres niveles propuestos por este trabajo son:

- **Hight Level User Interface (HLUI)**:
- **Platform User Interface (PUI)**:
- **Final User Interface (FUI)**:

A nivel HLUI se describen todos los aspectos necesarios para la generación de la aplicación, se denomina de alto nivel porque no tiene aspectos propios de ninguna plataforma ni tipo de dispositivo sino que es común a todos, por lo que a este nivel aún no se define la división en ventanas ni otros aspectos dependientes de la plataforma o tipo de dispositivo para el que se genere [41, 42]. El nivel HLUI incluye los meta-modelos de dominio, tareas, vista, acceso a datos y dispositivo. El modelo de tareas se incluye en este nivel y no en un nivel propio como pasa en UsiXML porque no es independiente, aquí colabora con (y necesita) el resto de modelos, por ejemplo las tareas que van a tener una vista asociada se enlazan con un layout del modelo de vistas. Nuestro modelo de tareas describe las tareas a ejecutar y su descomposición, pero también refleja conceptos de un modelo de diálogo ya que existe el concepto de transición entre tareas (con paso de parámetros). El diálogo debe definirse en este nivel y posteriormente derivar un diálogo concreto para el dispositivo objetivo (cuando ya conozcamos la división de la aplicación en pantallas).

El modelo de vista también se define en el nivel HLUI, en él se definen layouts para "representar" tareas, pero aún no se habla de pantallas concretas. Se utilizan dos ideas de Android y WP7, por un lado el concepto de vista estaría basado en los *Fragment* de Android. Se definen "partes" de la interfaz pero luego si las tareas que se representan en esas partes se renderizan en una única ventana esas partes se combinan para formar una única vista. Por otro lado se utiliza el concepto de binding del patrón MVVM. Como tenemos la tarea y la vista relacionadas y cada tarea tiene un *DataContext* (propiedades del modelo de dominio que maneja la tarea) los widgets se pueden enlazar con esas propiedades para poblar los controles con datos o permitir al usuario introducir datos en la aplicación.

Una vez que se tienen todas las instancias de los modelos definidos en el nivel HLUI LIZARD genera mediante el uso de transformaciones instancias de modelos definidos al nivel PUI. En el nivel PUI habría un modelo por cada plataforma/versión. El nivel

PUI no tiene una equivalencia directa en CAMELEON/UsiXML, conceptualmente estaría situado entre los niveles concretos y final.

El último nivel referido al código fuente de la aplicación final (FUI) es totalmente equivalente al nivel FUI de CAMELEON, la única diferencia es que al generar a partir de un nivel propio de la plataforma (PUI) ese modelo está conceptualmente más próximo a lo que se quiere generar, por lo que la fase de generación de código resultará más sencilla y también será más sencillo adaptar ese código a la plataforma de destino (patrones de diseño, arquitectónicos, guías de estilo, controles, etc.) por lo que se podrá obtener una interfaz de una mayor calidad y que tenga en cuenta aspectos tan importantes como la usabilidad.

4.2 Arquitectura

El sistema creado está basado en un conjunto de módulos que dan soporte al proceso completo de generación de aplicaciones. La Figura 13 muestra un diagrama completo de la arquitectura indicando los diferentes niveles de abstracción de la interfaz (HLUI, PUI y FUI) así como cada uno de los módulos que componen el sistema indicando para cada uno de ellos las tecnologías utilizadas y sus entradas y salidas. El proceso para la obtención de aplicaciones finales es un proceso lineal que se inicia con la descripción de todos los aspectos necesarios de la aplicación a desarrollar (HLUI) y finaliza con la obtención del código fuente (FUI), aunque se podría añadir un nuevo módulo final para la generación de instalables. La arquitectura del sistema está basada en MDE, utilizándose para su implementación varias de las tecnologías incluidas en el proyecto EMP (*Eclipse Modeling Project*). Los módulos que forman el sistema global y que iremos describiendo a lo largo de este apartado son los siguientes:

- Módulo de meta-modelado
- Módulo de creación de instancias
- Módulo de transformaciones de modelos
- Módulo de generación de código

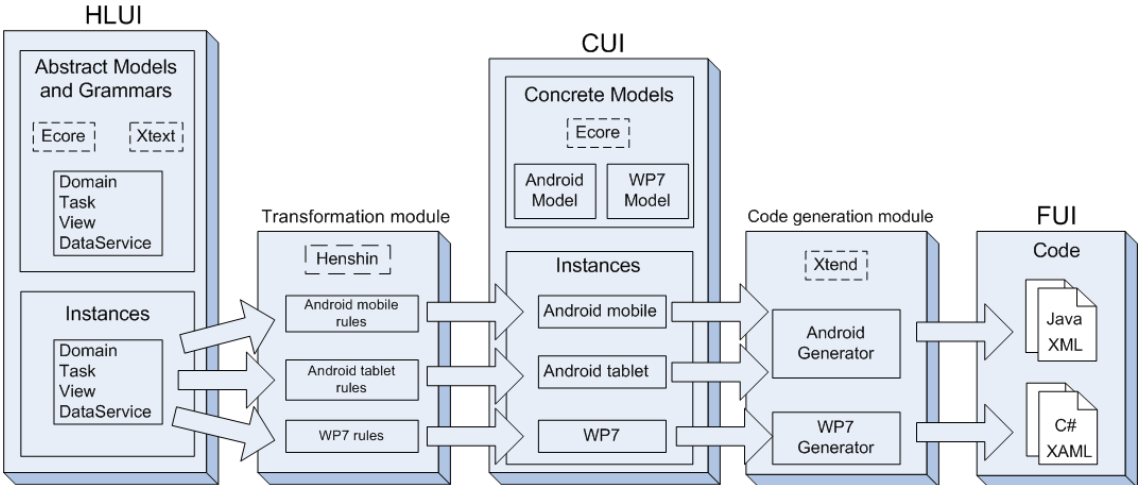


Figura 13. Diagrama de la arquitectura del sistema LIZARD.

4.2.1 Módulo de meta-modelado

En el módulo de meta-modelado se usa el meta-modelo Ecore para la creación de modelos que nos permitan describir todos los aspectos necesarios para la generación de las aplicaciones. Los modelos creados están basados en la expresividad del lenguaje UsiXML, que está a su vez basado en los distintos niveles de abstracción definidos en el proyecto CAMELEON, aunque con ciertas diferencias derivadas principalmente del uso de una única modalidad (interfaces gráficas) y el objetivo de obtener una interfaz lo más adaptada posible a la plataforma objetivo y garantizar su calidad y usabilidad.

El proyecto CAMELEON definía cuatro niveles de abstracción: *Task and domain, Abstract User Interface, Concrete User Interface* y *Final User Interface*, mientras que UsiXML define un conjunto de meta-modelos que permiten describir todos los aspectos necesarios para la generación de interfaces de usuario en cada uno de los niveles de abstracción definidos en CAMELEON.

LIZARD se basa en estos conceptos, y utiliza varios niveles de abstracción (HLUI, PUI y FUI) así como un conjunto de meta-modelos Ecore y herramientas que permiten describir todos los aspectos necesarios para la generación de aplicaciones. En el nivel HLUI se definen modelos que describen la aplicación a desarrollar de forma genérica e independiente de la plataforma y el tipo de dispositivo. El nivel PUI define modelos para cada una de las plataformas objetivo, en nuestro caso existirá un modelo Android y otro WP7, aunque el sistema es flexible y permite añadir nuevos modelos para otras plataformas o incluso versiones de éstas. El desarrollador de la aplicación es el encargado de crear instancias de los modelos definidos en el nivel HLUI, mientras que el sistema a partir de estos modelos genéricos obtendrá modelos específicos para cada plataforma usando el módulo de transformación de modelos. A partir de los modelos específicos definidos a nivel de PUI el módulo de generación de código obtendrá el código fuente final perteneciente al último nivel de abstracción (FUI).

Al igual que sucede en UsiXML los modelos definidos a un nivel de abstracción elevado son comunes a todas las plataformas y tipos de dispositivos, sin embargo y a diferencia de UsiXML se podrán definir modelos concretos para cada plataforma, versión, tipo de dispositivo, etc. (nivel PUI). Esta aproximación ofrece dos grandes ventajas:

- Permite describir cada versión de la aplicación final de forma mucho más detallada, diferenciando patrones de diseño, patrones arquitectónicos, controles gráficos específicos, etc. entre las distintas plataformas y tipos de dispositivos.
- Facilita en gran medida la fase de generación de código, ya que los modelos concretos están conceptualmente mucho más próximos al código final a generar.

Además de los modelos Ecore se han creado también gramáticas Xtext equivalentes que permitirán el uso de lenguajes textuales para la creación de instancias propias del nivel HLUI (dominio, tareas, vista y acceso a datos).

A continuación, se describe brevemente cada uno de los modelos necesarios.

4.2.1.1 Modelo de dominio

El modelo de dominio (Figura 14) permite describir las entidades que maneja la aplicación y las relaciones entre ellas. Este modelo nos permitirá por lo tanto definir conceptos como Libro, Película, etc. y sus propiedades; título, carátula, etc.

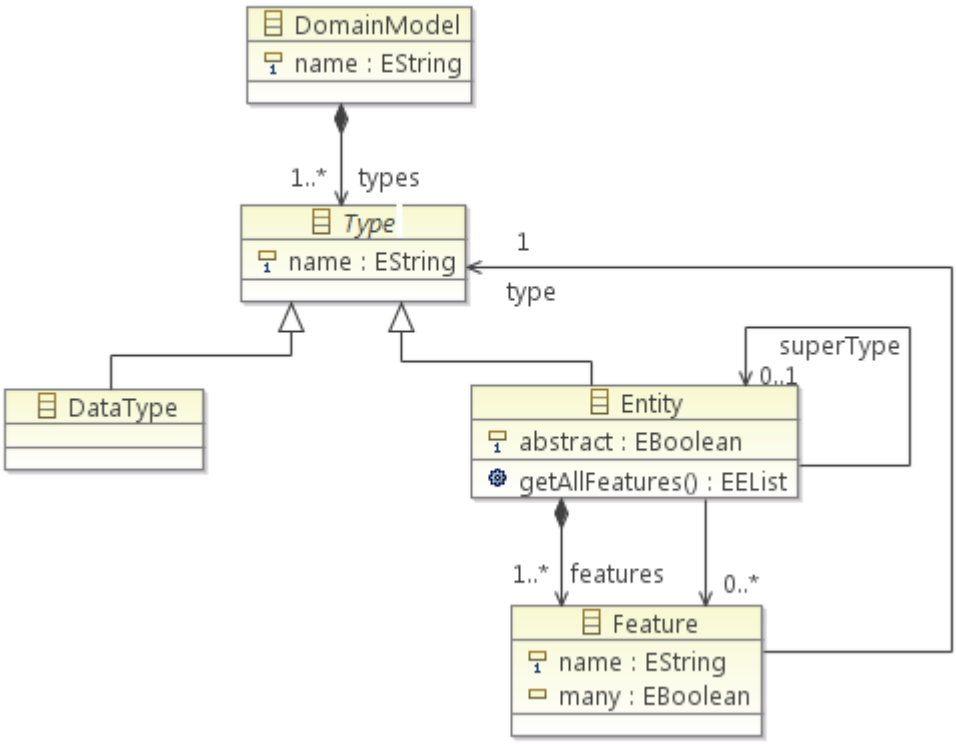


Figura 14. Diagrama del modelo de dominio.

El siguiente fragmento de código muestra la gramática Xtext creada para permitir al usuario del sistema describir la aplicación a desarrollar de forma textual. Las dos formas de definir la aplicación son equivalentes y ofrecen la misma expresividad, es una elección del usuario utilizar la versión con la que se encuentre más cómodo.

```
grammar es.uniovi.euitio.miw.tfm.xtext.domaindsl.DomainDsl with
org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "platform:/resource/UI_Metamodel_v5/model/DomainModel.ecore"

DomainModel returns DomainModel:
        'Domainmodel' name=ID
        '{'
        (types+=Type)+
        '}';

Type returns Type:
        DataType | Entity;

DataType returns DataType:
        'datatype' name=ID;

Entity returns Entity:
        (abstract?='abstract')? 'entity' name=ID ('extends' superType=[Entity])?
        '{'
```

```
        (features+=Feature)*
        ('toString' ':' (toStringFeatures+=[Feature] (','
toStringFeatures+=[Feature])*))?
        '}';

Feature returns Feature:
        type=[Type] (many?='[]')? name=ID;
```

4.2.1.2 Modelo de tareas

El modelo de tareas (Figura 15) permite describir las acciones que podrá realizar el usuario mediante el uso de la aplicación. Está basado en el modelo de tareas de UsiXML pero define además el flujo de ejecución de la aplicación (diálogo), describiendo para cada tarea sus posibles transiciones a otras tareas así como los parámetros que intercambian. Otro aspecto importante es que describe el contexto de datos manejados por la tarea, es decir, las entidades de dominio que utiliza la tarea. Las instancias de este modelo permitirán por lo tanto al desarrollador indicar las tareas que el usuario llevará a cabo mediante el uso de la aplicación. Por ejemplo visualizar la lista de películas, los datos que son necesarios para completar la tarea, en este caso una lista de películas y el flujo de tareas que permite al usuario llevar a cabo otras tareas relacionadas, como podría ser ver el detalle de una única película desencadenándose por lo tanto una transición a dicha tarea.

En el listando de código que aparece a continuación se muestra la gramática Xtext equivalente al modelo de tareas definido mediante el meta-modelo Ecore.

```
grammar es.uniovi.euitio.miw.tfm.xtext.taskdsl.TaskDsl with
org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

import "platform:/resource/UI Metamodel v5/model/TaskModel.ecore"

import "platform:/resource/UI Metamodel v5/model/DomainModel.ecore" as domainmodel

import "platform:/resource/UI Metamodel v5/model/ViewModel.ecore" as viewmodel

import "platform:/resource/UI Metamodel v5/model/DataServiceModel.ecore" as
dataservicemodel

TaskModel returns TaskModel:
        'Taskmodel'    name=ID
        '{'
                initTask=Task
    '}';


Task returns Task:
        Task_Impl | CompoundTask;

CompoundTask returns CompoundTask:
        'CompoundTask' name=ID
        '{'
                'title' ':' title=EString
                'relationshipType' ':' relationshipType=CompoundRelationshipType
                ('layouts' ':' layouts+=[viewmodel::Layout|EString] ","
layouts+=[viewmodel::Layout|EString])* )?
                ('services' ':' services+=[dataservicemodel::DataService|EString] ","
services+=[dataservicemodel::DataService|EString])*)?
                (context=DataContext)?
                (transitions+=Transition)*
                (subtasks+=Task)+
    '}';

Task_Impl returns Task:
        'Task'  name=ID
        '{'
```

```
                'title' ':' title=EString
                ('layouts' ':' layouts+=[viewmodel::Layout|EString] ("," 
layouts+=[viewmodel::Layout|EString])* )?
                ('services' ':' services+=[dataservicemodel::DataService|EString] ("," 
services+=[dataservicemodel::DataService|EString])*)?
                (context=DataContext)?
                (transitions+=Transition)*
    '}';

enum CompoundRelationshipType returns CompoundRelationshipType:
        sequential = 'sequential' | independent = 'independent' | masterDetail = 
'masterDetail';

EString returns ecore::EString:
        STRING | ID;

Transition returns Transition:
        'Transition' name=ID
        '{'
                'type' ':' type=TransitionType
                'target' ':' transitionTarget=[Task|EString]
                (parameters+=Param)*
    '}';

DataContext returns DataContext:
        'DataContext' name=ID
        '{'
                (properties+=Property)+
                (invoke+=DataServiceInvoke)*
    '}';

enum TransitionType returns TransitionType:
        navigation = 'navigation' | enabling = 'enabling';

Param returns Param:
        'Param'
        '{'
                'contextProp' ':' contextProperty=[domainmodel::Feature|EString]
                (mappings+=ParamMapping)+
    '}';

ParamMapping returns ParamMapping:
        'maps' outputElement=[domainmodel::Feature|EString] '->' 
mapsTo=[domainmodel::Feature|EString]
        ;

Property returns domainmodel::Feature:
        type=[domainmodel::Type] (many?='[]')? name=ID;


DataServiceInvoke returns DataServiceInvoke:
        'init' dataContextFeature=[domainmodel::Feature|EString] '=' 
serviceOperation=[dataservicemodel::Operation|EString]
        ;
```

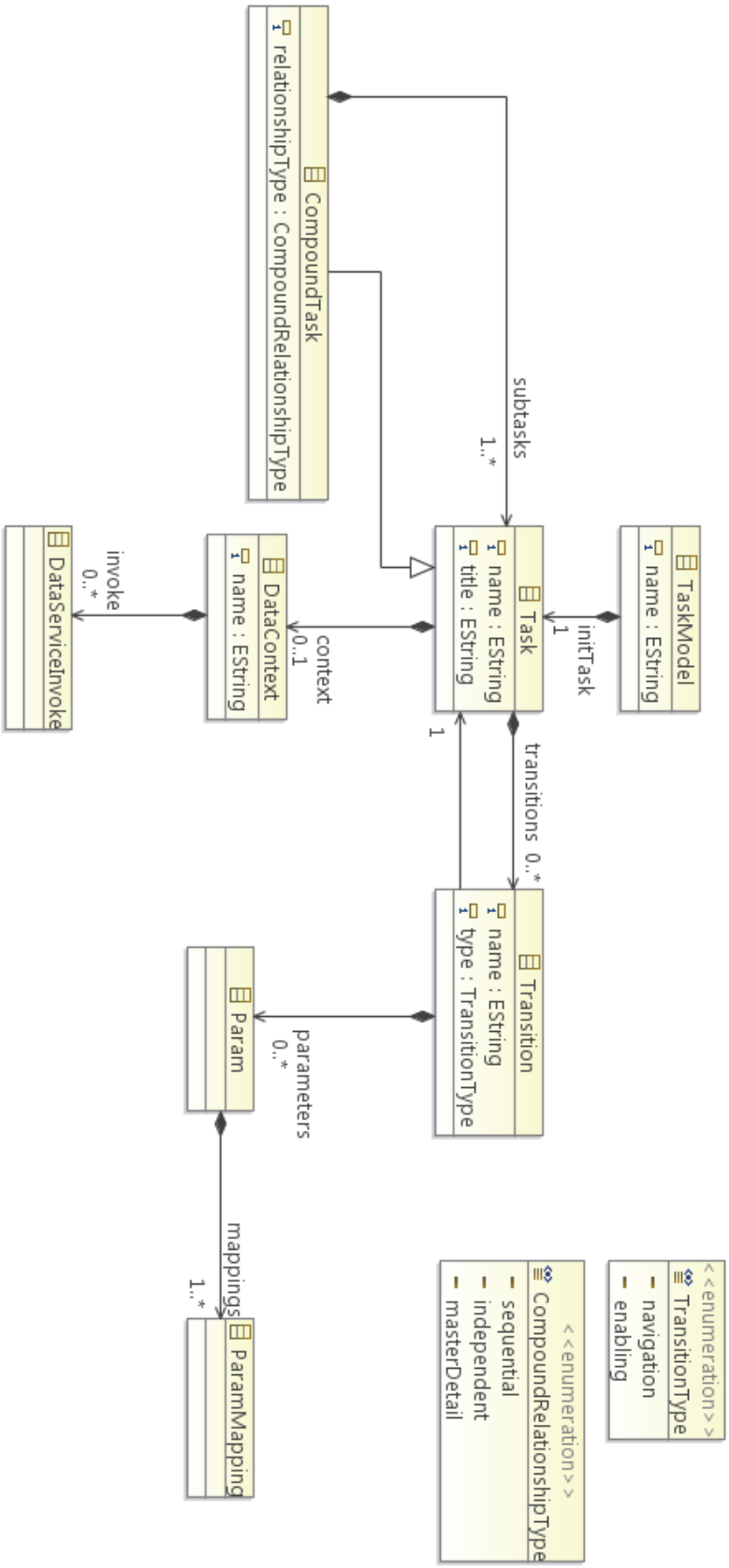


Figura 15. Diagrama del modelo de tareas.

4.2.1.3 Modelo de vista

El modelo de vista describe la apariencia visual de la aplicación, es decir, cómo se presentará cada una de las tareas del modelo anterior, por lo que principalmente definirá a un nivel abstracto los widgets y layouts a utilizar. Además este modelo permite al desarrollador definir los eventos a los que responderán los diferentes widgets de la interfaz de usuario. La Figura 16 muestra los principales conceptos que maneja este modelo .A este nivel no se habla aún de pantallas ya que su distribución dependerá de las características de la plataforma y tipo de dispositivo. A nivel conceptual sería equivalente al modelo concreto de UsiXML para una modalidad de interfaz gráfica, en el que se definen una serie de widgets comunes a la mayoría de toolkits de programación de interfaces gráficas. De esta forma el desarrollador podrá indicar que quiere presentar al usuario la tarea de mostrar el detalle de una película permitiéndole visualizar la carátula, título, nombre del director, etc. pudiendo además indicar la distribución de los datos mediante layouts.

A continuación se muestra la gramática Xtext equivalente al modelo de vista y que permite al usuario la definición de vistas de forma textual.

```
grammar es.uniovi.euitio.miw.tfm.xtext.viewdsl.ViewDsl with
org.eclipse.xtext.common.Terminals

import "platform:/resource/UI_Metamodel_v5/model/ViewModel.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "platform:/resource/UI_Metamodel_v5/model/TaskModel.ecore" as taskmodel
import "platform:/resource/UI_Metamodel_v5/model/DomainModel.ecore" as domainmodel

ViewModel returns ViewModel:
        'ViewModel' name=ID
        '{'
        (layouts+=Layout)+
        '}';

Layout returns Layout:
        LinearLayout;

Action returns Action:
        Navigate | InvokeService | WebBrowse | PlayAudio;

Component returns Component:
        LinearLayout | Button | TextView | ListView | ImageView | Label;

EString returns ecore::EString:
        STRING | ID;

Listener returns Listener:
        'Listener' name=ID
        '{'
        event=Event
        (conditions+=Condition)*
        (actions+=Action)+
        '}';

ComponentSize returns ComponentSize:
        STRING;

enum ScreenSize returns ScreenSize:
        ALL='ALL' | LARGE='LARGE' | NORMAL='NORMAL' | XLARGE='XLARGE' | SMALL='SMALL';

enum ScreenOrientation returns ScreenOrientation:
        BOTH='BOTH' | PORTRAIT='PORTRAIT' | LANDSCAPE='LANDSCAPE';

LinearLayout returns LinearLayout:
        'LinearLayout' name=ID
        '{'
        ('width' ':' width=ComponentSize)?
        ('height' ':' height=ComponentSize)?
```

```
        ('screenSizeSupport' ':'screenSizeSupport=ScreenSize)
        ('screenOrientationSupport' ':' screenOrientationSupport=ScreenOrientation)
        ('orientation' ':' orientation=Orientation)
        ('task' ':' task=[taskmodel::Task|EString])?
        (listeners+=Listener)*
        (components+=Component)*
        '}';

Event returns Event:
        'event' ':' type=EventType;

Condition returns Condition:
        {Condition}
        'Condition';

enum EventType returns EventType:
        onClick='onClick';

Navigate returns Navigate:
        'Navigate' '(' 'transitionRef' ':' transitionRef=[taskmodel::Transition|EString]
')';

InvokeService returns InvokeService:
        {InvokeService}
        'InvokeService';

WebBrowse returns WebBrowse:
        'WebBrowse' '(' 'uri' ':' uri=[domainmodel::Feature|EString] ')';

PlayAudio returns PlayAudio:
        'PlayAudio' '(' 'uri' ':' uri=[domainmodel::Feature|EString] ')';

Button returns Button:
        'Button' name=ID
        '{'
        ('width' ':' width=ComponentSize)?
        ('height' ':' height=ComponentSize)?
        ('text' ':' text=EString)?
        (bindings+=Binding)*
        (listeners+=Listener)*
        '}';

TextView returns TextView:
        'TextView' name=ID
        '{'
        ('width' ':' width=ComponentSize)?
        ('height' ':' height=ComponentSize)?
        ('text' ':' text=EString)?
        (bindings+=Binding)*
        (listeners+=Listener)*
        '}';

ListView returns ListView:
        'ListView' name=ID
        '{'
        ('width' ':' width=ComponentSize)?
        ('height' ':' height=ComponentSize)?
        'itemsSource' ':' itemsSource=[domainmodel::Feature|EString]
        'selectedItem' ':' selectedItem=[domainmodel::Feature|EString]
        (bindings+=Binding)*
        (listeners+=Listener)*
         'itemsLayout'
         '{'
               listItemLayout=Component
         '}'
        '}';

ImageView returns ImageView:
        'ImageView'    name=ID
        '{'
        ('width' ':' width=ComponentSize)?
        ('height' ':' height=ComponentSize)?
        ('imagePath' ':' imagePath=EString)?
        (bindings+=Binding)*
        (listeners+=Listener)*
        '}';

Label returns Label:
```

```
        'Label' name=ID
        '{'
        ('width' ':' width=ComponentSize)?
        ('height' ':' height=ComponentSize)?
        ('text' ':' text=EString)?
        ('textSize' ':' textSize=TextSizeValue)?
        (multiline?='multiline')?
        (bindings+=Binding)*
        (listeners+=Listener)*
        '}';

Binding returns Binding:
        'bind' widgetAttributeToBind=EString 'to'
contextAttribute=[domainmodel::Feature|EString];

enum TextSizeValue returns TextSizeValue:
        MEDIUM='MEDIUM' | XXSMALL='XXSMALL' | XSMALL='XSMALL' | SMALL='SMALL' |
LARGE='LARGE' | XLARGE='XLARGE' |
        XXLARGE='XXLARGE';

enum Orientation returns Orientation:
        VERTICAL='VERTICAL' | HORIZONTAL='HORIZONTAL';
```

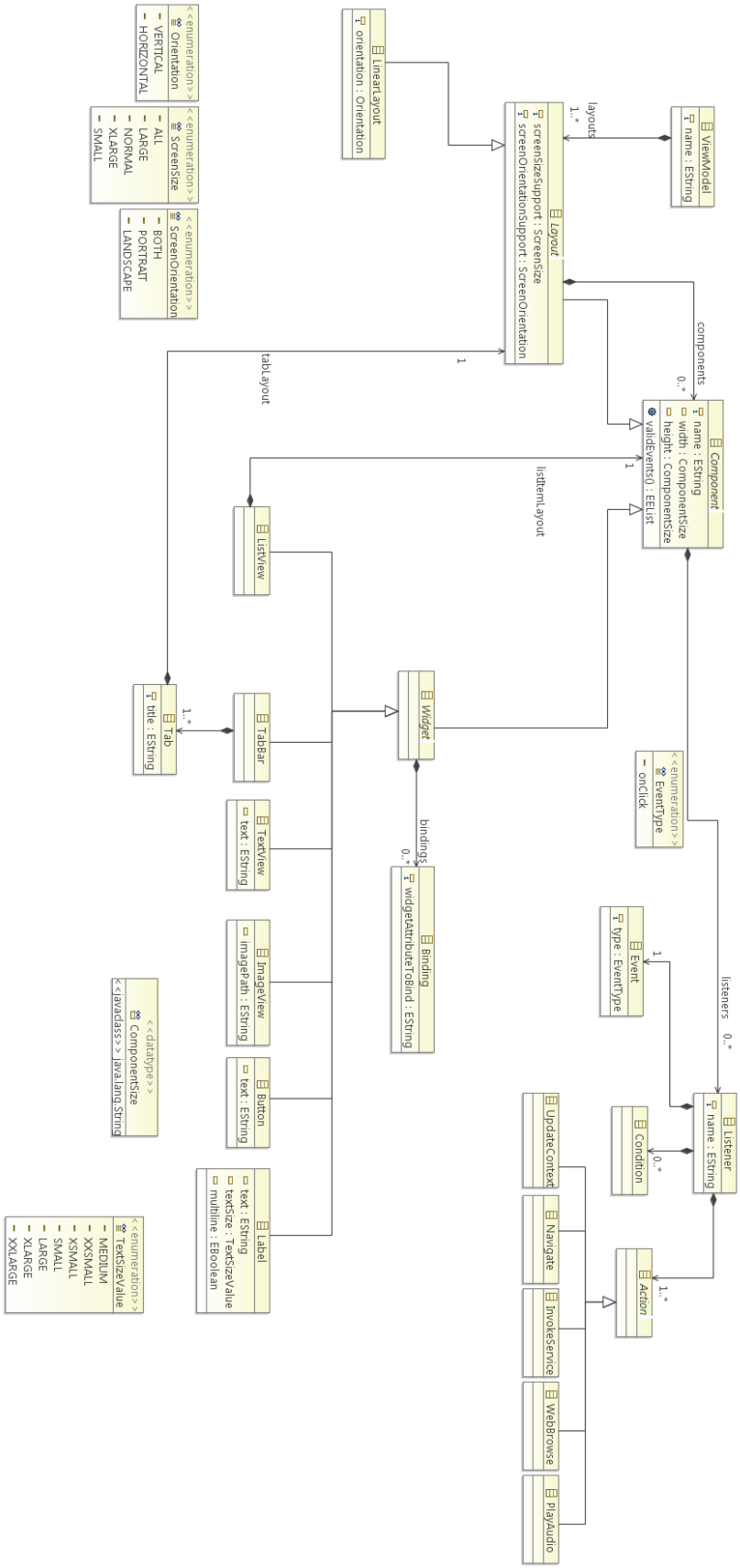


Figura 16. Diagrama del modelo de vista.

4.2.1.4 Modelo de acceso a datos

El modelo de acceso a datos (Figura 17) permite describir una interfaz para la obtención de los datos que debe manejar la interfaz gráfica. En la fase de generación de código se desacopla la obtención de estos datos con las fuentes de datos y tecnologías subyacentes mediante el uso de interfaces, factorías e inyección de dependencias. Resulta evidente que para poder mostrar la lista de películas necesitaremos recupera esa información bien sea de la memoria interna del dispositivo, mediante la invocación a un servicio remoto, etc. Desde el punto de vista de la interfaz lo importante es conocer qué datos podemos solicitar no cómo se obtienen. Este modelo nos permitirá por lo tanto la posibilidad de consultar la lista de películas, el detalle de una película a partir de su identificador, etc.

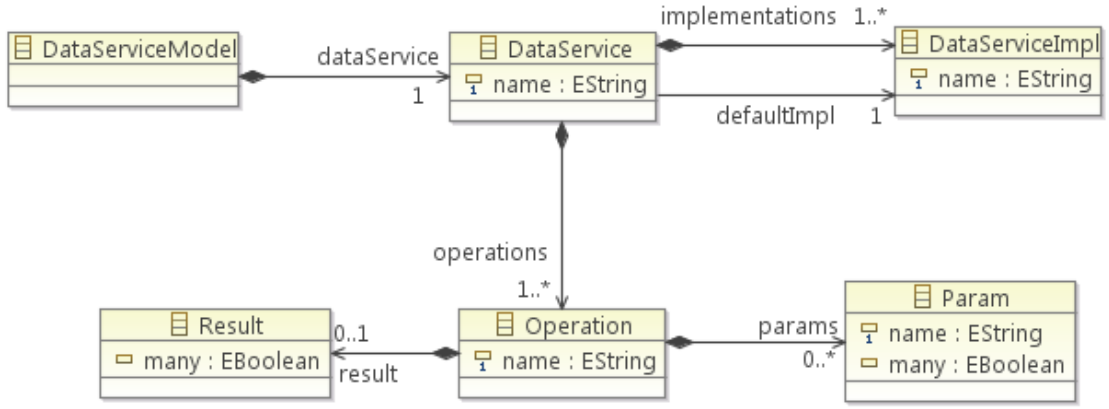


Figura 17. Diagrama del modelo de acceso a datos.

En el siguiente listado de código se muestra la gramática Xtext necesaria para la definición de un DSL de acceso a datos equivalente al modelo visto anteriormente.

```
grammar es.uniovi.euitio.miw.tfm.xtext.datadsl.DataDsl with
org.eclipse.xtext.common.Terminals

import "platform:/resource/UI_Metamodel_v5/model/DataServiceModel.ecore"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

import "platform:/resource/UI_Metamodel_v5/model/DomainModel.ecore" as domainmodel

DataServiceModel returns DataServiceModel:
        'DataServiceModel'
        '{'
                dataService=DataService
    '}';

DataService returns DataService:
        'DataService' name=ID
        '{'
                (operations+=Operation)+
                'implementations' ':' implementations+=DataServiceImpl ( ","
implementations+=DataServiceImpl)*
                'defaultImpl' ':' defaultImpl=[DataServiceImpl]
    '}';

EString returns ecore::EString:
        STRING | ID;

DataServiceImpl returns DataServiceImpl:
        name=ID
```

```
;

Operation returns Operation:
        'op' (result=Result)? name=ID '(' (params+=Param ( "," params+=Param)*)? ')'
;

Param returns Param:
        type=[domainmodel::Type] (many?='[]')? name=ID;

Result returns Result:
        type=[domainmodel::Type|EString] (many?='[]')?
;
```

4.2.1.5 Modelo de dispositivo

El modelo de dispositivo permite definir el conjunto de dispositivos objetivo para los que se pretende obtener una versión final de la aplicación. Para cada dispositivo se pueden definir un conjunto de propiedades tales como su plataforma, tipo (tablet o móvil), tamaño de pantalla, resolución, etc. estas propiedades pueden tenerse en cuenta posteriormente tanto para la aplicación de unas u otras transformaciones como para la generación del código fuente final. De esta forma el desarrollador podrá indicar por ejemplo que quiere generar la aplicación para un teléfono móvil WP7 y para una tablet Android. El sistema por lo tanto realizaría las transformaciones apropiadas en cada caso y obtendría dos modelos pertenecientes al nivel PUI a partir de los cuales se generaría el código fuente final para cada dispositivo. La Figura 18 muestra las entidades y propiedades que permiten la descripción de los dispositivos objetivo.

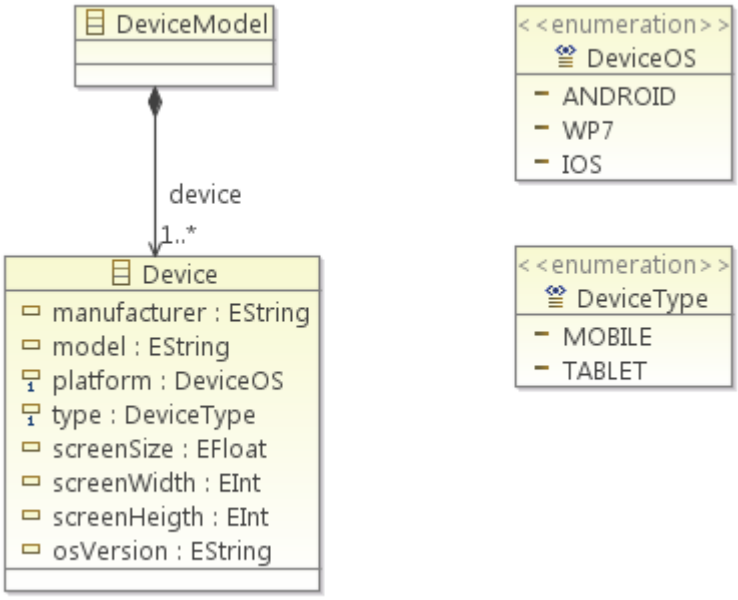


Figura 18. Diagrama del modelo de dispositivo.

4.2.1.6 Modelos específicos de la plataforma

Los modelos específicos de la plataforma permiten definir en detalle una plataforma objetivo. El sistema generará instancias de estos modelos aplicando transformaciones sobre el conjunto de modelos anteriores definidos a nivel HLUI, estas instancias servirán de entrada para el módulo de generación de código.

Mediante el uso de estos modelos concretos podremos definir el API de una plataforma o patrones arquitectónicos y conceptos propios de ella. Por ejemplo en el modelo para WP7 definimos el patrón arquitectónico MVVM. Las reglas de transformación para dicha plataforma crearán *ViewModels* asociados a cada una de las vistas. Inicialmente este trabajo incluye dos modelos, uno para la plataforma Android (Figura 19) y otro para WP7 (Figura 20), no obstante el sistema desarrollado es totalmente flexible y sería relativamente sencillo añadir nuevas plataformas o versiones de éstas.

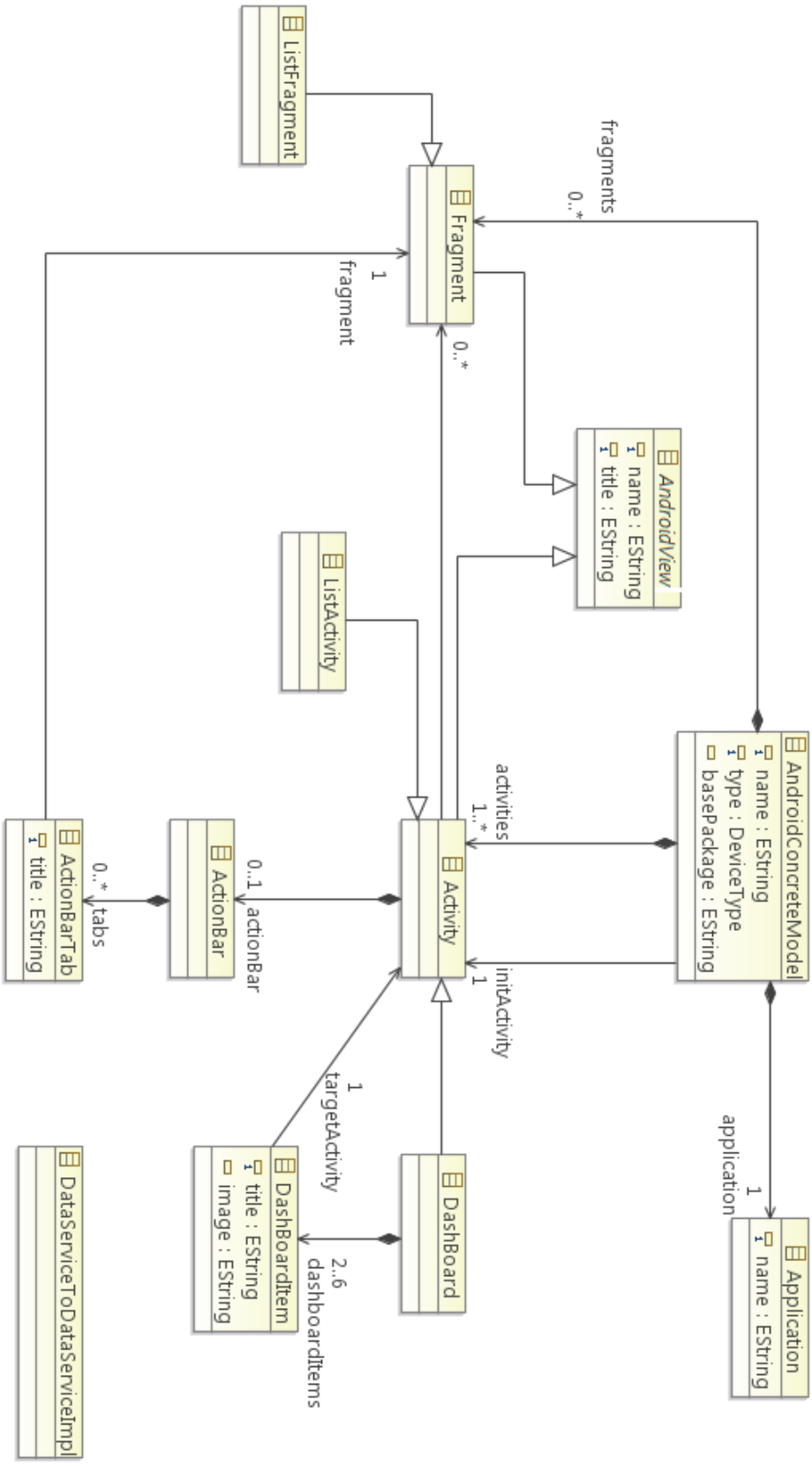


Figura 19. Diagrama del modelo concreto para la plataforma Android.

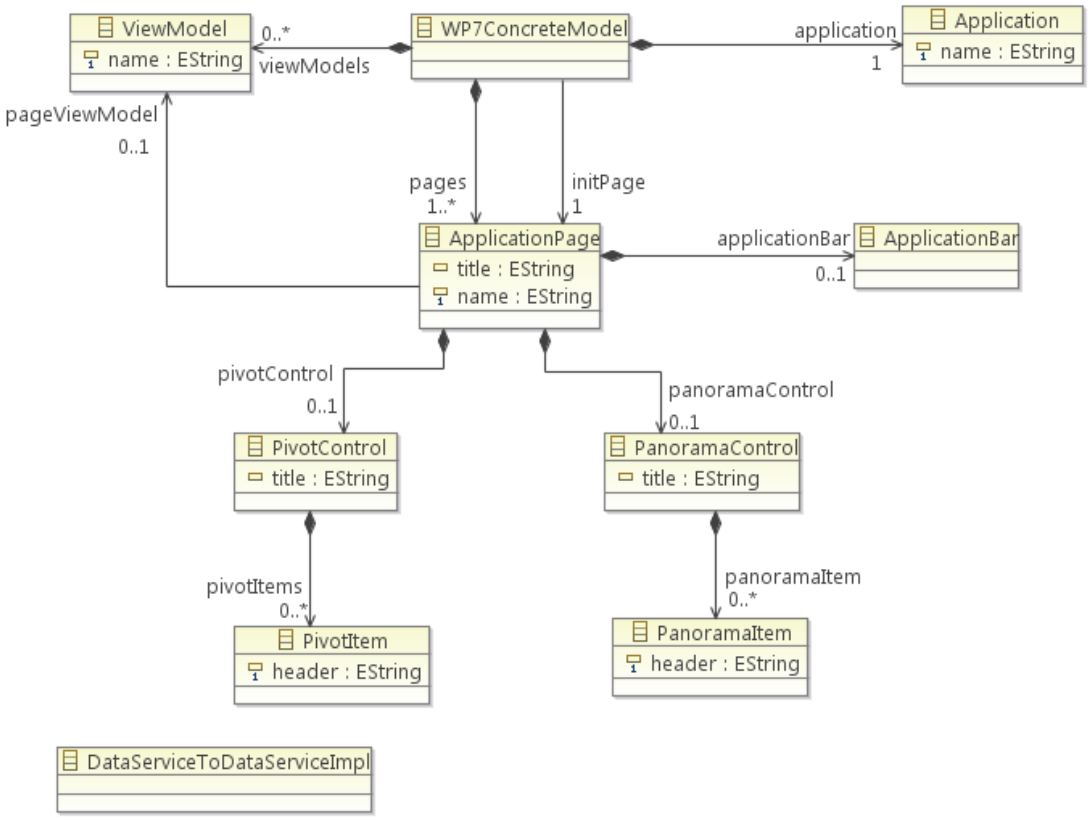


Figura 20. Diagrama del modelo concreto de la plataforma WP7.

4.2.2 Módulo de creación de instancias

EMF ofrece grandes facilidades para la creación de instancias de los modelos descritos anteriormente. El desarrollador podrá usar los editores para el IDE Eclipse para crear de forma sencilla instancias de cada modelo. La Figura 21 muestra el uso del editor gráfico facilitado por EMF para la creación de instancias de un modelo, en este caso se muestra la creación de una vista. La parte superior de la imagen muestra la jerarquía de elementos que componen la instancia mientras que en la parte inferior se proporciona al usuario una vista de propiedades que le permite asignar valores al elemento seleccionado en la parte superior.

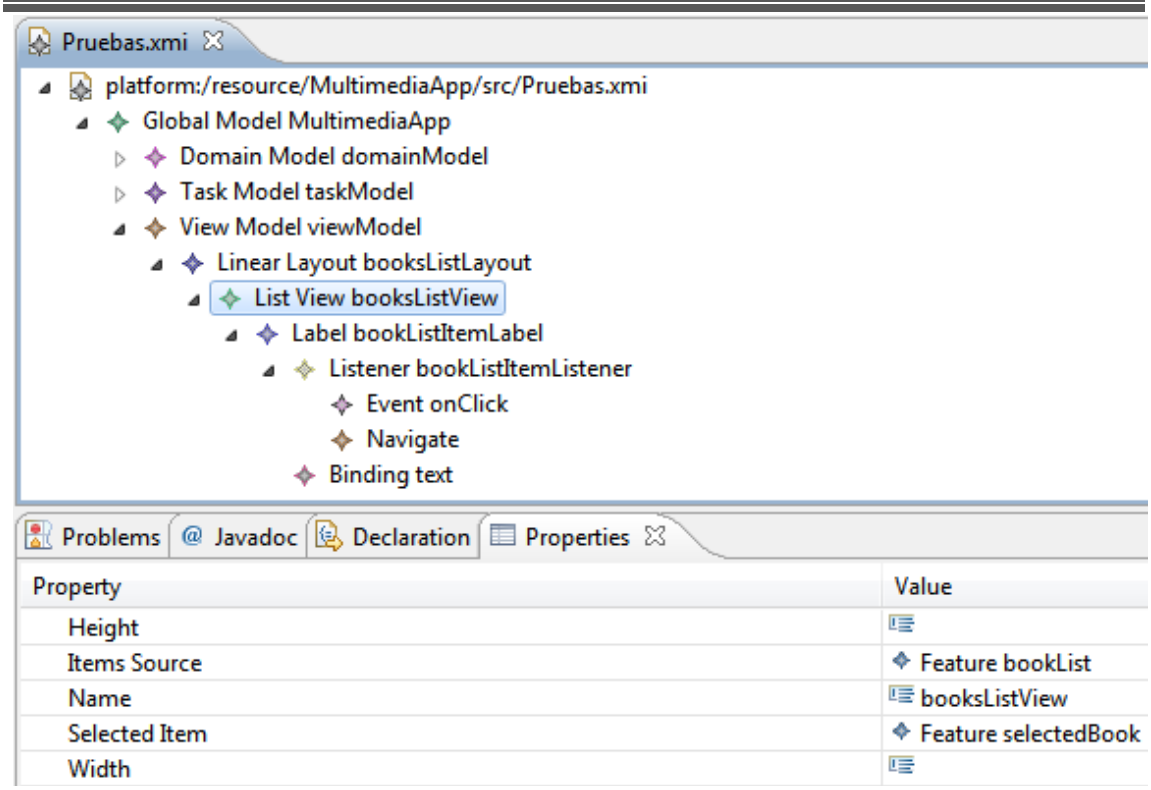


Figura 21. Definición de una vista usando el editor gráfico.

Como se ha comentado anteriormente también se ofrece la posibilidad de definir estas instancias mediante el uso de DSLs definidos mediante gramáticas Xtext. La integración con el IDE Eclipse es total y el usuario dispondrá de un gran número de características que facilitarán su tarea, repercutirán de forma satisfactoria en su rendimiento y contribuirán a que se encuentre más cómodo con el uso de la plataforma, como pueden ser syntax coloring, quick fixes, content assist, template proposals, etc.

Mediante el uso de los Ecore Model Editors o los editores textuales el desarrollador ha de crear instancias de los modelos de dominio, tareas, visual, de acceso a datos y dispositivo. La plataforma usará esas instancias como entradas del módulo de transformación de modelos y generará instancias de los modelos definidos a nivel PUI para cada tipo de dispositivo definido en el modelo de dispositivo.

multimediaApp.viewdsl

```
ViewModel viewmodel {

    LinearLayout booksListLayout{
        screenSizeSupport : ALL
        screenOrientationSupport : BOTH
        orientation : VERTICAL
        task : booksListTask

        ListView booksListView{
            itemsSource : bookList
            selectedItem : selectedBook
            itemsLayout {
                Label bookListItemLabel {
                    textSize : XXLARGE
                    bind "text" to title

                    Listener bookListItemListener {
                        event : onClick
                        Navigate ( transitionRef : toBookDetail)
                    }
                }

            }
        }
    }
```

Figura 22. Definición de una vista mediante lenguaje textual.

4.2.3 Módulo de transformación de modelos

El módulo de transformación de modelos utiliza como entrada instancias de los modelos definidos a nivel HLUI por el usuario para obtener como resultado instancias de modelos a nivel PUI de cada plataforma y tipo de dispositivo objetivo. Este módulo está implementado mediante la utilización de Henshin. Henshin es un lenguaje declarativo para transformación de modelos basado en conceptos de transformación de grafos. Además de los conceptos básicos de reglas de transformación está enriquecido con potentes *condiciones de aplicación* y un cálculo de atributos flexible basado en Java. Henshin permite la definición de estructuras de control de aplicación de reglas de forma modular mediante el uso de *transformation units*. Otro aspecto interesante para el uso de Henshin es que ofrece una gran integración con el resto de herramientas de EMP, lo que hace que encaje perfectamente con el resto de módulos del sistema.

El módulo de transformación de modelos permite obtener una versión de la aplicación totalmente adaptada a la plataforma y tipo de dispositivo objetivo, ya que traduce los modelos genéricos a modelos concretos. Estas transformaciones posibilitan la creación de aplicaciones que utilicen los patrones de diseño, patrones arquitectónicos y controles específicos para cada una de las versiones a generar.

La creación de reglas Henshin se puede realizar mediante el uso del editor jerárquico (Figura 23) o un editor gráfico (Figura 24).

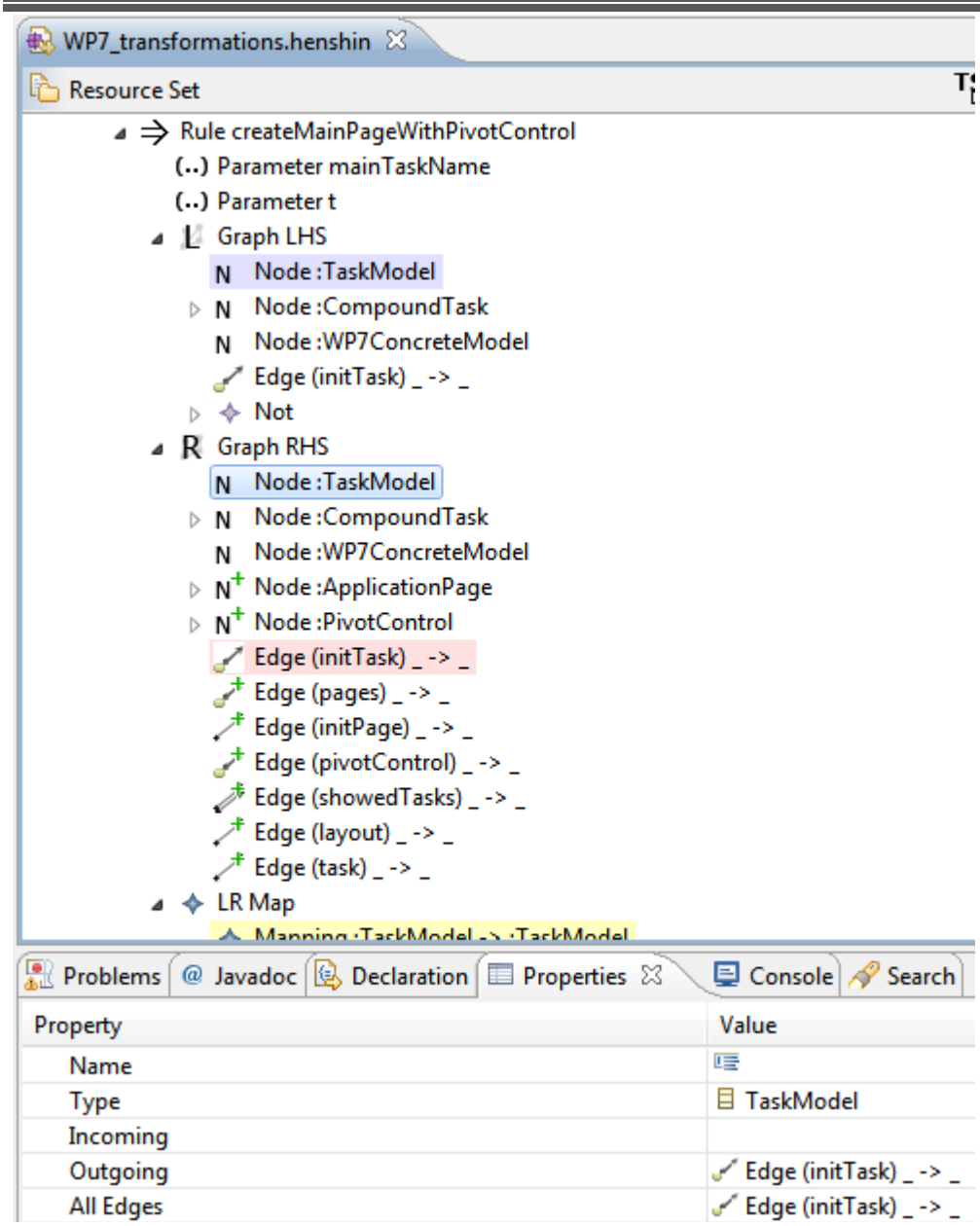


Figura 23. Editor jerárquico de reglas Henshin

En el Capítulo 6 veremos algunas de las reglas que posibilitan esta adaptación y cómo gracias a este módulo se obtienen tres versiones de la aplicación de ejemplo totalmente distintas a partir de una descripción común realizada por el usuario.

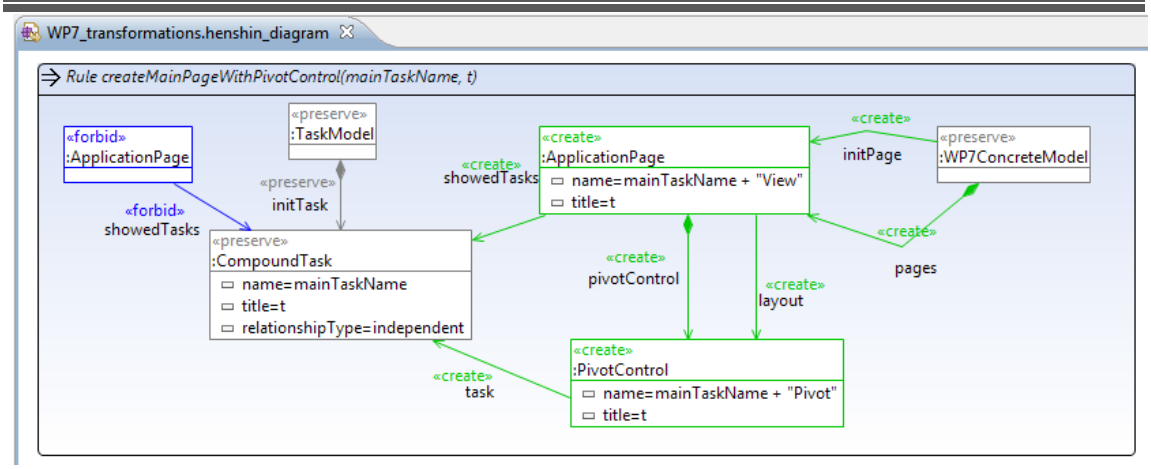


Figura 24. Editor gráfico de reglas Henshin

Henshin permite aplicar las transformaciones de forma programática y mediante el uso de un intérprete en el IDE Eclipse. La Figura 25 muestra la previsualización de una transformación utilizando el intérprete de Henshin. En la parte superior se muestra una descripción de los cambios que se aplican en la instancia del modelo inicial. En la parte inferior izquierda puede verse el resultado de la aplicación de la transformación mientras que en la parte derecha se muestra el modelo original. El uso de colores para indicar los nodos eliminados y añadidos facilita la comparación de las dos instancias.

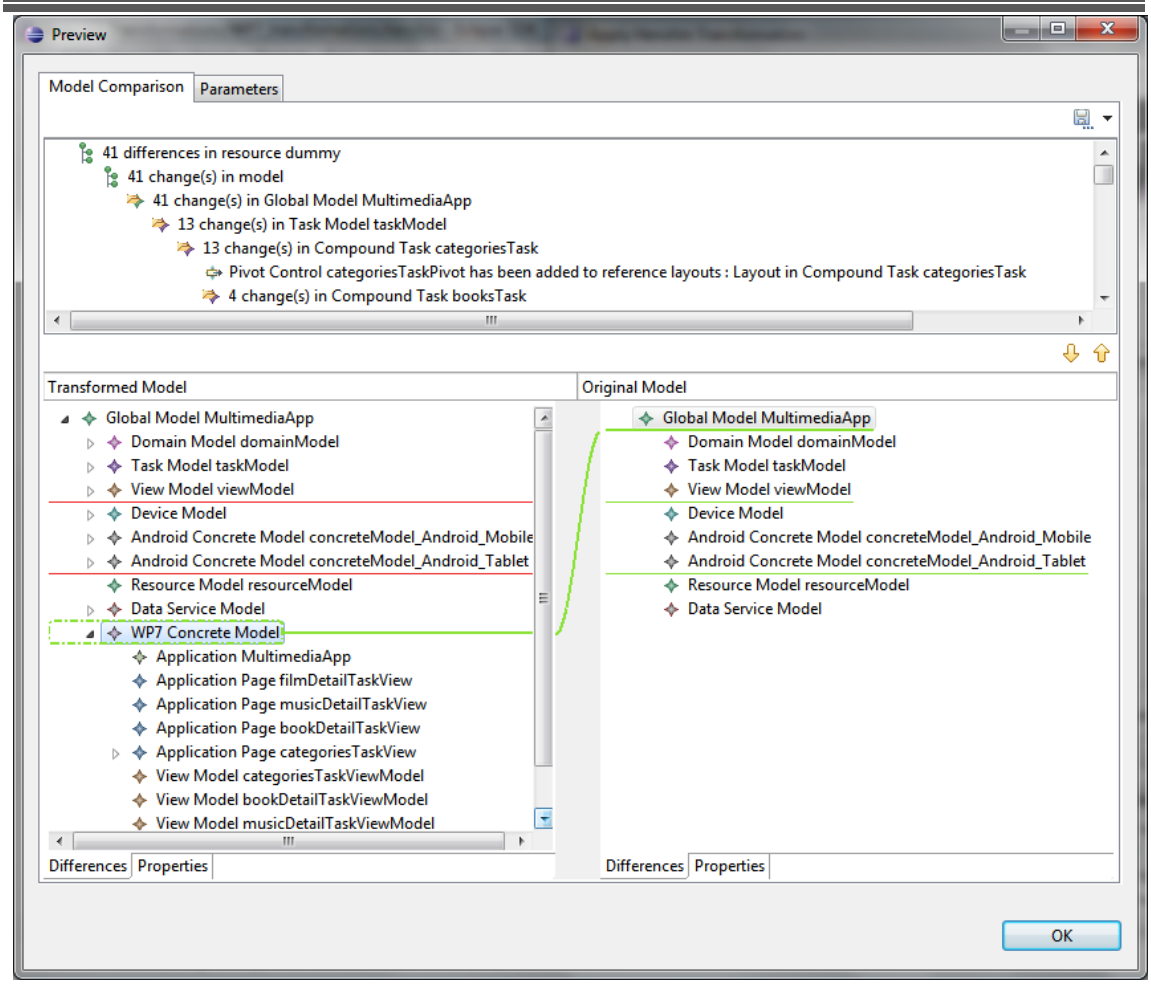


Figura 25. Previsualización de la aplicación de transformaciones en Henshin.

4.2.4 Módulo de generación de código

El último paso para la obtención de la aplicación final consiste en el módulo de generación de código fuente. Este módulo explorará los modelos concretos obtenidos tras las transformaciones anteriores para generar como salida código fuente en el lenguaje de programación de cada una de las plataformas objetivo, así como los ficheros de definición de interfaces o cualquier otro fichero necesario.

El módulo de generación de código está implementado utilizado Xtend. Xtend es un lenguaje estáticamente tipado integrado con la Java Virtual Machine pero que se traduce a Java en lugar de a bytecode. A pesar de ser un lenguaje de propósito general incluye una serie de características que hacen muy adecuado su uso como lenguaje para la generación de código. Xtend facilita el acceso a las instancias de los modelos Ecore para su recorrido y al sistema de archivos para la generación de los ficheros de salida, ofrece Template expressions con manejo automático del formato del código de salida, multiple dispatch (polymorphic method invocation) que evita la necesidad de crear visitors, extension methods, lambda expressions, etc.

El disponer de un modelo concreto para cada plataforma objetivo facilita la fase de generación de código y permite que los resultados obtenidos sean muy similares a los que obtendría de forma manual un desarrollador con experiencia en cada una de las plataformas. Por ejemplo la aplicación para WP7 resultante utiliza un contenedor

de inyección de dependencias (*Funq*) que permite desacoplar tipos concretos del código que depende de estos tipos. Utiliza también el patrón arquitectónico MVVM que facilita la total separación en el desarrollo de la interfaz de usuario del código de negocio, permitiendo la eliminación del *code-behind* mediante el uso de *bindings* entre el código de la vista (XAML) y cada *ViewModel*.

Capítulo 5. Metodología de Trabajo

Como en todo proyecto de investigación se ha comenzado por el estudio en profundidad del estado de la técnica, analizando los distintos trabajos ya existentes y las técnicas y herramientas seguidas por la comunidad investigadora para resolver el problema de la generación de interfaces de usuario multiplataforma. El análisis completo de las propuestas más interesantes se presenta en el Capítulo 3. Este análisis refleja un gran interés en la investigación para la generación de interfaces de usuario multiplataforma y manifiesta que la mayoría de las propuestas más relevantes se basan en el uso del paradigma MBUI (Model-Based UI) el cual propone el uso de modelos de alto nivel para permitir a los desarrolladores especificar y analizar las aplicaciones centrándose más en su semántica que en conceptos de bajo nivel propios de la implementación. Si bien muchos de los trabajos desarrollados permiten obtener código ejecutable para varios dispositivos generalmente la adaptación de la aplicación consiste en una traducción directa trasladando la misma experiencia de usuario entre todas las plataformas. Esto supone un problema importante y dificulta la obtención de interfaces de calidad y que ofrezcan un nivel aceptable de usabilidad. Por lo tanto este trabajo se basa en algunas de las propuestas existentes intentando ofrecer una solución para la generación de interfaces que se adapten a las guías de estilo de cada plataforma y los patrones de interfaces de usuario y arquitectónicos recomendados en cada caso.

Una vez analizado en profundidad el estado de la técnica y fijados los objetivos del trabajo se ha definido una metodología de trabajo iterativa, dividiendo el proceso en varias fases, con el objetivo de ir incrementando poco a poco la complejidad de los modelos utilizados y en cada iteración añadir soporte a nuevas funcionalidades. De esta forma las nuevas funcionalidades añadidas en los modelos se trasladan en primer lugar al módulo de transformación de modelos (añadiendo nuevas reglas) y posteriormente al módulo de generación de código. Finalmente se valida el correcto funcionamiento de las nuevas características en las aplicaciones finales generadas o se corrigen los posibles errores detectados en cualquiera de las fases.

Capítulo 6. Resultados Obtenidos

Con el fin de probar el sistema y demostrar la viabilidad de la investigación llevada a cabo se ha creado una aplicación de ejemplo utilizando la plataforma desarrollada. En los siguientes apartados describiremos el prototipo utilizado para demostrar el correcto funcionamiento del sistema y veremos paso a paso cómo ha sido el proceso de creación. Posteriormente analizaremos y discutiremos los resultados obtenidos.

6.1 Prototipo de ejemplo

El ejemplo desarrollado consiste en una pequeña aplicación que permite al usuario consultar los detalles de varios productos agrupados por categorías. El sistema deberá ser capaz de crear una aplicación nativa para cada dispositivo objetivo (Android móvil, Android tablet y WP7) a partir de una única descripción de la aplicación. Cada una de las aplicaciones obtenidas seguirá los patrones arquitectónicos, de diseño y guías de estilo de la plataforma y tipo de dispositivo adecuado en cada caso [43].

En primer lugar la aplicación permitirá al usuario seleccionar una de las tres categorías disponibles (libros, discos y películas). Una vez seleccionada la categoría se le mostrará una lista de sus elementos pudiendo elegir uno de ellos para ver sus detalles o realizar alguna acción en función de su tipo (reproducir un tema del disco o abrir el navegador web con la página de IMDb de una película).

A nivel de interfaz gráfica las aplicaciones generadas para cada dispositivo variarán principalmente en dos aspectos. Cada una de ellas ofrecerá una solución diferente al usuario para seleccionar la categoría. Una vez seleccionada la categoría sus elementos y el detalle de uno de ellos también se ofrecerán al usuario de forma distinta. En esta decisión influyen principalmente aspectos como el tamaño de pantalla del dispositivo, los componentes gráficos de los que dispone la plataforma o las guías de estilo y recomendaciones para cada una de ellas. El objetivo es obtener una aplicación lo más adaptada posible al dispositivo objetivo y que facilite así su manejo a los usuarios con experiencia en cada tipo de dispositivo.

La navegación principal de la aplicación que permitirá al usuario la selección de la categoría se realiza de tres formas diferentes en función de la plataforma y del tipo de dispositivo. El sistema de transformación de modelos (ver apartado 4.2.3) es el encargado de generar en cada caso el patrón necesario mediante la ejecución de unas u otras reglas:

1. Android tablet (Figura 26): Se utiliza un ActionBar en la parte superior de la pantalla que permite acceder a diferentes categorías mediante el uso de pestañas.
2. Android móvil (Figura 27): Se utiliza una pantalla de inicio con un botón de acceso por categoría (patrón de diseño Dashboard/ Springboard [11])
3. Windows Phone 7 (Figura 28): Se utiliza un control Pivot, control propio de WP7 que permite mostrar en la misma página un conjunto relacionado de

datos pudiendo navegar de una a otra vista mediante gestos a izquierda y derecha sobre la pantalla.

Figura 26. Navegación principal para tablets Android mediante el uso de un ActionBar con pestañas.

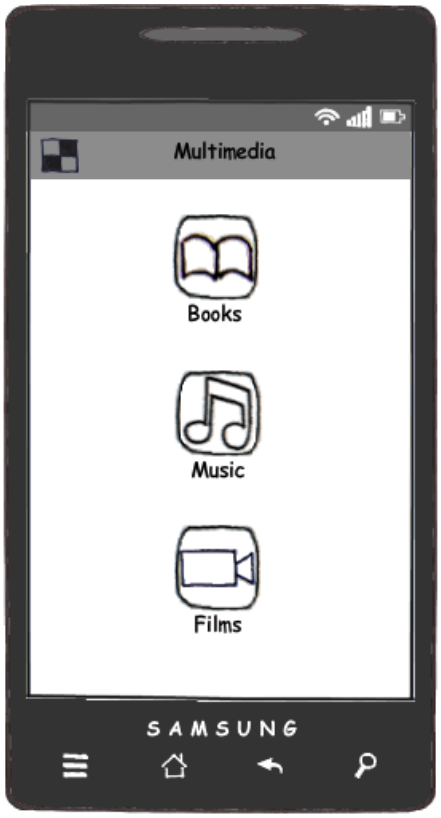


Figura 27. Navegación principal con DashBoard en móviles Android

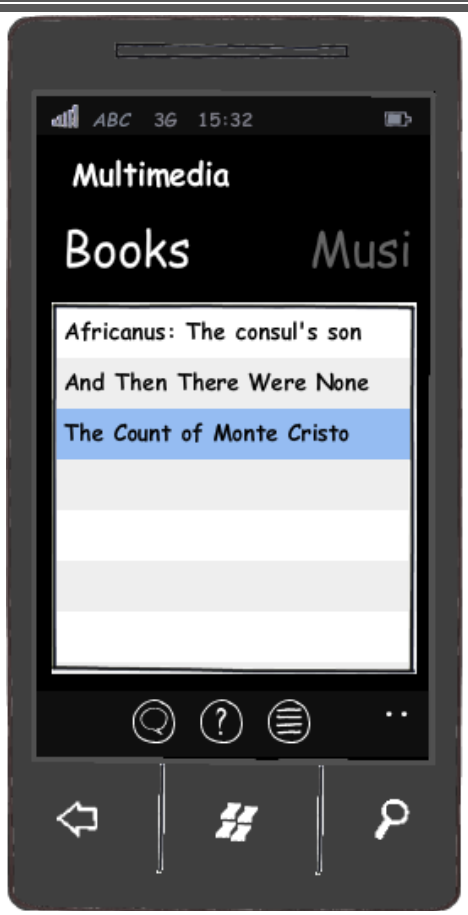


Figura 28. Navegación principal mediante control Pivot en Windows Phone 7

Para representar la información maestro/detalle (lista de elementos de cada categoría y detalle individual de un elemento) se utilizan dos alternativas diferentes:

1. Los móviles debido a su reducido tamaño de pantalla utilizan dos vistas independientes (Figura 29). La primera vista presenta la lista de elementos, al seleccionar un elemento se accede a la segunda vista que muestra su detalle.
2. La tablet al disponer de un tamaño de pantalla mucho mayor presenta el patrón maestro/detalle en una única vista formada por dos paneles. El panel izquierdo muestra la lista de elementos, la selección de uno de estos elementos provoca que su detalle se muestre en el panel de la derecha (Figura 30).

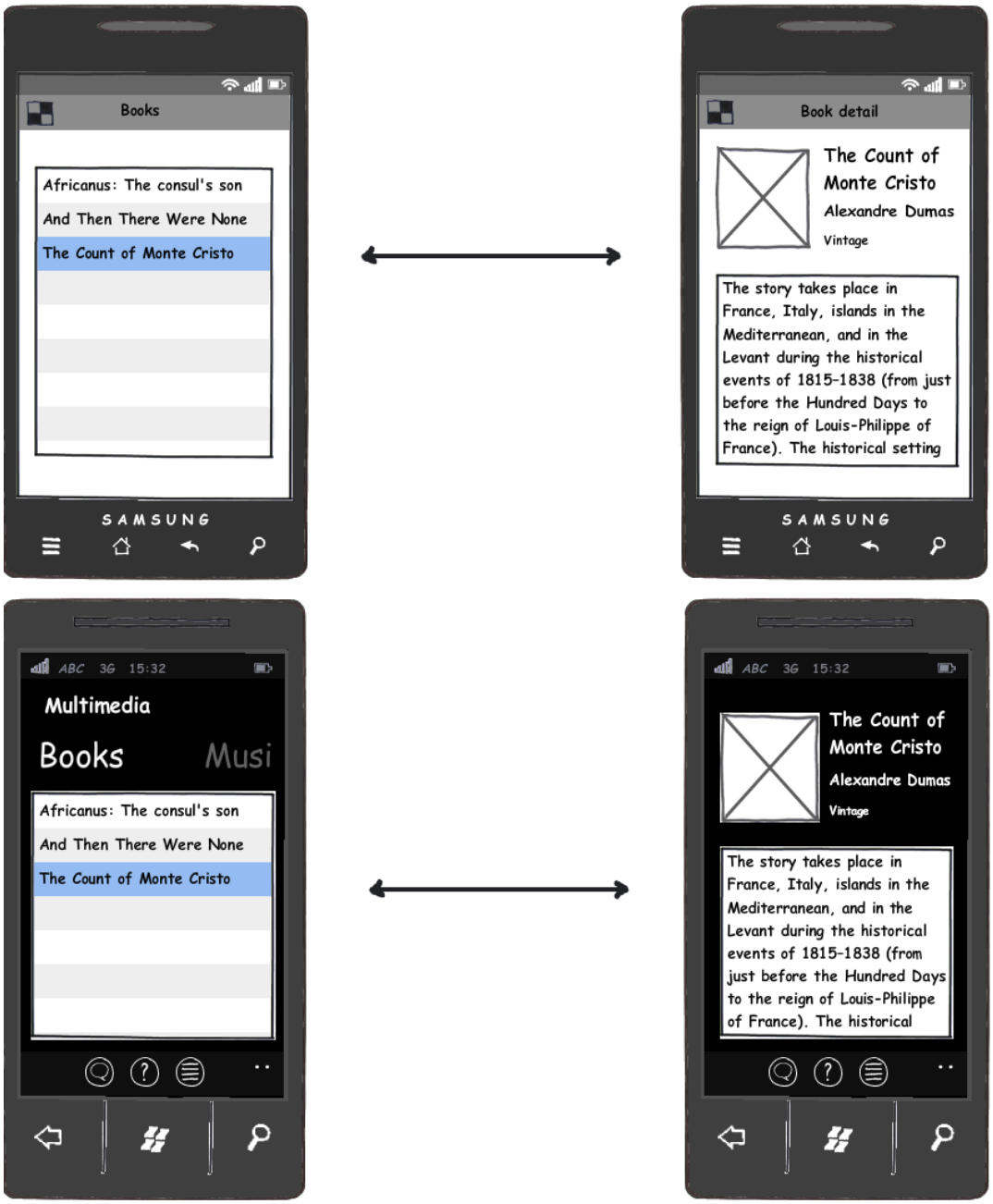


Figura 29. Maestro-detalle en pantallas de tamaño reducido. Una vista para la lista de elementos con una transición al detalle

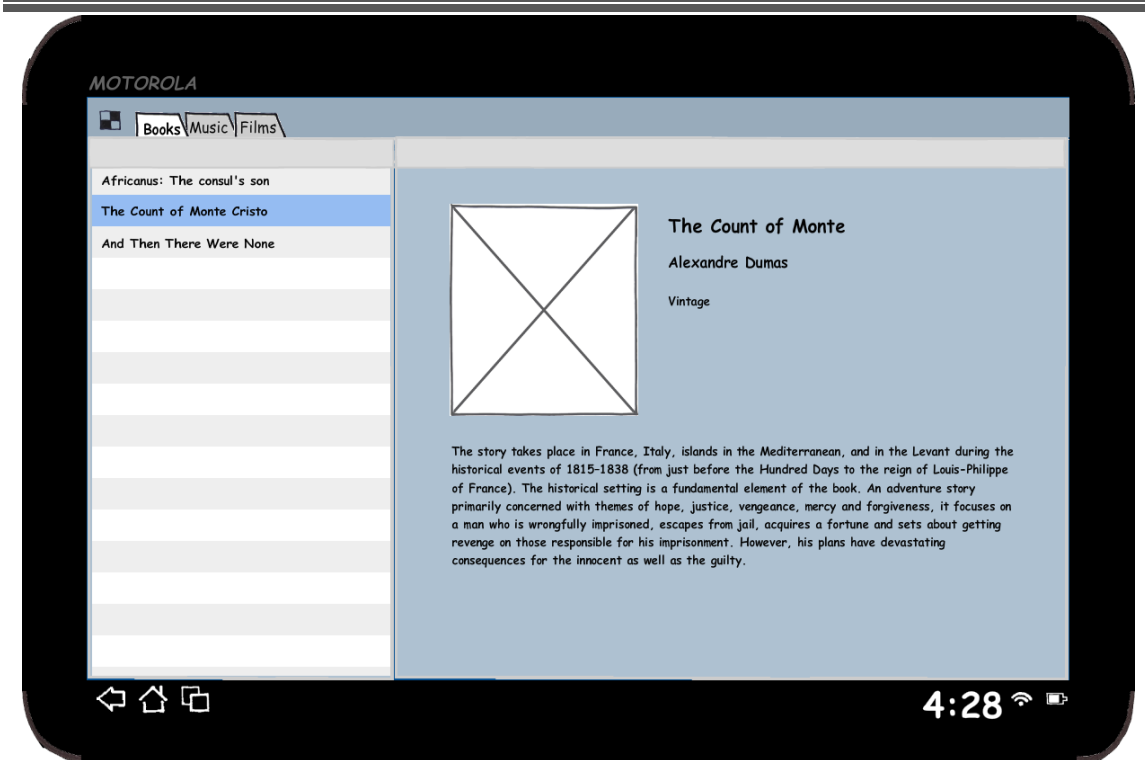


Figura 30. Maestro detalle en una única pantalla. Lista de elementos en el panel de la izquierda y detalle del elemento seleccionado en el panel de la derecha

6.2 Desarrollo del prototipo

Este apartado intenta describir el proceso completo de generación de una aplicación móvil desde la descripción de alto nivel hasta el código fuente final para cada una de las plataformas. A lo largo de la explicación se irá haciendo referencia a aspectos de la arquitectura descrita anteriormente (ver Capítulo 4).

Para el desarrollo de una aplicación basada en este framework el desarrollador debe seguir una serie de pasos. Cada uno de ellos relacionado con un módulo de la arquitectura.

Paso 1: Descripción de alto nivel de la aplicación

El desarrollador debe describir todos los aspectos necesarios para la generación de la aplicación. Esta descripción consistirá en la creación de instancias de los meta-modelos descritos anteriormente (apartados 4.2.1.1 - 4.2.1.5). El desarrollador podrá crear estas instancias en el IDE Eclipse usando el editor de modelos Ecore o de forma textual usando el DSL adecuado en cada caso conforme a la gramática definida para cada uno de los lenguajes. La Figura 31 muestra la definición del modelo de dominio de la aplicación de ejemplo creada con el editor de modelos Ecore.

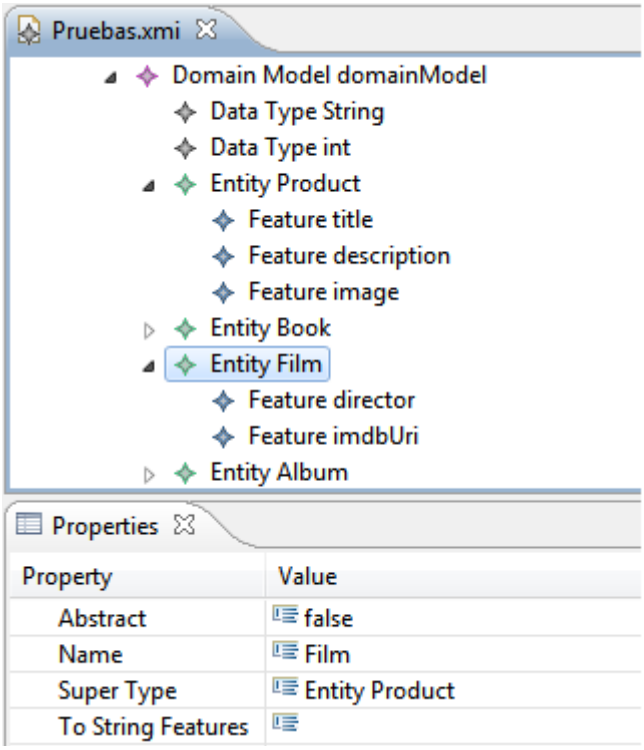


Figura 31. Definición del modelo de dominio en el editor de modelos Ecore.

En la Figura 32 se puede observar la misma definición pero esta vez realizada de forma textual mediante el uso del DSL de dominio definido a tal efecto.

multimediaApp.domaindsl

```
Domainmodel multimediadomain{
    datatype String
    datatype int

    abstract entity Product{
        String title
        String description
        String image
        toString : title
    }

    entity Book extends Product{
        String author
        String editorial
    }

    entity Film extends Product{
        String director
        String imdbUri
    }

    entity Album extends Product{
        int year
        String band
        String songUri
    }
}
```

Figura 32. Definición del modelo de dominio usando el DSL.

Bien haciendo uso del editor de modelos Ecore o de los diferentes DSLs el desarrollador deberá describir los siguientes aspectos necesarios para la generación de la aplicación, que iremos viendo de forma detallada en los siguientes apartados.

- Entidades del dominio que manejará la aplicación.
- Las tareas que podrá realizar el usuario con la aplicación.
- Las vistas (layouts) y los controles que formarán la interfaz gráfica.
- Las fuentes de datos que permitirán poblar los controles de la interfaz y mostrar datos al usuario.
- Los dispositivos para los que se desea generar una versión de la aplicación.

Paso 1.1: Descripción de las entidades del dominio.

Como hemos comentado anteriormente el modelo de dominio ha de permitir al desarrollador describir de forma sencilla todos los aspectos relacionados con las entidades de dominio que manejará la aplicación. Esto es, las entidades, relaciones entre ellas, sus atributos, tipos etc.

Tanto en la Figura 31 como en la Figura 32 se pueden ver las entidades definidas para el ejemplo desarrollado (Product, Book, Film and Album), así como sus atributos (title, image, director, etc.). Podemos ver también como se describen otros aspectos a los que da soporte el meta-modelo de dominio. Para cada uno de los atributos de una entidad se puede definir su tipo y cardinalidad. El meta-modelo de dominio

permite también definir una jerarquía de herencia entre entidades (Book, Film and Album heredan de Product, que se declara como abstracta).

Paso 1.2: Descripción de las tareas que se pueden realizar con la aplicación.

El modelo de tareas permitirá al desarrollador definir varios aspectos que se detallan a continuación y pueden verse en forma de ejemplo en la Figura 33.

- El conjunto de tareas que pueden ser llevadas a cabo por la aplicación.
- La relación entre tareas. Se ofrece la posibilidad de declarar tareas compuestas que estarán formadas por varias subtareas. Por ejemplo, la tarea para mostrar los álbumes musicales estará formada por una subtarea para mostrar la lista de discos y otra subtarea para mostrar el detalle de un disco.
- El contexto de datos que maneja cada tarea, para lo que se usarán relaciones con el modelo de dominio. Por ejemplo la tarea que muestra la lista de discos deberá manejar una lista de entidades de dominio de tipo álbum (cardinalidad *many*) y una propiedad de tipo álbum para almacenar el disco seleccionado por el usuario.
- Las transiciones entre tareas. Por ejemplo habrá una transición entre la tarea que muestra la lista de discos y la tarea que muestra su detalle.
- Los parámetros que se pasan de una a otra tarea en las transiciones y el mapeo de estos parámetros entre los contextos de ambas tareas. Por ejemplo, la tarea que muestra la lista de discos deberá pasar el álbum seleccionado por el usuario a la tarea que muestra su detalle. El modelo de tareas permite además relacionar los datos que se pasan de una tarea (propiedades del contexto de la tarea de origen) con propiedades del contexto de la tarea de destino. Por ejemplo en el caso de los discos vemos como la propiedad 'musicTitle' definida en el contexto de la tarea de destino (musicDetailTask) se mapea con el atributo 'title' del disco seleccionado (propiedad 'selectedMusic' de la tarea de origen).

multimediaApp.taskdsl

```
Task musicListTask{
    title : "Albums"
    layouts : musicListLayout
    services : multimediaDataService

    DataContext musicListContext{
        Album [] musicList
        Album selectedMusic
        init musicList = getAlbums
    }

    Transition toMusicDetail{
        type : navigation
        target : musicDetailTask
        Param {
            contextProp : selectedMusic
            maps ^title -> musicTitle
            maps band -> musicBand
            maps description -> musicDescription
            maps image -> musicImage
            maps year -> musicYear
            maps songUri -> musicSongUri
        }
    }
}

Task musicDetailTask{
    title : "Album detail"
    layouts : musicDetailLayout
    DataContext musicDetailContext{
        String musicTitle
        String musicBand
        String musicDescription
        String musicImage
        int musicYear
        String musicSongUri
    }
}
```

Figura 33. Definición de la parte del modelo de tareas relacionado con la visualización de la lista de Discos y el detalle de un disco

Paso 1.3: Descripción de las vistas de la aplicación.

El modelo de vista describirá el layout de cada una de las pantallas que formarán la aplicación así como los componentes gráficos utilizados en ellas. Además permitirá también definir los eventos que se producen al interactuar el usuario con los distintos componentes gráficos. La Figura 34 muestra la definición de la vista que permite visualizar la lista de discos. El layout definido está formado por un único componente gráfico de tipo *ListView*.

multimediaApp.viewdsl

```
LinearLayout musicListLayout{
    screenSizeSupport : ALL
    screenOrientationSupport : BOTH
    orientation : VERTICAL
    task : musicListTask

    ListView musicListView{
        itemsSource : musicList
        selectedItem : selectedMusic
        itemsLayout {
            Label musicListItemLabel {
                textSize : XXLARGE
                bind "text" to title

                Listener musicListItemListener {
                    event : onClick
                    Navigate ( transitionRef : toMusicDetail)
                }
            }

        }
    }
}
```

Figura 34. Definición de la vista para mostrar la lista de discos.

En el ejemplo de la Figura 34 habrá que tener en cuenta varios aspectos interesantes. En primer lugar la propiedad *task* del layout indica que la vista está relacionada con la tarea *musicListTask* descrita anteriormente (ver Figura 33). Esta relación nos permitirá enlazar los controles visuales (widgets) con propiedades del contexto de datos de la tarea. De esta forma podemos indicar por ejemplo que el control ListView tiene como fuente de datos la propiedad musicList (lista de álbumes definida en el contexto de la tarea) y que el resultado de la selección de un elemento (selectedItem) se almacene en la propiedad *selectedMusic* definida igualmente en el contexto de la tarea. Dentro del ListView definimos el layout que tendrá cada uno de sus ítems, en este caso una simple *Label*. Podemos ver que la propiedad '*text*' de la Label se enlaza con la propiedad *title* de un disco, con lo que cada uno de los elementos de la lista mostrará el título de un disco. Por último otro aspecto importante a tener en cuenta en el modelo de vistas es la definición de *Listeners* que permitirán asociar ciertas operaciones a las acciones llevadas a cabo por el usuario sobre los controles de la interfaz gráfica. En el ejemplo de la Figura 34 se declara un *Listener* sobre el evento *onClick* que provocará una transición al detalle del disco.

Paso 1.4: Descripción del acceso a datos.

Este modelo permite definir los servicios de acceso a datos que serán necesarios para poblar las interfaces gráficas. El modelo permite definir cada uno de los métodos necesarios así como sus tipos de retorno y parámetros necesarios. En el ejemplo que estamos describiendo será necesario definir métodos que nos permitan obtener listas de libros, discos y películas con los que poblaremos los *ListView* correspondientes (Figura 35).

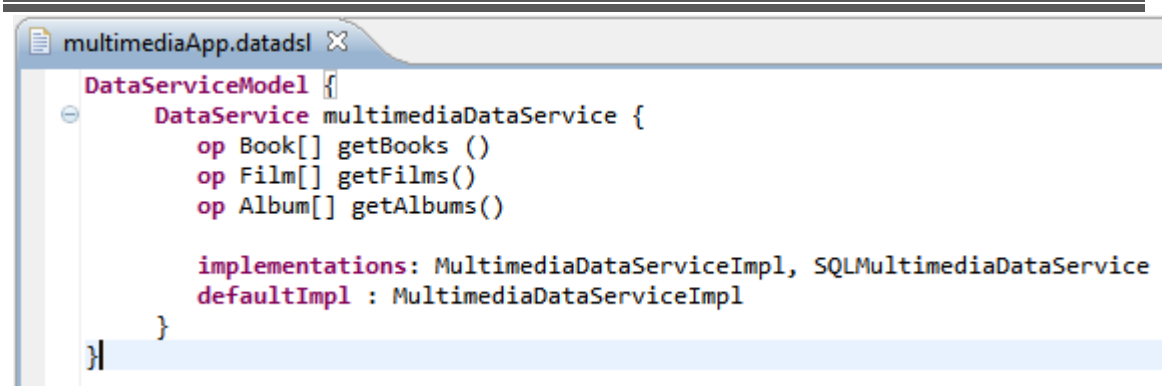


Figura 35. Definición del modelo de acceso a datos

El modelo de datos permite indicar varias implementaciones para un *DataService*, de esta forma para cada plataforma o aplicación final se podrá seleccionar la opción más apropiada a nivel de modelo. No obstante el código generado desacopla totalmente los datos con los mecanismos subyacentes de acceso. Para este fin en Android se hace uso de los patrones *Façade* y *Factory Method* [7], mientras que en WP7 se utiliza el framework de inyección de dependencias *Funq* [41].

Paso 1.5: Descripción de los dispositivos objetivo

La plataforma soporta actualmente la generación de código para las plataformas Android y Windows Phone 7. Otro aspecto importante a tener en cuenta para la generación es el tipo de dispositivo, ofreciendo soporte a teléfonos móviles de las dos plataformas y tablets Android. El modelo de dispositivo nos permitirá definir los dispositivos objetivo de la aplicación así como un conjunto de propiedades (tamaño de pantalla, resolución, etc) que pueden ser utilizadas tanto en las condiciones de aplicación de las reglas (módulo de transformación de modelos) como en la generación de código.

Paso 2: Transformación de los modelos HLUI a modelos PUI

En el paso anterior vimos todos los modelos que permiten describir cada uno de los aspectos necesarios para la generación de la aplicación. Todos esos modelos eran comunes a las distintas plataformas (Android y WP7) y tipos de dispositivos (móviles y tablets). En este paso se toman todos los modelos vistos anteriormente como entrada del sistema de transformaciones, obteniéndose como salida un nuevo modelo para cada tipo de dispositivo definido en el modelo de dispositivos (Paso 1.5: Descripción de los dispositivos objetivo). Este es el paso clave que permite adaptar la aplicación a los patrones arquitectónicos, patrones de diseño, guías de estilo y controles propios de cada plataforma y tipo de dispositivo. Para realizar estas transformaciones se usa Henshin, un lenguaje de transformación de modelos basado en conceptos de transformación de grafos que ofrece una perfecta integración con EFM.

El sistema de transformaciones está formado básicamente por dos elementos: grafos (modelos EMF) y *transformation units*. Las *transformation units* son estructuras de control que permiten controlar el orden de aplicación de las reglas. La *transformation unit* más básica es una regla en sí mismo que se corresponde con una única aplicación de esa regla. Una regla consiste en un grafo de precondición (*left-hand side, LHS*), un grafo de postcondición (*right-hand side, RHS*) y opcionalmente restricciones sobre la aplicación de la regla (*application conditions*). Una regla de transformación se

aplica sobre una instancia de un modelo EMF buscando el grafo de la precondición en el modelo y sustituyéndolo por el grafo de la postcondición si las *application conditions* lo permiten.

Henshin dispone de una notación gráfica que facilita en gran medida la creación de reglas así como su comprensión en comparación con otros sistemas de transformación como ATL y QVT. La notación gráfica de Henshin está basada en colores y anotaciones. Los nodos del grafo LHS se representan en color gris y mediante la anotación *«preserve»*. Los nuevos nodos que se crearán en la RHS tras la aplicación de la regla se representan en verde con la anotación *«create»*, mientras que en rojo y con la anotación *«delete»* se representan aquellos nodos que aparecían en la LHS y que serán eliminados tras la aplicación de la regla. Finalmente las *application condition* se representarán en color marrón y con la anotación *«require»* o en color azul y con la anotación *«forbid»* en el caso de tratarse de *NACs* (Negative Application Condition, una NAC es una condición adicional de aplicación que contiene un grafo con el cual el modelo inicial no puede casar para que se aplique la regla).

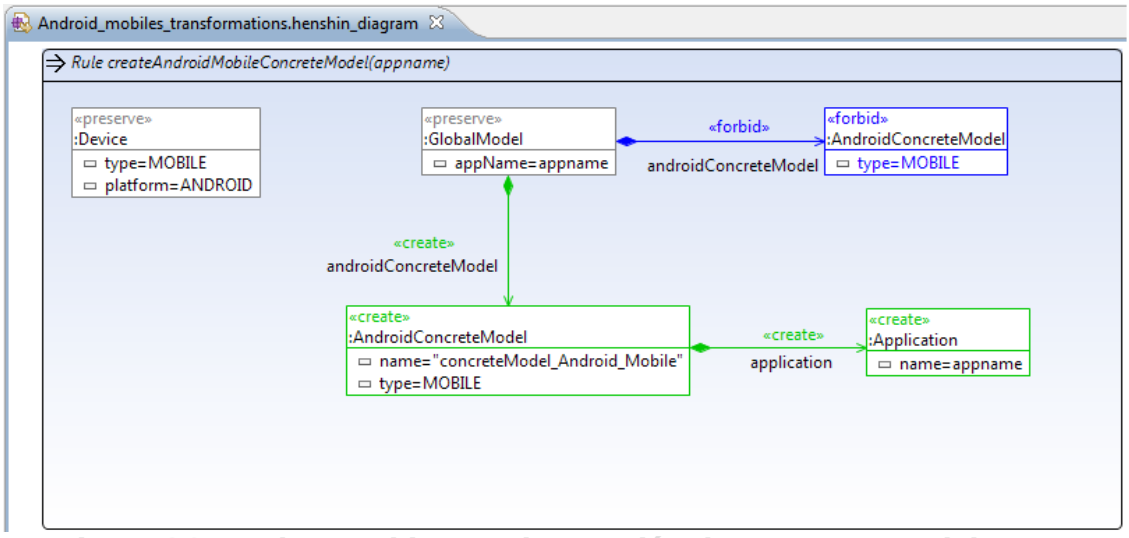


Figura 36. Regla Henshin para la creación de un nuevo modelo para teléfonos móviles de la plataforma Android

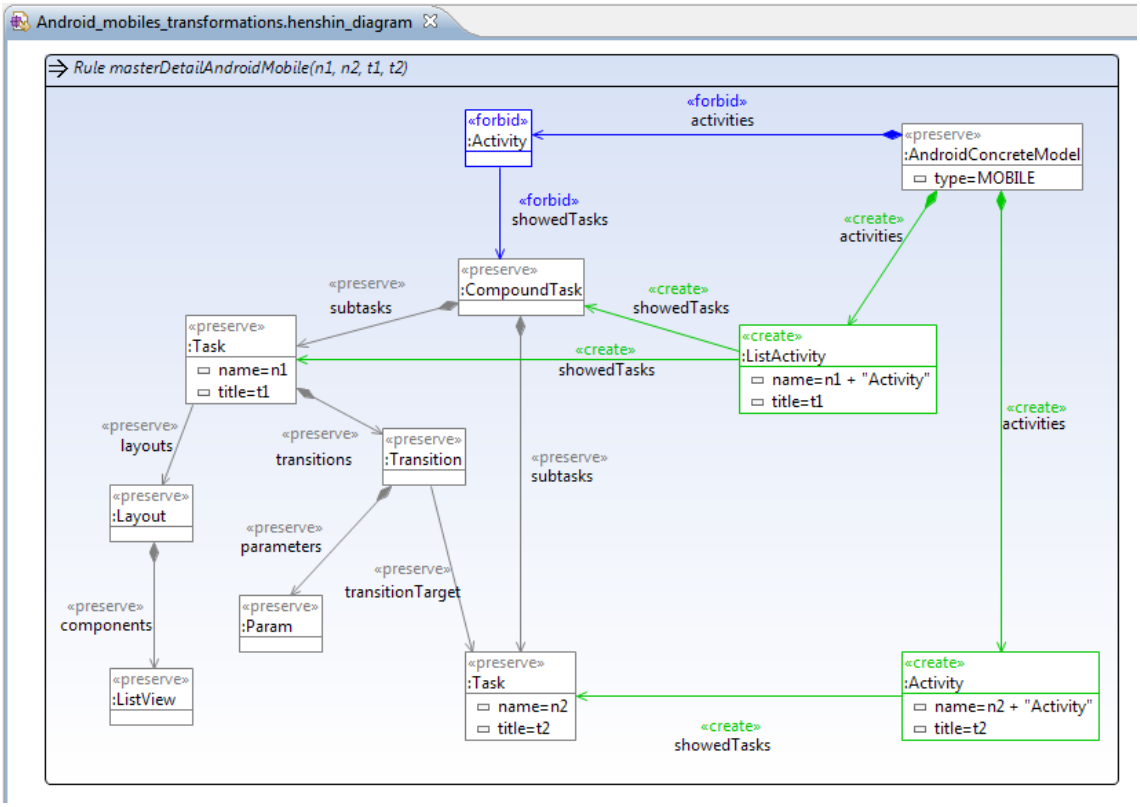
La Figura 36 muestra la regla inicial para la creación de un modelo para teléfonos móviles de la plataforma Android. Como precondición (LHS) se establece la necesidad de que el modelo inicial incluya un dispositivo de tipo *MOBILE* y plataforma *ANDROID*, en caso afirmativo se creará un nuevo *AndroidConcreteModel* de tipo *MOBILE* que contendrá un nodo de tipo *Application*. La NAC definida servirá como condición de control evitando que la regla se ejecute más de una vez. La ejecución del resto de reglas definidas para los teléfonos móviles de la plataforma Android completará el nuevo modelo con todos los nodos necesarios para la posterior generación del código fuente de la aplicación [44].

Para el prototipo de ejemplo se han creado tres sistemas de transformaciones, uno para la generación del modelo para teléfonos Android, otro para tablets Android y finalmente uno que creará un modelo para dispositivos de la plataforma WP7. Cada uno de estos sistemas de transformación está formado por un conjunto de reglas y *transformation units* (*loops y sequential units*) que controlan su aplicación. Como

veíamos en la descripción del ejemplo en el apartado anterior mediante el uso de algunas de estas reglas conseguiremos adaptar la aplicación resultante a diferentes patrones de diseño utilizados con frecuencia en las distintas plataformas y descritos en sus guías de estilo o buenas prácticas. Como ya se comentó previamente en la descripción del ejemplo se han utilizado principalmente dos transformaciones cuyo propósito será definir el sistema de navegación principal de la aplicación (*ActionBar* con pestañas para tablets Android, *DashBoard* en móviles Android y *Pivot Control* en WP7) así como la representación más adecuada para la visualización de los datos maestro/detalle (una única vista en dispositivos de tipo tablet y dos vistas independientes para teléfonos móviles con tamaño de pantalla más reducido).

Para explicar el funcionamiento del módulo de transformaciones utilizaremos la aplicación ejemplo que estamos describiendo en la versión para teléfonos móviles Android, las reglas utilizadas para tabletas Android y dispositivos WP7 son similares a éstas. No obstante al final de este apartado mostraremos las reglas equivalentes utilizadas para ofrecer una navegabilidad alternativa basada en un *ActionBar* con pestañas en dispositivos de tipo tablet. Mediante este ejemplo podremos ver las posibilidades de adaptación que nos ofrecen las reglas y cómo a partir de una misma descripción se obtienen dos versiones de la interfaz totalmente diferentes.

La Figura 37



muestra la regla a ejecutar para mostrar los datos maestro/detalle (lista de elementos de una categoría y detalle de uno de esos elementos) en dos pantallas diferentes. Al tratarse de una aplicación Android se creará un *ListActivity* para mostrar la lista de elementos y otro *Activity* para la visualización del detalle. Como podemos ver la LHS (grafo de precondición) obliga a la existencia de ciertos nodos en la instancia del modelo EMF. Para que se cumpla la precondición y que por tanto se ejecute la regla el modelo deberá tener una tarea compuesta con dos subtareas habiendo una transición de una a otra subtarea con paso de parámetros. Es necesario también que la primera de las subtareas incluya un componente de tipo *ListView* en

el layout asociado. Mediante la invocación de esta regla en un *transformation unit* de tipo *loop* se obtendrá el mismo resultado para cada una de las tres categorías de la aplicación de ejemplo.

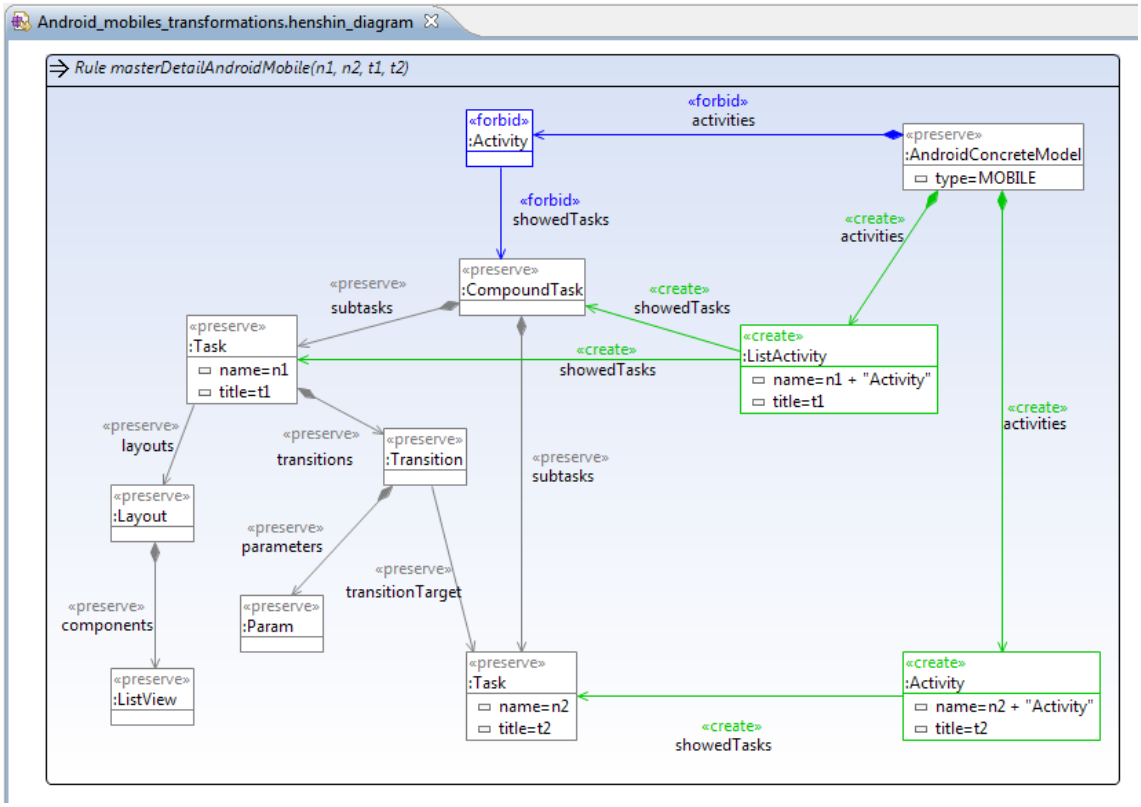


Figura 37. Regla para la visualización de maestro/detalle en dos Activities independientes

Las siguientes figuras (Figura 38, Figura 39 y Figura 40) muestran las reglas y *transformation units* necesarias para la creación del patrón de diseño *Dashboard* como método de navegación principal de la aplicación para móviles Android. La generación de los patrones para el resto de dispositivos sería muy similar, obteniéndose en lugar de un *Dashboard* una navegación basada en un *ActionBar*

con pestañas o en un control *Pivot* para tablets Android y dispositivos WP7 respectivamente.

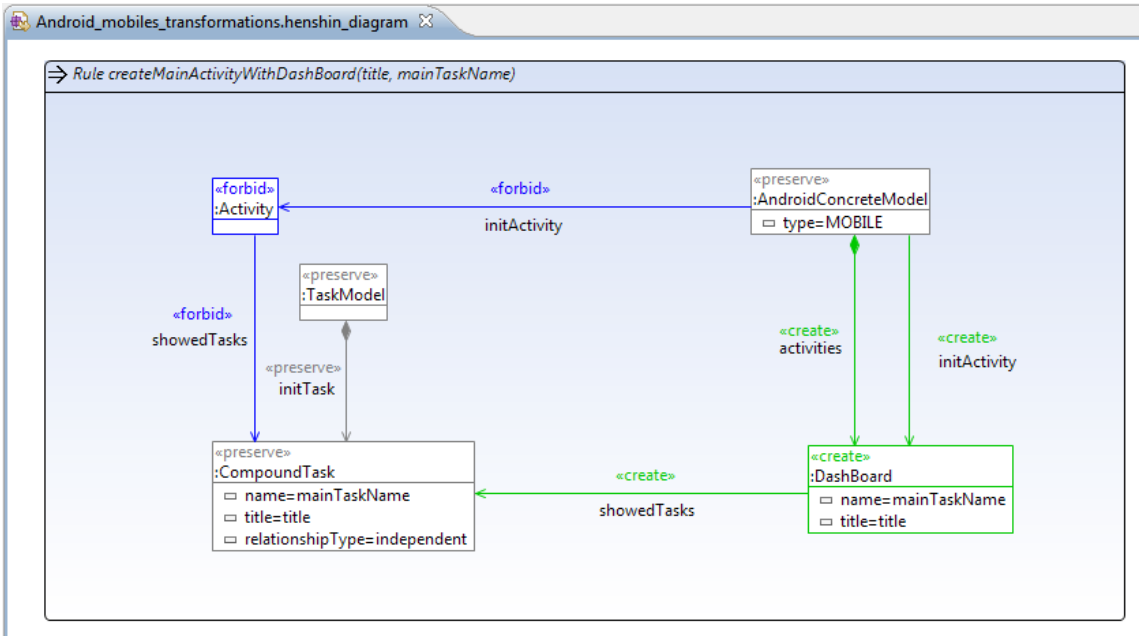


Figura 38. Regla para la creación de un Dashboard para la navegación principal en teléfonos Android

La ejecución de la regla de la Figura 38 creará un DashBoard para el AndroidConcreteModel encargado de dar una vista a la tarea principal (*initTask* del modelo de tareas). Los valores de algunas propiedades de la *CompoundTask*, como el título, se utilizan para inicializar otras propiedades de los nuevos nodos mediante el uso de variables en Henshin.

La regla mostrada en la Figura 39 creará un DashBoardItem para las subtareas de la tarea principal. Cada *DashBoardItem* creado apuntará al *Activity* asociado con la tarea encargada de mostrar la lista de elementos de una categoría (libros, discos o películas). Posteriormente en la fase de generación de código se creará un fichero XML para definir el layout del *DashBoard*, una clase *Activity* y un widget de tipo botón por cada *DashBoardItem* que permitirá al usuario navegar a la lista de elementos de cada categoría.

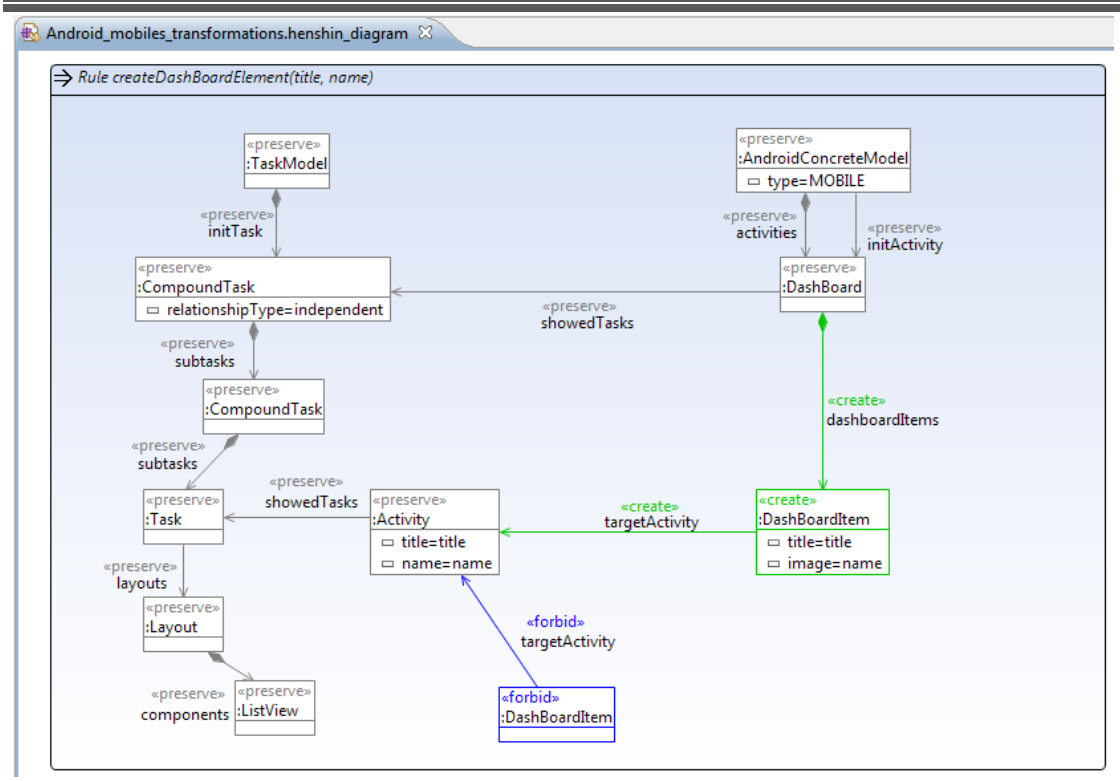


Figura 39. Regla para la creación de los DashBoardItems para navegar a cada subtarea de la tarea principal

La Figura 40 muestra las transformations units que controlan la ejecución de las reglas mostradas anteriormente para la creación de un DashBoard. La segunda transformation unit (*createNavigationWithDashBoard*), de tipo secuencial, es realmente la encargada de desencadenar todo el proceso necesario para la generación de una navegación basada en un DashBoard. En primer lugar invocará a la regla de la Figura 38 encargada de generar el DashBoard, y posteriormente a la *transformation unit createAllDashBoardElement* de tipo loop que ejecutará en un bucle la regla de la Figura 39 para crear un DashBoardItem por cada subtarea de la *CompounTask* inicial hasta que se produzca una condición de parada ocasionada por la NAC definida en dicha regla.

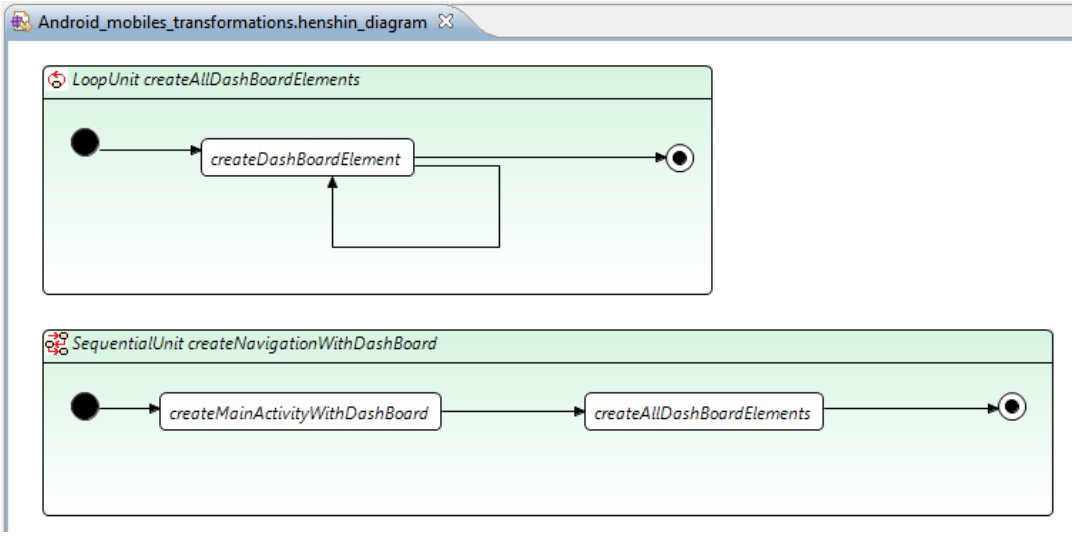


Figura 40. Transformation units para el control de la ejecución de las reglas anteriores

Para hacernos una idea de las posibilidades y la flexibilidad que nos ofrecen las reglas de transformación veremos cómo se puede obtener una interfaz totalmente distinta a partir de la misma descripción gracias a la aplicación de un conjunto de reglas diferentes.

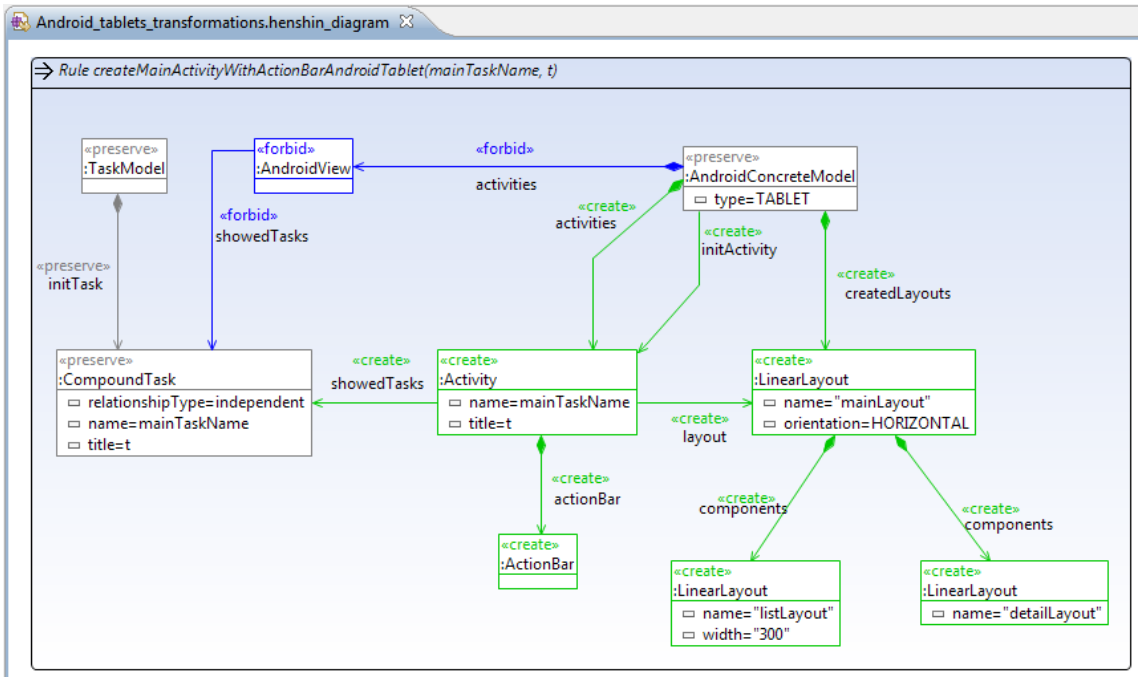


Figura 41. Regla Henshin para crear un Activity principal, su layout y un ActionBar

La Figura 41 muestra la regla Henshin inicial utilizada para crear una navegación basada en un *ActionBar* con pestañas. Mediante la ejecución de esta regla se crea un *Activity* asociado a la tarea principal (*initTask*) con un *ActionBar*. Posteriormente mediante la ejecución de la regla definida en la Figura 42 (invocada en un

transformation unit de tipo *loop*) se creará una pestaña (*ActionBarTab*) para acceder a cada tarea hija de la tarea inicial, es decir, una pestaña por cada categoría.

Estas últimas reglas definidas (Figura 41 y Figura 42) son una alternativa a las reglas de navegación para móviles Android para el uso de un *DashBoard* (Figura 38 y Figura 39). Vemos cómo a partir de una única descripción la potencia y la flexibilidad que nos ofrece Henshin nos permite obtener dos patrones de navegación, y por tanto dos aplicaciones, totalmente diferentes. El resultado de la navegación mediante *DashBoard* puede observarse en la Figura 2 mientras que en la Figura 5 podemos ver cómo las últimas reglas aplicadas para dispositivos tablet producen una navegación basada en un *ActionBar* con pestañas.

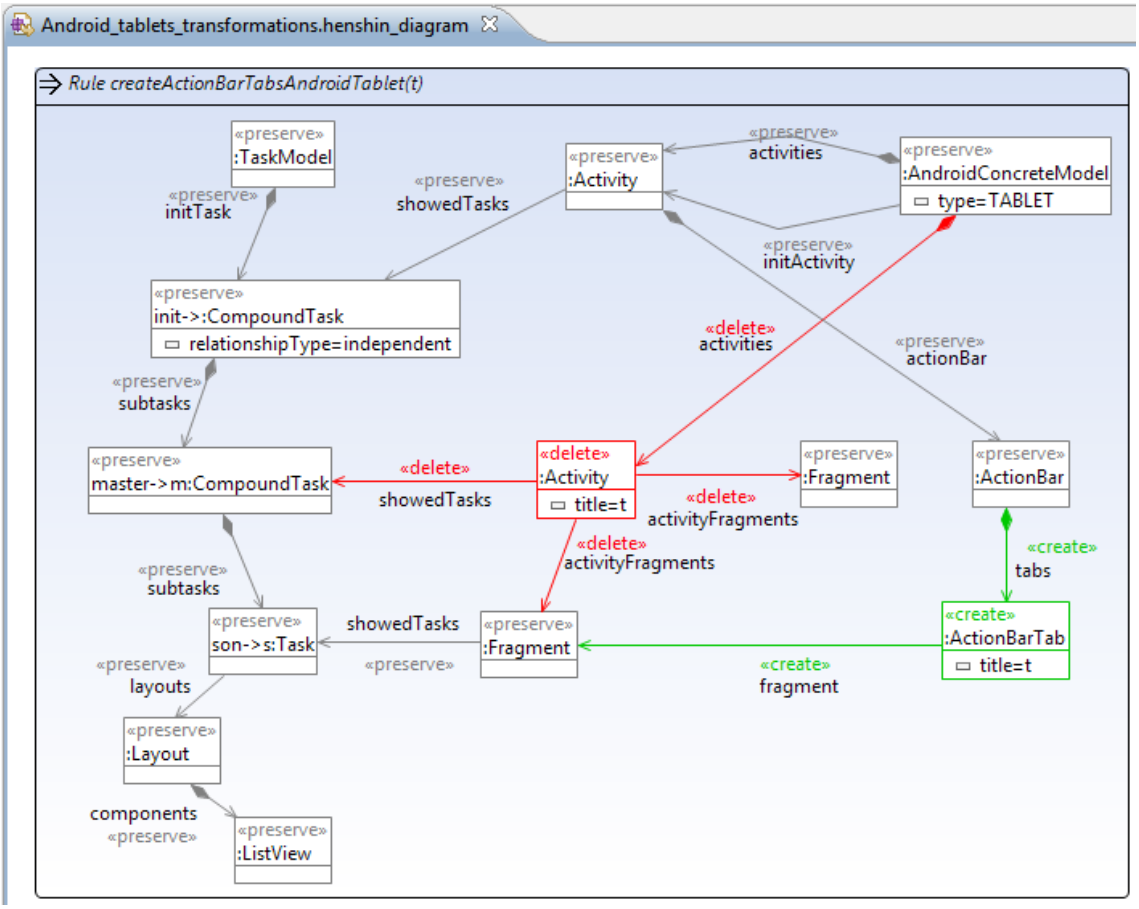


Figura 42. Regla Henshin para la creación de una pestaña para cada categoría

Un aspecto muy importante a tener en cuenta es que las reglas Henshin no son más que instancias de un meta-modelo, por lo que pueden ser creadas dinámicamente de forma sencilla. La idea para la creación de una herramienta final basada en esta aproximación sería combinar el uso de estructuras pregeneradas (descripción del patrón básico de la regla) con el uso de otras partes de la regla creadas dinámicamente, generalmente a partir del análisis de la instancia del modelo de una aplicación concreta.

Paso 3: Generación de código fuente

Una vez finalizado el proceso de transformación basado en Henshin y descrito en el paso anterior ya disponemos de una descripción completa de cada una de las versiones de la aplicación a desarrollar, por lo que podremos comenzar con el proceso de generación de código fuente [45].

El módulo de generación de código buscará en el modelo obtenido tras la ejecución de las transformaciones los distintos modelos finales creados para cada plataforma y tipo de dispositivo.

Como hemos comentado anteriormente en la descripción de la arquitectura el módulo de generación está basado en el lenguaje Xtend. Una vez más Xtend ofrece una integración total con el proyecto EMF.

La Figura 43 muestra parte de la clase Xtend utilizada para generar el código de las vistas para WP7. Esta clase implementa la interfaz *IGenerator*, que nos obliga a implementar el método *doGenerate*. Dicho método recibe como parámetro *Resource* y *IFileSystemAccess*, el parámetro *Resource* nos permite acceder a la instancias de los modelos y facilitándonos la tarea de iterar sus elementos, filtrarlos, etc. Por ejemplo en la línea 39 vemos cómo podemos filtrar todos los elementos de tipo *ApplicationPage* (representación de una vista o página en el modelo de WP7) e iterarlos para generar tanto el código C# de la clase como el código XAML. El parámetro *IFileSystemAccess* nos ofrece acceso a disco para generar los ficheros de código fuente de salida mediante el método *generateFile*, al que le indicamos como primer parámetro la ruta y nombre del fichero de salida y en segundo lugar su contenido (en nuestro ejemplo el resultado de la invocación a los métodos *compileCodeBehind* y *compileXAML* según el código a generar). A partir de la línea 45 se muestra el código Xtend para la generación de code behind. En la primera línea del método (46) se crea e inicializa un objeto de tipo *BasicNamespaceManager*, que nos permitirá almacenar en una colección los *namespaces* necesarios, la segunda línea (47) invoca a la sobrecarga del método *compileCodeBehind* (línea 61) y almacena el resultado en la variable *mainMethod.* Posteriormente se itera los namespaces añadidos y se escriben en el fichero de salida precedidos por la palabra reservada *using*. Vemos como de forma muy sencilla Xtend nos permite preprocesar el código de salida y almacenarlo en una variable para conocer los namespaces que necesitamos utilizar pudiendo volcarlos al principio del fichero de código fuente. En la clase Xtend que estamos comentando podemos ver algunas de las facilidades que nos ofrece Xtend para la generación de código. Vemos por ejemplo como los dos métodos *compileCodeBehind* utilizan la triple comilla simple para facilitar la creación de plantillas, todo el código incluido entre esos caracteres se volcará al fichero de salida a excepción del código entre los símbolos «», que nos permitirá definir código Xtend para el uso de condicionales, bucles, etc. Las facilidades de Xtend son aún mayores, ya que automáticamente se encarga también de manejar el formateo del código de salida, por lo que el código obtenido mantendrá el espaciado y tabulaciones adecuadas para facilitar su legibilidad.

Como ya habíamos comentado el uso del patrón MVVM evita la necesidad de generar code behind, sólo es necesario crear la clase con el constructor sin parámetros e invocar al método *InitializeComponent.*

WP7ViewsCodeGenerator.xtend

```
31 class WP7ViewsCodeGenerator implements IGenerator{
32
33     @Inject extension WP7GeneratorExtensions
34     GlobalModel gm
35
36     override doGenerate(Resource resource, IFileSystemAccess fsa) {
37         gm = resource.allContents.filter(typeof(GlobalModel)).head
38
39         for (e : resource.allContents.toIterable.filter(typeof(ApplicationPage))) {
40             fsa.generateFile(baseDirectory + 'View/' + e.name.toFirstUpper + ".xaml.cs", e.compileCodeBehind)
41             fsa.generateFile(baseDirectory + 'View/' + e.name.toFirstUpper + ".xaml", e.compileXAML)
42         }
43     }
44
45     def compileCodeBehind(ApplicationPage it) '''
46         «val usingManager = new BasicNamespaceManager()»
47         «val mainMethod = compileCodeBehind(usingManager)»
48
49         «IF !usingManager.getNamespaces.empty»
50             «FOR i : usingManager.getNamespaces»
51                 using «i»;
52             «ENDFOR»
53         «ENDIF»
54
55         namespace «gm.appName».View
56         {
57             «mainMethod»
58         }
59     '''
60
61     def compileCodeBehind(ApplicationPage it, BasicNamespaceManager im) '''
62         public partial class «name.toFirstUpper» : «im.addNamespace("Microsoft.Phone.Controls")»PhoneApplicationPage
63         {
64             public «name.toFirstUpper»()
65             {
66                 InitializeComponent();
67             }
68         }
```

Figura 43. Generación de código para las vistas en WP7

La Figura 44 muestra parte del código Xtend necesario para la generación de XAML, al igual que sucedía anteriormente se usan plantillas para su obtención. La línea 87 demuestra realmente toda la potencia que nos ofrece Xtend. En dicha línea se utiliza la función de extensión *map* de las listas, *map* espera una función como parámetro e invoca a esa función para cada elemento de la lista retornando otra lista que contiene los resultados de las invocaciones a la función. En nuestro caso para cada componente de la lista *components* se invoca a la función *compileComponent* (mostrada en la Figura 45). Posteriormente mediante la función *join* se juntan todos los resultados en un *String* para volcarlos al fichero de salida.

WP7ViewsCodeGenerator.xtend

```
71     def compileXAML(ApplicationPage it) '''
72         «val namespaceManager = new BasicNamespaceManager()»
73         «val body = layout?.compileApplicationPageBody(namespaceManager)»
74         <phone:PhoneApplicationPage
75             «compileXAMLNamespaces(namespaceManager)»
76             «compileApplicationPageProperties»>
77
78             «body»
79
80         </phone:PhoneApplicationPage>
81     '''
82     def dispatch compileApplicationPageBody(Layout it,BasicNamespaceManager im){}
83
84     def dispatch compileApplicationPageBody(LinearLayout it, BasicNamespaceManager im)'''
85         <Grid x:Name="LayoutRoot" Background="Transparent">
86             <StackPanel Orientation="«orientation.toString.toLowerCase.toFirstUpper»" Margin="8">
87                 «components.map[compileComponent].join»
88             </StackPanel>
89         </Grid>
90     '''
```

Figura 44. Generación de código XAML

En la Figura 45 vemos otra de las facilidades proporcionadas por Xtend para la generación de código. Los métodos *compileComponent* mostrados en esa figura están

sobrecargados con diferentes parámetros y todos ellos utilizan en su definición la palabra reservada *dispatch*. El uso de la palabra reservada *dispatch* convierte los métodos sobrecargados en métodos *dynamically dispatched*, es decir, se utiliza el *runtime type* del parámetro para resolver la sobrecarga. Esto facilita en gran medida la generación de código y evita la necesidad del uso del patrón *Visitor* [7]. En nuestro caso vemos cómo podemos utilizar una única plantilla (método *compileComponent*, línea 98) para la generación del código XAML de la mayoría de los widgets (*TextBlock*, *TextBox*, *Image*, etc.) redefiniendo este método sólo en aquellos casos en los que sea necesario, como sucede por ejemplo para el componente *ListView* que debe incluir las etiquetas *XAML* necesarias para definir sus ítems.

WP7ViewsCodeGenerator.xtend

```
92    def dispatch compileComponent(LinearLayout it)'''
93        <StackPanel Orientation="«orientation.toString.toLowerCase.toFirstUpper»" Margin="8">
94            «components.map[compileComponent].join»
95        </StackPanel>
96    '''
97
98    def dispatch compileComponent(Widget it)'''
99        «IF listeners.empty || isWidgetIntoListView»
100           <«componentName» «compileWidgetProperties»/>
101       «ELSE»
102           <«componentName» «compileWidgetProperties»>
103               «compileAllListeners»
104           </«componentName»>
105       «ENDIF»
106   '''
107
108   def dispatch compileComponent(ListView it)'''
109       <ListBox «compileWidgetProperties»>
110           «compileAllListeners»
111           <ListBox.ItemTemplate>
112               <DataTemplate>
113                   «listItemLayout.compileComponent»
114               </DataTemplate>
115           </ListBox.ItemTemplate>
116       </ListBox>
117   '''
```

Figura 45. Código Xtend para la generación de los componentes gráficos en XAML

Un aspecto clave que repercute directamente en la sencillez de la fase de generación de código es la creación de modelos específicos para cada tipo de generación (por plataforma y tipo de dispositivo). En comparación con UsiXML resultará mucho más sencilla esta fase ya que partimos de un modelo más cercano al dispositivo final (en contraposición con el modelo concreto de UsiXML con semántica común a todos los dispositivos objetivo).

6.3 Resultados obtenidos

Como se ha comentado anteriormente el objetivo principal del sistema es permitir el desarrollo de aplicaciones multidispositivo (centrándose principalmente en su interfaz de usuario) para obtener unos resultados lo más adaptados posible a los distintos modos de interacción y siguiendo las pautas y estilos más adecuados en cada caso. En los apartados anteriores se ha descrito la aplicación de ejemplo desarrollada para validar el sistema y el conjunto de pasos necesarios para construir dicha aplicación. A continuación se mostrarán capturas de pantalla de la aplicación real ejecutada en tres tipos de dispositivos diferentes para comprobar cómo gracias al nuevo nivel de abstracción introducido en este trabajo (PUI) y a la flexibilidad y potencia de las transformaciones de modelos obtenemos tres versiones de la misma aplicación totalmente diferentes a nivel de interfaz de usuario. Las plataformas seleccionadas para la generación de la aplicación han sido Android y WP7, obteniéndose en el caso de Android una versión para smartphones y otra para tablets.

6.3.1 Aplicación Android para smartphones

Para la selección de la categoría la aplicación Android para smartphones presenta un *Activity* independiente con un botón de acceso a cada una de las opciones (Figura 46). Este es un patrón de uso común en dispositivos móviles para facilitar la navegación por las distintas opciones de la aplicación (conocido como patrón DashBoard ó Springboard), siendo referenciado en varias ocasiones por los expertos de Android en interfaces de usuario [47] así como en catálogos de patrones de diseño de interfaces [10].

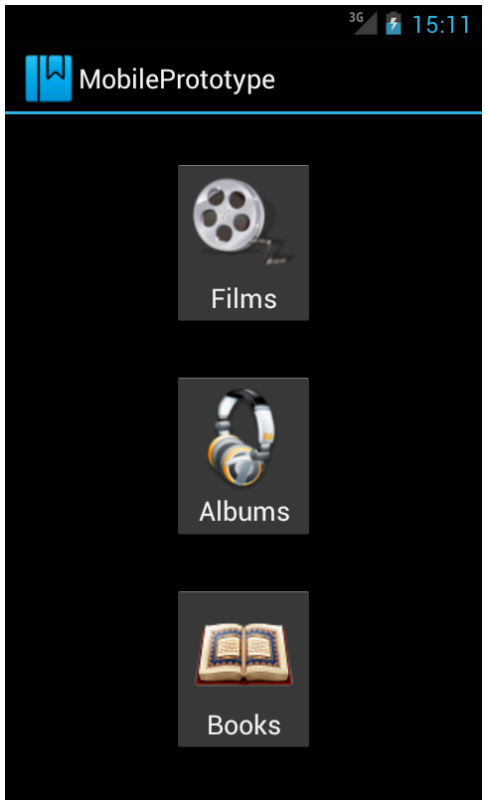


Figura 46. Selección de la categoría en móviles Android

Una vez elegida la categoría deseada se le presenta al usuario la posibilidad de seleccionar un producto de entre todos los pertenecientes a la categoría y mostrados en forma de lista (Figura 47).

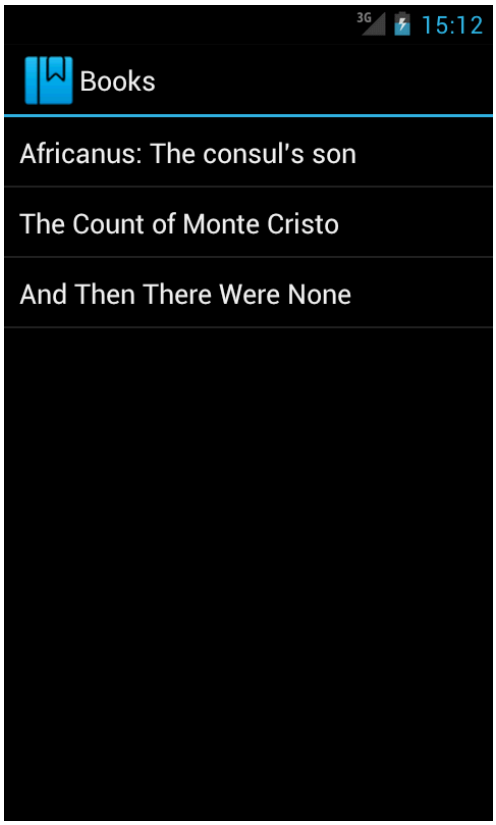


Figura 47. Lista de elementos disponibles de una categoría

A continuación el usuario podrá elegir uno de esos productos para acceder a su detalle completo (Figura 48).

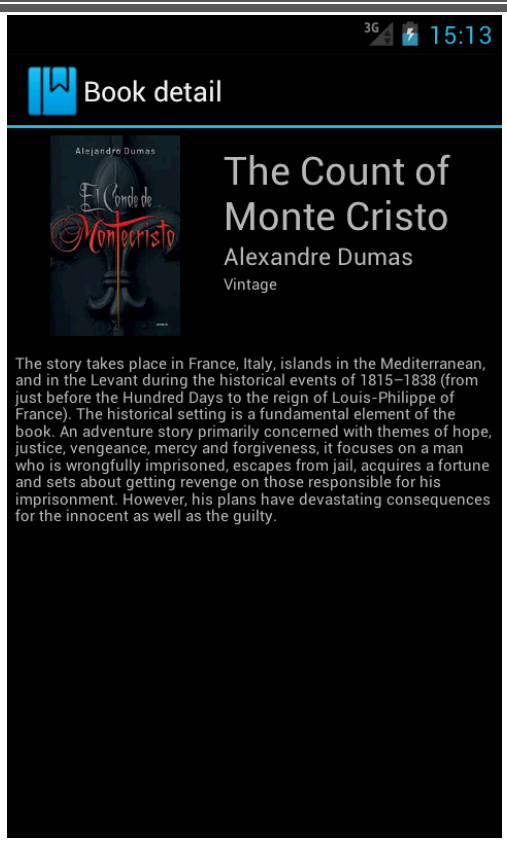


Figura 48. Detalle de uno de los productos

6.3.2 Aplicación Android para tablets

La Figura 49 muestra la interfaz de usuario generada para dispositivos tablet Android. Como podemos ver y a diferencia de la versión para móviles se utiliza una única pantalla para mostrar todas las tareas que puede llevar a cabo el usuario mediante el uso de la aplicación. El método de navegación utilizado en este caso en lugar de estar basado en el patrón Dashboard se basa en el uso de un ActionBar renderizado en la parte superior de la pantalla y con una pestaña para cada una de las categorías. Una vez iniciada la aplicación se mostrará la primera de las categorías y el usuario podrá cambiarla simplemente accediendo a las otras pestañas. A diferencia de lo que sucedía en la aplicación para móviles la tarea que permite la selección de un producto y su detalle se muestran también en una única pantalla. Este patrón de diseño de interfaces conocido como Master-detail es usado generalmente en dispositivos con tamaños de pantalla grandes, como las tablets, ya que facilita la visualización de los datos y el cambio entre elementos.

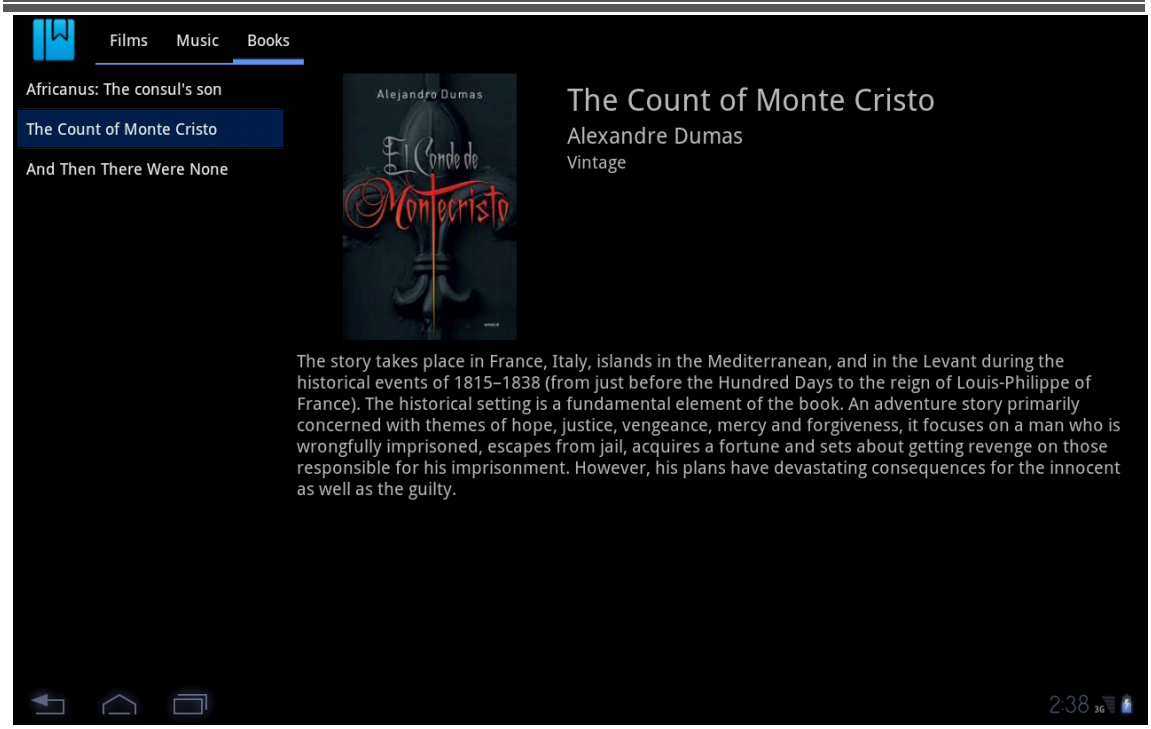


Figura 49. Interfaz de usuario generada para tablets Android

6.3.3 Aplicación WP7

La Figura 50 muestra la pantalla inicial generada por el sistema para dispositivos WP7. Esta aplicación está basada en el uso de un control propio de WP7, el control Pivot, que permite cambiar de forma sencilla la vista o página mostrada por la aplicación mediante gestos a izquierda o derecha sobre la pantalla. Vemos una vez más cómo la apariencia de esta aplicación tiene poco que ver con las versiones generadas tanto para móviles como para tablets Android. En este caso cuando el usuario accede a la aplicación se le muestra la lista de productos de la categoría inicial al igual que sucedía con la aplicación para tablets Android, sin embargo el mecanismo de navegación entre las categorías es diferente realizándose mediante los gestos descritos anteriormente en lugar de accediendo a las pestañas. En este caso la información de la lista de productos y el detalle también se muestran en dos pantallas diferentes debido al menor tamaño de pantalla en comparación con los dispositivos de tipo tablet.

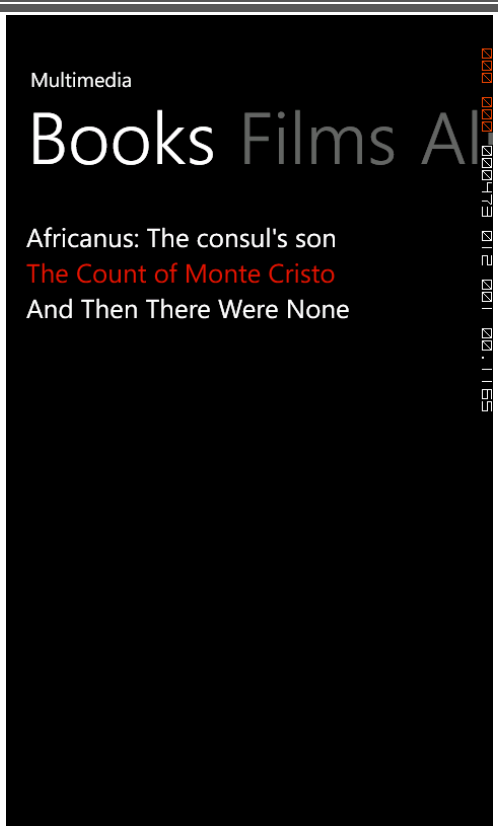


Figura 50. Pantalla inicial de la aplicación para WP7

La siguiente figura (Figura 51) muestra la vista de detalle de un producto, que como comentábamos se visualiza en una pantalla independiente.

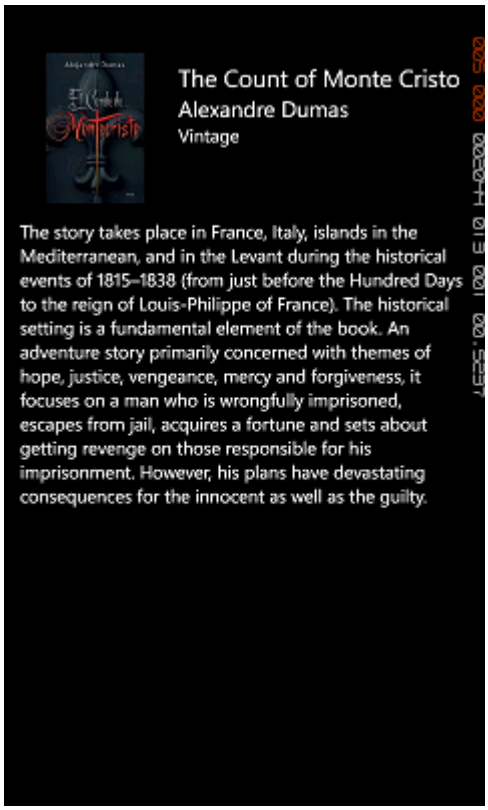


Figura 51. Pantalla independiente para el detalle de un producto

6.4 Conclusiones sobre los resultados

Como hemos visto en los apartados anteriores al presentar las interfaces gráficas de las tres versiones distintas de la misma aplicación generadas para cada tipo de dispositivo y plataforma los resultados obtenidos a nivel de interfaz difieren bastante de una a otra aplicación. Cada una de las interfaces está adaptada a una serie de patrones de interfaz de usuario propios de su plataforma o del tipo de dispositivo (móvil o tablet). Podemos concluir que la interfaz resultante se adapta perfectamente al dispositivo concreto para el que se genera lo que facilita su uso por parte de los usuarios.

El modelo intermedio propuesto para describir en detalle la plataforma (PUI) y el uso del módulo de transformaciones basado en reglas Henshin permiten obtener varias representaciones de una misma aplicación a partir de una única descripción a más alto nivel (HLUI). Además de la adaptación visual y como ya hemos comentado el uso del modelo PUI también nos permite la aplicación de patrones arquitectónicos en el código fuente resultante y en general facilita el uso de buenas prácticas durante la fase de generación de código.

Capítulo 7. Conclusiones y Trabajo Futuro

7.1 Conclusiones

Este trabajo sugiere el uso de un nivel intermedio entre los niveles CUI y FUI del framework CAMELEON, al que llamamos PUI (Platform User Interface). El nivel PUI incluye modelos de las plataformas objetivo y facilita la transformación de CUI a FUI teniendo en cuenta los patrones de diseño y arquitectónicos recomendados para cada plataforma así como las guías de estilo y recomendaciones de cada fabricante. En los ejemplos desarrollados hemos visto como a partir de una única descripción a alto nivel (HLUI) se han obtenido tres versiones distintas adaptadas en cada caso a la plataforma y tipo de dispositivo objetivo. Las versiones finales tienen en cuenta patrones arquitectónicos (MVVM en WP7 y arquitectura basada en el Activities en Android) y patrones de diseño de interfaces de usuario (DashBoard en teléfonos Android, ActionBar en tablets Android y control Pivot en WP7).

La flexibilidad en el diseño del módulo de meta-modelado basado EMF y del módulo de transformaciones basado en Henshin permitirá añadir nuevas plataformas objetivo o versiones de éstas y nuevas reglas que ofrezcan la posibilidad de aplicar nuevos patrones.

7.2 Trabajo Futuro

LIZARD basa su arquitectura en varias de las herramientas de EMP, lo que facilitaría una integración futura con el proyecto UsiXML que utiliza modelos Ecore. Una posibilidad sería la de sustituir los modelos propuestos en el nivel HLUI de LIZARD por los modelos de UsiXML para los niveles TC, AUI y CUI, lo que permitiría una independencia de la modalidad de uso (LIZARD sólo contempla el uso de interfaces gráficas). Tras esta integración las herramientas ya desarrolladas para UsiXML o nuevas herramientas podrían utilizar las ventajas que proporciona el nuevo nivel PUI organizando la generación de código de una manera más eficiente y facilitando la creación de aplicaciones que sigan las guías de diseño de cada fabricante y los patrones de diseño más adecuados en cada caso.

Otro punto clave a destacar como trabajo futuro sería la inclusión en el sistema de nuevos modelos definidos a nivel PUI que permitiesen la generación para nuevas plataformas, no sólo en el ámbito de dispositivos móviles sino también para sistemas operativos de escritorio. El mayor esfuerzo estaría en la necesidad de crear nuevas reglas para la plataforma y nuevos generadores de código.

Un aspecto que también puede resultar interesante para incluir en herramientas de este tipo sería la posibilidad de introducir de forma sencilla código creado de forma manual por la comunidad de desarrolladores. Un ejemplo de esto se puede ver en la implementación realizada para el modelo de acceso a datos. La idea sería ofrecer

implementaciones por defecto para algunas interfaces y que el desarrollador pueda cambiar la implementación por otra a nivel de modelo PUI. El generador de código sería el encargado de configurar la implementación adecuada en el Factory Method o en la configuración del contenedor de inyección de dependencias.

7.3 Difusión de los Resultados

El framework LIZARD ha sido presentado en forma de artículo de investigación a la revista "*Expert Systems with Applications"* [46], que se ubica en la sección *Computer Science, Artificial Intelligence* del Journal Citation Reports con un factor de impacto en 2011 de 2,203 (primer cuartil de la categoría mencionada). En el anexo **¡Error! No se encuentra el origen de la referencia.** se incluye la versión completa del artículo enviado.

Capítulo 8. Referencias

[1] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, "A Unifying Reference Framework for Multi-Target User Interfaces," Interacting with Computers, vol. vol, pp. 15no3289-308, 2003.

[2] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "USIXML : a Language Supporting Multi-Path Development of User Interfaces," Management.

[3] F. Paterno', C. Santoro, and L. D. Spano, "MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments," ACM Transactions on Computer-Human Interaction, vol. 16, no. 4, pp. 1-30, Nov. 2009.

[4] S. Berti, F. Correani, F. Paternò, C. Santoro, and V. G. Moruzzi, "The TERESA XML Language for the Description of Interactive Systems at Multiple Abstraction Levels."

[5] B. Michotte and J. Vanderdonckt, "GrafiXML, a Multi-target User Interface Builder Based on UsiXML," Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08), pp. 15-22, 2008.

[6] S. Berti, F. Correani, G. Mori, F. Paternò, C. Santoro, and V. G. Moruzzi, "TERESA : A Transformation-based Environment for Designing and Developing Multi-Device Interfaces," Evaluation, pp. 1-2, 2004.

[7] Miguel Garcia, David Llewellyn-Jones, Francisco Ortin, Madjid Merabti. Applying dynamic separation of aspects to distributed systems security: a case study. IET Software, volume 6, issue 3, pp. 231-248. June 2012.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional, 1995.

[9] M. Potel, "MVP : Model-View-Presenter The Taligent Programming Model for C ++ and Java," 1996.

[10] M. Fowler, "Presentation Model," Essay, July, 2004.

[11] T. Neil, Mobile Design Pattern Gallery. 2012.

[12] J. Guerrero-garcia, J. M. Gonzalez-, J. Vanderdonckt, J. Guerrero-garcía, J. M. González-calleros, and J. Muñoz-arteaga, "A theoretical survey of user interface description languages : Preliminary results," 2009.

[13] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, "UIML : an appliance-independent XML user interface language," vol. 31, pp. 1695-1708, 1999.

[14] A. Puerta and J. Eisenstein, "XIML : A Universal Language for User Interfaces," Components, 2001.

[15] F. Paternò, C. Mancini, and S. Meniconi, “ConcurTaskTrees: A diagrammatic notation for specifying task models,” pp. 362-369, 1997.

[16] “UsiXML.” [Online]. Available: http://www.usixml.org/.

[17] G. Calvary, J. Coutaz, and D. Thevenin, “The CAMELEON Reference Framework.” [Online]. Available: http://giove.isti.cnr.it/projects/cameleon.html.

[18] G. Taentzer, “AGG: A Graph Transformation Environment for Modeling and Validation of Software.”

[19] J. Lara and H. Vangheluwe, “AToM 3: A Tool for Multi-formalism and Meta-modelling,” Fundamental approaches to software engineering, pp. 174-188, 2002.

[20] D. Varró and A. Balogh, “The model transformation language of the VIATRA2 framework,” Science of Computer Programming, vol. 68, no. 3, pp. 214-234, Oct. 2007.

[21] J. Warmer and A. Kleppe, The object constraint language: precise modeling with UML. Addison-Wesley, 1998.

[22] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, “ATL: a QVT-like transformation language,” in Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, 2006, pp. 719-720.

[23] “Eclipse M2M Project,” 2011. [Online]. Available: http://www.eclipse.org/m2m/.

[24] JBoss, “Drools,” 2011. [Online]. Available: http://www.jboss.org/drools/.

[25] C. L. Forgy, “Rete : A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem,” vol. 19, no. 3597, pp. 17-37, 1982.

[26] “EMF Tiger,” 2011. [Online]. Available: http://www.eclipse.org/proposals/emf-tiger/.

[27] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, “Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations,” pp. 121-135, 2010.

[28] J. D. Lara and G. Taentzer, “Automated Model Transformation and Its Validation Using AToM3 and AGG,” Diagrammatic Representation and Inference, pp. 257-270, 2004.

[29] D. Varró and A. Pataricza, “Automated Formal Verification of Model Tranformations,” Communication, no. Otka 038027, 2000.

[30] R. Heckel, J. M. Küster, and G. Taentzer, “Confluence of Typed Attributed Graph Transformation Systems,” Graph Transformation, 2002.

[31] R. Heckel and A. Wagner, “Ensuring Consistency of Conditional Graph Grammars - A Constructive Approach -,” Electronic Notes in Theoretical Computer Science, vol. 2, pp. 118-126, 1995.

[32] D. Varró, “Automated Model Transformations for the Analysis of IT Systems,” 2003.

[33] E. Biermann, C. Ermel, L. Lambers, U. Prange, O. Runge, and G. Taentzer, "Introduction to AGG and EMF Tiger by modeling a Conference Scheduling System," International Journal on Software Tools for Technology Transfer, vol. 12, no. 3–4, pp. 245-261, May 2010.

[34] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, Fundamentals of algebraic graph transformation. Springer-Verlag New York Inc, 2006.

[35] E. Biermann, C. Ermel, and G. Taentzer, "Precise Semantics of EMF Model Transformations by Graph Transformation," Cycle, pp. 1-15.

[36] A. Stanciulescu, Q. Limbourg, J. Vanderdonckt, B. Michotte, and F. Montero, "A transformational approach for multimodal web user interfaces based on UsiXML," Development, pp. 259-266, 2005.

[37] Francisco Ortin, Diego Díez Redondo. Designing an Adaptable Heterogeneous Abstract Machine by means of Reflection. Information and Software Technology, volume 47, issue 2, pp. 81-94. February 2005.

[38] S. Efftinge, "oAW xText : A framework for textual DSLs," 2006.

[39] "Xtext." [Online]. Available: http://www.eclipse.org/Xtext/.

[40] "Xtend User Guide." 2012.

[41] Ignacio Marin, Antonio Campos, José Quiroga, Patricia Miravet, Francisco Ortin. Device Independence approach for ICT-based PFTL Solutions. International Conference on Paperless Freight Transport Logistics (e-Freight), Munich (Germany). May 2011.

[42] "Funq." [Online]. Available: http://funq.codeplex.com/.

[43] C. Nesladek, G. Bauer, R. Fulcher, C. Robertson, and J. Palmer, "Android UI design patterns," 2010. [Online]. Available: http://www.google.com/events/io/2010/sessions/android-ui-design-patterns.html.

[44] Patricia Miravet, Ignacio Marín, Francisco Ortin, Abel Rionda. DIMAG: a framework for automatic generation of mobile applications for multiple platforms. 6th ACM International Conference on Mobile Technology, Application & Systems (Mobility), Nice (France). September 2009.

[45] Patricia Miravet, Ignacio Marin, Francisco Ortin, Javier Rodriguez. Framework for the declarative implementation of native mobile applications. IET Software (to be published).

[46] "Expert Systems with Applications." [Online]. Available: http://www.journals.elsevier.com/expert-systems-with-applications/.

[47] Francisco Ortin, José Manuel Redondo, J. Baltasar García Perez-Schofield. Efficient virtual machine support of runtime structural reflection. Science of Computer Programming, volume 74, issue 10, pp. 836-860. August 2009.

[48] Francisco Ortin, Daniel Zapico, Juan M. Cueva. Design Patterns for Teaching Type Checking in a Compiler Construction Course. IEEE Transactions on Education, volume 50, issue 3, pp. 273-283. August 2007.

[49] José Manuel Redondo, Francisco Ortin, Juan Manuel Cueva. Optimizing Reflective Primitives of Dynamic Languages. International Journal of Software Engineering and Knowledge Engineering, volume 18, issue 6, pp. 759-783. September 2008.