UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Derivación automática de código Android a partir de modelos gráficos
Autores: Agustín Vosou, Juan Pablo Martínez
Director: Claudia Pons
Codirector: -
Asesor profesional: -
Carrera: Licenciatura en Sistemas

Resumen

El Desarrollo de Software Dirigido por Modelos (Model Driven Development), es un paradigma cuyo uso ha venido en creciente aumento durante los últimos años y el mismo pretende mejorar, estandarizar y automatizar la construcción del software basándose en un proceso guiado por modelos y soportado por potentes herramientas.
Es evidente que el desarrollo de software dirigido por modelos se encuentra en constante crecimiento, y esto queda reflejado por la gran cantidad de herramientas que existen para modelar aplicaciones, las cuales permiten transformar un modelo en código fuente.
En el mundo de las comunicaciones, sin embargo, para el caso específico de las plataformas Android, las herramientas existentes, con el adicional de ser limitadas en cuanto al código o binario generado, no permiten modelar y/o representar en forma completa una aplicación.
Por estos motivos se ha decidido encarar el desarrollo de una herramienta que permita modelar una aplicación en forma íntegra para luego transformarlo en código fuente Android, con el fin de que el mismo pueda ser manipulado para lo que oportunamente se necesite.

Palabras Claves

Model Driven Development, Eclipse IDE, Ecore, Metamodelos, Modelos, Transformación modelo a texto, Eclipse Modeling Framework, Graphical Modeling Framework, Acceleo, Android.

Conclusiones

El mundo del desarrollo de software dirigido por modelos, específicamente para aplicaciones Android, resultan ser limitadas en cuanto al código o binario generado ya que no permiten representar en forma completa una aplicación. Por este motivo, a través del desarrollo de una herramienta, se logró efectivizar la generación de código fuente teniendo como origen un modelo definido por el usuario. Se consigue realizar una herramienta de software libre, cuyo modelo es totalmente independiente de la plataforma.

Trabajos Realizados

Se diseñaron e implementaron una serie de plugins para Eclipse IDE con el fin de brindar la posibilidad de generar un modelo gráfico y luego realizar su transformación a código fuente Android. El código se genera dentro de un proyecto totalmente configurado para ser ejecutado en cualquier dispositivo. La generación también provee de documentación embebida en el código con el fin de que el usuario tenga la posibilidad de realizar los cambios que se requieran.

Trabajos Futuros

Se propone la ampliación de la cantidad de Widgets disponibles para tener un metamodelo más completo.
La creación de componentes más complejos para la creación automática de CRUD, interactuar con el hardware del dispositivo, etc.
La generación de código fuente para otras plataformas, por ejemplo: dispositivos móviles, computadoras de escritorio, etc.
La posibilidad de cambiar el modelo sin perder cambios realizados en los fuentes por el desarrollador.

Fecha de la presentación: Septiembre 2014

Agradecimientos

Agustín Vosou

A mis padres Virginia y Víctor, por haberme dado la posibilidad de estudiar y apoyarme todo este tiempo. También a mis hermanos Lu, Vicky y Chipi por su confianza y soporte en la convivencia.

A mi novia Dani por su amor y apoyo constante en todo momento.

A mi compañero de tesis y amigo, Juampi.

Juan Pablo Martínez

A mi novia Vicky por su constante apoyo y presencia en estos últimos años de mi vida.

A mis viejos, Nélida y Carlos por haberme regalado la posibilidad de haber podido elegir la carrera que siempre quise.

A mi hermana Ita, mis amigos de la facultad, de Junín y al pequeño Rober.

A mi abuela que sigue estando presente.

Índice

1. Introducción 1

1.1. Motivación 1

1.2. Objetivos 3

2. Organización de la tesis 4

2.1.1. Conceptos Básicos 4

2.1.2. Descripción de tecnologías 4

2.1.3. Diseño de la solución 4

2.1.4. Implementación de la solución 4

2.1.5. Caso de estudio 4

2.1.6. Conclusiones y Trabajo futuro 4

3. Conceptos básicos 6

3.1. MDD (Model Driven Development) 6

3.1.1. Modelos 6

3.2. MDA (Model Driven Architecture) 9

3.3. Metamodelos 9

3.4. Transformación 10

3.4.1. OMG 11

3.4.2. MOF y la arquitectura de cuatro capas 11

3.4.3. MOFM2T 13

3.5. UML 13

3.5.1. Diagramas de estructura 14

3.5.2. Diagramas de comportamiento 15

3.5.3. Diagramas de interacción 15

4. Descripción de tecnologías 17

4.1. Eclipse IDE 18

4.1.1. Arquitectura de Eclipse IDE 19

4.1.2. El componente plugin y su arquitectura 21

4.2. EMF 21

4.2.1. Meta metamodelo Ecore 22

4.2.2. OCL 22

4.3. GEF 23

4.4. GMF 24

4.4.1. Definición del editor gráfico 25
4.5. Acceleo 26
4.5.1. Características 26
4.6. Sistema Operativo Android 27
4.6.1. Arquitectura 27
4.6.2. Android Development Tools (ADT) 28
4.6.3. Android API Level 29
5. Implementación de la solución 30
5.1. Diseño de la solución 30
5.1.1. Definición del metamodelo 30
5.1.2. Desarrollo de una herramienta gráfica para la creación de modelos 31
5.1.3. Transformación del modelo a código fuente Android 31
5.2. Definición del metamodelo 32
5.2.1. Creación del metamodelo 32
5.3. Desarrollo de una herramienta gráfica para la creación de modelos 41
5.3.1. Generación del modelo de dominio (Domain Gen Model) 41
5.3.2. Definición de elementos gráficos del modelo (Graphical Def Model) 42
5.3.3. Definición de la paleta de herramientas (Tooling Def Model) 43
5.3.4. Mapeo del modelo (Mapping Model) 44
5.3.5. Generación final del editor gráfico (Diagram Editor Gen Model) 46
5.3.6. Configuración del código del dominio 47
5.4. Transformación del modelo a código fuente Android 49
5.5. Generación de plugins para la importación a Eclipse IDE 55
6. Casos de estudio 57
6.1. Introducción 57
6.1.1. Consideraciones generales en el modelado de la aplicación 60
6.2. Aplicación “My Notes” 60
6.2.1. Intención 60
6.2.2. Mockup y modelo de la aplicación 60
6.2.3. Generación y customización de código 65
6.3. Aplicación “Shopping List” 69
6.3.1. Intención 69
6.3.2. Mockup y modelo de la aplicación 70
6.3.3. Generación y customización de código 76
6.4. Ejecución de los ejemplos 81
7. Conclusiones y Trabajos futuros 83

7.1. Conclusiones 83
7.2. Trabajos futuros 84
8. Referencias 86

Lista de Figuras

Figura 1 - Ciclo del desarrollo de software dirigido por modelos 8
Figura 2 - Transformación M2T: Modelo UML a Java 10
Figura 3 - Arquitectura de cuatro capas 13
Figura 4 - Arquitectura de Eclipse IDE 20
Figura 5 - Transformación PIM - PSM - Código fuente 25
Figura 6 - Workflow de creación de un diagrama gráfico 26
Figura 7 - Arquitectura de SO Android 28
Figura 8 - Componentes Screen y Layout 34
Figura 9 - Representación de la clase Widget 35
Figura 10 - Subclases de Widget 36
Figura 11 - Menu y MenuItem 36
Figura 12 - Actions y sus subclases 37
Figura 13 - Creación del archivo Ecore 38
Figura 14 - Configuración final del Ecore con el metamodelo 39
Figura 15 - Primer nivel del diagrama del metamodelo con el nodo Root como contenedor 39
Figura 16 - Derivación y creación del genmodel 41
Figura 17 - Generación del código del dominio 42
Figura 18 - Derivación y creación del gmfgraph 42
Figura 19 - Configuración del archivo gmfgraph 43
Figura 20 - Derivación y creación del gmftool 43
Figura 21 - Configuración del archivo gmftool 44
Figura 22 - Combinación para generar el gmfmap 45
Figura 23 - Mapeo gmfgraph con gmtool desde el Wizard del Dashboard 45
Figura 24 - Archivo final gmfmap 46
Figura 25 - Transformación del gmfmap a gmfgen 47
Figura 26 - Iconos del tooling para crear el modelo 48
Figura 27 - Generación del MainGenerator.mtl 50
Figura 28 - Ejemplo de template principal de Acceleo 50
Figura 29 - Estructura de proyecto Android 54
Figura 30 - Configuración inicial del proyecto Android 58
Figura 31 - Finalización de la configuración del proyecto Android 59
Figura 32 - Pantalla principal de “My Notes” 61

Figura 33 - Composición de una nota 61
Figura 34 - Pantalla de creación/edición de una nota 62
Figura 35 - Menú contextual invocado desde una nota 62
Figura 36 - Ventana de confirmación de eliminación de una nota 63
Figura 37 - Menú de opciones de la aplicación 63
Figura 38 - Modelo gráfico de “My Notes” 64
Figura 39 - Pantalla principal de “Shopping List” 70
Figura 40 - Menú de opciones de la pantalla principal 70
Figura 41 - Pantalla de creación/edición de listas de compras 71
Figura 42 - Menú contextual invocado a partir de una lista de compras 71
Figura 43 - Ventana de confirmación de eliminación de una lista 72
Figura 44 - Pantalla con productos pertenecientes a una lista 72
Figura 45 - Menú de opciones de la pantalla de productos 73
Figura 46 - Pantalla de creación/edición de un producto 73
Figura 47 - Composición de un producto 74
Figura 48 - Menú contextual invocado desde un producto 74
Figura 49 - Ventana de confirmación de eliminación de un producto 74
Figura 50 - Diagrama relacionado a las listas de compras 75
Figura 51 - Diagrama relacionado con los items 76

1. Introducción

1.1. Motivación

El Desarrollo de Software Dirigido por Modelos (Model Driven Development), de ahora en adelante llamado MDD, es un paradigma cuyo uso ha venido en creciente aumento durante los últimos años y el mismo pretende mejorar, estandarizar y automatizar la construcción del software basándose en un proceso guiado por modelos y soportado por potentes herramientas. Este paradigma asigna a los modelos (representación de un objeto en la vida real) un rol central y activo, desde los cuales se genera el código fuente (dependientes de una plataforma) a través de pasos de transformación y/o refinamiento. De esta forma, MDD permite reducir los costos de desarrollo de software, adaptarlo rápidamente a los cambios tecnológicos y a cambios en los requisitos, siempre manteniendo la consistencia entre los modelos y el código del software.

La Arquitectura Dirigida por Modelos (MDA, del inglés Model Driven Architecture), es una de las iniciativas más conocidas y extendidas dentro del ámbito de MDD. MDA es un concepto promovido por el OMG (Object Management Group) a partir de 2000, con el objetivo de afrontar los desafíos de integración de las aplicaciones y los continuos cambios tecnológicos. MDA es una arquitectura que proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos, siguiendo el proceso MDD. La idea principal de MDA es utilizar modelos como el núcleo del desarrollo y, de esta forma, separar los aspectos específicos de la plataforma del proceso de desarrollo de software.

Con el objetivo de aplicar MDD generando código fuente Android a partir de un modelo dado, se analizaron diferentes herramientas de las cuáles se esperaba que cumplan con una serie de requerimientos mínimos propuestos:

- que genere código Android,
- que sea software libre,
- que presente un metamodelo rico y/o expresivo y

- que la creación del modelo sea independiente del código fuente Android.

Actualmente existen algunas herramientas que permiten modelar una aplicación para una posterior generación de código Android, ellas son:

- App Inventor (versión 2)
    - Descripción: aplicación desarrollada por el MIT[1] para crear aplicaciones Android construyendo el código en forma de rompecabezas con piezas que se van uniendo. Posee un editor web Drag & Drop para ir enlazando estos bloques con el fin de crear la lógica.
    - Características: facilidad y simplicidad para crear aplicaciones Android. Es Freeware. Contra: sólo devuelve el código compilado (APK[2]) sin posibilidad de ver el fuente. Sólo es ejecutable en plataformas Windows.

- DroidBreeder (versión 02-04-2013)
    - Descripción: es una aplicación que sirve para generar ejemplos de diferentes funcionalidades de Android, por ejemplo: ejemplos de un connection manager, ejemplos con ListView, ejemplos de conexión con base de datos, etc.
    - Características: genera código fuente de los ejemplos predefinidos que provee, pero no permite modelar la aplicación. Es una plataforma de código libre. Contras: genera código Android para versiones de Android <= 2.1 (o Level 7) y sólo corre en plataforma Windows.

- DroidDraw (versión dd.r1.b22)
    - Descripción: es una de las aplicaciones más completas en cuanto a las plataformas en las que corre: Windows, Mac OS X, Linux. El objetivo primordial es el de crear interfaces gráficas para Android.

[1] El Instituto Tecnológico de Massachusetts es una universidad privada localizada en Cambridge, Massachusetts.
[2] Un archivo con extensión .APK es un paquete para el sistema operativo Android. Este formato es una variante del formato JAR de Java y se usa para distribuir e instalar componentes empaquetados para la plataforma Android para smartphones y tablets.

- Características: no permite generar código fuente. Como observación: permite generar algunos archivos XML. Posee un editor "Drag and Drop"[3] para agregar los elementos (o Widgets) en la pantalla e ir construyendo los Layouts. Permite generar APKs.

A partir del análisis de estas herramientas, se llegó a la determinación que ninguna de ellas se adapta a los objetivos mencionados, ya que la mayoría de ellas termina generando una aplicación compilada o siendo herramientas que presentan serias limitaciones a la hora de representar un modelo de aplicación. Por estos motivos, se decidió realizar la implementación de una herramienta que cumpla con los objetivos anteriormente mencionados.

1.2. Objetivos

El objetivo de la tesina constará en la realización de una herramienta de software que permita generar código fuente Android en forma automática a partir de un modelo definido gráficamente por el usuario. Para esto se trabajará en:

- La definición de un metamodelo. El mismo permitirá que la creación de los modelos sea lo más expresiva posible.
- Armado de una herramienta que permita construir el modelo gráficamente. Esto permitirá que un usuario pueda armar el modelo en base a elementos gráficos, por lo cual evitará tener que depender de un lenguaje técnico que desconoce.
- Desarrollo de una herramienta que permita crear un proyecto Android basado en el modelo previamente construido. De esta manera se tomará el modelo construido gráficamente como punto de partida para derivarlo en código fuente.
- Se proveerá de ejemplos de código fuente de prueba para poder comprobar la funcionalidad tanto en entornos virtuales como en dispositivos móviles o de cualquier índole. Como característica extra, el código fuente se generará con código documentado con el fin de brindar una ayuda a aquellos usuarios que quieran modificar el código generado.

[3] Drag and Drop (arrastrar y soltar) es una expresión informática que se refiere a la acción de mover con el ratón objetos de una ventana a otra o entre partes de una misma ventana.

2. Organización de la tesis

El desarrollo del presente informe se encuentra dividido en los siguientes capítulos:

2.1.1. Conceptos Básicos

En este capítulo se mencionan los conceptos básicos para una correcta comprensión acerca de los temas y conceptos que se van a desarrollar en los capítulos posteriores.

2.1.2. Descripción de tecnologías

En este capítulo se describen las tecnologías a utilizar en el desarrollo de la implementación. Es primordial que la descripción de las herramientas tecnológicas sea completa y esté acompañada de una correcta justificación.

2.1.3. Diseño de la solución

En este capítulo se describe el diseño de la solución propuesta. Se propone un marco teórico que aborda la solución del problema.

2.1.4. Implementación de la solución

En este capítulo se explica paso a paso cómo se implementa la solución. Se presentan imágenes descriptivas y todos los recursos necesarios para la correcta interpretación de la misma.

2.1.5. Caso de estudio

En este capítulo se proponen dos ejemplos, los cuales se proponen y se implementan haciendo uso de los plugins creados y de esta forma, se intenta demostrar la potencialidad de la implementación desarrollada.

2.1.6. Conclusiones y Trabajo futuro

Se concluye en base a la situación actual del desarrollo de software dirigido por modelos, el resultado de la implementación de la herramienta y se proponen los trabajos futuros.

3. Conceptos básicos

En este capítulo se describen los conceptos básicos para una comprensión posterior en el desarrollo de la implementación del código fuente.

3.1. MDD (Model Driven Development)

El Desarrollo de Software Dirigido por Modelos MDD (en inglés: Model Driven software Development) se ha convertido en un nuevo paradigma de desarrollo software. MDD promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. El adjetivo “dirigido” (driven) en MDD, a diferencia de “basado” (based), enfatiza que este paradigma asigna a los modelos un rol central y activo: son al menos tan importantes como el código fuente. Los modelos se van generando desde los más abstractos a los más concretos a través de pasos de transformación y/o refinamientos, hasta llegar al código aplicando una última transformación. La transformación entre modelos constituye el motor de MDD.

Los puntos claves de la iniciativa MDD son los siguientes:

1. El uso de un mayor nivel de abstracción en la especificación tanto del problema a resolver como de la solución correspondiente, en relación con los métodos tradicionales de desarrollo de software.
2. El aumento de confianza en la automatización asistida por computadora para soportar el análisis, el diseño y la ejecución.
3. El uso de estándares industriales como medio para facilitar las comunicaciones, la interacción entre diferentes aplicaciones y productos, y la especialización tecnológica.

3.1.1. Modelos

Un modelo es una representación conceptual o física a escala de un proceso o sistema, con el fin de analizar su naturaleza, desarrollar o comprobar hipótesis o supuestos y permitir una mejor comprensión del fenómeno real al cual el modelo representa y permitir así perfeccionar los diseños, antes de iniciar| la construcción de las obras u

objetos reales. Se utilizan con frecuencia para el estudio de represas, puentes, puertos, aeronaves, etc. Muchas veces, para obras complejas como por ejemplo una vivienda, se requiere la construcción de más de un modelo. Por ejemplo, un modelo general de la disposición de los ambientes de la vivienda con sus puertas y ventanas, un modelo específico a una escala mayor para la instalación eléctrica y sanitaria, uno diferente para mostrar la estética de la fachada y el entorno, etc.

Los modelos que nos interesan son aquellos relevantes para el desarrollo de software, sin embargo estos modelos no sólo representan al software; cuando un sistema de software soporta un determinado negocio, el modelo de dicho negocio es también relevante.

MDD contempla cuatro tipos de modelos. Algunos de ellos permiten describir a un sistema de manera independiente a los conceptos técnicos que involucra su implementación sobre una plataforma de software. Otros, en cambio, tienen como finalidad primaria describir tales conceptos técnicos. Teniendo en cuenta lo anterior, los tipos de modelos son:

- *CIM:* El modelo independiente de la computación (en inglés: Computation Independent Model). Un CIM es una vista del sistema desde un punto de vista independiente de la computación. Un CIM no muestra detalles de la estructura del sistema. Usualmente al CIM se lo llama modelo del dominio y en su construcción se utiliza un vocabulario que resulta familiar para los expertos en el dominio en cuestión. Se asume que los usuarios a quienes está destinado el CIM (los expertos de dominio) no poseen conocimientos técnicos acerca de los artefactos que se usarán para implementar el sistema. El CIM juega un papel muy importante en reducir la brecha entre los expertos en el dominio y sus requisitos por un lado, y los expertos en diseñar y construir artefactos de software por el otro.

- *PIM:* El modelo independiente de la plataforma (en inglés, Platform Independent Model). Un PIM es un modelo con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación. Dentro del PIM el sistema se modela desde el punto de vista de cómo se soporta mejor al negocio, sin tener en cuenta cómo va a ser implementado: ignora los sistemas

operativos, los lenguajes de programación, el hardware, la topología de red, etc. Por lo tanto un PIM puede luego ser implementado sobre diferentes plataformas específicas.

- *PSM:* El modelo específico de la plataforma (en inglés, Platform Specific Model). Como siguiente paso, un PIM se transforma en uno o más PSMs (Platform Specific Models). Cada PSM representa la proyección del PIM en una plataforma específica. Un PIM puede generar múltiples PSMs, cada uno para una tecnología en particular. Generalmente, los PSMs deben colaborar entre sí para una solución completa y consistente. Por ejemplo, un PSM para Java contiene términos como clase, interfaz, etc. Un PSM para una base de datos relacional contiene términos como tabla, columna, clave foránea, etc.

- *IM:* El modelo de la implementación (en inglés, Implementation Model) es el paso final en el desarrollo es la transformación de cada PSM a código fuente. Ya que el PSM está orientado al dominio tecnológico específico, esta transformación es directa.

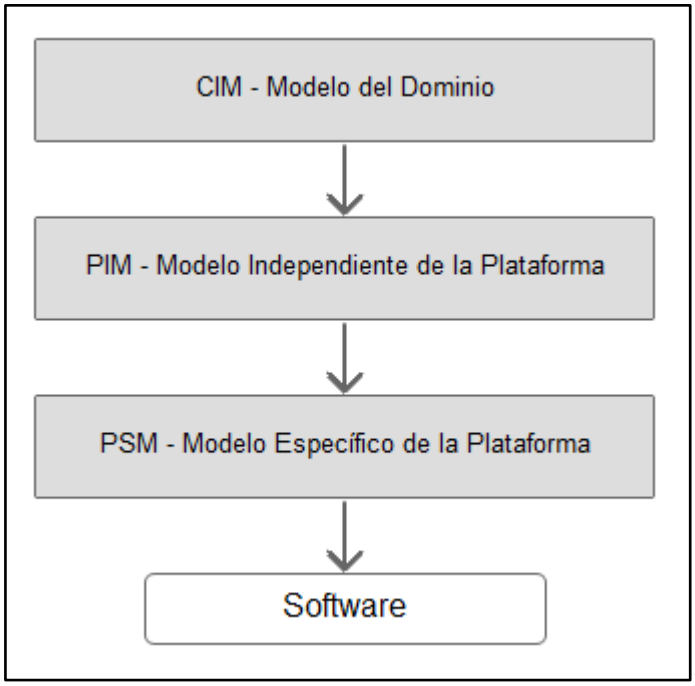

Figura 1 - Ciclo del desarrollo de software dirigido por modelos

3.2. MDA (Model Driven Architecture)

La arquitectura dirigida por modelos (Model-Driven, Architecture o MDA) es un acercamiento al diseño de software, propuesto y patrocinado por el *Object Management Group*. MDA se ha concebido para dar soporte a la ingeniería dirigida a modelos de los sistemas de software. MDA es una arquitectura que proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos.

Uno de los principales objetivos de MDA es separar el diseño de la arquitectura y de las tecnologías de construcción, facilitando que el diseño y la arquitectura puedan ser alterados independientemente. El diseño alberga los requerimientos funcionales (casos de uso) mientras que la arquitectura proporciona la infraestructura a través de la cual se hacen efectivos requerimientos no funcionales como la escalabilidad, fiabilidad o rendimiento. MDA se asegura de que el modelo independiente de la plataforma (PIM), el cual representa un diseño conceptual que concreta los requerimientos funcionales, sobrevive a los cambios que se produzcan en las tecnologías de fabricación y en las arquitecturas software. De particular importancia en MDA es la noción de transformación de modelos. Uno de los estándares definidos para la transformación de modelos se denomina QVT.

3.3. Metamodelos

Básicamente, un metamodelo (OMG, 2003) es un modelo que define el lenguaje formal para representar un modelo. En otras palabras, el metamodelo de un modelo describe qué elementos se pueden usar en el modelo y cómo pueden ser conectados. Por ejemplo, el lenguaje UML establece que dentro de un modelo se pueden usar los conceptos clase, atributo, asociación, entre otros.

Como un metamodelo es también un modelo, es necesario que exista un lenguaje para definir metamodelos de manera precisa. En esta instancia aparece el concepto de meta-metamodelo (OMG, 2003) y el mismo es un modelo que define el lenguaje formal para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y modelo.

3.4. Transformación

Una transformación consiste en una colección de reglas, las cuales son especificaciones no ambiguas de las formas en que un modelo, o parte de él, puede ser usado para crear otro modelo. Entonces, decimos que es una herramienta que toma un PIM como entrada y lo transforma en un PSM. La misma herramienta u otra, tomará el PSM y lo transformará a código fuente, por ejemplo. Estas transformaciones son esenciales en el proceso de desarrollo de MDD. Por lo general existen dos tipos de transformaciones:

- Modelo a modelo (M2M): cuyo principal objetivo es el de crear su modelo destino como una instancia de un metamodelo específico. Un ejemplo de esto es la transformación que existe de PIM a PSM.
- Modelo a texto (M2T): el origen es un modelo y su destino es un documento de texto. Un ejemplo de esto es la transformación que se lleva a cabo de PSM a Código Fuente.

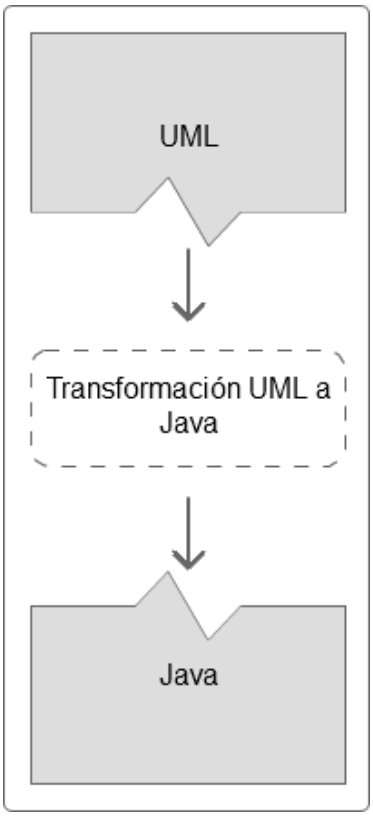

Figura 2 - Transformación M2T: Modelo UML a Java

3.4.1. OMG

El OMG (de sus siglas en inglés Grupo de Gestión de Objetos) es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI[4], etc. Es una organización sin ánimo de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas. El grupo está formado por diversas compañías y organizaciones con distintos privilegios dentro de la misma.

3.4.2. MOF y la arquitectura de cuatro capas

El objetivo es el de proveer de un sistema de tipos y una interfaz para que dichos tipos puedan ser creados y manipulados.

La especificación define un núcleo del modelo MOF que incluye un conjunto relativamente pequeño de construcciones para el modelado de información orientado a objetos. Luego, el modelo MOF puede ser extendido por herencia y composición para definir un modelo de información más rica que soporte construcciones adicionales. Alternativamente, el modelo MOF puede ser usado como un modelo para definir modelos de información. En este contexto, el modelo MOF se refiere como un meta-metamodelo porque es usado para definir metamodelos como por ejemplo el UML.

MOF está diseñado siguiendo una arquitectura de cuatro capas. Esta es una propuesta de la OMG para organizar y estandarizar los conceptos de modelado, desde los más abstractos a los más concretos. Por ejemplo, el metamodelo UML está definido en una de estas capas. La arquitectura de cuatro capas es un esquema comprobado que permite definir con precisión las semánticas requeridas por modelos complejos. Otras ventajas asociadas a esta propuesta son:

- Valida las construcciones del núcleo recursivamente verificando las sucesivas meta-capas.
- Provee las bases arquitecturales para definir futuras extensiones al metamodelo UML.

[4] XML Metadata Interchange es una especificación para el intercambio de diagramas. Especifica cómo representar un modelo.

- Suministra bases arquitecturales para alinear el metamodelo UML con otros estándares basados en la arquitectura de modelado de cuatro capas.

Las arquitectura de cuatro capas se divide en niveles: M3 (MOF), M2 (UML), M1 (User Model), y M0 (Instancias).

- *Meta-metamodelo:* La función principal de esta capa es definir el lenguaje para especificar un metamodelo. Un meta-metamodelo define un modelo en un nivel más alto de abstracción que un metamodelo, y es más compacto que el metamodelo que describe. Un meta-metamodelo puede definir múltiples metamodelos, y puede haber múltiples meta-metamodelos asociados con cada metamodelo.

  Dentro del OMG, MOF es el lenguaje estándar de esta capa. Ejemplos de meta-meta objetos en la capa de meta meta-modelado serían: meta-clase, meta-atributo y meta-operación.

- *Metamodelo:* Un metamodelo es una instancia de un meta-metamodelo. La responsabilidad primaria de esta capa es de definir un lenguaje para especificar modelos. Los metamodelos son típicamente más elaborados que los meta-metamodelos que los describen, especialmente cuando describen semánticas dinámicas. Ejemplos de meta objetos en la capa de meta-modelado son: clase, atributo, operación y componente.

- *Modelo:* Un modelo es una instancia de un metamodelo. La función principal de la capa de modelo es de definir un lenguaje que describa el dominio de la información. Ejemplos de objetos en esta capa son: Auto (Clase), Modelo (Atributo), arrancar (Operación).

- *Objetos de usuario:* Los objetos de usuario son las instancias de un modelo. La responsabilidad principal de la capa de objetos de usuario es de describir un dominio de información específico. Se utiliza para describir objetos del mundo real.

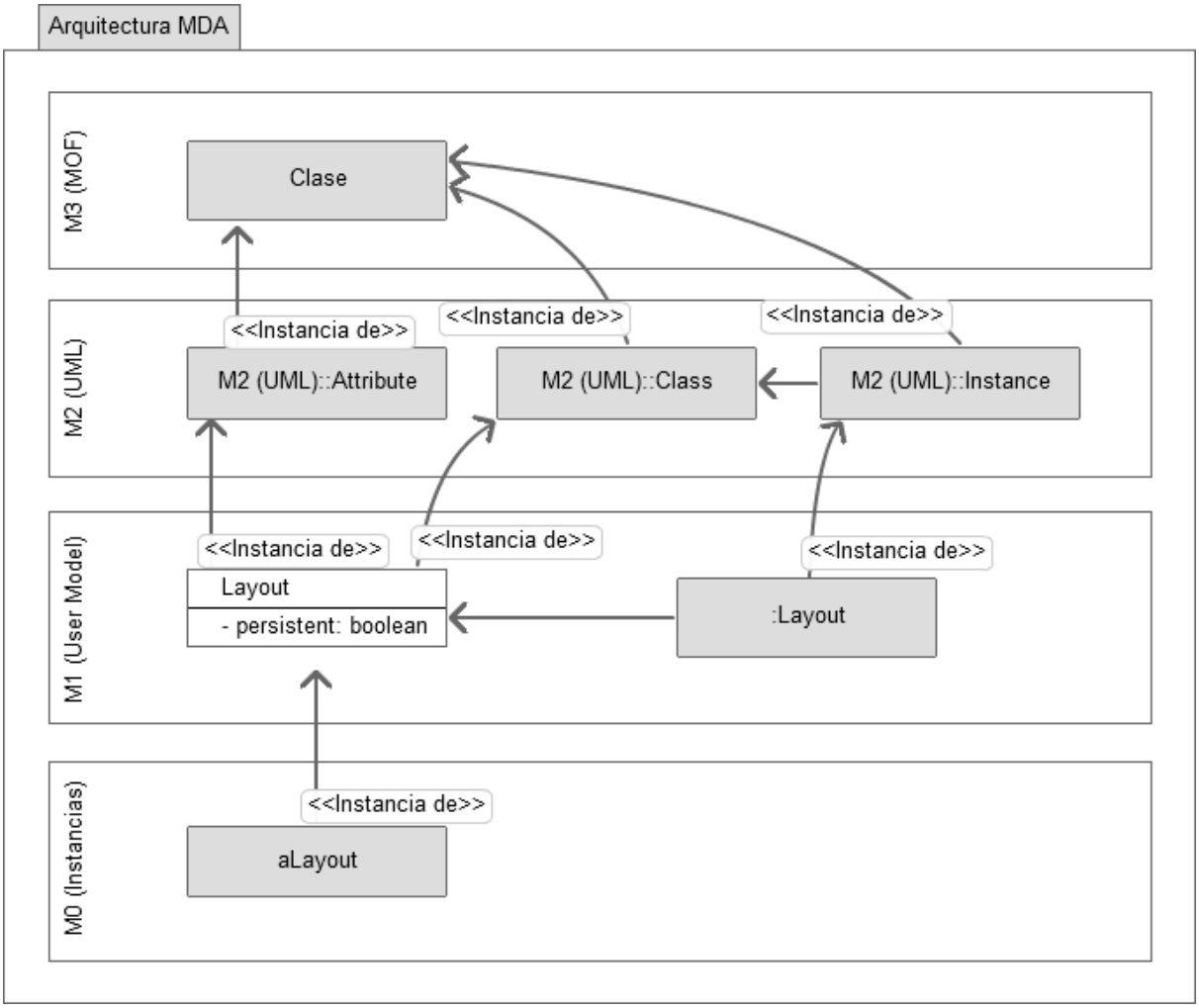

Figura 3 - Arquitectura de cuatro capas

3.4.3. MOFM2T

MOFM2T o Lenguaje de transformación de modelo a texto basado en MOF (Meta Object Facility) y es una especificación de la OMG para la transformación de modelos a lenguajes de programación. Puede ser usado para expresar transformaciones que transforman un modelo a texto (M2T). Este re usa algunos conceptos de MOF.

3.5. UML

Lenguaje Unificado de Modelado (UML o Unified Modeling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como

procesos de negocio, funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y compuestos reciclados.

Es importante remarcar que UML es un "lenguaje de modelado" para especificar o para describir métodos o procesos. Se utiliza para definir un sistema, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo.

Se puede aplicar en el desarrollo de software gran variedad de formas para dar soporte a una metodología de desarrollo de software (tal como el Proceso Unificado Racional o RUP), pero no específica en sí mismo qué metodología o proceso usar.

UML no puede compararse con la programación estructurada, pues UML significa Lenguaje Unificado de Modelado, no es programación, solo se diagrama la realidad de una utilización en un requerimiento. Mientras que, programación estructurada, es una forma de programar como lo es la orientación a objetos, la programación orientada a objetos viene siendo un complemento perfecto de UML, pero no por eso se toma UML sólo para lenguajes orientados a objetos.

UML cuenta con varios tipos de diagramas, los cuales muestran diferentes aspectos de las entidades representadas:

- Diagramas de estructura,
- diagramas de comportamiento y
- diagramas de interacción.

3.5.1. Diagramas de estructura

Enfatizan los elementos que deben existir en el sistema modelado. Dentro de este tipo se encuentran:

- Diagrama de clases: describe la estructura de un sistema mostrando las clases del mismo, sus atributos y las relaciones entre clases.
- Diagrama de componentes: describe cómo un sistema de software es dividido en componentes y muestra las dependencias entre dichos componentes.

- Diagrama de objetos: muestra una vista completa o parcial de la estructura del sistema modelado en un momento específico.
- Diagrama de estructura compuesta: describe la estructura interna de una clase y las colaboraciones que su estructura permite.
- Diagrama de despliegue: describe el modelo de hardware utilizado en la implementación del sistema, los entornos de ejecución y los artefactos desplegados sobre el hardware.
- Diagrama de paquetes: muestra como un sistema es dividido en grupos lógicos, especificando las dependencias entre estos grupos.

3.5.2. Diagramas de comportamiento

Describen lo que debe suceder en el sistema modelado. Dentro de esta categoría se ubican:

- Diagrama de actividades: representa los flujos de trabajo, de negocios, operacionales y de control entre los componentes del sistema.
- Diagrama de casos de uso: muestra la funcionalidad provista por un sistema en términos de actores y sus objetivos representados como casos de usos. Además especifica las dependencias entre dichos casos de uso.
- Diagrama de estados: muestran los posibles estados de un componente del sistema y cómo los estímulos externos provocan los cambios de estado.

3.5.3. Diagramas de interacción

Son un subtipo de diagramas de comportamiento que enfatiza el flujo de control y de datos entre los elementos del sistema modelado. Dentro de esta categoría se encuentran:

- Diagrama de secuencia: muestra cómo los objetos se comunican entre sí a través del envío de mensajes. También indica el tiempo de vida de cada objeto a partir de los mensajes que envía y recibe.
- Diagrama de comunicación: es una versión simplificada del diagrama de colaboración. Muestra las interacciones entre los objetos o partes en términos de secuencias de mensajes. Representa una combinación de información tomada

desde los diagramas de clases, secuencia y casos de usos. Describe tanto las estructuras estáticas como el comportamiento dinámico de un sistema.

- Diagrama de tiempos: son un tipo específico de diagramas de interacción donde el foco está puesto en las restricciones temporales

4. Descripción de tecnologías

En el presente capítulo se hace mención de las tecnologías utilizadas. Básicamente se describe lo que es el Eclipse IDE con sus plugins, para finalizar describiendo a la plataforma Android OS.

Es importante remarcar y justificar la elección de las herramientas tecnológicas antes de comenzar a describirlas. De esta forma, se opta por Eclipse IDE por los siguientes motivos:

- El entorno de desarrollo integrado (IDE) de Eclipse emplea plugins para proporcionar toda su funcionalidad, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no.
- Este mecanismo de plugins es una plataforma ligera para componentes de software. Adicionalmente al permitirle a Eclipse extenderse usando otros lenguajes de programación como son C/C++ y Python, permite a Eclipse IDE trabajar con lenguajes para procesado de texto como LaTeX, etc.
- La arquitectura plugin permite escribir cualquier extensión deseada en el ambiente. Se provee soporte para Java y CVS en el SDK de Eclipse. Y no tiene porqué ser usado únicamente para soportar otros Lenguajes de programación.
- La definición de un el proyecto Eclipse IDE es: "una especie de herramienta universal, un IDE abierto y extensible para todo y nada en particular".
- Los autores se encuentran muy familiarizados con el IDE, por el hecho de ser la herramienta frecuente como entorno de desarrollo de programación.

La elección de la plataforma Android está dada por los siguientes puntos:

- Diversidad de dispositivos. Android como sistema operativo es utilizado por una gran cantidad de fabricantes y por lo tanto contamos con amplias posibilidades de elección. Además, cada una de estas marcas cuenta con dispositivos de diferente gama en prestaciones y coste.

- Es libre. Android es un Sistema Operativo libre y de código abierto[5], existe una amplia comunidad donde la modificación, personalización y mejoras de las ROMs[6] originales brindan muchas posibilidades de mejora y actualización, contando con versiones alternativas a los Sistemas Oficiales.
- Inmerso en el mundo de Google Play Store[7]. A día de hoy se cuenta con más de 1.300.000 aplicaciones disponibles. Además, Google mantiene una mentalidad abierta (SDK libre) a las aplicaciones expuestas en su tienda y por tanto el crecimiento en aplicaciones es cada día mayor.
- Abanico amplio de dispositivos. Android, ha conseguido extenderse a multitud de dispositivos, móviles, netbook, tablets, navegadores, relojes, etc. Esto hace que Android sea al día de hoy un sistema multifunción, donde los fabricantes cuentan con mayor posibilidad de integración tecnológica.

4.1. Eclipse IDE

Eclipse IDE es un entorno de desarrollo integrado, de código abierto, desarrollado en Java y multiplataforma para desarrollar aplicaciones. Con la ayuda de los denominados "plugins" se pueden desarrollar aplicaciones en diferentes lenguajes: C, PHP, Python, Ruby, Scala, etc.

Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas. Un ejemplo es el recientemente creado Eclipse Modeling Project, cubriendo la mayoría de las áreas de Model Driven Engineering.

Eclipse IDE fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Este IDE es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

---

[5] Código abierto es la expresión con la que se conoce al software distribuido y desarrollado libremente. Se focaliza más en los beneficios prácticos (acceso al código fuente) que en cuestiones éticas o de libertad que tanto se destacan en el software libre.
[6] ROM es un archivo que contiene en este caso, un Sistema Operativo. Puede ser Android, etc.
[7] Google Play Store es una plataforma de distribución digital de aplicaciones móviles para los dispositivos con sistema operativo Android, así como una tienda en línea desarrollada y operada por Google.

Dispone de un Editor de texto con resaltado de sintaxis. La compilación es en tiempo real. Tiene pruebas unitarias con JUnit, asistentes o Wizard[8] para creación de proyectos, clases, tests, etc., y refactorización.

Asimismo, a través de plugins libremente disponibles es posible añadir control de versiones[9] con Subversion e integración con Hibernate[10], entre otros.

4.1.1. Arquitectura de Eclipse IDE

La arquitectura de Eclipse IDE se compone de los siguientes componentes:

- Platform Runtime (también llamado Kernel[11]) y la arquitectura del plugin: Su tarea es determinar cuáles son los plugins disponibles en el directorio de plugins de Eclipse IDE. Cada plugin tiene un fichero XML manifest[12] que lista los elementos que necesita de otros plugins así como los puntos de extensión que ofrece. Como la cantidad de plugins puede ser muy grande, sólo se cargan los necesarios en el momento de ser utilizados con el objeto de minimizar el tiempo de arranque de Eclipse IDE y recursos.
- Workspaces o Espacios de trabajo: Maneja los recursos del usuario, organizados en uno o más proyectos. Cada proyecto corresponde a un directorio en el directorio de trabajo de Eclipse IDE, y contienen archivos y carpetas.
- Workbench & UI Toolkit o Área de desarrollo: La interfaz de usuario de Eclipse IDE está construido alrededor de un espacio de trabajo que proporciona la estructura general y presenta una interfaz de usuario ampliable para el usuario.
- UI Integration o Interfaz de usuario: Muestra los menús y herramientas, y se organiza en perspectivas que configuran los editores de código y las vistas. A diferencia de muchas aplicaciones escritas en Java, Eclipse IDE tiene el aspecto

---

[8] Wizard es un asistente o guía paso a paso que ayuda a un usuario a realizar una tarea específica.
[9] Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo.
[10] Herramienta de Mapeo objeto-relacional (ORM) para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación.
[11] Kernel es un software que constituye una parte fundamental del sistema operativo y se define como la parte que se ejecuta en modo privilegiado. Es el principal responsable de facilitar a los distintos programas acceso seguro al hardware de la computadora o en forma básica, es el encargado de gestionar recursos, a través de servicios de llamada al sistema.
[12] En la plataforma Java un archivo manifest es un archivo específico contenido en un archivo Jar. Se usa para definir datos relativos a la extensión y al paquete.

y se comporta como una aplicación nativa. No está programada en Swing, sino en SWT (Standard Widget Toolkit) y Jface (juego de herramientas construida sobre SWT), que emula los gráficos nativos de cada sistema operativo. Este ha sido un aspecto discutido sobre Eclipse IDE, porque SWT debe ser portada a cada sistema operativo para interactuar con el sistema gráfico. En los proyectos de Java puede usarse AWT y Swing salvo cuando se desarrolle un plugin para Eclipse IDE.

- Team Support o Ayuda al equipo: Este plugin facilita el uso de un sistema de control de versiones para manejar los recursos en un proyecto del usuario y define el proceso necesario para guardar y recuperar de un repositorio.
- Help o Documentación: Al igual que el propio Eclipse IDE, el componente de ayuda es un sistema de documentación extensible. Los proveedores de herramientas pueden añadir documentación en formato HTML y, usando XML, definir una estructura de navegación.

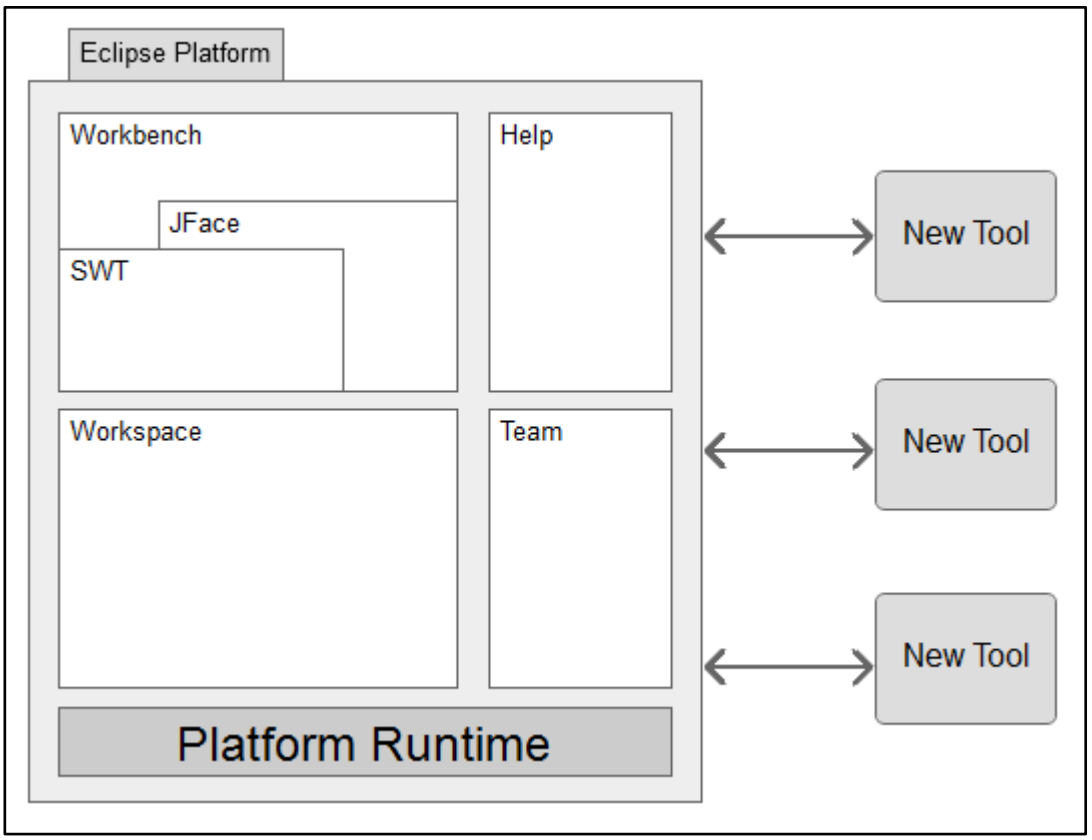

Figura 4 - Arquitectura de Eclipse IDE

4.1.2. El componente plugin y su arquitectura

Un plugin es la mínima unidad de la Plataforma Eclipse IDE que puede ser desarrollada y distribuida separadamente. Usualmente una pequeña utilidad se escribe como un simple plugin, mientras que una utilidad compleja tiene su funcionalidad repartida entre varios plugins. Salvo el Kernel conocido como Plataforma Runtime, toda la funcionalidad de la Plataforma Eclipse IDE está realizada con plugins.

Estos se desarrollan en código Java que se guarda en una librería JAR, junto con algunos recursos como imágenes, páginas HTML, etc. e información de sólo lectura. Estas librerías de plugins junto a la información de solo lectura son guardadas en un directorio del sistema de archivos o en una URL dentro de un servidor. Este es un mecanismo que permite que los plugins puedan ser sintetizados por muchos fragmentos separados, cada uno en su propio directorio o URL. Cada plugin tiene un archivo denominado manifest que declara las dependencias a otros plugins. El modelo de dependencias es simple: un plugin declara un número de puntos de extensiones, y un número de extensiones a uno o más puntos de extensión en otros plugins.

4.2. EMF

El proyecto EMF es un framework[13] para modelado, que permite la generación automática de código para construir herramientas y otras aplicaciones a partir de modelos de datos estructurados. Se usa, por ejemplo, para implementar XML Schema Infoset Modelo (XSD), Servicio de Data Objects (SDO), UML2, y Web Tools Platform (WTP) para los proyectos Eclipse IDE. Además EMF se utiliza en productos comerciales, como Omondo, EclipseUML, IBM Rational y productos WebSphere.

EMF permite usar un modelo como el punto de partida para la generación de código, e iterativamente refinar el modelo y regenerar el código, hasta obtener el código requerido. Aunque también prevé la posibilidad de que el programador necesite modificar ese código, es decir, se contempla la posibilidad de que el usuario edite las clases generadas, para agregar o editar métodos y variables de instancia. Siempre se puede

[13] Un framework es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software.

regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas durante la regeneración.

El código generado incluye clases Java para manipular instancias de ese modelo como así también clases adaptadoras para visualizar y editar las propiedades de las instancias desde la vista "propiedades" de Eclipse IDE. Además se provee un editor básico en forma de árbol para crear instancias del modelo. Y por último, incluye un conjunto de casos de prueba para permitir verificar propiedades.

4.2.1. Meta metamodelo Ecore

En EMF los modelos se especifican usando un meta-metamodelo llamado Ecore. Este el corazón de EMF. De esta manera, para generar código a partir de la definición de un modelo, deben seguirse los siguientes pasos:

Definición del metamodelo: desde un documento XML o diagrama de clases UML se interpreta mediante EMF y se genera una instancia del meta metamodelo Ecore. La instancia creada se guardará en un archivo XMI dentro del espacio de trabajo.

Generación del código: EMF nos brinda la posibilidad de, una vez obtenido el XMI, generar el código java. Este se generará en forma de plugin y existirán cuatro tipos:

- Model Code: código java de la implementación del modelo, por ejemplo, crear instancias, etc.
- Edit Code: permitirá editar propiedades de las instancias a través de una forma visual en Eclipse.
- Editor Code: genera un editor para el modelo.
- Test code: casos de pruebas para verificar propiedades de los elementos.

4.2.2. OCL

El plugin OCL permite la definición de restricciones sobre el modelo Ecore para luego evaluarlas y determinar si el modelo está correctamente formado.

OCL es un lenguaje libre de efectos laterales. Esto significa que ninguna expresión OCL puede modificar elementos del modelo.

OCL se usa frecuentemente para especificar pre y post condiciones para las operaciones. Una tercera clase de expresiones definidas sobre las operaciones es la expresión body, la cual define el valor de la operación en términos de sus parámetros y las propiedades disponibles dentro del contexto.

Una vez más, al generar el código, se crea el método definido en la clase el cual tiene definida una implementación. El contexto de la expresión OCL es una operación, lo cual asegura que los nombres y tipos de los parámetros son visibles.

4.3. GEF

GEF permite a los desarrolladores mapear rápidamente cualquier modelo existente con un ambiente de edición gráfico. El ambiente gráfico es el SWT basado en el plugin de dibujo Draw2D (el cual es parte del componente GEF). El desarrollador puede sacar ventaja de muchas operaciones comunes previstas en GEF y/o extendiéndolas para un dominio específico.

GEF es apropiado para crear gran variedad de aplicaciones, incluyendo:

- Constructores GUI (o interfaz gráfica de usuario).
- Editores de diagramas UML (como por ejemplo workflow y diagramas de modelado de clases).
- Editores de texto WYSIWYG[14] como HTML.

GEF no asume que se tiene que construir alguna de estas aplicaciones por lo que el dominio de la aplicación es neutro dando una gran flexibilidad en la construcción de aplicaciones.

[14] WYSIWYG es el acrónimo de “What You See Is What You Get” (en español, "lo que ves es lo que obtienes"). Se aplica a los procesadores de texto y otros editores de texto con formato (como los editores de HTML) que permiten escribir un documento viendo directamente el resultado final.

4.4. GMF

GMF es un framework de código abierto que permite construir editores gráficos, también desarrollado para el entorno Eclipse IDE. Está basado en los plugins EMF y GEF. Algunos ejemplos de editores generados con GMF son los editores UML, de Ecore, de procesos de negocio y de flujo.

Los editores gráficos generados con GMF están completamente integrados a Eclipse y comparten las mismas características con otros editores, como vista overview, la posibilidad de exportar el diagrama como imagen, cambiar el color y la fuente de los elementos del diagrama, hacer zoom animado del diagrama, imprimirlo.

El primer paso es la definición del metamodelo del dominio (archivo con extensión Ecore), donde se define el metamodelo en términos del meta-metamodelo Ecore. A partir de allí, se derivan otros modelos necesarios para la generación del editor: el modelo de definición gráfica específica las figuras que pueden ser dibujadas en el editor y el modelo de definición de tooling contiene información de los elementos en la paleta del editor, los menús, etc. Una vez definidos estos modelos, habrá que combinar sus elementos.

Esto se hace a través de un modelo que los relaciona, es decir, relaciona los elementos del modelo de dominio con representaciones del modelo gráfico y elementos del modelo de tooling. Estas relaciones definen el modelo de mapping (archivo con extensión *gmfmap*).

Por último, una vez definidos todos los modelos, y relacionados a través del archivo mapping, GMF provee un generador de modelo que permite definir los detalles de implementación anteriores a la fase de generación de código (archivo con extensión *gmfgen*). La generación de código producirá un plugin editor que interrelaciona la notación con el modelo de dominio. También provee persistencia y sincronización de ambos.

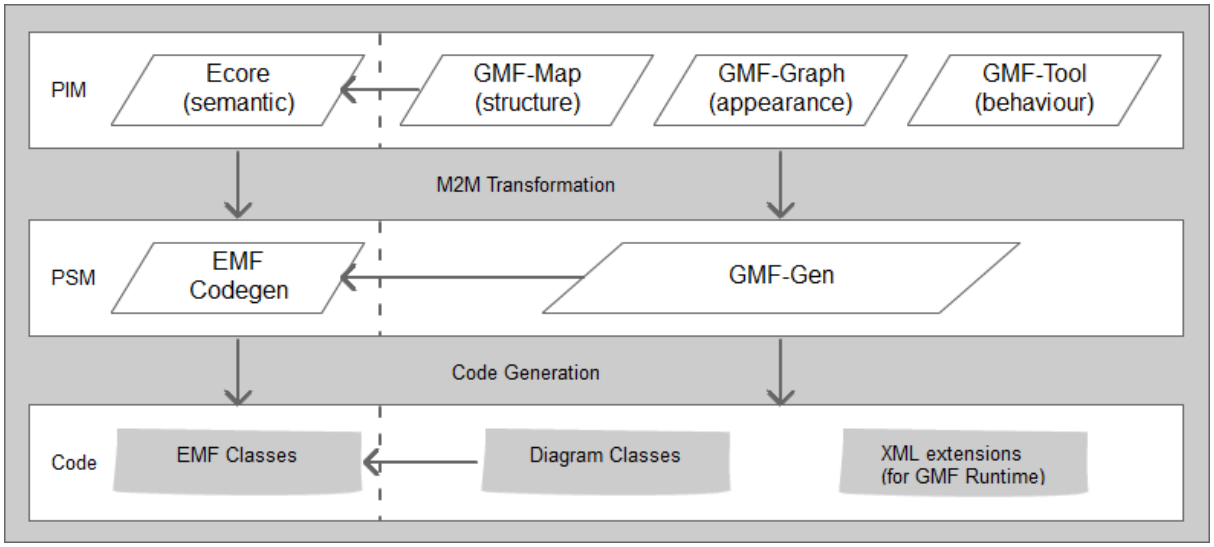

Figura 5 - Transformación PIM - PSM - Código fuente

4.4.1. Definición del editor gráfico

- Modelo de Dominio (Domain Model): se usa el editor para especificar el metamodelo que se quiere instanciar para luego generar el código necesario para manipularlo usando EMF. Se pueden utilizar diferentes metamodelos: Ecore, UML, etc.
- Generación del Modelo de Dominio (Domain Gen Model): se usa para generar el código del modelo de dominio con EMF. Se usa la extensión genmodel.
- Modelo de definición gráfica (Graphical Def Model): se usa para definir nodos o elementos que se mostrarán en el diagrama.
- Modelo de definición de herramientas (Tooling Def Model): se especifica la paleta con la cual se podrá armar el crear el diagrama. Se eligen las acciones que se desencadenan, etc.
- Modelo de Mapeo (Mapping Model): se utiliza para relacionar la definición gráfica con la definición de herramientas.
- Generación de código para el editor gráfico (Diagram Editor Gen Model): se usa para generar el editor gráfico GMF en base al código EMF generado por el archivo *genmodel*.

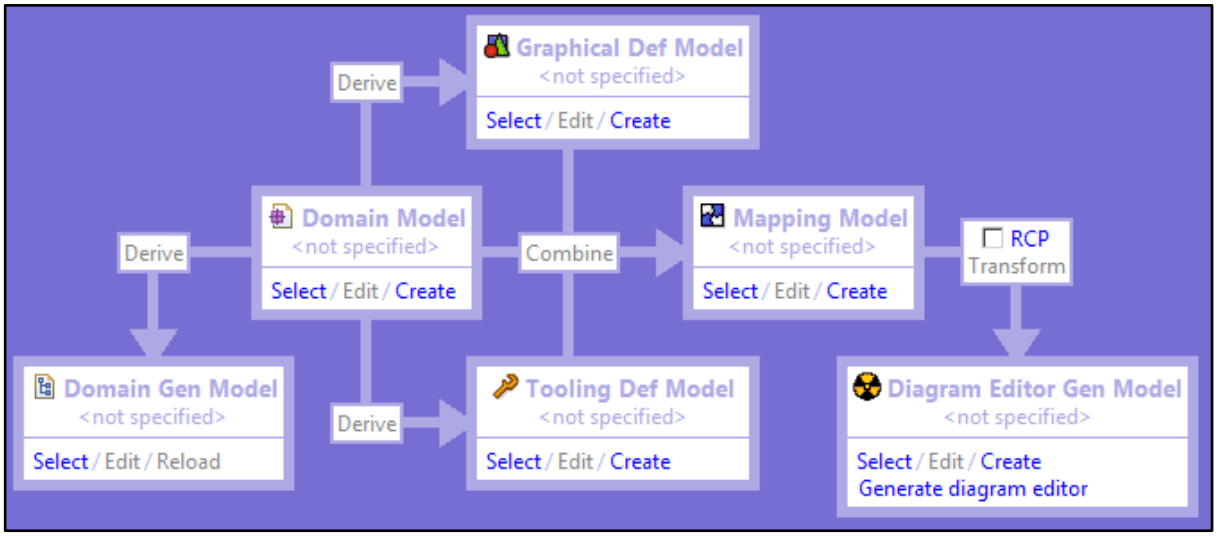

Figura 6 - Workflow de creación de un diagrama gráfico

4.5. Acceleo

Acceleo es una herramienta de código abierto que permite generar código fuente a partir de un modelo determinado. Es una implementación de MOFM2T (perteneciente a OMG) para realizar transformaciones modelo-a-texto (M2T).

Este lenguaje de generación de código utiliza una aproximación basada en “templates”. Con este enfoque, un template está compuesto por código que será procesado en base a elementos de entrada. La mayoría de estas partes son expresiones las cuales son usadas para seleccionar y extraer información del modelo. En Eclipse IDE dichas expresiones están basadas en la implementación del lenguaje OCL.

4.5.1. Características

Acceleo provee de herramientas para la generación de código basado en modelos EMF. Gracias a estas herramientas, por ejemplo, Acceleo provee la generación incremental. Esto brinda la posibilidad de generar código desde un modelo, modificar el código generado y luego volver a generar el código del modelo, sin haber perdido los cambios del código previamente modificado. También permite:

- Generar código desde cualquier metamodelo compatible con EMF como UML1, UML2 y DSL[15].
- Personalización de la generación de código con templates definidos por el usuario.
- Generación de código en diferentes lenguajes: C, Java, Python, etc.

4.6. Sistema Operativo Android

Android es un sistema operativo basado en el Kernel de Linux, diseñado principalmente para dispositivos móviles con pantalla táctil, como teléfonos inteligentes o tabletas, y también para relojes inteligentes, televisores y automóviles, etc.

4.6.1. Arquitectura

- Aplicaciones: Las aplicaciones base incluyen un cliente de correo electrónico, programa de SMS, calendario, mapas, navegador, contactos y otros. Todas las aplicaciones están escritas en lenguaje de programación Java.
- Marco de trabajo de aplicaciones (Applications Framework): Los desarrolladores tienen acceso completo a los mismos APIs del framework usados por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes; cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación puede luego hacer uso de esas capacidades (sujeto a reglas de seguridad del framework). Este mismo mecanismo permite que los componentes sean reemplazados por el usuario.
- Bibliotecas: Android incluye un conjunto de bibliotecas de C/C++ usadas por varios componentes del sistema. Estas características se exponen a los desarrolladores a través del marco de trabajo de aplicaciones de Android; algunas son: System C library (implementación biblioteca C estándar), bibliotecas de medios, bibliotecas de gráficos, 3D y SQLite[16], entre otras.
- Runtime de Android: Android incluye un set de bibliotecas base que proporcionan la mayor parte de las funciones disponibles en las bibliotecas base del lenguaje Java. Cada aplicación Android corre su propio proceso, con su propia instancia de

[15] DSL o Lenguaje específico del dominio determina un lenguaje específico para crear modelos para un dominio determinado.
[16] SQLite es un sistema de gestión de bases de datos relacional compatible con ACID, contenida en una relativamente pequeña (275 KB) biblioteca escrita en C.

la máquina virtual Dalvik. Esta máquina virtual denominada Dalvik ha sido escrito de forma que un dispositivo puede correr múltiples máquinas virtuales de forma eficiente. Dalvik ejecuta archivos en el formato Dalvik Executable, el cual está optimizado para memoria mínima.

- Núcleo Linux: Android depende de Linux para los servicios base del sistema como seguridad, gestión de memoria, gestión de procesos, pila de red y modelo de controladores. El núcleo también actúa como una capa de abstracción entre el hardware y el resto de la pila de software

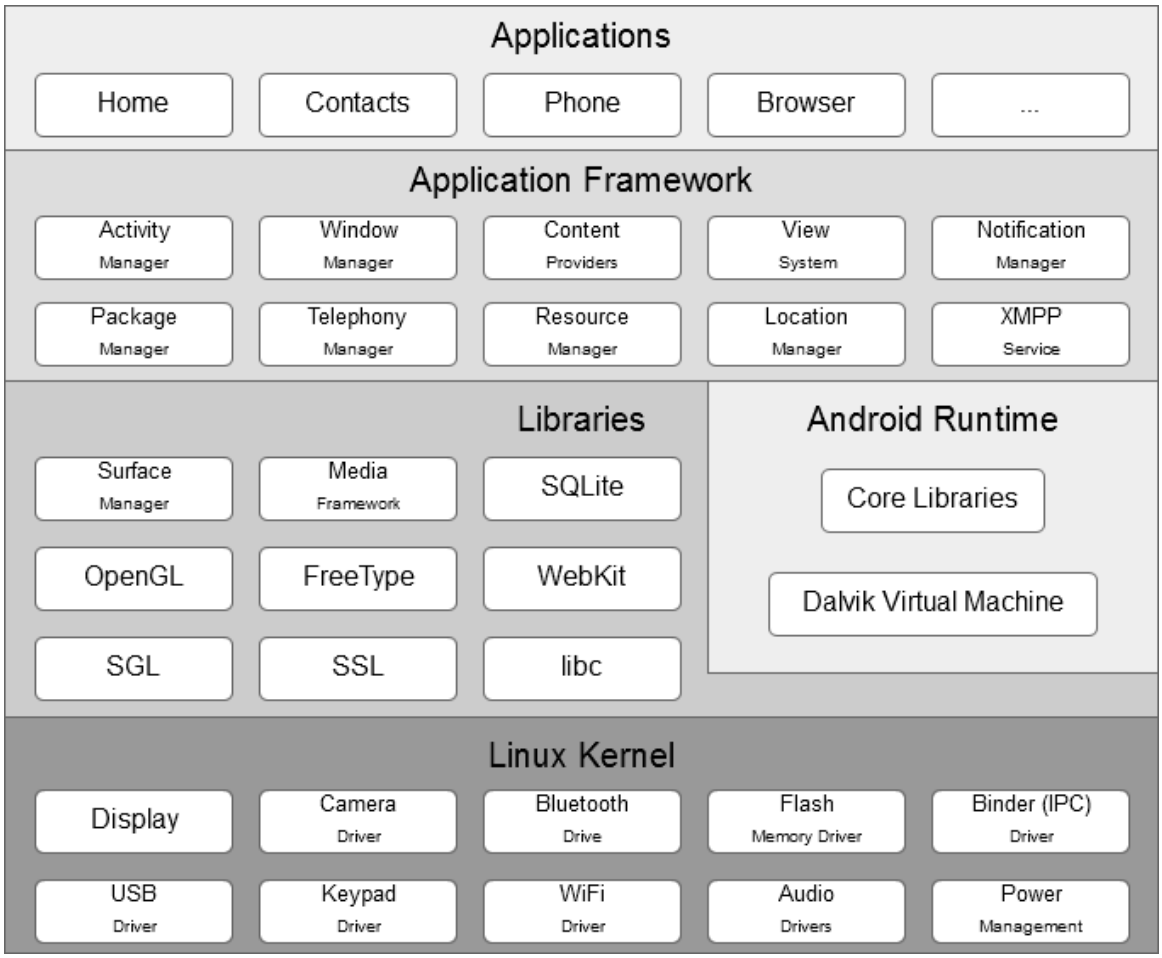

Figura 7 - Arquitectura de SO Android

4.6.2. Android Development Tools (ADT)

ADT es un plugin para desarrollar aplicaciones Android en Eclipse IDE. El mismo provee de las siguientes herramientas:

- Eclipse IDE y el plugin ADT

- Android SDK Tools: es el kit de desarrollo de software. Con él podremos desarrollar aplicaciones y ejecutar un emulador del sistema Android en la versión que se necesite. Todas las aplicaciones Android se desarrollan en lenguaje Java.
- Una versión de la plataforma Android y una imagen completa del SO Android para el emulador: de esta forma se podrán ejecutar aplicaciones sin necesidad de hacerlo sobre otro dispositivo.

4.6.3. Android API Level

API[17] Level (Nivel de API) es un valor entero que identifica de forma exclusiva una revisión de la API que ofrece una versión de la plataforma Android.

La plataforma Android proporciona una API que las aplicaciones pueden utilizar para interactuar con el sistema Android subyacente. La API se compone de:

- Un conjunto básico de paquetes y clases.
- Un conjunto de elementos y atributos XML para declarar el archivo de manifiesto.
- Un conjunto de elementos y atributos XML para declarar y el acceder a los recursos.
- Un conjunto de permisos que las aplicaciones pueden solicitar, así como refuerzos de permisos incluidos en el sistema.

Cada versión nueva del SO Android puede incluir actualizaciones a nuevas versiones de la API. A continuación se mencionan las versiones de Android más destacadas, junto con su versión y API Level:

Versión	API Level	Código o Nombre
Android 1.6	4	Donut
Android 2.3.2	9	GingerBread
Android 3.0	11	HoneyComb
Android 4.0	14	Ice Cream Sandwich
Android 4.1	16	Jelly Bean
Android 4.4	19	KitKat

[17] Es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

5. Implementación de la solución

En el presente capítulo se describe el diseño de una solución que permitirá generar código fuente Android a partir de modelos generados gráficamente en base a un metamodelo definido. Luego, se detalla la implementación concreta de los módulos que la componen. Como resultado se irán obteniendo un conjunto de plugins de Eclipse IDE que juntos brindan a los usuarios una herramienta visual para crear modelos y derivar código fuente Android.

5.1. Diseño de la solución

El diseño de la solución se dividió en tres etapas, cada una de ellas con un propósito diferente:

- La definición del metamodelo,
- el desarrollo de una herramienta gráfica para la creación de modelos; y
- la transformación del modelo a código fuente Android.

5.1.1. Definición del metamodelo

Como resultado de esta etapa se espera obtener un metamodelo que permitirá crear modelos que representen aplicaciones y sus distintos elementos; es por este motivo que lo llamaremos “Application”. El mismo se construirá sobre un conjunto de premisas que se describen a continuación.

- Simplicidad: que haya un conjunto reducido de elementos pero que sean representativos.
- Expresividad: que se puedan componer sus elementos para representar una amplia variedad de aplicaciones diferentes.
- Independencia de Android: que sirva para modelar cualquier aplicación, no solamente una Android. A pesar que este trabajo está basado en la generación de código fuente Android, se quiere un metamodelo que permita representar aplicaciones en general, lo más independiente de la plataforma posible.

5.1.2. Desarrollo de una herramienta gráfica para la creación de modelos

En esta etapa se pretende desarrollar una herramienta gráfica para simplificar la creación de modelos cuyo metamodelo sea el definido en el punto anterior. La herramienta gráfica deberá respetar los siguientes principios:

- Debe ser intuitiva, para esto se construirá una pantalla basada en dos partes:

    - Una parte donde se diseñe o grafique el modelo.
    - Otra parte donde se brinden las herramientas crear el modelo.

    De esta manera el usuario tendrá a la izquierda de la pantalla un editor y en la derecha la paleta de herramientas, la cual proveerá de los elementos necesarios, para arrastrar, soltar y armar el modelo. Dicho modelo se armará uniendo los elementos, unos con los otros, y también el mismo proveerá de validaciones automáticas que permitirán o no realizar conexiones entre los elementos.

- Debe ser amigable: la herramienta debe ser de fácil uso. Este concepto está estrechamente asociado al concepto anterior. Para esto, tanto en la paleta de herramientas como en el editor, los elementos se podrán identificar con íconos representativos. Estos íconos representan en forma gráfica el elemento que se está manipulando. De este modo, con el solo hecho de visualizar un elemento el reconocimiento visual será sencillo, independientemente del nombre que le haya dado al mismo.

5.1.3. Transformación del modelo a código fuente Android

Esta etapa tiene como objetivo tomar los modelos creados según el metamodelo del primer punto (creados con la herramienta desarrollada en el paso anterior o manualmente) y generar código fuente Android.

Para este punto se pretende que el código Android generado cumpla con las prácticas de programación propuestas por Android y que además, se cuente con la

documentación en código necesaria, para saber para qué versiones o API Level de Android estará destinado.

5.2. Definición del metamodelo

En el siguiente apartado se definirá el metamodelo correspondiente y como resultado obtendremos la implementación del mismo utilizando la herramienta de meta-modelado Ecore. Es importante remarcar que este punto de la implementación es realizado una única vez y se incluye como parte del desarrollo de esta tesis. Para futuras extensiones de la tesis, según el contexto en el cual se aplique, será necesario modificar tanto este paso, como los siguientes, ya que una modificación en este nivel requiere que se tenga que volver a regenerar los siguientes niveles.

5.2.1. Creación del metamodelo

Para crear nuestro metamodelo utilizamos el framework de modelado EMF, que viene incluido con el plugin GMF. En una primera instancia se crea un proyecto Java el cual contendrá el metamodelo a diseñar. Luego, desde el menú contextual se dispone a crear un metamodelo Ecore y una vez finalizado se procede a realizar la configuración del mismo.

La creación del metamodelo debe cumplir el requisito esencial de de ser rico y expresivo. Llamamos rico, al hecho de ser lo más abarcativo posible, brindando las herramientas suficientes como para poder modelar la aplicación lo más completa posible. Decimos expresivo ya que trataremos de realizar el mismo de la forma más simple y fácil de entender y manipular.

Teniendo en cuenta las características descritas se empieza definiendo el metamodelo con una clase o componente elemental denominado *Application*, cuya función principal será la de actuar como entidad principal y representará a una aplicación cualquiera en sí.

Antes de continuar explicando cada componente del mismo, nos concentramos en dividir el metamodelo en cuatro grupos importantes con el fin de que la explicación resulte más sencilla de entender. Estos grupos son:

1. Screens
    a. Layout
2. Widgets
3. Menus
    a. Menús de opciones
    b. Menús contextuales
4. Actions

Screens son elementos que representan en forma visual, una instancia en un momento dado de la aplicación. En sí, una Screen no es nada sin un componente Layout, que es el cual le termina de dar vida a la entidad Screen. El Layout es quien contiene otros componentes, como por ejemplo, los Widgets. También, un Layout puede contener otros Layouts.

Algunos atributos interesantes de algunos componentes de este primer grupo son:

- *Application*

    - *PackageName*: el nombre del package que se utilizará para almacenar los archivos java que se generen.
    - *VersionCode*: por defecto 0. Debe ir un valor entero. Sirve para llevar un control de actualizaciones, una versión nueva puede consultar este código, y si es menor, puede decidir de actualizar o no.
    - *VersionName*: por defecto vacío. Es una valor de tipo *String* y se utiliza para describir las versiones de tipo: <mayor>.<menor>.<específico>

- *Layout*

    - *Orientation*: puede tomar dos valores: horizontal y/o vertical. Según la elección, los elementos dentro del *Layout* se ubicaran en forma horizontal (uno al lado del otro) o en forma vertical (uno debajo del otro).
    - *Persistent*: define si el *Layout* es persistente o no. Por defecto se encuentra en *false*. En caso que sean persistentes, se persistirá cada uno de los componentes que estén contenidos, insertando automáticamente un botón que permite realizar el guardado.

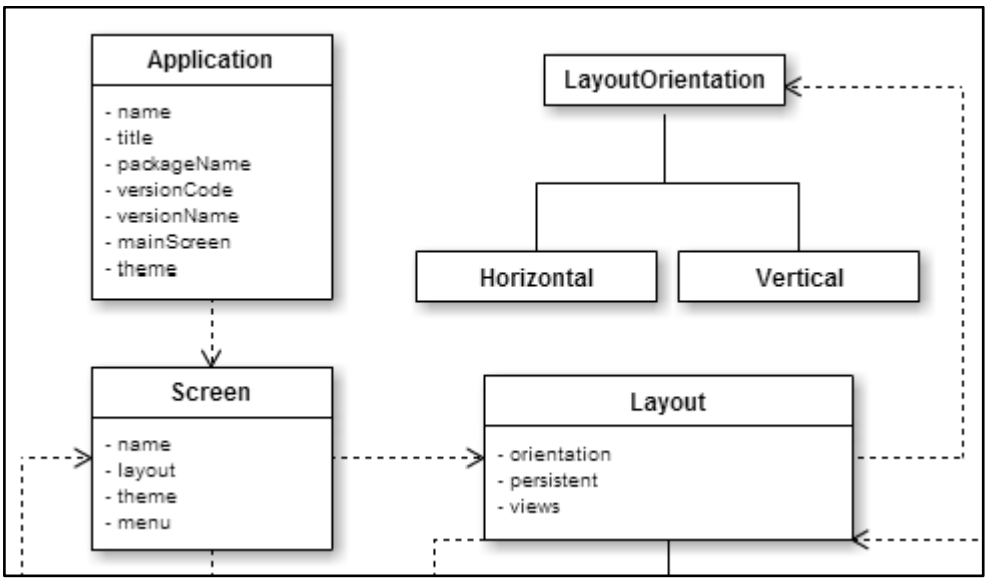

Figura 8 - Componentes Screen y Layout

Una vez finalizada la explicación del concepto de Screen estaremos en condiciones de definir los componentes Widgets y Menu. Widgets son controles o componentes interactivos que forman parte y son contenidos en un componente *Screen*. Widgets son los botones, campos para texto, casilleros de selección, etc. En el metamodelo propuesto se tienen en cuenta los siguientes controles o componentes (con algunas propiedades importantes):

- *Button*: elemento especialmente diseñado para realizar algún cambio en la aplicación. El mismo lanza una acción en el momento en el cual se lo presiona.
- *Label*: elemento para mostrar texto.
- *Checkbox*: elemento o casillero de selección. La acción de presionar sobre uno de estos elementos colocará un “check” o una marca dentro de la casilla, indicando que dicho casillero esta seleccionado.
- *EditText*: diseñado esencialmente para el ingreso de texto. Soporta varios tipos de ingreso de texto con sus validaciones correspondientes:

    - *Email*: sólo soporta direcciones de email. Se provee de una validación para el ingreso exclusivo de direcciones de email.
    - Password: oculta los caracteres ingresados. Soporta caracteres alfanuméricos y especiales.
    - *Multiline*: permite el ingreso de más de una línea de texto.

  - *Text*: permite el ingreso de sólo una línea de texto.
  - *Numeric*: permite sólo el ingreso de números.

- *RadioGroup*: contenedor del componente RadioButton, donde sólo se permitirá la elección de un único elemento.
- *ToggleButton*: casillero de selección con dos estados posibles.

  - *textOn* y *textOff*: atributos no obligatorios para personalizar los textos de *On* y *Off*.

- *Spinner*: lista desplegable donde sólo se permitirá la selección de un único elemento.
- *List*: se encarga de renderizar una lista desplazable de elementos. Una lista está compuesta por un *Layout* que configura el renderizado de cada elemento y por defecto provee de una barra de desplazamiento vertical para poder acceder a ítems no visibles por limitaciones de la pantalla de un dispositivo.

  - *itemContextualMenu*: El mismo soporta la utilización de un Menú Contextual para cada uno de los elementos de la lista.
  - *itemLayout*: su composición se basa en un *Layout* que configura la visualización de cada elemento de la lista. En caso que el *Layout* sea persistente los elementos de la misma se recuperarán de la base de datos; por el contrario, cuando sea no persistente se generarán datos aleatorios y de prueba para poder comprobar cómo se visualizan los elementos de la misma.

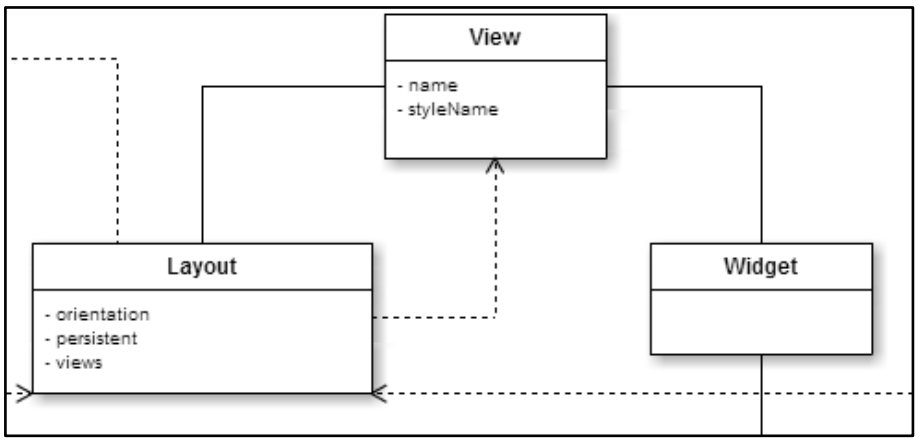

Figura 9 - Representación de la clase Widget

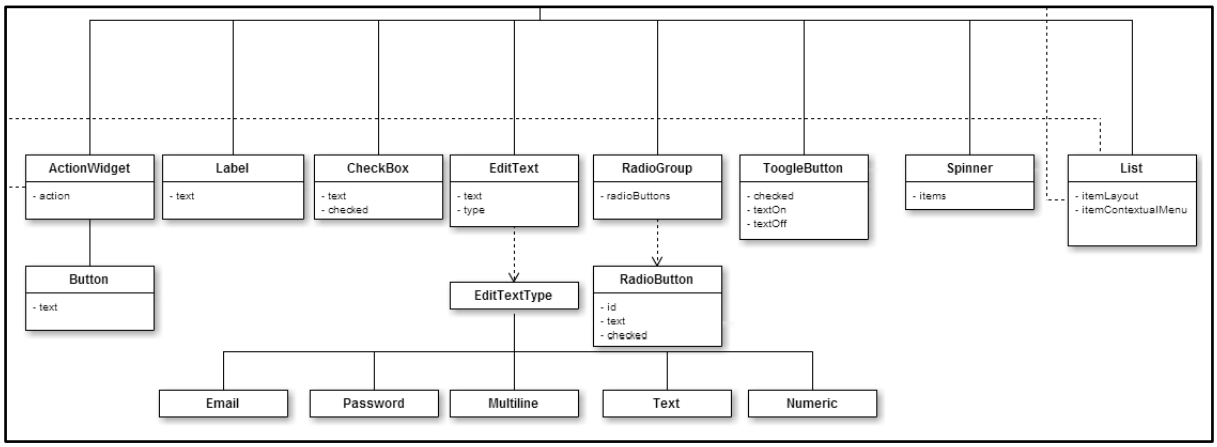

Figura 10 - Subclases de Widget

Un elemento *Menu* no es más que una lista de opciones la cual cada una está destinada a realizar algún tipo de acción específica. Existen dos tipos de componentes Menu:

- De opciones
- Menú contextual

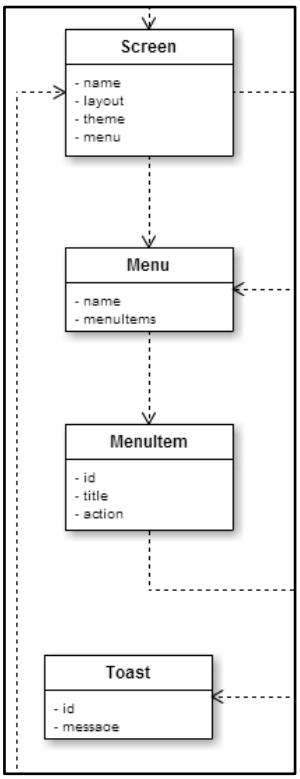

Figura 11 - Menu y MenuItem

El tipo variará dependiendo del contexto donde se lo integre. Si se encuentra dentro de un elemento *List*, será un menú contextual. Dentro de un elemento *Screen*, será un

menú de opciones. Cada *Menu* tiene opciones, y estas opciones están formadas por *MenuItems*.

Algunas propiedades:

- *MenuItem*
    - *action*: el ítem puede ejecutar una acción o *Action* predefinida.

El componente *Action* son acciones predefinidas en el metamodelo, con el fin de ayudar o establecer acciones por defecto al usuario. Las mismas pueden ejecutar las siguientes acciones:

- *CloseScreen*: acción para cerrar el componente *Screen* que lo invocó.
- *CloseApplication*: acción para cerrar o finalizar el proceso del *Application*.
- *ChangeScreen*: acción para cambiar de pantalla, ocultando la actual y mostrando una nueva.
- *ShowConfirmation*: muestra una ventana de confirmación compuesto por un mensaje y dos botones para cancelar o confirmación una acción.

Como una funcionalidad extra, una vez ejecutada la acción se puede disparar un *Toast*, cuya función principal es la de mostrar en forma no invasiva, una mensaje inmediatamente luego de realizar cualquiera de las acciones descritas arriba.

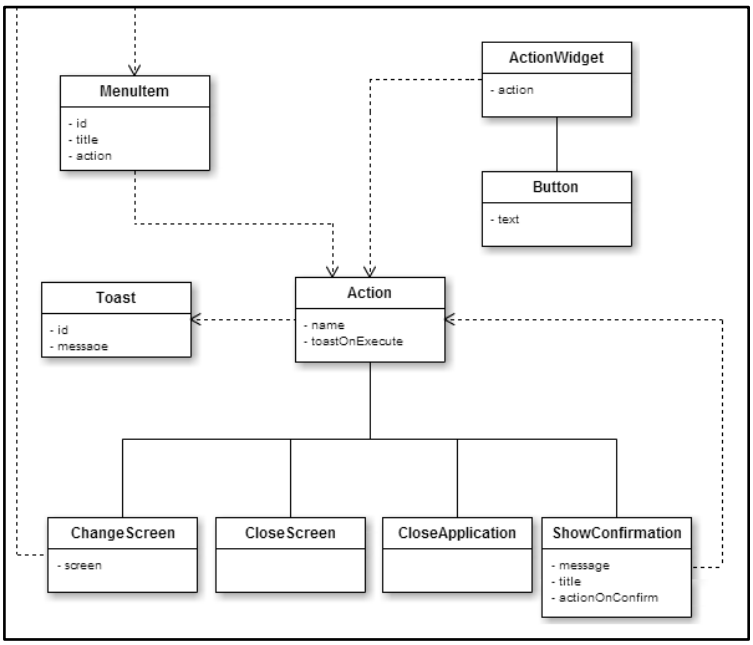

Figura 12 - Actions y sus subclases

Una vez finalizada la creación del metamodelo, se procede a representar el metamodelo en un sistema informático. Para realizar esto utilizaremos Eclipse IDE en su versión 3.6.2 como herramienta de desarrollo. Nativamente Eclipse IDE no cuenta con funcionalidad para poder almacenar un metamodelo, entonces para poder llevar esto a cabo necesitaremos la ayuda de un plugin denominado GMF (Graphical Modeling Framework) cuya instalación incluye herramientas necesarias para crear nuestro metamodelo. En esta instancia utilizaremos Ecore como herramienta para representar el metamodelo en un sistema informático.

A continuación, se procede a instalar GMF desde "Install New Software" en Eclipse IDE. Se utiliza el siguiente "update site"[18] para obtener el plugin:

- *http://download.eclipse.org/modeling/gmp/gmf-tooling/updates/releases-2.4.0/*

Finalizada la instalación, creamos un proyecto nuevo (ar.edu.unlp.info.mdd.tesis.metamodel) y ahí mismo creamos el archivo Ecore con el Wizard que se muestra en la imagen:

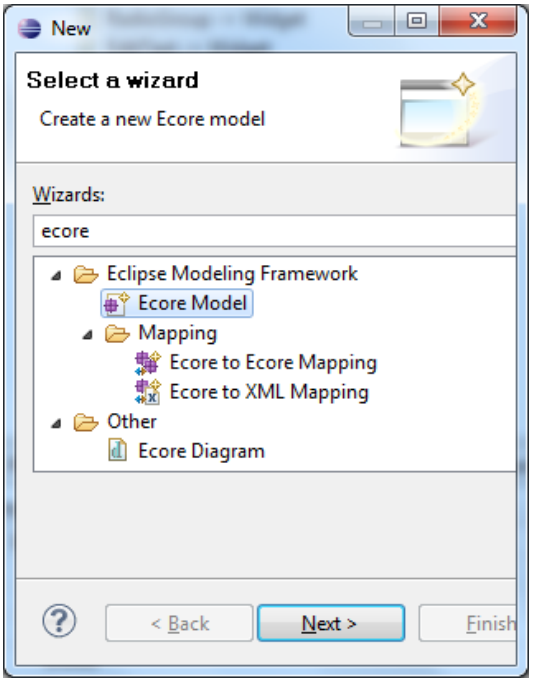

Figura 13 - Creación del archivo Ecore

[18] Update Site se compone de una URL donde existen los recursos necesarios para que Eclipse IDE pueda descargar e instalar sus plugins.

La creación de los elementos es moderadamente sencilla, a continuación se describe en la figura, el archivo Ecore final con el metamodelo recientemente mencionado:

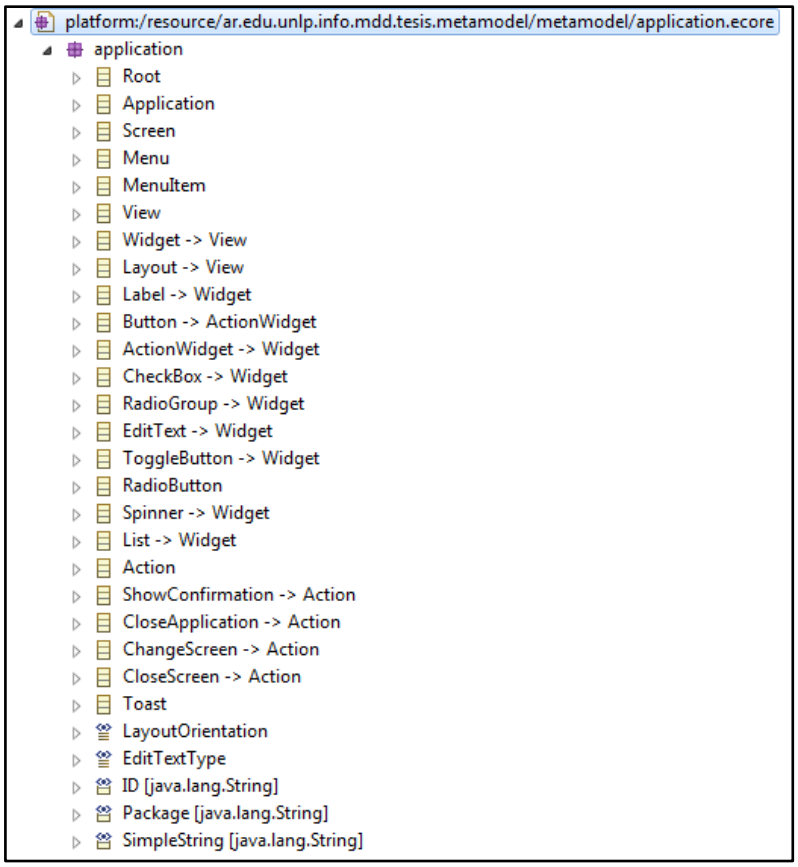

Figura 14 - Configuración final del Ecore con el metamodelo

Se observa la existencia de una clase *Root*. Esta clase actuará como raíz principal, contenedora de todas las demás clases o elementos del metamodelo, como lo indica la siguiente figura a continuación:

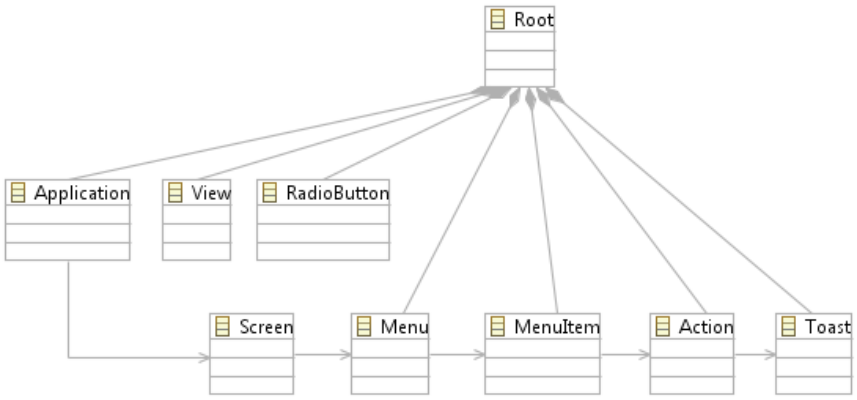

Figura 15 - Primer nivel del diagrama del metamodelo con el nodo Root como contenedor

Es esencial el hecho de poseer un elemento, en nuestro caso el elemento *Root*, que contenga todos los elementos del metamodelo. Cuando decimos todos, nos referimos a todas las clases menos aquellas que sean subclases. Ejemplos de estas serían: *View*, *RadioButton*, *Application*, *Screen*, *Menu*, *MenuItem*, *Action* y *Toast*.

El fin de esta clase *Root* es para que exista una correcta identificación de las clases a la hora de mostrar y seleccionar las clases a utilizar, en la generación del código para el editor gráfico, específicamente en la instancia de generación del *gmfgraph* y *gmftool*. De esta forma, el Wizard sólo mostrará correctamente los nodos siempre y cuando haya un elemento general que los contenga a todos.

Para la correcta configuración del Ecore, las siguientes son algunas de las propiedades que más comúnmente se utilizan:

- *Containment:* se utiliza para especificar que un elemento A contiene a uno B.
- *Unique:* define si el mismo valor puede ocurrir más de una vez.
- *Abstract:* se considera abstract a la clase que nunca se instancia, por ejemplo, en nuestro caso, la clase *View*.
- *Lower Bound y Upper Bound:* número mínimo o máximo que los valores pueden ocurrir de dicho elemento.
- *Name:* representa el nombre del elemento. Es obligatorio.
- *EType:* representa el tipo del elemento. El tipo puede ser uno nativo o algún elemento del Ecore.
- *ID:* se utiliza para identificar unívocamente un elemento.

EMF, a su vez, permite crear tipos de datos denominados *EDataType*, además de *EClass*, cuyo objetivo es el de representar una clase o elemento, para aplicar en los atributos de cada uno de los elementos.

En el metamodelo se definen tres tipos de datos customizados: *SimpleString*, *ID* y *Package*. También existe la posibilidad de definir tipos enumerativos, denominados *EEnum*, donde en desarrollo práctico lo utilizamos para definir tipos de datos cuyos valores están preestablecidos, ejemplo: *LayoutOrientation* y *EditTextType*. Esto se ha realizado con el fin de acotar el vocabulario a utilizar tanto en nombres como en IDs de

los elementos. De esta forma evitamos que el usuario ingrese caracteres no aceptados que luego podrían llegar a provocar errores en la generación de código.

Como paso final, en esta instancia, armaremos un plugin con el metamodelo, estableciéndose de esta manera, el primer plugin de la solución propuesta.

5.3. Desarrollo de una herramienta gráfica para la creación de modelos

En este apartado se mencionan los pasos para generar una herramienta gráfica para que el usuario pueda crear su modelo de aplicación, con los componentes definidos en el metamodelo. Como resultado se obtendrá un editor gráfico compuesto de un editor para armar el modelo y una barra de herramientas donde se dispondrán los elementos necesarios para crearlo. El editor debe cumplir dos condiciones: ser intuitivo y sencillo. La generación de la herramienta gráfica consta de seis partes, que se describirán a lo largo del apartado. Para la generación de cada uno de los pasos se hará uso del framework de modelado GMF, el cual provee de un editor visual llamado GMF Dashboard[19] para la generación rápida del editor.

5.3.1. Generación del modelo de dominio (Domain Gen Model)

Una vez que tenemos el metamodelo Ecore definido, armado y validado, se procede a generar el archivo *genmodel* para poder generar el código de dominio mediante la opción “Derive”.

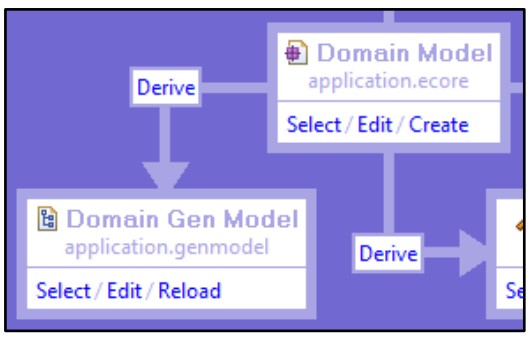

Figura 16 - Derivación y creación del genmodel

[19] GMF Dashboard es una vista que viene incluida en el plugin GMF para la creación de editores gráficos.

Este paso es sencillo e insume poco tiempo, ya que lo único que resta hacer, es generar el código haciendo uso de la opción “Generate All” del *genmodel*. Una vez generado el código, se obtendrá el código dividido en cuatro proyectos, denominados Model, Edit, Editor y Test. Es importante mencionar que cada uno de los proyectos se generará con la configuración necesaria para luego, poder generar plugins.

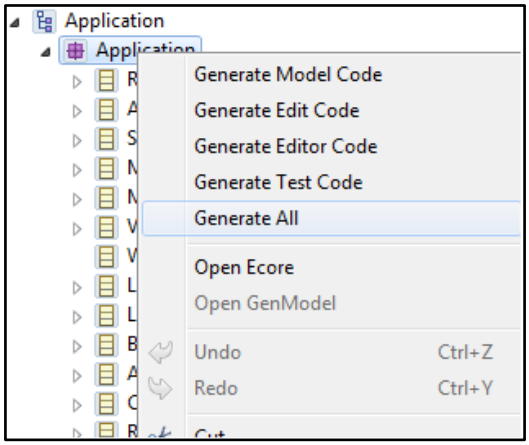

Figura 17 - Generación del código del dominio

5.3.2. Definición de elementos gráficos del modelo (Graphical Def Model)

En esta parte se definen los elementos gráficos del modelo. Para esto, utilizando el Wizard del Dashboard de GMF, primero se seleccionará la opción “Derive” y luego se elegirá cuáles serán, basándose en nuestro metamodelo Ecore, los nodos (nodes), links (connections) y etiquetas (labels) que se tendrán en cuenta para mostrar en el editor. Los nodos representan la elección de una clase en sí, los links o connections se utilizan para determinar que dicho atributo contiene otros nodos, y las etiquetas o labels identifica un atributo con un valor determinado.

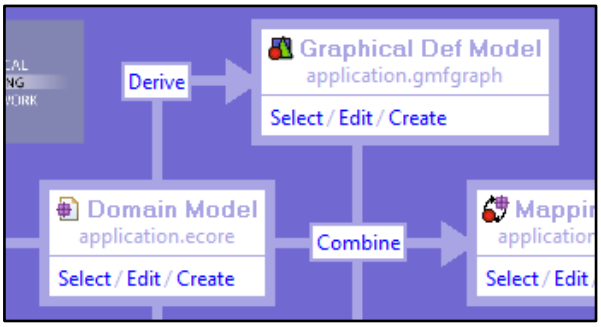

Figura 18 - Derivación y creación del gmfgraph

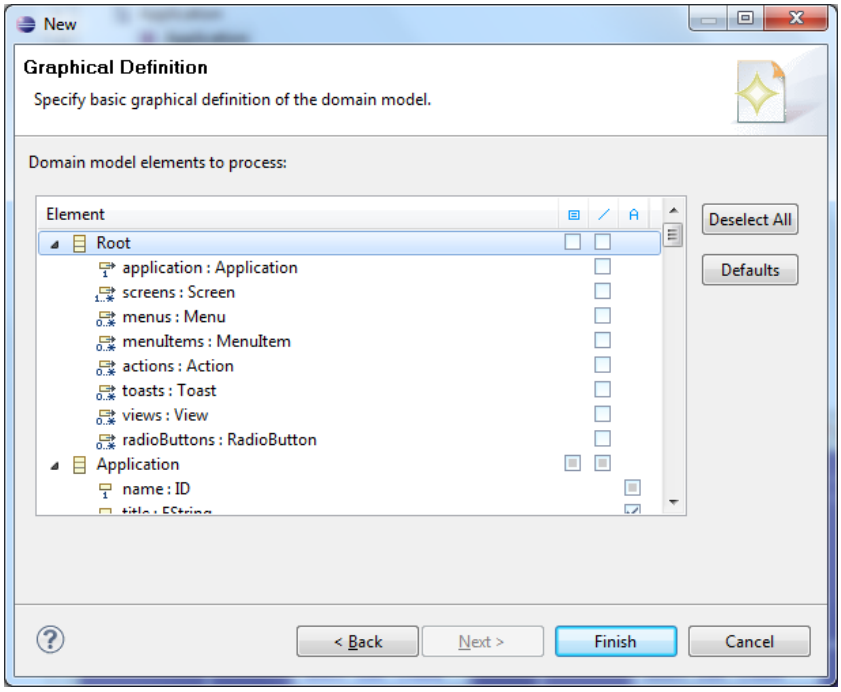

Figura 19 - Configuración del archivo gmfgraph

Finalizada la configuración, obtendremos un archivo con extensión *gmfgraph*.

5.3.3. Definición de la paleta de herramientas (Tooling Def Model)

Se pretende definir la paleta de herramientas, cuyo objetivo es la de proveer los elementos para insertar en el editor y poder construir el modelo. Nuevamente, utilizamos la opción “Derive” del Dashboard para configurar la paleta.

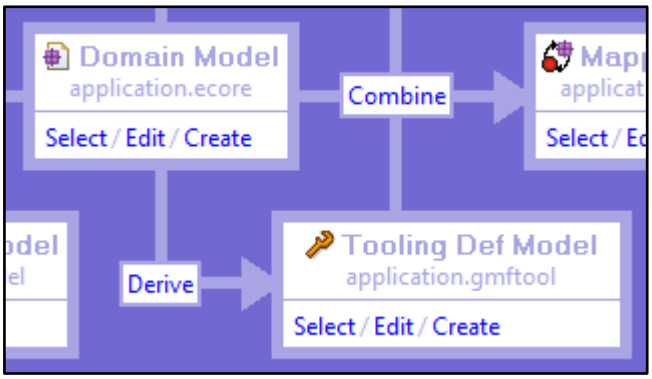

Figura 20 - Derivación y creación del gmftool

En este paso es primordial elegir correctamente qué elementos visualizar y cuáles no. La configuración es semejante a la realizada para el editor con la excepción que en este

caso sólo vamos a tener que elegir entre nodos y links, sin tener la opción de elegir etiquetas.

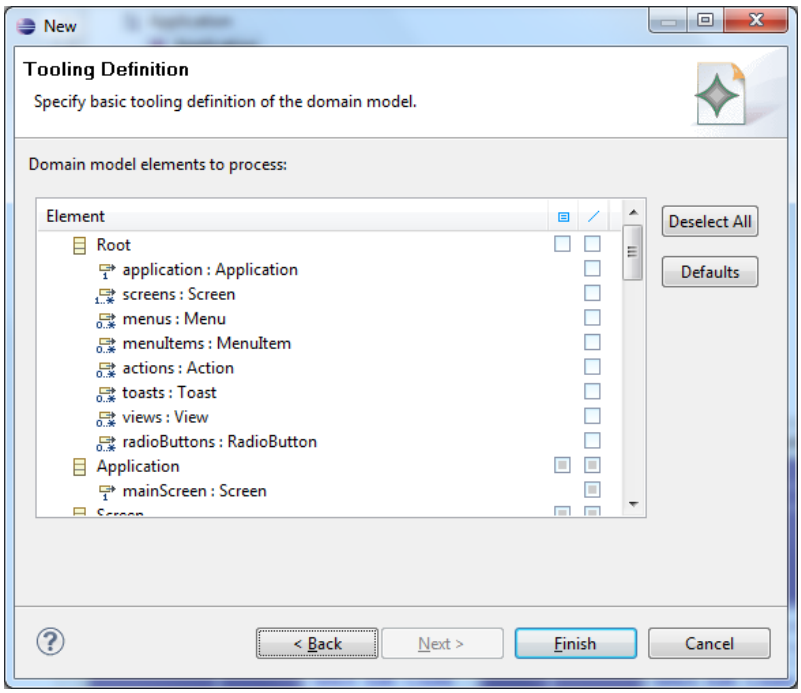

Figura 21 - Configuración del archivo gmftool

Finalizada la configuración, se obtiene un archivo con extensión *gmftool*.

Para una futura organización se decide crear agrupaciones dentro del *gmftool*. Con estas agrupaciones dividiremos los componentes en cuatro importantes grupos:

- General: *Application*, *Screen*, *Layout*, etc.
- Menu: *Menu*, *MenuItem*, etc.
- Widgets: *Label*, *Button*, etc.
- Actions: *ShowConfirmation*, *CloseApplication*, etc.

Para realizar esto, en “Palette ApplicationPalette, New Child” se agregan “Tool Group”, y se procede en agrupar las “Creation Tool”.

5.3.4. Mapeo del modelo (Mapping Model)

El mapeo se utiliza para unir el metamodelo desarrollado, con los elementos gráficos y la paleta de herramientas. Técnicamente se utiliza el archivo Ecore, el archivo *gmfgraph*

y el *gmftool*. La unión de los tres archivos sirven para poder continuar con la generación del editor gráfico final y también, para que este funcione correctamente. El hecho de que haya que lidiar con tres archivos lleva a que la fusión de los mismos se torne complicada, ya que más allá de tener la posibilidad de realizarlo automáticamente con el Wizard del Dashboard, el mismo a veces no genera algunas relaciones en forma correcta y por este motivo habrá que realizarlas a mano. La combinación se realiza mediante la opción "Combine" del Dashboard.

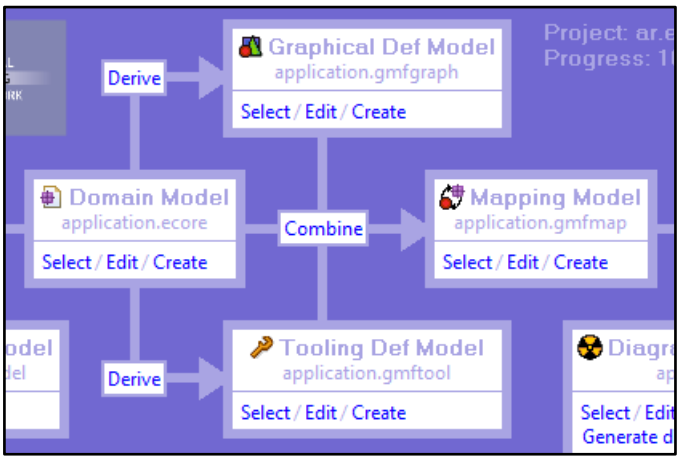

Figura 22 - Combinación para generar el gmfmap

Una vez dentro de la configuración o ejecución del Wizard, el mapping o mapeo consiste esencialmente en combinar Nodos y Links en forma armónica. Esto es, tratar de que cada nodo, cumpla con las uniones o relaciones que debe tener con otros nodos.

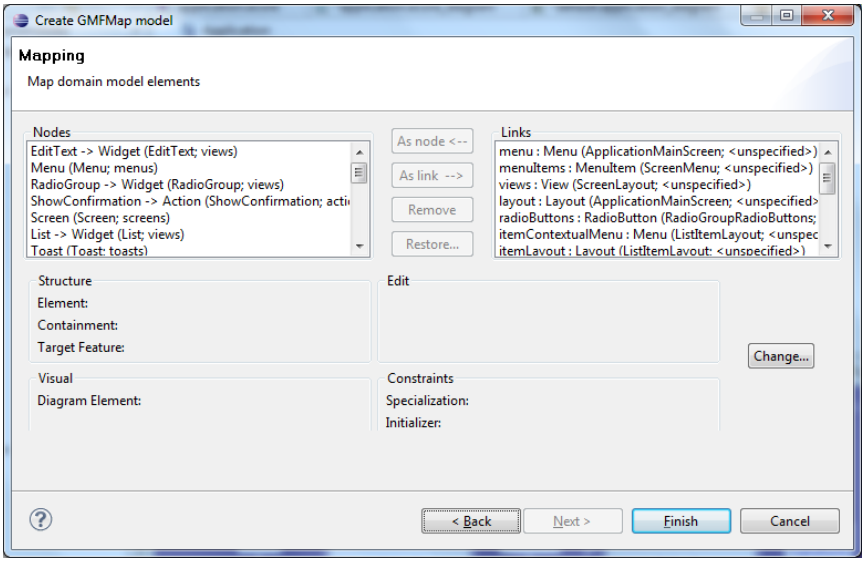

Figura 23 - Mapeo gmfgraph con gmftool desde el Wizard del Dashboard

Como dijimos anteriormente, una vez finalizado y generado el archivo de mapeo, llamado *gmfmap*, habrá que realizar modificaciones en forma manual.

Para terminar de configurar el archivo de mapeo, habrá que modificar cada "Top Node Reference" y cada "Link Mapping" para verificar que todo se haya generado correctamente, caso contrario se modificará o eliminará lo que sea necesario para que esto suceda. Finalizado el mapeo manual, habrá que validar el archivo para evitar problemas u omisiones en las modificaciones.

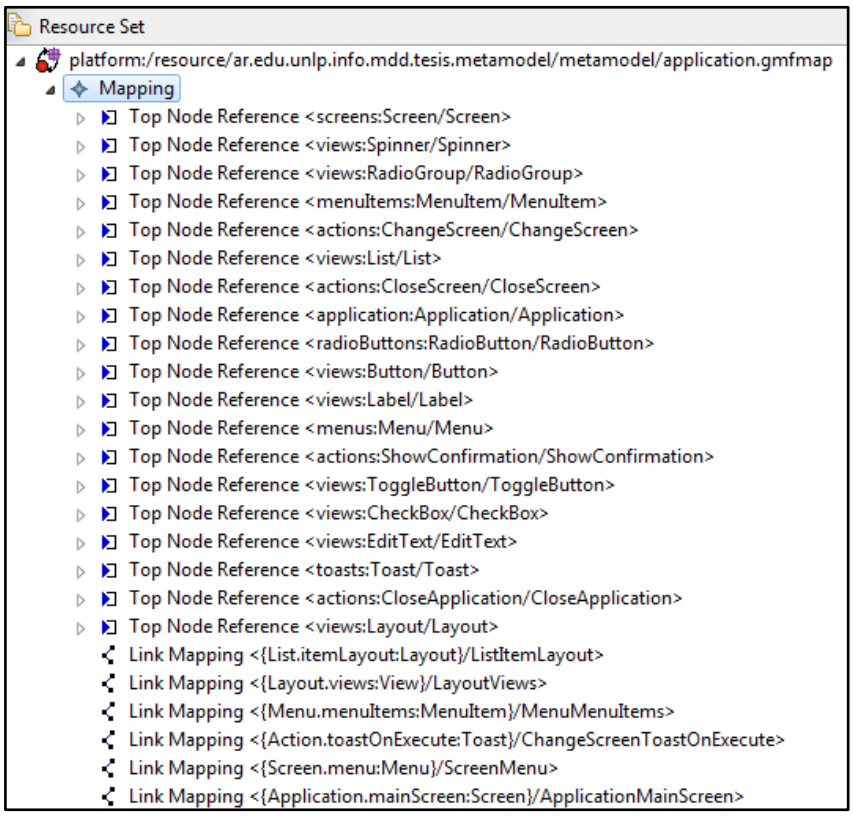

Figura 24 - Archivo final gmfmap

5.3.5. Generación final del editor gráfico (Diagram Editor Gen Model)

En este paso, siguiendo el normal curso del diagrama del Dashboard, ya habremos hecho un importante avance para el objetivo final.

En esta etapa, como en uno de los pasos anteriores, lo único que hay que hacer es transformar y crear el archivo *gmfgen* para luego poder generar el editor haciendo uso

de la opción “Transform” del Dashboard. Una vez generado el archivo mencionado, se configura el mismo para más adelante, poder desarrollar una funcionalidad denominada “Live Validation”. Para esto dentro del *gmfgen*, en “Gen Diagram RootEditPart” debemos setear en “True” los siguientes dos atributos:

- “Validation Enabled” y
- “Validation Decorators”.

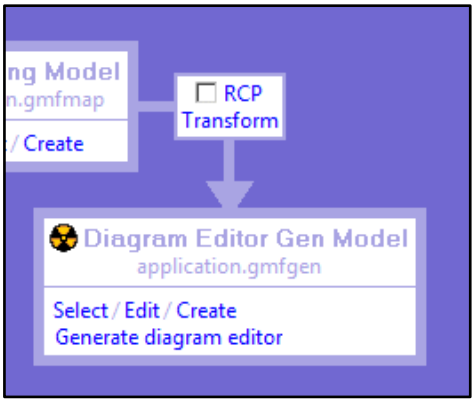

Figura 25 - Transformación del gmfmap a gmfgen

Para finalizar, con un único click en “Transform” daremos final al workflow o circuito y se habilitará el botón “Generate Diagram Editor” para generar el fuente del mismo.
Generado el fuente, se crea un proyecto, nuevamente con la estructura y configuración para generar un plugin.

5.3.6. Configuración del código del dominio

La configuración del código de dominio o generado, consta de tres partes importantes. Esta configuración brinda al editor una forma más sencilla e intuitiva a la hora de crear el modelo. Estas tres partes son:

- el desarrollo de código para realizar lo que se llama “Live Validation”,
- la creación de íconos para representar los componentes del metamodelo, y
- la modificación de los mensajes a mostrar de cada componente del editor.

Live Validation: Es importante para un desarrollo como el propuesto, definir formas lo menos invasivas posibles para lo que es la validación. Para esto, se ha desarrollado la ejecución automática de un validador de modelos. Este validador se accionará siempre

y cuando el usuario intente guardar el modelo. Esta modificación del código involucra el proyecto que contiene el código del editor generado por *gmfgen*. El método que habrá que modificar será el *doSaveDocument()* del archivo java denominado *ApplicationDocumentProvider*. Esto se realiza con el fin descrito más arriba, donde dentro del mismo, se agrega una línea de código para validar el modelo:

```
protected void doSaveDocument(IProgressMonitor monitor, Object element,
       IDocument document, boolean overwrite) throws CoreException {
              ValidateAction.runValidation((org.eclipse.gmf.runtime.notation.View)
document.getContent());
              [ ... ]
}
```

Iconos de representación de componentes del metamodelo: consideramos muy importante, la interacción que el usuario tenga con la interfaz, y para esto damos una gran relevancia a la representación de los componentes del metamodelo, identificándolos con iconos descriptivos. Para esto, se han creado un pack de íconos que se adaptan y abarcan cada componente del metamodelo.

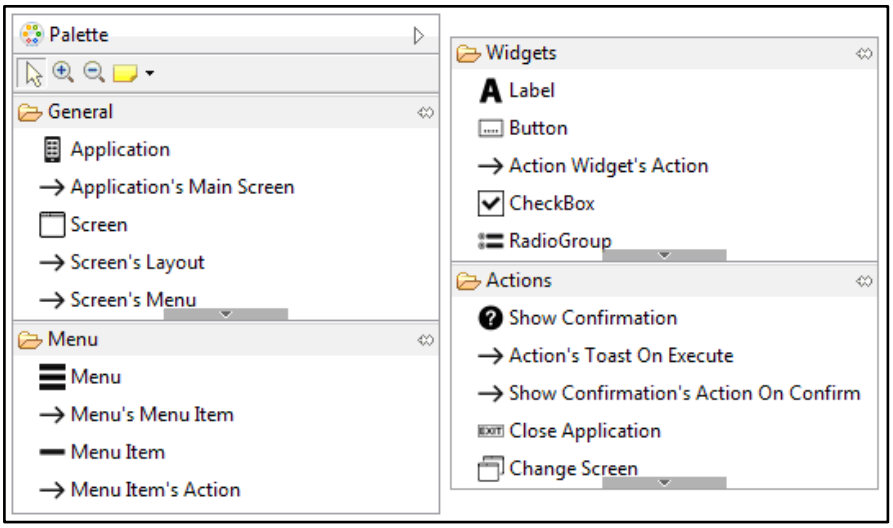

Figura 26 - Iconos del tooling para crear el modelo

Los íconos que se generan se almacenan en uno de los proyectos generados, en este caso, el proyecto “ar.edu.unlp.info.mdd.tesis.metamodel.edit” ser el contenedor de los mismos.

Modificación de mensajes: la modificación de los mensajes que se presentan en la pantalla tienen como fin la clarificación de los textos mostrados en el editor. Estos textos

representan casi la mayoría de los mensajes que aparecen en el diagram, tanto del editor en sí como el de la paleta de herramientas. Para esto se modifica el archivo "messages.properties" del proyecto ar.edu.unlp.info.mdd.tesis.metamodel.diagram.

En esta instancia se tiene todo preparado y configurado como para poder generar los plugins restantes.

5.4. Transformación del modelo a código fuente Android

La transformación del modelo a código fuente es una de las partes más importantes del desarrollo de esta implementación. En esta instancia es donde ocurre la transformación del modelo desarrollado por el usuario, hasta llegar al código fuente Android.

Como se ha mencionado más arriba, en esta instancia se generará el código con la ayuda de un framework de transformación de modelos denominado Acceleo. Esta herramienta de transformación de un modelo a código fuente trabaja con archivos denominados "templates". Estos templates son archivos con extensión MTL, cuya finalidad es la de generar archivos con código fuente.

Como primer paso es necesario instalar este plugin y lo hacemos como hemos hecho con GMF, en "Install New Software" agregamos la el siguiente update site:

- *http://download.eclipse.org/modeling/m2t/acceleo/updates/releases*

En este caso elegimos e instalamos Acceleo en su versión 3.4.0.

Acceleo, tiene como característica principal el uso de un archivo principal denominado "MainGenerator" y es el punto de inicio de las transformaciones que se realizarán. Este archivo principal se puede crear mediante un Wizard denominado "Acceleo Module File", cuyo fin será el de realizar las configuraciones iniciales y poder elegir el metamodelo con el cual trabajar. En nuestro caso utilizaremos el metamodelo creado en los apartados anteriores. En el campo "Metamodel's URL" elegiremos este último y en "Type", elegimos el elemento *Root* del metamodelo.

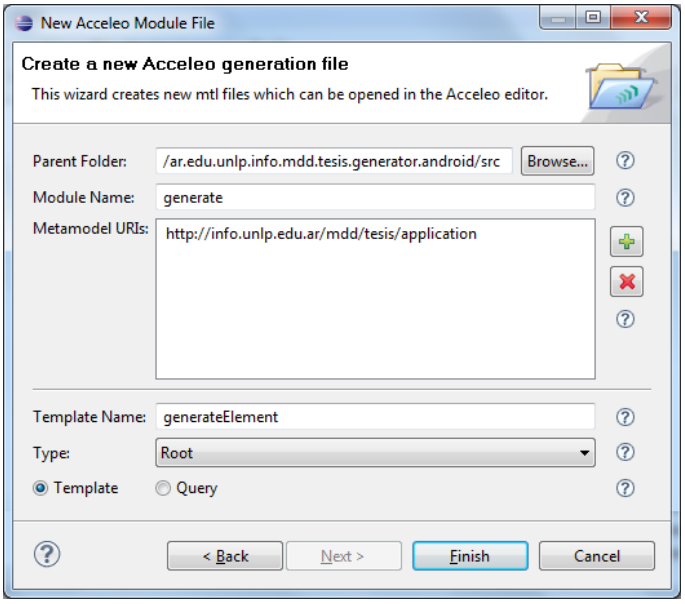

Figura 27 - Generación del MainGenerator.mtl

Este primer archivo, el "MainGenerator.mtl" actuará como origen o génesis y será el cual Acceleo tomará para empezará a interpretar y convertir lo que se encuentre implementado en los todos templates que estén referenciados desde este. El archivo para poder ser tomado como template principal, deberá contener el tag o comentario *[comment @main/]*.

Un archivo Acceleo es de extensión MTL y puede contener cuatro tipos de tags: *module*, *import*, *template* y *query*.

El tag *module* denomina el nombre del archivo o módulo. *Import* se utiliza para importar otros MTLs. *Template* para definir el cuerpo general del código a generar y el *Query* se utiliza para definir funciones que devuelven porciones de código que se pueden repetir más de una vez.

```
[comment encoding = UTF-8 /]
[comment]
 Contributors:
     Martinez, Juan Pablo
     Vosou, Agustin
[/comment]

[module mainGenerator('http://info.unlp.edu.ar/mdd/tesis/application')]

[import ar::edu::unlp::info::mdd::tesis::generator::android::files::resources::AndroidManifestXmlFile /]

[template public mainGenerator(root : Root)]
    [comment @main/]
    [generateAndroidManifestXmlFile(root)/]
[/template]
```

Figura 28 - Ejemplo de template principal de Acceleo

Una vez generado el template principal, hemos decidido definir la arquitectura la de las transformaciones en diferentes packages[20]:

- main
- commons
- files
    - widgets
    - resources
    - helpers
    - lists
    - database

Cada package contiene uno o más templates para la transformación. En el caso del package "main", contiene el *MainGenerator*, cuya finalidad es la de actuar como iniciador del proceso de ejecución de las transformaciones. También contiene las referencias a los otros templates que se deseen utilizar.

El package "commons" es el encargado de contener archivos cuya función sea la de asistir y proveer alguna funcionalidad secundaria en un template. Concretamente en este package existen los siguientes archivos:

- *DBHelpers*: define queries y templates relacionados con la base de datos.
- *FileHelpers*: define queries y templates relacionados con los archivos. También define nombre de archivos, rutas y packages.
- *Helpers*: define queries y templates que pueden ser de uso común en todo el proyecto.

El package "files.widgets" contiene el MTL para la generación de las *Activities* y algunos de sus *Widgets* (*Button*, *Label*, etc.). El archivo de este package es:

- *ActivitiesJavaFile*: este archivo define un archivo java por cada *Activity* de un proyecto Android.

[20] Un paquete es un espacio de nombres que organiza un conjunto de clases e interfaces relacionadas.

El package "files.resources" contiene los MTLs para generar los XMLs de configuración de Android, como por ejemplo: *Layout*, *Menus*, *Strings*, etc. El package contiene:

- *AndroidManifestXmlFile*: este archivo genera el archivo XML perteneciente al *AndroidManifest*.
- *DimensXmlFile*: este archivo genera uno o varios archivos XML pertenecientes al Dimens, cuya función es la de especificar las dimensiones dependiendo el dispositivo.
- *LayoutXmlFile*: la función de este archivo es la de generar uno o varios XML con los Layouts de la aplicación.
- *MenusXmlFile*: genera el o los archivos XML por cada menú existente en el proyecto.
- *StringsXmlFile*: este archivo genera el archivo XML que contendrá todos los textos del proyecto.
- *StylesXmlFile*: crea el o los archivos XML de los estilos, dependiendo el API Level.

El package "files.helpers" contiene MTLs para generar archivos Java de constantes, entidades, etc. Este contiene:

- *ConstantsJavaFile*: genera un archivo Java con las constantes que utilizan los Layouts si son persistentes.
- *GenericEntityJavaFile*: es un template que define la creación genérica de entidades Java que representan Layouts, es invocado desde diferentes templates, para la generación de diferentes tipos de entidades; por ejemplo entidades de Layouts persistentes y no persistentes. Para su generación se tienen en cuenta los siguientes Widgets (que almacenan datos), como por ejemplo:
    - *EditText*
    - *Checkbox*
    - *ToggleButton*
    - *Spinner*
    - *RadioGroup*

- *GenericLayoutXmlFile*: es un template que define la creación genérica de archivos XML para representar Layouts. De la misma manera que GenericEntityJavaFile, se llama desde varios templates.

El package "files.lists" contiene los templates para generar los archivos Java necesarios para generar código para los componentes *List*. El contenido es:

- *ListAdapterJavaFile*: genera archivos Java por cada *ListAdapter*. Sirve para manipular Widgets de tipo *List*.
- *ListEntityJavaFile*: genera el archivo Java de la entidad lista, por cada ocurrencia en el proyecto.

Y finalmente, el package "files.database" contiene los templates necesarios para crear los archivos Java para realizar la persistencia de los datos. En nuestro caso utilizamos SQLite como motor de base de datos. Los templates son:

- *CustomViewBinderJavaFile*: genera la clase *CustomViewBinder* en un archivo Java. Sirve para setear los valores persistidos a los Widgets: *Spinner*, *RadioGroup* y *Checkbox*.
- *DBAdapterJavaFile*: indica que si contiene Layouts persistentes, genera el adaptador para la base de datos. Tiene funciones de creación de la misma y algunas consultas básicas.
- *DBAdapterProviderJavaFile*: genera el archivo Java para el DBAdapterProvider. Entre otras cosas, puede ejecutar consultas personalizadas a la base de datos.
- *PersistentLayoutEntityJavaFile*: genera un archivo Java por cada entidad persistente en el proyecto.

La arquitectura descrita cumple el objetivo final de generar el código fuente para un proyecto Android completo. La arquitectura, dependiendo del modelo creado, genera una estructura de proyecto Android organizada de la siguiente manera:

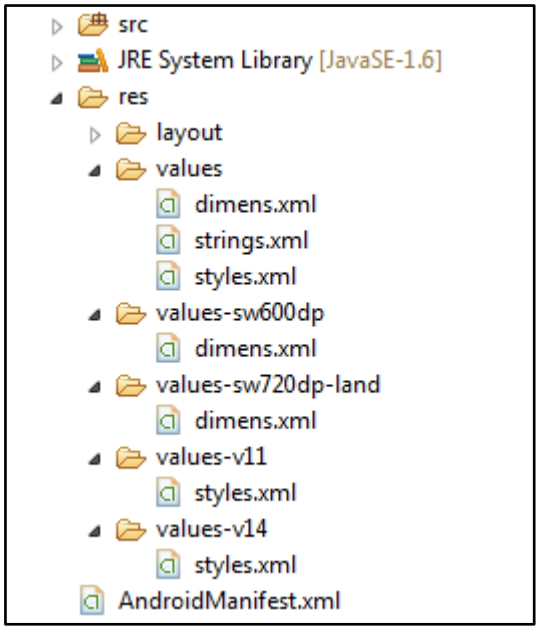

Figura 29 - Estructura de proyecto Android

Por lo general, el contenido de un proyecto Android es el representado en la imagen. En el directorio raíz encontramos el archivo Manifest, cuyo fin es la configuración general de la aplicación, el directorio "src" y el directorio "res". El directorio "src" contiene las clases Java necesarias, ubicadas en packages según su función. El directorio "res" contiene todo lo necesario para la configuración particular de diferentes aspectos del proyecto. Por ejemplo dentro de "res", el directorio Layout, contendrá los Layouts de cada pantalla, el directorio "values" tendrá recursos de textos, estilos, etc., Por último las carpetas "values-v<api-level>" tendrán estilos por cada API Level y las carpetas "values-sw<dimensión-en-dp>" los archivos XML con especificaciones para cada dimensión de pantalla que el usuario desee especificar.

Con respecto al API Level y las versiones de Android, en la implementación se ha decidido implementar el código para los rangos:

- API Level 11 al 17
- API Level 14 al 17

El motivo por el cual se tomó la decisión de abarcar el rango del nivel 11 al nivel 17, fue por el hecho de poder abarcar Android 3.0, cuyo nivel es el 11 y por lo general la mayoría de las tablets se empezaron a producir con dicho SO. El nivel 14 pertenece a Android 4.0 y en la actualidad la mayoría de tablets y celulares vienen con dicho SO.

Se toma como tope la API Level 17 (Android 4.2), ya que al momento de la implementación de la parte práctica, este nivel es el más estable en la cual la API se encontraba.

5.5. Generación de plugins para la importación a Eclipse IDE

En este último apartado se pretende dar una breve explicación sobre la generación de plugins en Eclipse IDE. En el IDE, la generación de plugins es moderadamente simple, bastará con ubicar el o los archivos plugin.xml de cada proyecto, editarlo con el "Plugin Manifest Editor", exportar los mismos y luego seleccionar todos los que están disponibles, previamente habiéndose configurado adecuadamente (en la solapa de "Overview") los siguientes campos:

- "provider name" y el
- "plugin name".

Elegiremos un lugar donde exportarlos, que por lo general será el directorio "plugins" de Eclipse IDE y con esto se finalizará la operación. La próxima vez que se ejecute Eclipse IDE, los mismos se encontrarán disponibles para poder utilizarlos.

El resultado de la generación será el siguiente:

- ar.edu.unlp.info.mdd.tesis.metamodel.diagram_1.0.0
    - Plugin que contiene el editor gráfico para modelar la aplicación.
- Plugins generados y luego utilizados por el proyecto diagram:
    - ar.edu.unlp.info.mdd.tesis.metamodel.edit_1.0.0
    - ar.edu.unlp.info.mdd.tesis.metamodel.editor_1.0.0
    - ar.edu.unlp.info.mdd.tesis.metamodel.tests_1.0.0
- ar.edu.unlp.info.mdd.tesis.metamodel_1.0.0
    - Plugin que contiene el metamodelo

En el caso del plugin de Acceleo, habrá que generar un proyecto aparte, ya que para el correcto funcionamiento del mismo, es necesario generar el "Acceleo UI Launcher".

Para esto en el proyecto donde se encuentran los templates de Acceleo se ejecuta “Create Acceleo UI Launcher Project” en donde se inicia un Wizard. Una vez iniciado, se realizan algunas configuraciones, siendo la más importante aquella donde se especifica el directorio donde se generará el código. En el caso de nuestra implementación, será el directorio raíz. Una vez finalizado el asistente, se generará un nuevo proyecto, el cual deberá volver a generarse como plugin. Al final se obtiene:

- ar.edu.unlp.info.mdd.tesis.generator.android_1.0.0
    - Plugin que contiene el código de transformación del modelo a código Android.
- ar.edu.unlp.info.mdd.tesis.generator.android.ui_1.0.0
    - Plugin necesario para que se realice la ejecución de la transformación del código contenido en el plugin anterior.

6. Casos de estudio

En el desarrollo del capítulo se presentan dos desarrollos los cuales intentan ser lo más abarcativos posibles en cuanto a las aplicaciones en general. Como aplicaciones de ejemplo se modelará:

- Una aplicación llamada “Shopping List”, que intentar simular una o varias listas de compras y
- otra aplicación denominada “My Notes”, cuyo objetivo ser la de generar notas breves para realizar comentarios, recordatorios, etc.

6.1. Introducción

Como punto inicial, es necesaria una correcta configuración del entorno para poder modelar y luego transformar el modelo a código. Para esto es requisito contar con las versiones de los siguientes plugins instalados en nuestro Eclipse IDE:

- Eclipse 3.6.2 (Helios)
- GMF 2.4.0
- Acceleo 3.4.0
- Android ADT 22.0.1

En cuanto a Acceleo, es sumamente importante mencionar una configuración clave para la correcta interpretación del modelo gráfico a código fuente. Esta configuración setea a UTF-8 como codificación por defecto para todo el entorno de Eclipse IDE. Entonces se debe agregar al archivo “eclipse.ini” ubicado en la raíz del directorio donde se encuentra el ejecutable de Eclipse IDE, la siguiente línea:

- *-Dfile.Encoding=UTF8*

Una vez realizada la configuración de Eclipse IDE, necesitaremos tener los plugins generados en el capítulo anterior para luego instalarlos en la carpeta “plugins”:

- ar.edu.unlp.info.mdd.tesis.metamodel.diagram_1.0.0
- ar.edu.unlp.info.mdd.tesis.metamodel.edit_1.0.0
- ar.edu.unlp.info.mdd.tesis.metamodel.editor_1.0.0
- ar.edu.unlp.info.mdd.tesis.metamodel.tests_1.0.0
- ar.edu.unlp.info.mdd.tesis.metamodel_1.0.0
- ar.edu.unlp.info.mdd.tesis.generator.android_1.0.0
- ar.edu.unlp.info.mdd.tesis.generator.android.ui_1.0.0

Finalizada la instalación de los plugins necesarios, se continúa con la generación de un proyecto Android. Nuevamente hacemos uso del Wizard que provee Eclipse IDE para la creación de proyectos. En este caso se utiliza el de creación de proyectos Android. A continuación se elige “Android Application Project”.

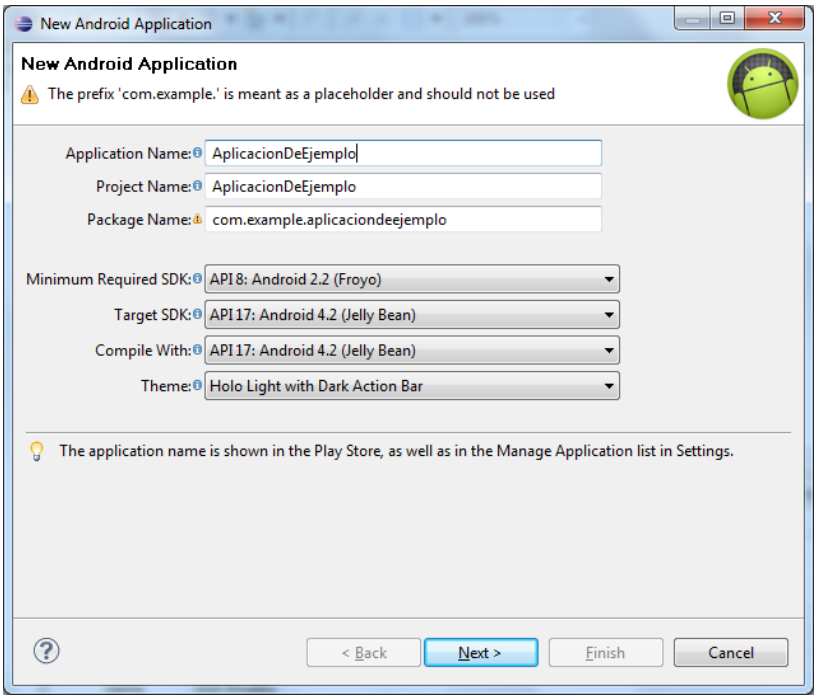

Figura 30 - Configuración inicial del proyecto Android

Se elige un nombre en “Application Name” y se continúa con la configuración. Es importante remarcar que la elección del “Minimum Required SDK” y los tres siguientes campos, es indistinto, ya que del ajuste de esto se encargarán los plugins desarrollados.

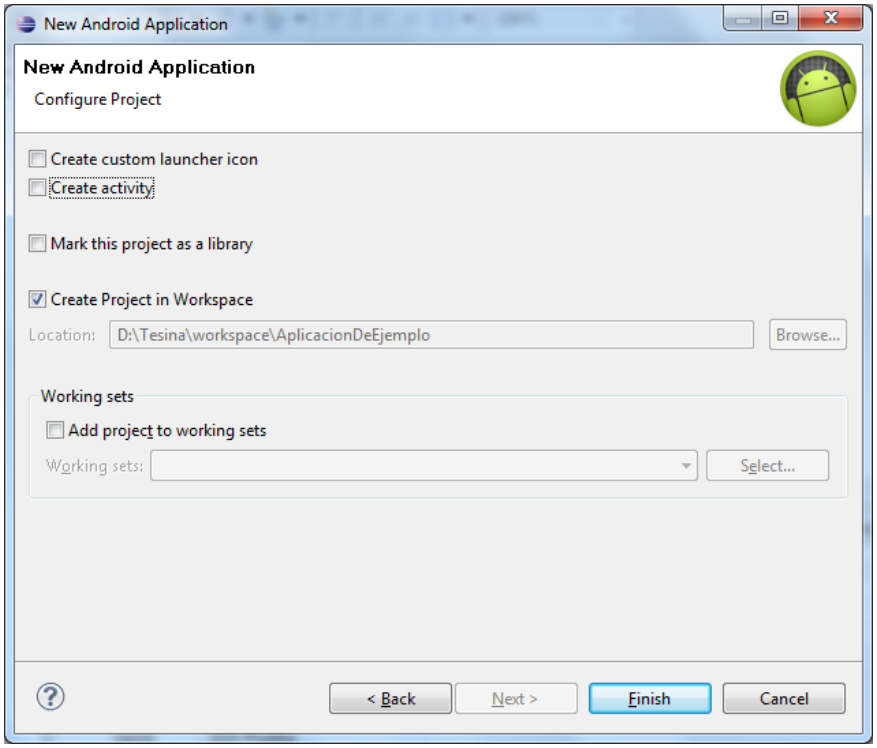

Figura 31 - Finalización de la configuración del proyecto Android

En la última imagen se destildan “Create custom launcher icon” y “Create activity” ya que es funcionalidad extra, totalmente innecesaria para este ejemplo.

Finalizada la creación del proyecto Android, se desea comenzar con la creación del modelo. Para esto, utilizando el Wizard de Eclipse IDE, dentro del proyecto Android recientemente creado, se invoca el mismo para la creación del “Application Diagram”. Este Wizard deberá crear los archivos en la raíz del proyecto Android. Este asistente constará de dos pantallas:

- Uno para elegir el nombre del “Application Diagram”
- y otro para elegir el nombre del “Application Domain Model”.

El primer archivo generado se utilizará para crear el modelo, el segundo será el archivo que se utilizará para la posterior generación de código fuente. Terminado el Wizard, se utilizar el “Application Diagram” para comenzar a modelar la aplicación.

Esta instancia, es el punto en común de cualquier aplicación a realizar. A continuación se describirán y explicarán las aplicaciones a realizar para luego continuar con la

generación de su código fuente y todas las modificaciones que se realicen en los mismos para lograr la aplicación final.

6.1.1. Consideraciones generales en el modelado de la aplicación

Es importante remarcar que a cada uno de los atributos de cada elemento no se le asignen valores conocidos que puedan llegar a estar ligadas a clases del SDK de Android o Java, como por ejemplo *List*, *Boolean*, *Activity*, etc. Todo esto se sugiere con el fin de evitar posibles errores de interpretación del código por parte del SDK. De esta forma se sugiere utilizar la notación Camelcase[21] compuesta de más de una palabra.

6.2. Aplicación "My Notes"

6.2.1. Intención

El objetivo de la aplicación "My Notes" es la de poder crear notas, ya sean observaciones, recordatorios, ayudas de memoria, etc. y de esta forma brindarle una utilidad extra al dispositivo tecnológico. Estas notas están compuestas por un título, un texto y una elección de colores para diferenciarlas. La aplicación debe permitir tanto crear, editar y eliminar las mismas. Se busca una interfaz amigable, sencilla e intuitiva.

6.2.2. Mockup y modelo de la aplicación

Como punto de partida, la pantalla principal se deberá visualizar de la siguiente manera:

[21] Camelcase es un estilo de escritura que se aplica a frases o palabras compuestas. Ejemplo: EstoEsUpperCamelCase.

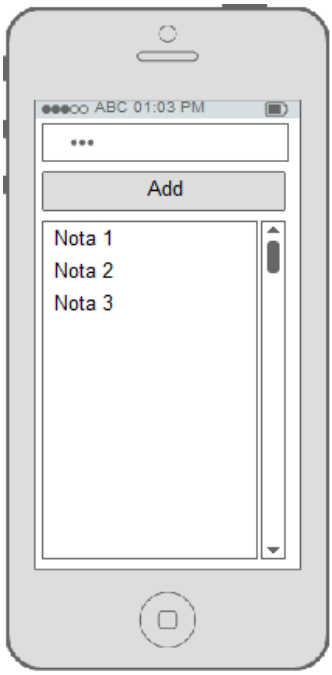

Figura 32 - Pantalla principal de "My Notes"

En la imagen se puede observar que la aplicación consta de diversos componentes, por orden de aparición:

- Un menú.
- Un botón con un texto "Add".
- Una lista que contiene ítems o notas (Nota 1, Nota, 2, etc.).
- Una barra lateral para poder realizar scrolling[22] sobre la lista.

La funcionalidad de esta pantalla principal es la de mostrar las notas que se han ido ingresando. A continuación se detalla el contenido de cada nota o ítem de la lista:

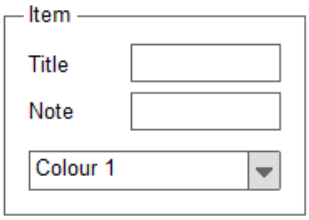

Figura 33 - Composición de una nota

[22] Se denomina scroll o desplazamiento al movimiento en 2D de los contenidos que conforman el escenario de una ventana que se muestra en una aplicación informática, por ejemplo, una página web visualizada en un navegador web.

Cada ítem está compuesto por un título y un cuerpo. El campo Title mostrará el título ingresado, y de una forma similar, el campo “Note” mostrará el texto o cuerpo de la nota. El último componente mostrará el color elegido para dicha nota.

Para agregar una nota, bastará con presionar sobre el botón “Add” mencionado anteriormente. La acción lanzará la siguiente pantalla:

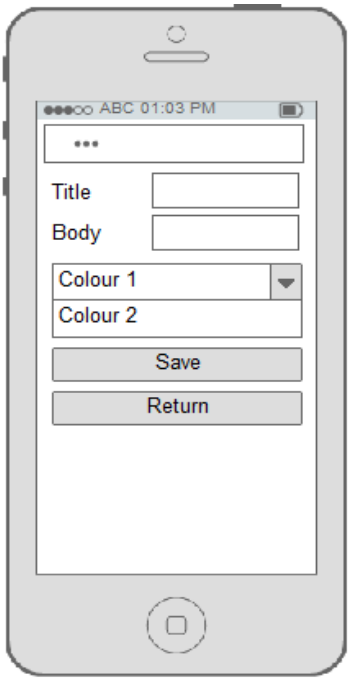

Figura 34 - Pantalla de creación/edición de una nota

En dicha pantalla se podrá crear una nota. Esta pantalla se ir completando campo por campo, hasta llegar al último cuyo fin será la de asignarle un color a la misma con el fin de resaltarla. Para guardar la nota será necesario presionar sobre el botón “Save”. El botón “Return” se utilizará para regresar a la pantalla anterior.

Habiendo regresado a la pantalla principal, manteniendo presionado uno de los ítems o notas, se desplegará un menú contextual de la siguiente forma:

Figura 35 - Menú contextual invocado desde una nota

El menú contextual consta de dos opciones: eliminar o “Remove” y editar o “Edit”. Como el nombre lo dice, la primera opción ofrece la posibilidad de eliminar la nota, previamente mostrando la siguiente ventana de confirmación:

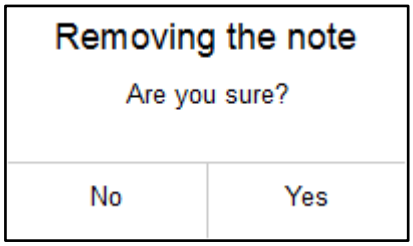

Figura 36 - Ventana de confirmación de eliminación de una nota

La ventana de confirmación pretende alertar al usuario de un inminente cambio crítico en la aplicación. En el caso de confirmar, se procederá a eliminar la nota. Caso contrario se cancelará la acción y el foco retornará a la pantalla desde donde se invocó.

“Edit” o Editar ofrecerá la opción de edición de la nota permitiendo modificar todos los campos utilizando la misma pantalla de creación.

Por último se cuenta con un menú de opciones, ubicado en la parte superior de la pantalla, en la cual se cuenta con la opción de salir de la aplicación.

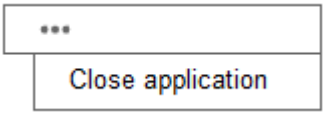

Figura 37 - Menú de opciones de la aplicación

A continuación se procede a modelar la aplicación, haciendo uso del Application Diagram. Finalmente, se obtiene el siguiente modelo:

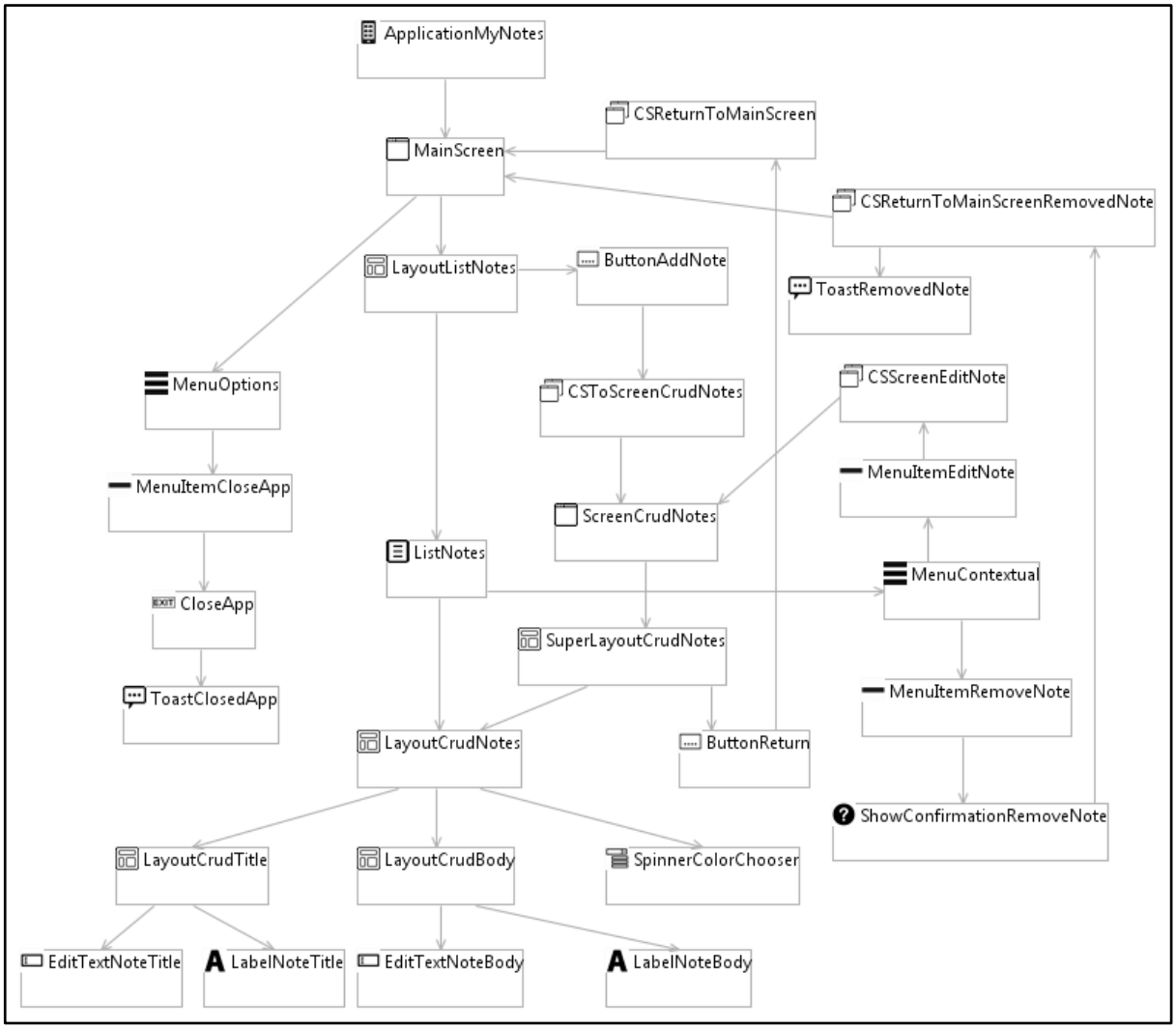

Figura 38 - Modelo gráfico de "My Notes"

Es importante remarcar dos características importantes del modelo:

- *LayoutCrudNotes* es de tipo "persistent"
- El *SpinnerColorChooser* tiene cargados los siguientes ítems: White, Red, Orange, etc.

Una vez finalizado se procede a guardar el modelo y en esta instancia se lanzará una autovalidación la cual indicará si el mismo es válido o no, esto es, si el modelo tiene atributos obligatorios vacíos, o si algún valor no concuerda con algún patrón especificado para el elemento. En caso de que la validación no sea la correcta, se sugiere verificar el problema y no continuar hasta que este valide.

6.2.3. Generación y customización de código

Finalizada la construcción del modelo, se procede a generar el código fuente. Esta vez se utilizará el archivo del "Application Domain Model" para generar el mismo. Se utiliza el menú para seleccionar "Acceleo Model to Text > Generate Application to Android" el cual completará el proyecto Android con todos los archivos necesarios para poder ejecutar la aplicación.

Una vez finalizado el procesamiento, se obtiene un proyecto Android listo para ser ejecutado. El mismo se encuentra configurado con una base de datos SQLite y todas las clases necesarias para persistir lo definido en el modelo. Se presentan algunos estilos básicos heredados del core de estilos de Android y cuenta con la mayoría de las funcionalidades modeladas, ya escritas en código fuente. Más allá de esta configuración hay que realizar algunos agregados para permitir lo siguiente:

- Eliminación de notas.
- Edición de notas.
- Conseguir una visualización más atractiva.

Eliminación de Notas

Para permitir la eliminación de una nota en particular, una de las modificaciones más críticas a realizar es la del agregado de una función que llame al método *delete()*, cuya funcionalidad ya se encuentra generada desde la generación automática del código. En este caso sólo hay que parametrizar el identificador de la nota en particular:

```
private void contextualMenuItemClickedMenuItemRemoveNote(final
        ContextMenu.ContextMenuInfo info) {
        [ ... ]
        actionConfirmedShowConfirmationRemoveNote(
                ((AdapterContextMenuInfo)info).id
        );
        [ ... ]
}
```

Para invocar al método *delete()* en *actionConfirmedShowConfirmationRemoveNote* de la forma:

```
private void actionConfirmedShowConfirmationRemoveNote(long id) {
        delete(DBAdapterProvider.LAYOUTCRUDNOTES_URI, null,
                new String[]{String.valueOf(id)}
        );
        [ ... ]
}
```

De esta forma, la funcionalidad para eliminar notas queda cubierta en forma satisfactoria. Todos estos cambios se realizan sobre la clase *MainScreen*.

Edición de Notas

Con respecto a la edición de notas, en esta instancia es necesario modificar un poco más de código que en el ejemplo anterior.

Para esto se modifica la función *contextualMenuItemClickedMenuItemEditNote* (cuya finalidad será la de lanzar una nueva ventana) para que antes del cambio de ventana se envíe el identificador de la nota seleccionada:

```
private void contextualMenuItemClickedMenuItemEditNote(
        final ContextMenu.ContextMenuInfo info) {
                [ ... ]
                intent.putExtra("id", ((AdapterContextMenuInfo)info).id);
                [ ... ]
}
```

Una vez agregada la llamada al método *putExtra()*, en la clase *MainScreen*, se procede a modificar la clase *ScreenCrudNotes*.

En el método *onCreate()* de esta última clase tendremos que obtener el parámetro enviado desde la clase *MainScreen*. Para esto se escribe lo siguiente:

```
private long id;
protected void onCreate(Bundle savedInstanceState) {
        [ ... ]
        Intent intent = getIntent();
        this.id = intent.getLongExtra("id", 0);
        [ ... ]
```

Lo que se intenta realizar, dentro de *ScreenCrudNotes*, es obtener un identificador de la nota, para buscar los datos en la base de datos con el fin de mostrarlos en la pantalla de creación de notas. Una vez mostrados, se debe modificar el comportamiento del botón "Save" para que en vez de hacer una inserción, se realice una actualización. A continuación se muestran los cambios para obtener los datos de la nota y ubicarlos en la pantalla de edición.

```
        [ ... ]
if (this.id == 0){
        [ ... ]
} else {
        layoutCrudNotes_insert_button.setOnClickListener(
                layoutCrudNotesUpdateListener
        );
        Cursor cursor = getContentResolver().query(DBAdapterProvider.LAYOUTCRUDNOTES_URI,
                null, null, new String[]{String.valueOf(this.id)}, null);
        cursor.moveToFirst();
        String title = cursor.getString(cursor.getColumnIndex(DBAdapter
                .COLUMN_LAYOUTCRUDNOTES_LAYOUTCRUDTITLEEDITTEXTNOTETITLE));
        String body = cursor.getString(cursor.getColumnIndex(DBAdapter
                .COLUMN_LAYOUTCRUDNOTES_LAYOUTCRUDBODYEDITTEXTNOTEBODY));
        String position = cursor.getString(cursor.getColumnIndex(DBAdapter
                .COLUMN_LAYOUTCRUDNOTES_LAYOUTCRUDNOTESSPINNERCOLORCHOOSER));
        ((EditText) findViewById(R.id.LayoutCrudTitle_EditTextNoteTitle)).setText(title);
        ((EditText) findViewById(R.id.LayoutCrudBody_EditTextNoteBody)).setText(body);
        ((Spinner) findViewById(R.id.LayoutCrudNotes_SpinnerColorChooser))
                .setSelection( Integer.valueOf(position));
}
```

El paso final es el de desarrollar el nuevo comportamiento del botón "Save":

```
View.OnClickListener layoutCrudNotesUpdateListener = new View.OnClickListener() {
        public void onClick(View v){
                String title =
                        ((EditText)findViewById(R.id.LayoutCrudTitle_EditTextNoteTitle))
                                .getText().toString();
                String body =
                        ((EditText)findViewById(R.id.LayoutCrudBody_EditTextNoteBody))
                                .getText().toString();
```

```
                int position =
                        ((Spinner)findViewById(R.id.LayoutCrudNotes_SpinnerColorChooser))
                                .getSelectedItemPosition();
                String[] selectionArgs = {String.valueOf(id)};
                ContentValues contentValues = new ContentValues();
                contentValues.put(DBAdapter
                        .COLUMN_LAYOUTCRUDNOTES_LAYOUTCRUDNOTESSPINNERCOLORCHOOSER,
                                position);
                contentValues.put(DBAdapter
                        .COLUMN_LAYOUTCRUDNOTES_LAYOUTCRUDTITLEEDITTEXTNOTETITLE,
                                title);
                contentValues.put(DBAdapter
                        .COLUMN_LAYOUTCRUDNOTES_LAYOUTCRUDBODYEDITTEXTNOTEBODY,
                                body);
                getContentResolver().update(DBAdapterProvider.LAYOUTCRUDNOTES_URI,
                        contentValues, null, selectionArgs);
                [ ... ]
                Toast.makeText(getApplicationContext(), R.string.beforeModification,
                Toast.LENGTH_SHORT).show();
        }
};
```

Visualización atractiva

Para lograr una visualización más atractiva, se desarrollan dos clases extras, *MyCursorAdapter* y *CustomColor*, las cuales son las encargadas de, en base a un color elegido en la creación o edición de una nota, colorear el fondo de un ítem en la lista de notas. Para esto también se define el archivo “color.xml” el cual contiene los colores a establecer en la nota.

Ejemplo de la clase MyCursorAdapter

```
class MyCursorAdapter extends SimpleCursorAdapter {

        private Context context;

        public MyCursorAdapter(Context context, int layout, Cursor c,
        String[] from, int[] to, int flags) {
```

```
            super(context, layout, c, from, to, flags);
            this.context = context;
        }

        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
            View view = super.getView(position, convertView, parent);
            Resources resources = this.context.getResources();
            Spinner spinner = (Spinner) view.findViewById(
                    R.id.LayoutCrudNotes_SpinnerColorChooser
            );
            String option = spinner.getSelectedItem().toString();

            /* Assignation of a background colour item chosen by the user */
            view.setBackgroundColor(resources.getColor(
                    CustomColor.get(option).getId())
            );
            return view;
        }
}
```

Otras de las modificaciones involucran algunos archivos XML. En general se modifican los archivos *Layout* con el fin de lograr que tanto los componentes *EditText* como los componentes *Button* ocupen por completo el ancho de la pantalla. Otra muy importante modificación involucra a todos los botones, los cuales se colorean de un color rojo degradado y se redondean sus esquinas. Esto último se logra con la ayuda de la creación de un archivo nuevo denominado “red_button.xml”.

Finalmente se generan los íconos, por cada resolución disponible, con el fin de representar a la aplicación en el menú de aplicaciones de un Android SO una vez que este se compile para su posterior ejecución.

6.3. Aplicación “Shopping List”

6.3.1. Intención

El objetivo de la aplicación es poder crear listas de compras, donde cada una permitirá el agregado de múltiples ítems o productos, con la capacidad de identificar los mismos

por un nombre, una cantidad, unidad y si el mismo fue comprado o no. Nuevamente se desea modelar una aplicación útil, sencilla e intuitiva.

6.3.2. Mockup y modelo de la aplicación

La ejecución de la aplicación comenzará con la siguiente pantalla principal:

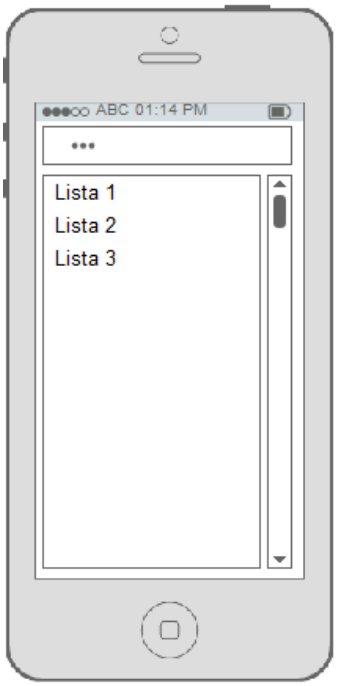

Figura 39 - Pantalla principal de "Shopping List"

En la pantalla se pueden observar algunos componentes:

- Un menú de opciones en la parte superior
- Una lista de Ítems o "Listas de compras". Esta lista contiene un ítem con un nombre, donde cada ítem especifica una Lista de Compras.
- Una barra para realizar scrolling y de esta forma para poder desplazar la lista.

Utilizando el menú superior encontramos las siguientes opciones:

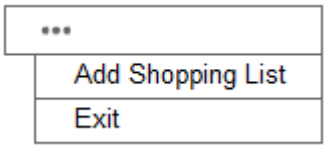

Figura 40 - Menú de opciones de la pantalla principal

El primer ítem es “Add Shopping List”, cuyo objetivo será la de agregar una lista nueva. El segundo pertenece al “Exit”, cuya acción cerrará la aplicación.

La acción de presionar sobre “Add Shopping List” creará una nueva lista y para esto, abrirá la siguiente ventana:

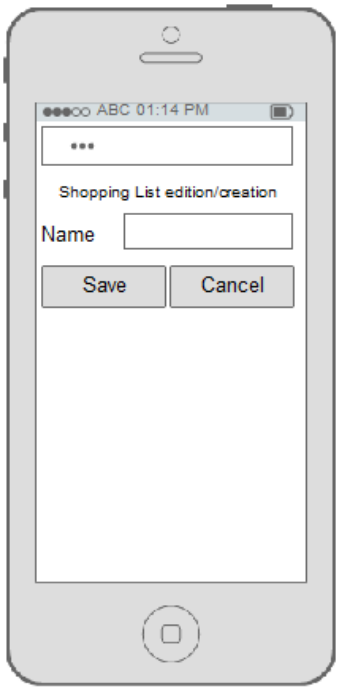

Figura 41 - Pantalla de creación/edición de listas de compras

Como se describe en la figura, para crear la lista, será necesaria la elección de un nombre.

Tanto el botón “Save” como el botón “Cancel”, hará que la pantalla se esconda y el foco vuelva a la pantalla principal. La diferencia entre un botón y el otro es que el primer botón guardará la lista, y el último cancelará la creación.

Una vez creada la lista y presionando sobre la misma recientemente creada se desplegará el siguiente menú contextual:

Figura 42 - Menú contextual invocado a partir de una lista de compras

De esta forma, el menú ofrece tres opciones, “Edit” cuyo objetivo ser el de mostrar la misma pantalla que se utilizó para crear la lista, pero en uso exclusivo para editar. La opción “Delete” mostrará una ventana de confirmación para proceder o no a la eliminación. A continuación se muestra la ventana de confirmación:

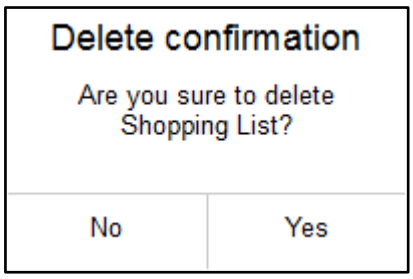

Figura 43 - Ventana de confirmación de eliminación de una lista

Y por último “Show Detail”, donde la elección de esta opción abrirá una nueva ventana para el agregado de productos a la lista:

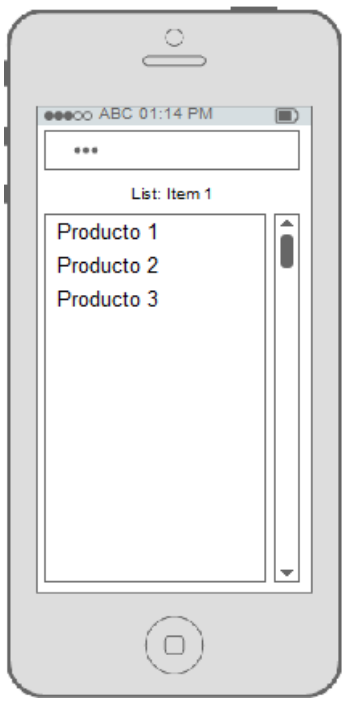

Figura 44 - Pantalla con productos pertenecientes a una lista

En esta última figura se puede observar nuevamente un menú superior, un título que indica el nombre de la lista y una lista de productos. Para poder agregar un producto, se hace uso del menú de opciones, el cual permitirá ejecutar las siguientes acciones:

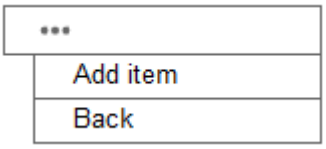

Figura 45 - Menú de opciones de la pantalla de productos

Las acciones descritas en la imagen son: “Add Item” donde se permite agregar un nuevo producto en la lista, y la acción “Back”, el cual hará que el foco retorne a la pantalla de productos.

La funcionalidad “Add Item” mostrará la siguiente pantalla:

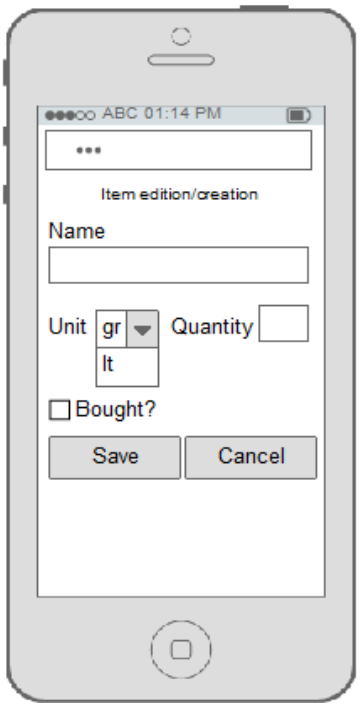

Figura 46 - Pantalla de creación/edición de un producto

En la figura se pueden visualizar los siguientes campos:

- Name: se utilizará para indicar el nombre del producto
- Unit: indicará la unidad en gramos, litros, kilogramos, etc.
- Quantity: define la cantidad de productos de dicho tipo.
- Bought: indica si fue comprado o no.

Finalmente se visualizan dos botones: “Save” y “Cancel”, cuya funcionalidad es la de guardar los cambios o cancelar los mismos. Ambos botones esconden la ventana y el foco regresa a la ventana que contiene la lista de productos.

Cada uno de los productos de la lista de compras se visualiza de la siguiente forma:

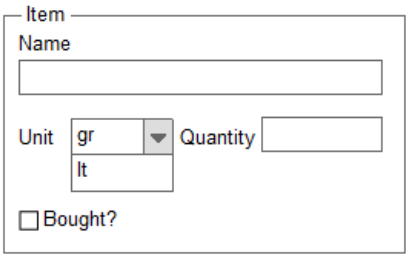

Figura 47 - Composición de un producto

Nuevamente, en esta instancia se puede invocar a otro menú contextual, presionando sobre uno de los productos:

Figura 48 - Menú contextual invocado desde un producto

En esta oportunidad el menú contextual brindará dos opciones: “Edit” en donde presionando sobre dicha opción, se mostrará la misma pantalla de creación de un producto, pero en modo “edición”. Esto servirá para modificar la información contenida. Finalmente encontramos la acción “Delete” la cual elimina el producto y luego retorna el foco a la pantalla de la lista de productos. A continuación se muestra la ventana de confirmación de eliminación de un producto:

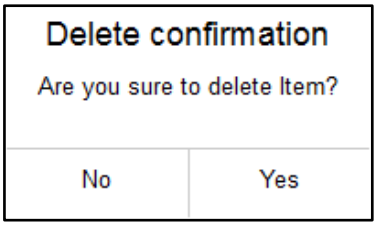

Figura 49 - Ventana de confirmación de eliminación de un producto

A continuación se procede a modelar la aplicación, haciendo uso del Application Diagram. Se obtiene el siguiente modelo, que para su mejor visualización se divide en

dos partes. En la primera de ellas se tiene lo relacionado con listas de compra, que incluye la pantalla que presenta el listado de listas de compra y la de creación y edición de listas, ambas con sus respectivos menús de opciones, menús contextuales y Layouts. En la segunda parte se incluye lo relacionado a los ítems, como la pantalla con el listado de los ítems de una lista de compras y la de creación y edición de los mismos; también con sus respectivos menús de opciones, menús contextuales y Layouts.

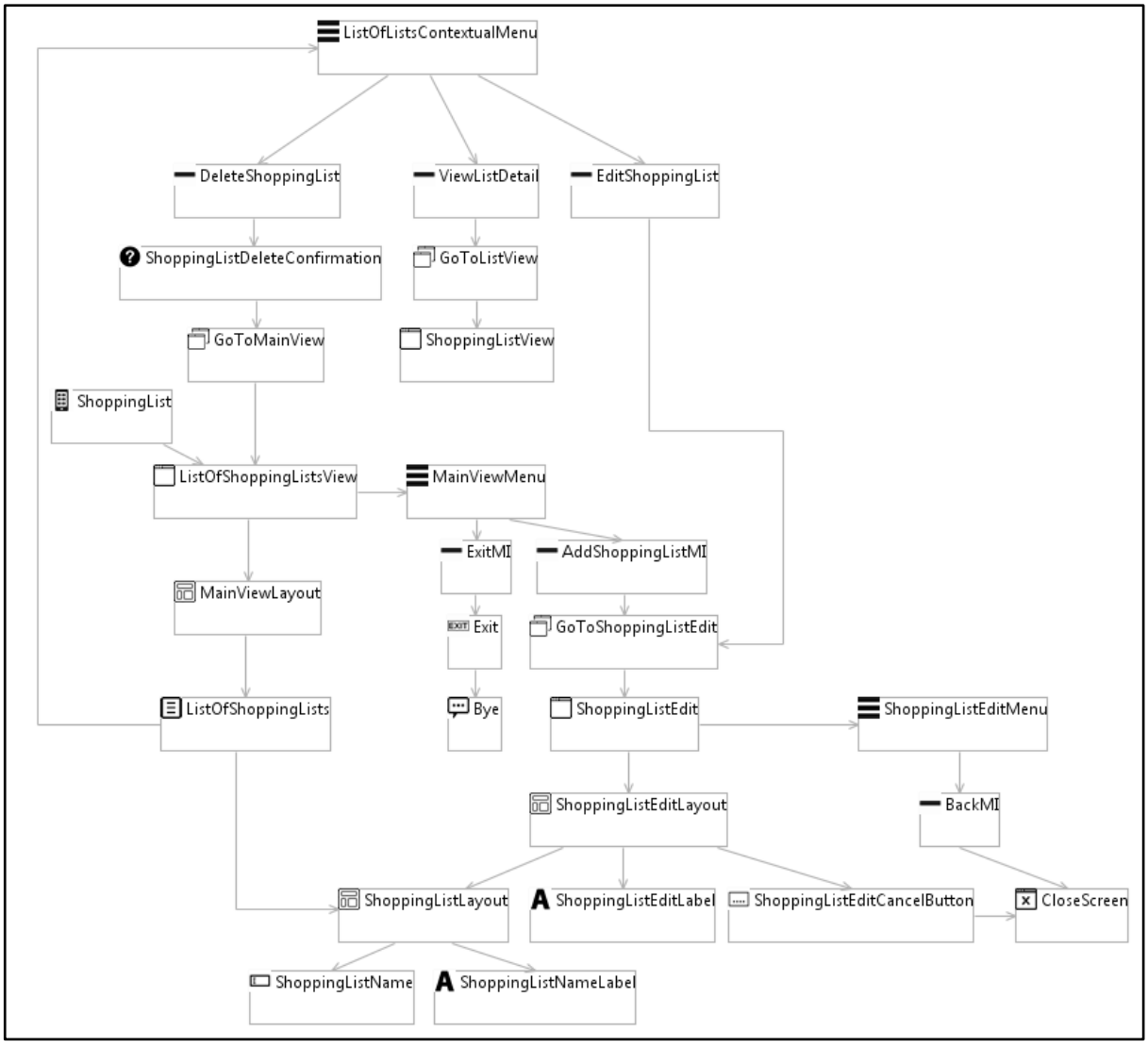

Figura 50 - Diagrama relacionado a las listas de compras

En este primer diagrama del modelo, el Layout *ShoppingListLayout* está configurado como *persistente*.

Figura 51 - Diagrama relacionado con los ítems

Por otro lado, en este diagrama se presentan dos elementos con características a destacar:

- El Layout *Item* es persistente.
- *UnitSpinner* es inicializado con diferentes elementos que representan medidas de unidades, como por ejemplo: gr, kg, piece, lt, cm^3, cm, mt, dozen, etc.

6.3.3. Generación y customización de código

Luego de finalizada la definición del modelo se procede con la generación del código fuente, de la misma forma y con características similares a la aplicación “My Notes”.

Como resultado del proceso se obtiene un proyecto Android listo para ser ejecutado, con las funcionalidades comentadas en puntos anteriores; aunque será necesario realizar algunos ajustes para adaptar o personalizar la siguiente funcionalidad:

- Permitir eliminación y edición de listas de compras.
- Permitir eliminación y edición de ítems.
- Relación de los productos con la lista a la cual pertenecen.
- Obtención de los productos de una lista en determinada.
- Modificación de los estilos.

Eliminación de Listas de Compras

Para la eliminación de una lista de compras en particular, se debe modificar el comportamiento del ítem del menú contextual DeleteShoppingList de ListOfShoppingListsView para que realice la llamada a la operación CRUD[23] deleteShoppingList.

La primera acción a realizar es la edición de la función que se encarga de realizar la llamada para eliminar una lista, para que este reciba el identificador de la lista a eliminar y luego lo envíe a la operación CRUD de borrado.

```
private void actionConfirmedShoppingListDeleteConfirmation(long listId) {
       getContentResolver().delete(DBAdapterProvider.SHOPPINGLISTLAYOUT_URI, null,
       new String[]{String.valueOf(listId)});
}
```

A continuación es necesario recuperar el identificador de la lista a eliminar y enviarlo como parámetro a la función modificada con anterioridad.

```
private void contextualMenuItemClickedDeleteShoppingList(ContextMenuInfo info) {
       [ ... ]
              public void onClick(DialogInterface dialog, int id) {
                     actionConfirmedShoppingListDeleteConfirmation(
                            ((AdapterContextMenuInfo)info).id
                     );
```

[23] CRUD es el acrónimo de Crear, Obtener, Actualizar y Borrar (del original en inglés: Create, Read, Update and Delete). Se usa para referirse a las funciones básicas en bases de datos o la capa de persistencia en un software.

```
            }
        [ ... ]
}
```

Edición de Listas de Compras

Para este otro requerimiento se debe cambiar el comportamiento del ítem del menú contextual EditShoppingList de la clase ListOfShoppingListsView para recuperar y enviar como parámetro el identificador de la lista a modificar a la pantalla ShoppingListEdit.

```
private void contextualMenuItemClickedEditShoppingList(ContextMenuInfo info) {
        Intent intent = new Intent(ListOfShoppingListsView.this,
                                   ShoppingListEdit.class);
        intent.putExtra("shoppingListId", ((AdapterContextMenuInfo)info).id);
        startActivity(intent);
}
```

El siguiente paso consta de la modificación de la clase ShoppingListEdit para que verifique si se le pasó como parámetro el identificador de una lista (en caso de edición) o no (en caso de creación). Si recibe el parámetro se encargará de recuperarla de la base de datos y cargar su contenido en los diferentes Widgets de la pantalla para su edición.

```
protected void onCreate(Bundle savedInstanceState) {
        [ ... ]
        this.shoppingListId = getIntent().getLongExtra("shoppingListId", 0);
        if (this.shoppingListId > 0) {
                this.loadShoppingList();
        }
        [ ... ]
}


private void loadShoppingList() {
        Cursor cursor = getContentResolver().query(
                DBAdapterProvider.SHOPPINGLISTLAYOUT_URI, null, null,
                new String[]{String.valueOf(this.shoppingListId)}, null);
        cursor.moveToFirst();


        String listName = cursor.getString(cursor.getColumnIndex(
```

```
            DBAdapter.COLUMN_SHOPPINGLISTLAYOUT_SHOPPINGLISTLAYOUTSHOPPINGLISTNAME));

        EditText nameEdit = (EditText)findViewById(
                                    R.id.ShoppingListLayout_ShoppingListName);
        nameEdit.setText(listName);
}
```

Por último se modifica el comportamiento del botón “Save” para que se realice una inserción o una actualización de la lista de compras dependiendo si previamente se recibió el identificador de la lista como parámetro.

```
View.OnClickListener shoppingListLayoutInsertListener = new View.OnClickListener(){
        public void onClick(View v){
                [ ... ]
                if (shoppingListId > 0) {
                        int result = getContentResolver().update(
                                DBAdapterProvider.SHOPPINGLISTLAYOUT_URI, contentValues,
                                null, new String[]{String.valueOf(shoppingListId)});
                } else {
                        Uri uri = getContentResolver().insert(
                                DBAdapterProvider.SHOPPINGLISTLAYOUT_URI,contentValues);
                }
                [ ... ]
        }
}
```

Eliminación y edición de ítems

Para permitir la eliminación y la edición de los ítems de una lista de compra se siguen pasos similares que para la eliminación y edición de listas de compras pero sobre las clases de las pantallas relacionadas con el listado y creación/edición de ítems.

Relación de los productos con la lista a la cual pertenecen

Lo que se quiere lograr en este punto es que cada producto sea agregado, es decir que pertenezca, a una lista de compras en particular.

En primer lugar se tiene que agregar a la tabla Item una clave foránea a la tabla ShoppingList y configurarla para realizar un borrado en cascada de todos los ítems al eliminar una lista. Para ello se debe editar la clase DBAdapter.

```
public static final String COLUMN_ITEM_SHOPPINGLISTID = "ShoppingList_ID";

private static final String CREATION_QUERY_ITEM = "CREATE TABLE " + [ ... ] +
       "FOREIGN KEY(" + COLUMN_ITEM_SHOPPINGLISTID +") REFERENCES "
       + TABLE_SHOPPINGLISTLAYOUT + "(" + COLUMN_ID + ") ON DELETE CASCADE);";
```

Seguidamente se debe modificar la clase ItemEdit para que reciba como parámetro el identificador de la lista y lo utilice al momento de crear o editar un ítem.

```
private long shoppingListId;

protected void onCreate(Bundle savedInstanceState) {
       [ ... ]
       this.shoppingListId = getIntent().getLongExtra("shoppingListId", 0);
       [ ... ]
}

View.OnClickListener itemInsertListener = new View.OnClickListener() {
       public void onClick(View v){
              ContentValues contentValues = new ContentValues();
              [ ... ]
              contentValues.put(DBAdapter.COLUMN_ITEM_SHOPPINGLISTID, shoppingListId);
              [ ... ]
       }
}
```

Obtención de los productos de una lista en determinada

El objetivo de esta funcionalidad es recuperar y listar únicamente los ítems correspondientes a una lista de compras cuando se visualiza su detalle.

La primera de las modificaciones a realizar es el agregado de un parámetro a la función getAllItem de la clase DBAdapter, que reciba el identificador de la lista de la cual se quieren recuperar los ítems.

```
public Cursor getAllItem(long shoppingListId){
       return getWritableDatabase().rawQuery("SELECT id _id,* FROM " + TABLE_ITEM +
       " WHERE " + COLUMN_ITEM_SHOPPINGLISTID + "=" + shoppingListId, null);
}
```

Luego, se modifica la clase ShoppingListView para adjuntar el identificador de la lista de la cual se quieren a recuperar los ítems al momento de realizar la llamada. La clase ShoppingListView recibe el identificador de la lista que se quiere visualizar el detalle de la misma manera que se hizo para la edición de una lista.

```
public Loader<Cursor> onCreateLoader(int loaderID, Bundle bundle) {
       return new CursorLoader(this, DBAdapterProvider.getUriForLoader(loaderID),
              null, null, new String[]{String.valueOf(shoppingListId)}, null);
}
```

Modificación de los estilos

Los cambios que se realizan para la estilización de la aplicación son casi completamente mediante la edición y creación de archivos XML.

En una primera etapa se reacomodan los Layouts contenedores de botones de inserción o cancelación para que queden uno al lado del otro, en vez de uno debajo del otro.

Por otro lado, se crea el archivo “rounded_button.xml” en el que se define el formato de presentación de todos los botones de la aplicación. Algunos de los cambios que se realizan sobre los botones afectan el color de fondo y sus esquinas redondeadas.

A su vez se modifica el archivo styles.xml para personalizar, entre otras cosas, los estilos definidos en el modelo. El ejemplo más claro es el caso del estilo *title*.

Finalmente se generan los íconos de la aplicación para cada una de las resoluciones disponibles.

6.4. Ejecución de los ejemplos

Finalizado desarrollo de las dos aplicaciones, las mismas estarán listas para ser ejecutadas. Para realizar la prueba de las aplicaciones se deberá contar con los siguientes requisitos:

- Tener un cable USB que conecte el dispositivo con Android OS con la PC.
- Tener instalado el driver del dispositivo instalado en la PC.

- Es requisito esencial tener configurado el dispositivo en modo “Depuración por USB”.

Cumpliendo los requisitos mencionados, y teniendo el dispositivo conectado a la PC se procede a ejecutar “Run As > Android Application” sobre el proyecto. Se debe tener en cuenta que para el éxito de la ejecución, además deberemos tener el dispositivo desbloqueado, o sea, con la pantalla encendida o activa. Luego, Eclipse IDE realiza una instalación del APK sobre el dispositivo, lo que significa que la aplicación quedará instalada hasta que el usuario explícitamente desee eliminarla. Una vez instalada, se busca en el menú de aplicaciones y se ejecuta.

La aplicación por lo general se mantiene activa en segundo plano, salvo que el usuario explícitamente desee cerrarlo desde el menú de opciones que provee la aplicación.

7. Conclusiones y Trabajos futuros

7.1. Conclusiones

Es evidente que el desarrollo de software dirigido por modelos se encuentra en constante crecimiento, y esto queda visible debido a la gran cantidad de herramientas que existen para modelar aplicaciones las cuales permiten transformar el modelo en código fuente.

En el mundo de las comunicaciones, sin embargo, para el caso específico de las plataformas Android, las herramientas existentes, con el adicional de ser limitadas en cuanto al código o binario generado, no permiten modelar y/o representar en forma completa una aplicación. Esto es, ya sea porque las que existen generan binarios y/o porque son extremadamente limitadas a la hora de permitir un modelo regular de aplicación.

Por estos motivos se ha decidido encarar el desarrollo de una herramienta que permita modelar una aplicación en forma íntegra para luego transformarlo en código fuente Android, con el fin de que el mismo pueda ser manipulado para lo que oportunamente se necesite.

En este informe se expone la potencialidad de MDD, introduciendo el concepto desde lo más básico para luego desarrollar una herramienta y demostrar sus capacidades con ejemplos prácticos.

A través del desarrollo de esta herramienta, se logró efectivizar de una manera lo suficientemente sencilla y amigable, la generación de código fuente teniendo como origen un modelo definido por el usuario. La creación de un metamodelo rico y expresivo fue positiva por el hecho de haber podido representar la funcionalidad de aplicaciones de diferentes tipos de forma simple y natural, como por ejemplo en los casos de estudio.

Se debe reconocer que el uso de Eclipse IDE ha facilitado el desarrollo de la misma, ya que por medio de la infinidad de plugins favoreció que el desarrollo resulte lo más sencillo y limpio posible.

El código Android generado fue probado y resulta compatible para la versión que se genera. Se han generado sus binarios y se ha probado en diferentes dispositivos con el fin de comprobar fehacientemente que la aplicación funciona en sintonía como el usuario lo modeló.

Finalmente se puede confirmar que se cumplen con el objetivo de desarrollar una herramienta que genera código Android, de software libre, cuyo metamodelo es rico y expresivo, y como punto extra, donde la generación del modelo es totalmente independiente del código fuente Android a generar. Adicionalmente se genera documentación suficiente como para su posterior modificación.

7.2. Trabajos futuros

En base a los conceptos planteados a lo largo de la tesina y a los resultados obtenidos, se proponen las siguientes líneas como trabajos futuros:

- Ampliación de la cantidad de Widgets disponibles: se propone la creación de otros Widgets o componentes con el fin de tener un metamodelo todavía más rico y expresivo. De esta forma el abanico de aplicaciones a crear será más extenso y al mismo tiempo brindará un nivel de complejidad más interesante. También se propone la creación de componentes o módulos cuyo objetivo sea el de agrupación de funcionalidad, por ejemplo:

    - ABM de algún tipo de “entidad” existente: El objetivo será el de, dada una entidad, poder generar un ABM y que el mismo módulo, genere las pantallas y Widgets necesarios para poder realizar este tipo de operaciones.
    - Interfaz de conexión con otros sistemas por medio de servicios RESTful, HATEOAS o Web Services: Se pretende establecer algún componente que mediante una sencilla comunicación no sólo provea de una forma de

comunicación con algún servicio, sino, también el hecho de poder manipular la información obtenida o a brindar.

- API de comunicación con el hardware del dispositivo en el cual se esté corriendo: El objetivo de esta funcionalidad será la de proveer un servicio para la manipulación del hardware del dispositivo. En el caso de un smartphone, poder comunicarse con el GPS, Acelerómetro, etc.

- Generación de código para otras plataformas: se pretende que se desarrollen módulos, los cuales generarán código fuente para plataformas como por ejemplo iOS, Blackberry, etc. Lo mismo es válido para aplicaciones desktop, generando código PHP, Ruby, etc. Gracias a la modularización del código, solamente bastará con, en base al metamodelo obtenido, crear una librería de transformación específica para dicho lenguaje de programación.

- Regeneración del código sin perder cambios realizados: se propone adaptar la actual herramienta para que permita mantener los cambios realizados por el usuario al momento de regenerar el código. De esta forma, en una supuesta regeneración, los cambios del usuario se mantendrían y se integrarían con el nuevo código derivado del modelo.

8. Referencias

- Desarrollo de software dirigido por modelos: conceptos teóricos y su aplicación práctica. Claudia Pons; Roxana Giandini; Gabriela Pérez. 1a ed. - La Plata: Universidad Nacional de La Plata, 2010.
- Testing basado en modelos. Especificación gráfica y derivación automática del código. Anselmo Francisco Abadía, Juan Ignacio Barisich. Tesina de Grado. Noviembre 2009.
- Graphical Modeling Framework [Online] / Autor: Eclipse - http://www.eclipse.org/modeling/gmp/
- Eclipse Modeling Framework [Online] / Autor: Eclipse - http://www.eclipse.org/modeling/emf/?project=emf
- Graphical Editing Framework [Online] / Autor: Eclipse - http://www.eclipse.org/gef/
- Acceleo [Online] / Autor: Acceleo - http://www.eclipse.org/acceleo/
- Guías de la API de Android [Online] /Autor: Android - http://developer.android.com/intl/es/guide/index.html
- AppInventor [Online] / Autor: MIT- http://appinventor.mit.edu/
- DroidBreeder [Online] / Autor: DroidBreeder - http://droidbreeder.sourceforge.net/
- DroidDraw [Online] / Autor: DroidDraw - http://www.droiddraw.org/
- Eclipse Platform Technical Overview [Online] / Autor: Eclipse - https://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html
- Kermeta.org [Online] / Autor: Kermeta - http://www.kermeta.org
- Graphical Modeling Framework Tooling [Online] / Autor: Eclipse - http://www.eclipse.org/gmf-tooling/
- EuGENia [Online] / Autor: EuGENia - http://www.eclipse.org/epsilon/doc/eugenia/
- Dashboard Android Development [Online] / Autor: Android - https://developer.android.com/intl/es/about/dashboards/index.html