Prólogo

Mi interés en el campo de la ingeniería del software se debe a que antes de los estudios universitarios ya había estudiado informática pero siempre desde un punto de vista práctico. La ingeniería del software me ha abierto una nueva forma de concebir la creación de software, más cercano a la teoría y de forma racional.

Hay varios motivos por los que me he decantado para hacer este proyecto frente a otras alternativas como pudieran ser una aplicación o algún tipo de convenio con empresa. Una aplicación sólo serviría para reforzar la parte de conocimientos prácticos que ya he adquirido en estos estudios y en estudios previos. Seguramente este tipo de proyectos me resultarían más sencillos pero no serían una experiencia tan enriquecedora como lo es este proyecto. La decisión de no hacerlo con una empresa fue algo más compleja ya que estaba descartando una posibilidad de encontrar un puesto de trabajo al terminar la carrera. Finalmente he decidido que quiero seguir estudiando y posiblemente hacer un doctorado, por tanto me resulta más atractivo realizar un proyecto de estudio de una tecnología con la universidad.

He decidido especializarme en la rama de la ingeniería del software, por tanto me parece que dedicar mi proyecto de fin de carrera es una buena forma de acercamiento.

1. Introducción

El proyecto servirá para automatizar una etapa del diseño de software, la *asignación de responsabilidades*. La automatización se hará en base a las preferencias y especificaciones del usuario. Para llevar a cabo dicha automatización se usará una tecnología de apoyo, *AndroMDA*.

Automatizando la etapa de asignación de responsabilidades nos acercamos a una nueva forma de crear software, en la cual el desarrollador trabaja a un nivel conceptual y queda aislado de la dependencia tecnológica de éste.

1.1 Antes de empezar...

¿Qué es una *responsabilidad*? Una responsabilidad en el ámbito de la ingeniería del software se podría definir como cada una de las tareas asignadas a los distintos elementos que componen el software.

¿Qué es una *capa*? Una capa es un módulo o conjunto de módulos independiente que se comunica u ofrece servicios a otras capas.

¿Qué es la asignación de responsabilidades a capas? La asignación de responsabilidades a capas es una de las etapas del diseño de software, en ella se decide qué capa de la arquitectura del sistema es la más adecuada para encargarse de asegurar que una funcionalidad se lleva a cabo de forma correcta.

La toma de decisiones se hace en base a una serie de estrategias, reglas y

preferencias establecidas por el diseñador, que en nuestro caso es el usuario del programa resultante del proyecto.

En el transcurso de esta memoria se responderá a estas preguntas de forma más amplia.

1.2 Objetivos del proyecto

El proyecto está dividido en tres grandes apartados. El primero, un trabajo de *investigación tecnológica*, en el cual se pretende adaptar una tecnología existente, *AndroMDA*, para que sirva de apoyo al proceso de detección de responsabilidades, para ello será necesario conocer su funcionamiento y las posibilidades que nos ofrece, así como determinar qué información adicional necesitamos y qué mecanismos nos ofrece *AndroMDA* para poder detectar las responsabilidades. En este apartado también se incluye el estudio de los diversos lenguajes de programación y de *scripting* necesarios para el desarrollo del proyecto.

El segundo apartado es determinar qué *responsabilidades* serán tratadas. Este apartado se centra en el estudio de las distintas alternativas de gestión tanto de las mencionadas responsabilidades como de su posterior asignación a capas, así como en la decisión de qué tratamiento o tratamientos tiene cada responsabilidad y qué criterios no funcionales se usarán para determinar la asignación de éstas. También se pretende ofrecer mecanismos y/o documentación para incrementar el número de

responsabilidades y sus tratamientos.

Finalmente se implementará, usando *AndroMDA,* la asignación de automática responsabilidades. Como resultado, dada una especificación de partida que cumple los requisitos establecidos en el apartado anterior, se obtendrá una lista de responsabilidades y sus tratamientos.

1.3 ¿Por qué AndroMDA?

Se ha optado por adaptar AndroMDA frente a la opción de desarrollar una plataforma propia para la detección de responsabilidades ya que se trata de una plataforma conocida, con una comunidad de usuarios y es de libre distribución.

AndroMDA es una plataforma de desarrollo de aplicaciones en el contexto de las tecnologías de soporte a el *Model Driven Architecture (MDA),* cuya finalidad es generar código en diversos lenguajes de programación a partir de diagramas UML.

El motor de esta plataforma nos permite interpretar diagramas UML estándar. Gracias a su alta extensibilidad podemos adaptarlo para detectar responsabilidades.

1.4 Alcance del proyecto

Este proyecto pretende ser un primer paso hacia la automatización

completa del proceso de asignación de responsabilidades a capas. La asignación se hará en base a las preferencias declaradas por el usuario y no a partir del análisis de los diagramas, esta posibilidad se podría implementar en los siguientes pasos.

2. Arquitectura en capas

La arquitectura en capas es una de las arquitecturas más usadas en la creación de software, su objetivo es separar el programa en unidades, estas unidades son las capas. Cada capa representa una colección cohesiva de servicios.

Los objetivos de organizar las aplicaciones en capas y sus ventajas son:

- Crear unidades de software lo más independiente posible, permitiendo así reducir el acoplamiento entre capas.

- Hacer que cada capa tenga una finalidad concreta dentro de la aplicación, permite un mantenimiento más eficaz y mejorar la reusabilidad de las capas. Además se consiguen resultados de más calidad ya que se pueden destinar especialistas al desarrollo de cada capa.

- La separación en capas nos otorga los beneficios de usar una estrategia de dividir y vencer.

Desde el punto de vista arquitectónico, la propiedad fundamental de las capas es la forma en que éstas interaccionan entre ellas o el uso entre unas y otras. Se recomienda seguir una arquitectura estricta, que permite a una capa comunicarse sólo con sus adyacentes, en la figura siguiente podemos ver un ejemplo de lo que no sería una arquitectura en capas estricta. Para entender esto podemos pensar en las pieles de una cebolla,

cada capa de la cebolla está en contacto con la anterior y la siguiente, de forma que sólo puede interaccionar con éstas y nunca saltándose capas. En este proyecto nos centraremos en arquitecturas en capas estrictas.

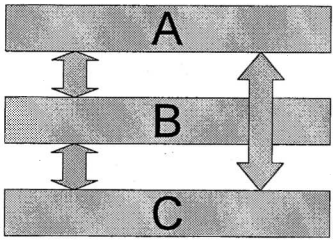

Fig. 1: Modelo no estricto

A continuación vemos diversas notaciones para representar la misma arquitectura en capas. Nótese que en el segundo y tercer diagrama deberían indicarse textualmente en qué sentidos está permitida la comunicación entre capas si se desea añadir alguna limitación.

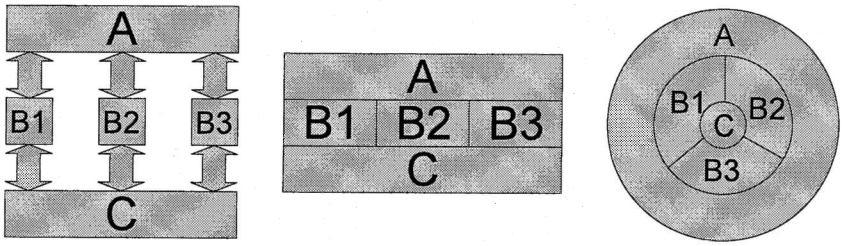

Fig. 2: Notaciones

Las arquitecturas en capas pueden tener formas como la expuesta en la figura 3. En este caso el software de A, B y C puede usar el software de D y de la misma forma D puede usar el software de A, B y C. En estos casos

se suele limitar la comunicación en uno de los sentidos ya que en caso contrario crearía una arquitectura poco sostenible. Es trabajo del diseñador especificar este tipo de limitaciones de uso.

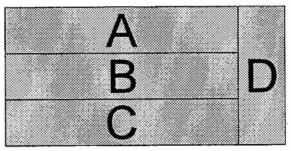

Fig. 3: Cuatro capas

2.1 Arquitectura en tres capas

Cuando aplicamos los conceptos de las arquitecturas en capas a un sistema de información (SI), obtenemos tres capas: presentación, dominio y persistencia. A este modelo se le suele llamar de forma genérica como "Arquitectura en tres capas".

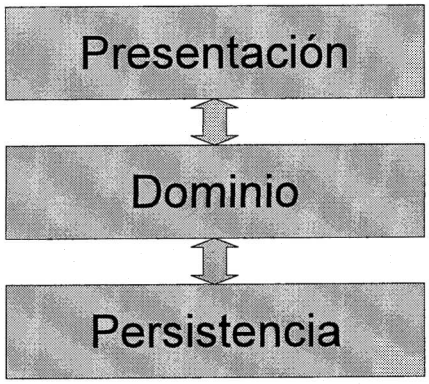

Fig. 4: Arquitectura en tres capas

La capa de presentación es la capa superior. Contiene todos los elementos de software que son visibles para el usuario como pueden ser las vistas o la navegación dentro de la aplicación. Por ejemplo, normalmente esta capa se encarga de gestionar el idioma de la interfaz de usuario y de conocer qué opciones están habilitadas o deshabilitadas.

La capa de dominio es el núcleo de la aplicación. En esta capa encontramos los algoritmos y las estructuras de datos en memoria durante las transacciones o sesiones. Además hace de nexo de unión entre las otras dos capas.

La capa de persistencia se encarga gestionar la comunicación entre la capa de dominio y el sistema gestor de bases de datos (SGBD) o el sistema de ficheros si la aplicación lo requiere. Esta capa conoce las estructuras de datos y como deben hacerse persistentes.

La navegabilidad entre las capas puede ser descendente, ascendente o como se muestra en la figura 4, en ambas direcciones.

3. Requisitos

3.1 Requisitos funcionales

- Adaptar AndroMDA para detectar las responsabilidades en el modelo UML.

- Detectar las responsabilidades de un diagrama de clases UML. Las responsabilidades tratadas son las responsabilidades de clave, cardinalidad de asociaciones y algunas restricciones textuales, de éstas últimas trataremos precondiciones y postcondiciones de operaciones.

- Asignar los tratamientos adecuados a cada responsabilidad según las estrategias y criterios especificadas por el usuario. Cada responsabilidad tendrá uno o varios tratamientos asignados.

- Generar una lista de responsabilidades y sus tratamientos en un formato estándar que permita su reusabilidad.

3.2 Requisitos no funcionales

En este proyecto se persiguen dos requisitos no funcionales, la *extensibilidad* y la *independencia tecnológica*. El motivo de tener estos requisitos es que este proyecto es un primer paso hacia la automatización de la asignación de responsabilidades, de forma que no nos interesa hacer un proyecto dependiente de una tecnología. En los siguientes pasos que se den es posible que se decida cambiarla o ampliarla.

Ambos requisitos se focalizarán en los siguientes apartados:

- UML básico y estándar, de esta forma evitaremos crear lazos tecnológicos entre este proyecto y los siguientes.
- Facilitar el proceso de añadir nuevas responsabilidades y tratamientos para éstas.
- Separación conceptual de la parte del proyecto ligada al AndroMDA y la parte del proyecto referente a la asignación de responsabilidades a capas.
- Documentación suficiente para que otra persona pueda retomar el proyecto.

4. AndroMDA

AndroMDA[1] es un entorno de trabajo de código abierto, que sigue el paradigma de arquitectura dirigida por modelos (*Model Driven Architecture*). Soporta la transformación de los modelos hechos con herramientas de diseño *UML* en componentes de las plataformas más conocidas (*J2EE*, *Spring*, *.NET*).

Dispone de una amplia gama de *cartridges*[2] que permiten generar código para las tecnologías más usadas hoy en día (*Axis*, *jBPM*, *Struts*, *JSF*, *Spring* y *Hibernate*).

Permite crear *cartridges* o adaptar uno existente usando el *meta cartridge*, un *cartridge* para desarrollar *cartridges*.

4.1 Model Driven Architecture (MDA)

La Arquitectura Dirigida por Modelos es un acercamiento al diseño de software propuesto y patrocinado por el *Object Management Group (OMG)*.

La idea de la arquitectura *MDA* consiste en desarrollar el software partiendo de un modelo, normalmente especificado en *UML*, de forma que el dominio y la lógica de la aplicación quedan separadas de la tecnología usada, facilitando que éstos puedan ser alterados independientemente.

1 Se pronuncia "Andrómeda"
2 Un cartridge es similar a un plugin, un incremento en las funcionalidades del programa.

El modelo inicial es el Modelo Independiente de la Plataforma (*PIM*), que es una abstracción del funcionamiento del software. Añadiendo información adicional en forma de marcas al modelo, obtendremos el PIM con marcas, estas marcas reflejan elementos propios de una plataforma específica. Finalmente aplicando la función de transformación obtendremos el Modelo Específico de la Plataforma (*PSM*), a partir del modelo *PSM* podemos generar código para la plataforma escogida.

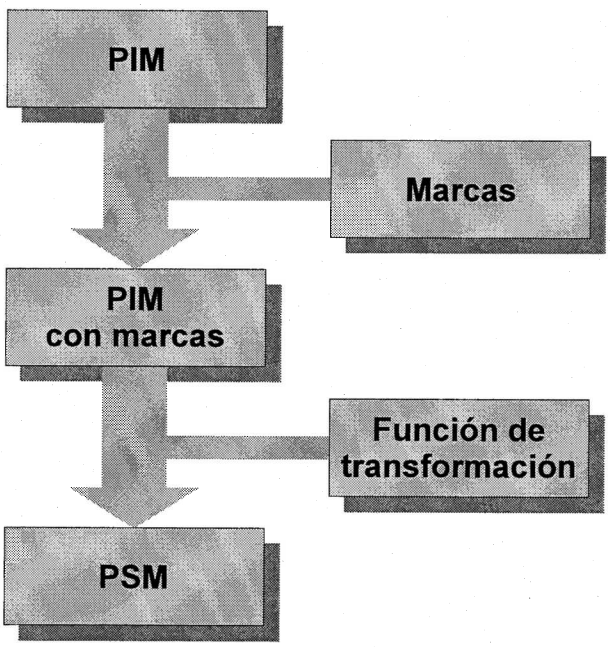

Fig. 1: Diagrama de transformación MDA

La arquitectura *MDA* persigue tres objetivos básicos: la portabilidad, la interoperabilidad y la reusabilidad, todos ellos a través de la separación arquitectónica.

4.2 Historia de AndroMDA

En Julio del 2002, se creó *UML2EJB,* el proyecto predecesor de *AndroMDA*. Se llamó *UML2EJB* porque la tecnología que más se demandaba para programar era *Enterprise JavaBeans (EJB) 2.0*, pero sólo se podía escribir el código a mano o bien usando asistentes de algún Entorno de Desarrollo Integrado (*IDE*). *UML2EJB* fue desarrollado para generar el código de los ejemplos del libro "*Enterprise JavaBeans GE-PACKT*". En este proyecto no existían el concepto *cartridge* ni de *metafacade*[3].

En Febrero del 2003, el proyecto *AndroMDA* fue aprobado en *SourceForge*. Los cambios que se introdujeron en la arquitectura de la plataforma hizo posible que *AndroMDA* creciera. Se dio soporte para cargar modelos *UML* desde varias herramientas CASE. Se introdujo el concepto de *cartridge*, el cual hizo que *AndroMDA* fuera más extensible. Junto con la primera versión de *AndroMDA* también se hizo el *cartridge* de *Hibernate*.

En Julio del 2003, se lanzó *AndroMDA 2.0* junto con los *cartridges* para *EJBs*, *Hibernate*, *Struts* y *Java*.

En Mayo del 2005, *AndroMDA 3.0* se hizo estable. En esta versión se hizo el código más robusto, pues hasta ese momento *AndroMDA* era una plataforma de desarrollo que fácilmente dejaba de funcionar. Aparte de las mejoras en los *cartridges* mantenidos por los desarrolladores de

3 Un metafacade es una extensión de las funcionalidades ofrecidas por un elemento del modelo.

AndroMDA cabe destacar dos nuevos aspectos, la introducción del concepto de *metafacade* junto con la aparición del *meta cartridge,* y el soporte de traducción de *Object Constraint Language (OCL*). *OCL* permite introducir comprobaciones en el modelo UML en tiempo de generación de código.

La versión actual de *AndroMDA* es la 3.2, que da soporte a las especificaciones en *UML* 2.0.

En este proyecto de fin de carrera inicialmente se trabajó con la versión 3.1 y posteriormente con la versión en desarrollo de *AndroMDA* 3.2, ya que se requería soporte de *UML* 2.0. Actualmente se está trabajando con la versión final de *AndroMDA* 3.2.

4.3 Aspectos destacados de AndroMDA

- Diseño modular: Prácticamente todas las partes de *AndroMDA* son intercambiables o se pueden quitar dependiendo de las necesidades del usuario.

- Soporte para herramientas comerciales de diseño *UML* como *MagicDraw*, *Poseidon y Enterprise Architect*.

- Soporte para las especificaciones *UML* 1.4 y 2.0 mediante un *metamodelo*[4]. Alternativamente, se puede diseñar otro *metamodelo*

4 Un metamodelo es un modelo que define la forma en que se representa otro modelo.

y generar código desde modelos basados en éste.

- Validación de los modelos de entrada usando *OCL* en las clases del *metamodelo*. Dispone de restricciones preconfiguradas para evitar los errores más comunes en el modelado. Por ejemplo, no pueden existir dos clases con el mismo nombre.

- Posibilidad de generar cualquier tipo de salida en formato texto usando plantillas (código fuente, *scripts* de bases de datos, páginas web, etc.). Las plantillas se pueden definir con lenguajes de plantillas como *Velocity* y *FreeMarker*.

- Suministro de *cartridges* para las arquitecturas más comunes (*EJB*, *Spring*, *Hibernate*, *Struts*, *JSF*, *Axis*, *jBPM*).

4.4 Conceptos relacionados con AndroMDA

4.4.1 Maven

Maven es un gestor de proyectos desarrollado por la fundación *Apache*, *AndroMDA* usa este gestor para su propio desarrollo y para los proyectos desarrollados con *AndroMDA*. Aunque su uso no es obligatorio, facilita considerablemente el trabajo de crear, modificar, configurar y compilar proyectos, además de permitir la integración de las diversas tecnologías usadas por *AndroMDA*.

4.4.2 Plantillas (Templates)

Las plantillas de *AndroMDA* permiten al desarrollador de *cartridges* definir cómo será la salida. Actualmente *AndroMDA* trabaja sobre todo con el lenguaje de plantillas *Velocity*, aunque tiene soporte para otros lenguajes. Dentro de estas plantillas tenemos acceso a las propiedades del *cartridge* y a los elementos del modelo.

En el ejemplo vemos una plantilla que ha tomado un conjunto de clases del modelo en la variable "$classes". Por cada clase obtiene todas sus responsabilidades y por cada responsabilidad, escribe los datos en formato *XML*.

```
#set ($generatedFile = "responsibilities.xml")
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="responsibilities.xsl"?>
<responsibilities>
#foreach($class in $classes)
#foreach($responsibility in $class.Responsibilities)
      <responsibility>
            <name>
                  $responsibility.Name
            </name>
            <modelElement>
                  $responsibility.ModelElement
            </modelElement>
            <modelElementName>
                  $responsibility.ModelElementName
            </modelElementName>
            <layer>
                  $responsibility.Layer
            </layer>
            <description>
                  $responsibility.Description
            </description>
      </responsibility>
#end #end
</responsibilities>
```

Código 1: Ejemplo de plantilla usando Velocity.

4.4.3 Stereotype

En *UML*, un *stereotype* es un concepto usado para encapsular comportamientos. En *AndroMDA* se usa como forma de comunicación entre la especificación y el diseño del software.

Los *stereotypes* son denotados entre los caracteres '<<' y '>>' o bien usando un icono.

Un *stereotype* nos indica el tipo o el significado de un elemento, por ejemplo en un diagrama de clases podemos marcar con el *stereotype* <<constructor>> al método de cada clase encargado de crear la instancia.

4.4.4 Tagged value

Un *tagged value* viene a ser como una variable del modelo. Cada elemento del modelo puede tener un valor para esa variable. Sirven para dar información extra y definir propiedades.

4.4.5 Metafacade

Cuando usamos *AndroMDA* cada elemento de un modelo se transforma en una instancia de una de las clases del metamodelo, por ejemplo un atributo del modelo de partida se transforma en una instancia de la clase "Atributo" del metamodelo. De esta forma podemos navegar por el modelo, una vez aplicada la transformación, con un nivel de abstracción más elevado. Pero esto no siempre es suficiente ya que el estándar *UML*

no es capaz de representar todas las situaciones. Siguiendo con el ejemplo anterior, en *UML* no podemos saber si un atributo es clave o no. Para solucionar esto indicaremos mediante un *stereotype* que ese atributo es clave.

El concepto de *metafacade* está muy ligado al concepto de *stereotype*, es la forma que nos ofrece *AndroMDA* para añadir nuevas funcionalidades a los *cartridges*. Un *metafacade* puede estar asociado con uno o más *stereotypes*. Por cada *stereotype* que usemos en nuestro *cartridge* podemos crear un *metafacade*.

Un *metafacade* es una extensión de una clase del metamodelo o una subclase de otro *metafacade*. El *metafacade* del ejemplo anterior sería una subclase del *metafacade* "AttributeFacade" de *AndroMDA* (v. fig.2). *AndroMDA* tiene hechos los *metafacades* que extienden los elementos del metamodelo. Esta jerarquía de clases permite saber de qué tipo de atributo se trata. En los *metafacades* se puede añadir atributos, métodos y restricciones en lenguaje *OCL*.

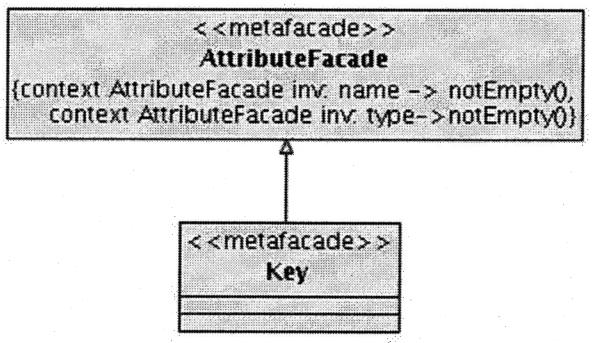

Fig. 2: Ejemplo de metafacade.

Desde el punto de vista de la arquitectura *MDA*, el conjunto de *metafacades* de un *cartridge* representa la función de transformación que genera el *PSM*. En *AndroMDA* el modelo *PSM* reside en memoria durante la ejecución del *cartridge.*

Cuando usamos *metafacades*, las plantillas se simplifican ya que todas las comprobaciones y transformaciones quedan centralizadas en los objetos *Java* resultantes de crear los *metafacades*.

4.5 Cartridges

Los *cartridges* proveen a *AndroMDA* la capacidad de procesar elementos del modelo. Para que *AndroMDA* sepa que elemento debe procesar hay que indicar un *metafacade* o bien un *stereotype*. Los cartridges procesan estos elementos usando las plantillas.

4.5.1 Cómo se usan los cartridges

Para poder usar un *cartridge* en un proyecto, usando *maven,* simplemente hay que añadir como dependencias el *plugin* de *maven* y los *cartridges* al fichero *"project.xml"* del proyecto.

```
<dependencies>
      ...
      <dependency>
            <groupId>${pom.groupId}</groupId>
            <artifactId>maven-andromda-plugin</artifactId>
            <version>${andromda.version}</version>
            <type>plugin</type>
      </dependency>
      <dependency>
```

```
            <groupId>${pom.groupId}</groupId>
            <artifactId>andromda-arc-cartridge</artifactId>
            <version>${andromda.version}</version>
            <type>jar</type>
        </dependency>
        ...
</dependencies>
```

Código 2: Ejemplo de "project.xml".

Como usuario del *cartridge*, *AndroMDA* permite un alto grado de adaptabilidad, esto se consigue mediante las opciones de substitución de elementos de un *cartridge ("mergeLocation")* y la de mezcla mediante etiquetas ("mergeMappingsUri"). Ambas opciones se definen en el fichero *"<proyecto>/mda/conf/andromda.xml"*.

```
...
<namespace name="arc">
      <properties>
            ...
            <property name="mergeLocation">
                  ${maven.src.dir}/custom
            </property>
            <property name="mergeMappingsUri">
                  file:${maven.conf.dir}/mappings/ArcMappings.xml
            </property>
            ...
      </properties>
</namespace>
...
```

Código 3: Ejemplo de "mda/conf/andromda.xml".

La propiedad *"mergeLocation"* es un directorio con la misma estructura que el *cartridge* (el *cartridge* en este caso sería *"arc"*, acrónimo de Asignación de Responsabilidades a Capas), de forma que los elementos en el directorio especificado tendrían preferencia sobre los elementos del propio *cartridge*.

La propiedad "*mergeMappingsUri*" es un fichero que contiene referencias a etiquetas del *cartridge* y su tratamiento, cada una de estas etiquetas está preparada por el *cartridge* para adaptarlo a las necesidades del usuario.

Al contrario que en el caso de "*mergeLocation*", donde es el usuario quien adapta el *cartridge*, cuando usamos la propiedad "*mergeMappingsUri*" es el propio *cartridge* quien está preparado para adaptarse a las necesidades del usuario. En el ejemplo siguiente se puede ver cómo el *cartridge* está preparado para que el usuario especifique sus propiedades.

```
<mappings name="ArcMappings">
      ...
      <mapping>
            <from>
                  <![CDATA[<!-- cartridge-property merge-point-->]]>
            </from>
            <to>
                  <![CDATA[<property reference="enableARCTest"/>]]>
            </to>
      </mapping>
      ...
</mappings>
```

Código 4: Ejemplo de "ArcMappings.xml".

4.5.2 Cómo se crean los cartridges

Un cartridge de *AndroMDA* está compuesto por cinco elementos:

- El descriptor del cartridge
- El descriptor de namespace

- Las plantillas
- El descriptor de metafacades
- Clases Java de los metafacades y clases auxiliares de apoyo.

Los tres primeros elementos son necesarios para cualquier cartridge, mientras que los dos últimos son opcionales.

4.5.2.1 El descriptor del cartridge

El descriptor del *cartridge* sirve para que *AndroMDA* pueda determinar qué funcionalidades tiene el *cartridge*, qué *metafacades* usa, *stereotypes*, plantillas, propiedades, etc. El descriptor del *cartridge* tiene que estar en el directorio *"META-INF/andromda"* y debe llamarse *"cartridge.xml"*. En el siguiente código se muestra un ejemplo y a continuación se detalla su contenido.

```
<cartridge>
      <templateEngine className="VelocityTemplateEngine">
            <macrolibrary name="templates/lib.vm"/>
      </templateEngine>

      <templateObject
            name="stringUtils"
            className="org.apache.commons.lang.StringUtils"/>
      <!-- cartridge-templateObject merge-point-->

      <resource
            path="templates/responsibilities.xsl"
            outputPattern="{0}"
            outlet="responsibilities"
            overwrite="false"/>
      <!-- cartridge-resource merge-point -->

      <property reference="enableTemplating"/>
```

```
    <!-- cartridge-property merge-point-->

    <condition name="enableTemplating">
        $enableTemplating.equalsIgnoreCase("true")
    </condition>
    <!-- condition merge-point-->

    <template
        path="templates/responsibilities.vsl"
        outputPattern="$generatedFile"
        outlet="responsibilities"
        overwrite="true"
        outputToSingleFile="true">
        <modelElements variable="classes">
            <modelElement>
                <type name=
    "org.andromda.cartridges.arc.metafacades.ClassLogicImpl"/>
            </modelElement>
        </modelElements>
    </template>
    <!-- cartridge-template merge-point -->
</cartridge>
```

Código 5: Ejemplo de "cartridge.xml".

Dentro del apartado de *templateEngines* podemos escoger qué motor de plantillas se usará en el *cartridge*, esto permite cambiar el lenguaje de las plantillas en cada *cartridge* de forma independiente. Además *AndroMDA* permite que se especifiquen librerías de macros para usarlas posteriormente en las plantillas.

Las opciones de *templateObject* sirven para especificar qué clases auxiliares estarán disponibles dentro de las plantillas. Éstas deberán tener un constructor por defecto y deben estar disponibles para el *cartridge*, ya sea internamente o externamente.

Los recursos (*resources*), son los ficheros que serán copiados tal cual en el lugar indicado, sin ser evaluados por el *template engine*. En el ejemplo se

está copiando una hoja de estilos, para poder ver la salida del *cartridge* de una forma más agradable. Es necesario indicar la ruta de los ficheros en el *path.* El *outlet* es el nombre lógico del lugar donde se copiará el fichero. Es interesante remarcar que los *outlets* usados por el *cartridge* estarán inicializados en el proyecto que usa el *cartridge* y no en el propio *cartridge*, de esta forma cada proyecto puede definir su propia estructura de directorios. La opción de *outputPattern* indica qué nombre tendrán los ficheros una vez copiados ("{0}" indica que tendrán el mismo nombre). Por último la opción de *overwrite* indica a *AndroMDA* si ese recurso debe ser copiado cada vez que se ejecute el *cartridge*, si el recurso se generase de forma dinámica, esta opción se tendría que marcar como verdadera.

Las opciones de propiedades (*property*), se usan para especificar propiedades que estarán disponibles dentro de la plantilla. De la misma forma que los *outlets*, las propiedades están inicializadas dentro del proyecto que usa el *cartridge*, posibilitando de esta forma que cada proyecto pueda decidir el comportamiento de los *cartridges* según sus necesidades.

Las condiciones (*condition*), permiten que el *cartridge* imponga restricciones sobre las propiedades que tiene definidas.

La parte donde se define el comportamiento de las plantillas (*template*), son sin duda la parte más importante del descriptor. Muchas de las

opciones ya se han explicado en el apartado de recursos, en este caso el *path* es la ruta hasta el fichero de plantilla y el *"$generatedFile"* usado en la opción de *outputPattern* indica que será la propia plantilla quien establecerá el nombre del fichero de salida. La opción *outputToSingleFile* sirve para que se genere un fichero para todos los elementos tratados en la plantilla, en caso de que no se indicara esta opción, se estaría creando un fichero para cada elemento.

Para cada plantilla se definen qué elementos serán accesibles mediante las opciones de *modelElements* y *modelElement*. Para cada opción se define un nombre de variable, que será accesible desde la plantilla y representará una instancia de un elemento con el *stereotype* o el tipo (*"type"*) indicado en el *modelElement*. En el caso de haber seleccionado previamente la opción de *outputToSingleFile*, las variables representarán una colección de elementos. En el ejemplo se usa la opción *type*, que permite especificar la clase concreta del *metafacade*.

Se puede observar que los descriptores tienen declarados *merge-points* en distintas partes del fichero para que el usuario del *cartridge* pueda adaptarlo a sus necesidades. Por ejemplo un usuario podría preferir un formato de salida distinto, en este caso sólo tiene que crear otra plantilla e indicarlo en su proyecto mediante un *merge-point*.

4.5.2.2 El descriptor de namespace

El descriptor de *namespace* es un fichero *XML* que permite a *AndroMDA* saber qué componentes se deben registrar y definir a qué propiedades tiene acceso el *cartridge*.

De la misma forma que el descriptor del *cartridge*, este fichero debe estar en el directorio *"META-INF/andromda"* y debe llamarse *"namespace.xml"*.

```
<namespace name="arc">
   <components>
      <component name="cartridge">
         <path>META-INF/andromda/cartridge.xml</path>
      </component>
      <component name="metafacades">
         <path>META-INF/andromda/metafacades.xml</path>
      </component>
   </components>
   <properties>
      <!-- namespace-propertyGroup merge-point -->
      <propertyGroup name="Outlets">
         <property name="responsibilities" required="false">
            <documentation>
              The directory to which responsibilities are generated.
            </documentation>
         </property>
      </propertyGroup>
      <propertyGroup name="Other">
         <property name="enableTemplating">
            <default>true</default>
            <documentation>
               Indicates whether or not the cartridge is allowed
               to use templates.
            </documentation>
         </property>
      </propertyGroup>
   </properties>
</namespace>
```

Código 6: Ejemplo de "namespace.xml".

En el ejemplo, los componentes son el *cartridge* y los *metafacades;* cuando *AndroMDA* descubra el *namespace* "arc" sabrá que ambos deben

ser registrados bajo el *namespace* "arc". Nótese que los componentes son en realidad descriptores de otras partes del *cartridge*.

Las propiedades de dividen en grupos, algunos de los cuales tienen un significado especial, como en el caso de los *outlets* (cuyo uso se explicó en el apartado del descriptor del *cartridge*). Para cada propiedad se puede indicar un valor por defecto, la documentación y si se trata de una propiedad requerida o no.

4.5.2.3 El descriptor de metafacades

El descriptor de *metafacades* es un fichero *XML* que permite a *AndroMDA* saber qué *metafacades* están disponibles en el *cartridge.* En este fichero se indica qué *stereotype* está asociado con cada *metafacade* o qué clase del metamodelo está asociada con cada *metafacade*.

De la misma forma que el descriptor del *cartridge* y el de *namespace,* este fichero debe estar en el directorio *"META-INF/andromda"* y debe llamarse *"metafacades.xml"*.

```
<metafacades>
   <metafacade
      class="org.andromda.cartridges.arc.metafacades.ClassLogicImpl"
      contextRoot="true">
         <mapping class="org.omg.uml.foundation.core.UmlClass$Impl">
         </mapping>
   </metafacade>
   <metafacade
      class="org.andromda.cartridges.arc.metafacades.ClaveLogicImpl"
      contextRoot="true">
      <mapping>
         <stereotype>KEY</stereotype>
         <context>
```

```
            org.andromda.cartridges.arc.metafacades.Class
        </context>
    </mapping>
  </metafacade>
</metafacades>
```

Código 7: Ejemplo de "metafacades.xml".

Por cada *metafacade* hay que indicar la clase que lo representa. La opción *contextRoot* indica si este metafacade será la raíz de un contexto. Las opciones de *mapping* le indican a *AndroMDA* cómo debe asociar el *metafacade* con los elementos del modelo, ya sea mediante una clase del metamodelo o bien usando un *stereotype*. La opción *context* sirve para restringir la asociación únicamente a los elementos que están dentro del contexto indicado. En el caso de no indicar el *mapping*, *AndroMDA* determinará qué elemento del *metamodelo* es el que se intenta representar con el *metafacade*.

4.6 Arquitectura del proyecto

Hasta este capítulo la arquitectura del proyecto (v. fig. 3) consta de tres partes.

- El proyecto del usuario. Donde se definen las preferencias y se crea el modelo *UML* de la aplicación que se quiere desarrollar.

- La detección de responsabilidades, que es llevada a cabo por el *cartridge* desarrollado en este proyecto, que a su vez usa el motor de la plataforma *AndroMDA* para interpretar el modelo *UML*.

- El resultado, un conjunto de responsabilidades obtenido de ejecutar el cartridge sobre el modelo *UML* del usuario.

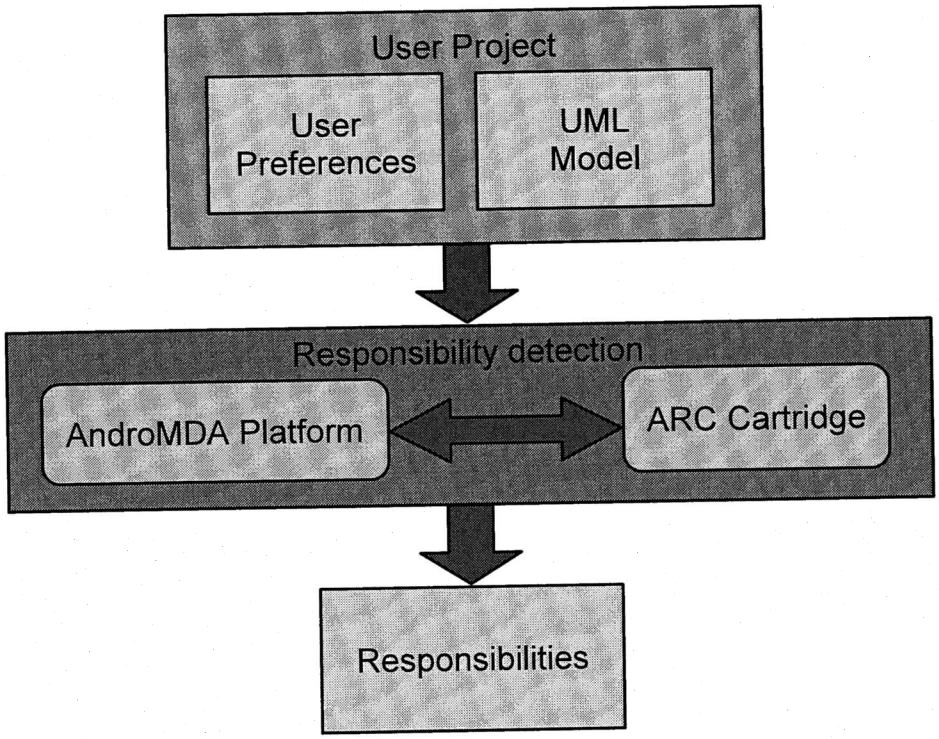

Fig. 3: Arquitectura del proyecto

En el capítulo 6 se verá cómo se asignan los tratamientos a estas responsabilidades y cómo queda modificada esta arquitectura.