FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Testing basado en modelos. Especificación gráfica y derivación automática del código

AUTORES: Anselmo Francisco Abadía, Juan Ignacio Barisich

DIRECTOR: Dra. Claudia Pons

CARRERA: Licenciatura en Sistemas

Resumen

La Ingeniería de Software Dirigida por Modelos (MDE) es uno de los paradigmas más prometedores en su género. Propone mejorar la construcción de software basándose en un proceso guiado por modelos soportado por potentes herramientas. Los modelos, orientados al dominio, constituyen el foco principal del desarrollo. El código, orientado a la plataforma, es generado automáticamente a partir del modelo.

UML es el lenguaje de modelado más utilizado en el desarrollo con MDE. Los perfiles UML, provistos a partir de la versión 2.0 de UML, se suman a éste para lograr definiciones más específicas de los dominios particulares. En especial, el perfil U2TP (UML 2.0 Testing Profile) es un estándar de la OMG que permite la especificación de casos prueba (tests) sobre un modelo UML. Debido a su reciente y no definitiva especificación, son pocas las herramientas de modelado que permiten la definición de casos de prueba con U2TP, más aún, ninguna de ellas permite la derivación automática del código fuente para los mismos.

A lo largo de esta tesis hemos desarrollado una herramienta de software que permite definir gráficamente modelos UML, aplicando el perfil U2TP para especificar los casos de prueba, y a partir de éste derivar automáticamente el código para las clases del dominio y los casos de prueba.

Líneas de Investigación

- Testing de software
- Testing basado en modelos
- Transformaciones de modelos
- Eclipse
- JUnit

Conclusiones

Con el desarrollo de nuestra herramienta hemos comprobado que el paradigma MDE es un enfoque muy poderoso y puede llevarse a la realidad con muy buenos resultados. Además, consideramos que brinda un esquema flexible y potencialmente puede ser aplicado en cualquier ámbito del desarrollo del software. Por otra parte, consideramos que nuestra herramienta aporta a la comunidad MDE una implementación funcional y extensible para el modelado de casos de prueba sobre modelos UML, avalada por el estándar U2TP de la OMG.

Trabajos Realizados

En esta tesis se han analizado los conceptos básicos de MDE, y se ha profundizado en el perfil U2TP. Además se diseñó una arquitectura modular para la herramienta y se implementaron los módulos (plugins de Eclipse) que permiten la definición gráfica de modelos UML con U2TP y su derivación a código fuente Java, utilizando JUnit para la generación de los casos de prueba.

Trabajos Futuros

Como trabajos futuros se proponen:

- Extender los plugins que hemos desarrollado para soportar la generación de código fuente de los conceptos del perfil U2TP que no fueron tratados.
- Implementar nuevos plugins de manera de permitir la derivación de código fuente para otros lenguajes de programación y frameworks de testing
- Analizar las posibilidades para realizar ingeniería inversa a partir de los modelos especificados con UML y U2TP

Fecha de la presentación: Noviembre de 2009

Agradecimientos

Primero quiero agradecer a mi familia por el apoyo que me dio durante estos diez años: a mis padres Raquel y Raúl, que me brindaron la posibilidad de estudiar, a mi hermana Antonela, mi Abuela Blanquita y mi abuelo Francisco que ya no está; a mi novia Analía, que me acompañó durante todo este tiempo; a mis tíos y a mis primos; mis cuñados y mis sobrinos; a todos ellos gracias por el aliento y el apoyo incondicional.
Les quiero agradecer a mis amigos de toda la vida, que hicieron más llevado este largo camino.
A las personas del Lifia que me formaron profesionalmente y a la Facultad de Informática.
Y, finalmente, a mi compañero de tesis, sin el cual no habría podido terminar este trabajo.

Anselmo

A mis padres Cristina y Roberto, por regalarme esta vida y hacerla tan hermosa.
A mi hermana Malala, por creer siempre en mi y apoyarme en cada paso.
A vos Cami, por tu amor y aguante en todos nuestros días.

Juani

Índice

Lista de figuras

1 Introducción

1.1 Motivación

La Ingeniería de Software Dirigida por Modelos MDE se ha convertido en un nuevo paradigma de desarrollo software. MDE promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. Este paradigma asigna a los modelos un rol central y activo: son los artefactos primarios desde los cuales se generan eficientes implementaciones a través de sucesivas transformaciones. Los modelos (orientados al dominio) constituyen el foco principal en el desarrollo de software. El código (orientado a la plataforma) es generado automáticamente aplicando una serie de transformaciones y/o refinamientos sobre el modelo. De esta forma, MDE 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.

MDA, acrónimo de Model Driven Architecture (Arquitectura Dirigida por Modelos), es una de las iniciativas más conocidas y extendidas dentro del ámbito de MDE. MDA es un concepto promovido por el OMG (Object Management Group) en Noviembre 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 MDE. 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.

Generalmente, UML (Unified Modeling Language) es el lenguaje elegido para definir los modelos en MDE. UML es un lenguaje gráfico de propósito general que permite el diseño y desarrollo de sistemas complejos orientados a objetos. Si bien es un lenguaje flexible para abordar la mayoría de los conceptos orientados a objetos, la especificación de casos de prueba (tests) y demás cuestiones de verificación no son provistas por UML, hasta incluso en su versión UML 2.0. Por eso, la OMG ha desarrollado un perfil de testing llamado UML 2.0 Testing Profile (U2TP) que se ha convertido en estándar oficial desde marzo de 2004.

Un perfil UML es una extensión específica de un dominio que es creado utilizando un mecanismo estandarizado de extensibilidad. En particular, U2TP proporciona un marco formal para la definición de un modelo de prueba bajo el acercamiento de caja negra. Aplicando este enfoque las pruebas se derivan de la especificación del sistema sometido a verificación, donde éste es considerado una “caja negra” cuyo comportamiento sólo se puede determinar estudiando sus entradas y salidas. Al ser un estándar tan reciente, son pocas las herramientas de modelado que permiten la definición de casos de prueba utilizando U2TP. Más aún, ninguna de ellas permite la derivación automática del código de los test.

El proyecto Eclipse Modeling Framework (EMF) es un Framework para modelado, que posibilita la generación automática de código para construir herramientas y otras aplicaciones a partir de modelos de datos estructurados. Toma un modelo como el punto de partida, e iterativamente permite refinar el modelo y regenerar el código, hasta obtener el código requerido. Por otra parte, UML2 es una implementación del estándar UML 2.0 basada en EMF. Permite la definición de modelos en UML para luego transformarlos al código correspondiente. Además permite la creación de perfiles para obtener extensiones de UML especializadas en dominios o contextos particulares. Esta característica es la que nos va a permitir crear una implementación concreta y funcional de U2TP. Con esta implementación se podrán definir los casos de test sobre un modelo UML para luego generar el código fuente correspondiente en un lenguaje en particular y en un Framework de testing, como JUnit o TTCN-3.

1.2 Objetivos

El objetivo de esta tesis es la realización de una herramienta de software que permita definir modelos de manera gráfica utilizando UML2 y UML2 Testing Profile (U2TP), y derivar código fuente a partir de los mismos. Para esto, se llevarán a cabo los siguientes desarrollos:

- Se va a extender el plugin UML2 de Eclipse adicionando una implementación del perfil U2TP.
- Se desarrollará un Framework que permita transformar con facilidad modelos creados con el plugin extendido anteriormente, para generar el código fuente correspondiente.
- Se implementará un caso de ejemplo que incluya la definición de un modelo utilizando U2TP y su respectiva transformación a código fuente Java, utilizando las librerías de JUnit y JMock.

2 Organización de la tesis

Como explicamos en el capítulo anterior, este trabajo de tesis tiene como objetivo principal la construcción de una herramienta de software que permita derivar código fuente a partir de modelos *UML* utilizando el perfil *U2TP*.

Inicialmente, en el capítulo 3, brindamos una introducción a los conceptos básicos del paradigma *MDE* como *MDA* y *MOF*. Además, realizamos una descripción del perfil U2TP haciendo foco principal en el paquete de *Test Architecture*, el cual es fundamental para nuestro trabajo.

En el capítulo 4, introducimos todas las herramientas tecnológicas que serán utilizadas a lo largo del desarrollo, como *Eclipse, EMF, GEF, TopCased* y *Acceleo*. Estas herramientas fueron seleccionadas para hacer reuso de los conceptos ya implementados por la comunidad.

Una vez realizada la explicación teórica y la explicación tecnológica, en el capítulo 5 proponemos nuestra solución a los objetivos de la tesis, brindando una descripción arquitectural y de diseño.

A partir de los conceptos del capítulo 5, explicamos en el capítulo 6 la implementación de la solución detallando individualmente cada una de las capas de la arquitectura del software propuesto.

En el capítulo 7 mostramos con un ejemplo el funcionamiento de nuestra solución. Generamos un modelo *UML* con el perfil *U2TP* y derivamos a partir de este, código fuente Java utilizando *JUnit* para la implementación de los casos de prueba.

Para finalizar, en los capítulos 8 y 9 se exponen las conclusiones obtenidas a partir de este trabajo y las líneas propuestas para trabajos futuros.

3 Conceptos básicos

En este capítulo se describen los conceptos básicos sobre metamodelos, como así también la arquitectura de metamodelos y los diferentes modelos que contempla MDA (Arquitectura Dirigida por modelos). Por último se detallan los conceptos básicos referidos a UML, UML Profile y UML2 Testing Profile.

3.1 MDE

A lo largo de esta década la Ingeniería de Software Dirigida por Modelos se ha convertido en un nuevo paradigma que propone mejorar la construcción de software a través de un proceso guiado por modelos, soportado por potentes herramientas que generan código a partir de modelos. MDE promete una mejora de la productividad y la calidad del software generado debido a que se reduce el salto semántico entre el dominio del problema y la solución. Además, las herramientas de generación pueden utilizar Frameworks, patrones y técnicas cuyo éxito se ha comprobado, acortando los tiempos de desarrollo.

3.1.1 Modelos de MDE

MDE 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:

- **Modelo independiente de computación (CIM)**

Un CIM es la visión de un sistema desde un punto de vista independiente de computación. En este tipo de modelos no se muestran detalles de la estructura de los sistemas. A veces es llamado modelo de dominio y en su especificación se utiliza un vocabulario que es familiar a los profesionales de este dominio.

Se supone que el usuario del CIM, el profesional del dominio, no conoce sobre los modelos o artefactos que serán utilizados para implementar el sistema. En este sentido, el CIM juega un papel importante para reducir la brecha entre los que son expertos en el dominio y sus requerimientos, y los que son expertos en el diseño y la implementación del software.

- **Modelo independiente de la plataforma (PIM)**

Un PIM permite la visión de un sistema desde un punto de vista independiente de la plataforma en que será implementado. En sí, representa un diseño conceptual que concreta los requerimientos funcionales del sistema, sin tener en cuenta cómo va a ser implementado: ignora los sistemas operativos, los lenguajes de programación, el hardware, etc. Esta clase de modelos es apropiada para el reuso de un modelo en diversas implementaciones, cada una desarrollada en una plataforma de software en particular.

- **Modelo específico de la plataforma (PSM)**

Un PSM es la visión de un sistema desde un punto de vista específico de la plataforma en que será implementado. Combina las definiciones contenidas en el PIM con los detalles que determinan cómo el sistema es proyectado en una plataforma en particular. Puede proveer más o menos detalles, dependiendo del propósito. Un PSM será una implementación si provee toda la información necesaria para construir un sistema y ponerlo en operación, o puede actuar como un PIM que es usado para futuros refinamientos a PSMs que puedan ser implementados directamente.

Un PSM que es una implementación proveerá una variedad de información diferente, que puede incluir código de programa, especificaciones de compilación, deploy, y otras disposiciones de configuración.

- **Modelo de implementación (IM)**

Se necesita un código final que trabaje en una plataforma específica. Por eso, como último paso del desarrollo, se transforma cada PSM a código fuente. Como cada PSM está orientado a una plataforma en particular, esta transformación es bastante directa.

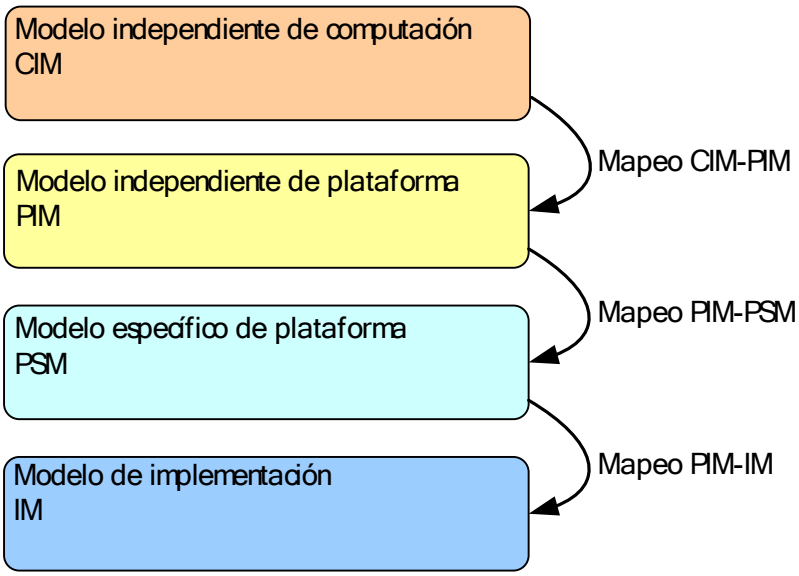

Figura 1 - Modelo MDE

3.2 MDA

Han surgido varios enfoques dentro del ámbito de MDE, como DSL o DSM, pero sin duda la iniciativa más conocida y extendida es MDA, acrónimo de Model Driven Architecture (Arquitectura Dirigida por Modelos). Presentada por el consorcio OMG en noviembre de 2000, MDA tiene como objetivo abordar los desafíos de integración de aplicaciones y los continuos cambios tecnológicos.

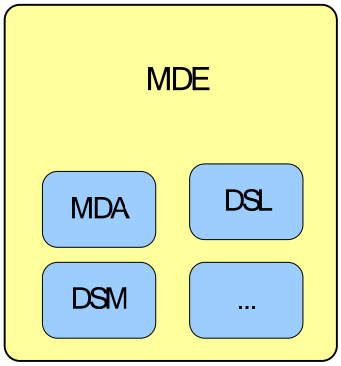

Figura 2 - MDA

La Arquitectura Dirigida por Modelos propone definir un conjunto de normas que especifiquen un conjunto de tecnologías interoperables, las cuales serán utilizadas para el desarrollo a través de modelos.

MDA comenzó con la conocida idea, y establecida desde hace mucho tiempo, de separar la especificación de un sistema de los detalles que determinan su implementación en una plataforma de software. De esta forma, MDA es una alternativa para:

- Especificar un sistema independiente de la plataforma que lo soporta.
- Especificar plataformas.
- Elegir una plataforma en particular para el sistema.
- Transformar la especificación del sistema en una para una plataforma particular.

Los problemas más significativos que aparecen en el desarrollo tradicional de software son: productividad, portabilidad e interoperabilidad. MDA trata de resolverlos de la siguiente forma:

- **Productividad**

El proceso de desarrollo de software tradicional, es conducido a menudo por el diseño de bajo nivel y el código. Aún si el proceso es iterativo e incremental, los documentos y los diagramas se producen sólo en las fases tempranas (etapa de requerimientos y etapa de análisis). Los documentos y los diagramas correspondientes creados en las primeras fases pierden rápidamente su valor tan pronto como la codificación comienza. La conexión entre los diagramas y el código se pierde mientras se progresa la fase de codificación. En vez de ser una especificación exacta del código, los diagramas se convierten frecuentemente en dibujos con poca relación. Cuando un sistema cambia, la distancia entre el código, el texto y los diagramas producidos en las primeras fases, crece. Los cambios se hacen a menudo sólo en el código, porque no hay tiempo disponible para actualizar los diagramas y otros documentos de alto nivel.

En MDE el foco está puesto en el desarrollo de modelos. Luego, el código fuente es generado utilizando procesos automatizados de transformaciones. Por supuesto, es necesario que alguien defina dichas transformaciones, lo cual es una tarea difícil y especializada. Pero éstas necesitan ser especificadas sólo una vez, y pueden ser aplicadas en el desarrollo de muchos sistemas. La utilización de herramientas de modelado y generación automática de código incrementa significativamente los tiempos de desarrollo y por ende la productividad de los desarrolladores.

- **Portabilidad**

La industria del software tiene una característica especial que la diferencia de otras industrias. Cada año, o cada menos tiempo, aparecen nuevas tecnologías y rápidamente llegan a ser populares (por ejemplo Java, Linux, XML, HTML, UML, NET, Flash, Servicios WEB, etc.). Las nuevas tecnologías ofrecen beneficios concretos para las compañías y muchas de ellas no pueden quedarse atrás, por lo tanto, tienen que utilizarlas muy rápidamente. El software ya existente puede seguir sin cambios, pero necesitará comunicarse con los nuevos sistemas que serán construidos usando una nueva tecnología. Dentro de MDE la portabilidad se lleva a cabo enfocando en el desarrollo de PIMs que son independientes de la plataforma. Un mismo PIM puede ser automáticamente transformado en muchos PSMs para diferentes plataformas. Por lo tanto, todo lo que se especifica en el nivel de PIM es completamente portable; sólo depende de las herramientas de transformación disponibles.

- **Interoperabilidad**

Los sistemas de software raramente funcionan en forma aislada. La mayoría de los sistemas necesitan comunicarse con otros que generalmente ya existen. Incluso cuando los sistemas se construyen completamente, usan a menudo múltiples tecnologías, a veces de versionados o épocas diferentes. Por ejemplo, un sistema WEB necesita usar una base de datos para almacenar su información. En MDE se pueden generar muchos PSMs a partir del mismo PIM, y éstos pueden estar relacionados. Estas relaciones se llaman bridges o puentes.

Como se mencionó anteriormente, en MDA los modelos cumplen el rol principal en el desarrollo de sistemas de software. Por eso es fundamental definir una sintaxis para lenguajes de creación y manipulación

de modelos. A continuación se presentará la técnica de metamodelado y la arquitectura en cuatro capas para modelado propuesta por la OMG.

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, etc.

Como un metamodelo es también un modelo, es necesario que exista un lenguaje para definir metamodelos de manera precisa. Un meta-metamodelo (OMG, 2003) 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.

3.3.1 MOF

La especificación MOF, acrónimo de Meta-Object Facility, es el fundamento de los ambientes estándares de la industria OMG, donde los modelos pueden ser exportados desde una aplicación, importados en otra, transportados a través de la red, almacenados en un repositorio y luego recuperados, renderizados en diferentes formatos, transformados y usados para generar código de aplicación. Estas funciones no se restringen sólo a modelos estructurales ó a modelos UML, sino que se aplican a todos los modelos basados en MOF.

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.

3.3.2 Arquitectura de metamodelo de cuatro capas

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

Como su nombre lo indica, esta arquitectura está compuesta por cuatro capas, las cuales se describen a continuación:

- **Meta-metamodelo**

La capa de meta-meta modelado es el fundamento de la arquitectura de meta modelado. 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. Aunque es deseable que los meta-metamodelos y

metamodelos relacionados compartan filosofías de diseño y construcciones comunes, esto no es una regla estricta. Eso si, cada capa necesita mantener su propia integridad del diseño.

Dentro del OMG, MOF es el lenguaje estándar de esta capa. Ejemplos de meta-meta objetos en la capa de meta-meta modelado son: 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. Ejemplos de objetos en esta capa son: Ford Falcon (Auto), 78 (Edad).

3.4 UML

Un modelo es una descripción de un sistema, o de una parte del mismo, escrita en un lenguaje bien definido. Para estar bien definido, un lenguaje debe tener sintaxis y semántica precisas, y debe poder ser interpretado y manipulado por una computadora. Entre los lenguajes de modelado que define OMG el más conocido y usado es sin duda UML (Unified Modeling Language). UML es un lenguaje gráfico para especificar, construir y documentar los artefactos que modelan un sistema. UML fue diseñado para ser un lenguaje de modelado de propósito general, por lo que puede utilizarse para especificar la mayoría de los sistemas basados en objetos o en componentes, y para modelar aplicaciones de muy diversos dominios de aplicación (telecomunicaciones, comercio, sanidad, etc.) y plataformas de objetos distribuidos (como por ejemplo J2EE, .NET o CORBA).

En UML hay 13 tipos diferentes de diagramas. Para comprenderlos correctamente, a veces es útil categorizarlos jerárquicamente de la siguiente manera:

- Los **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.

- Los **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.
- Los **Diagramas de Interacción** son un subtipo de diagramas de comportamiento que enfatizan el flujo de control y de datos entre los elementos del sistema modelado. Dentro de esta categoría se encuentran:
    - **Diagrama de secuencia:** muestra como 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.

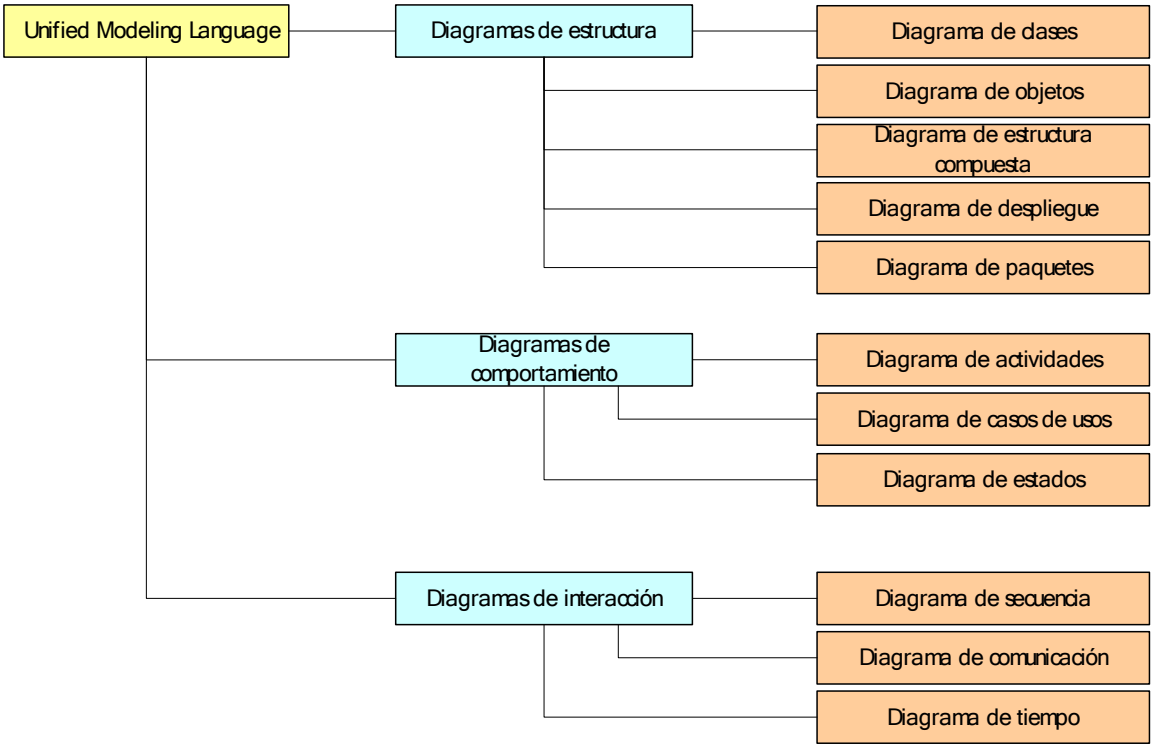

Figura 3 - Esquemas de diagramas UML

3.4.1 UML Profile

El hecho de que UML sea un lenguaje de propósito general proporciona una gran flexibilidad y expresividad a la hora de modelar sistemas. Sin embargo, hay numerosas ocasiones en las que es mejor contar con algún lenguaje más específico para modelar y representar los conceptos de ciertos dominios particulares. Esto sucede, por ejemplo, cuando la sintaxis o la semántica de UML no permiten expresar los conceptos específicos del dominio, o cuando se desea restringir y especializar los constructores propios de UML, que suelen ser demasiado genéricos y numerosos.

A la hora de definir lenguajes específicos de dominio OMG define dos alternativas, las cuales se corresponden con las dos situaciones mencionadas antes: o bien se define un nuevo lenguaje (alternativa a UML), o bien se extiende el propio UML, especializando algunos de sus conceptos y restringiendo otros, pero respetando la semántica original de los elementos de UML (clases, asociaciones, atributos, operaciones, transiciones, etc.). La primera opción es adoptada por lenguajes como CWM, puesto que la semántica de algunos de sus constructores no coinciden con la de UML. Para definir un nuevo lenguaje hay que describir su metamodelo utilizando MOF, que como se mencionó anteriormente, es un lenguaje para especificar lenguajes de modelado. Por otro lado, hay situaciones en las que es suficiente con extender el lenguaje UML utilizando una serie de conceptos recogidos en lo que se denominan Perfiles de UML (UML Profiles).

Cada una de estas dos alternativas presenta ventajas e inconvenientes. Por su parte, definir un nuevo lenguaje ad-hoc permite conseguir un mayor grado de expresividad, incrementando la correspondencia entre los elementos del modelo y los conceptos de un dominio en particular. Sin embargo, incluso cuando esos nuevos lenguajes se describen con MOF, el hecho de no respetar el metamodelo estándar de UML va a impedir que las herramientas UML existentes en el mercado puedan manejar sus conceptos de una forma natural. En general, resulta difícil decidir cuándo debemos crear un nuevo metamodelo, y cuándo es mejor definir una extensión de UML usando perfiles. Esta elección depende del tipo de sistema a modelar y de los requerimientos que impone el cliente.

Los Perfiles UML se definieron originalmente en la versión 1.3 de UML, aunque era difícil saber si se estaban aplicando correctamente debido a que su definición era un tanto ambigua. La versión UML 2.0 mejoró substancialmente la definición de perfiles, determinando de forma más clara las relaciones permitidas entre los elementos del modelo a especificar y el uso de las metaclases de un metamodelo dentro de un perfil. En concreto, el paquete Profiles de UML 2.0 define una serie de mecanismos para extender y adaptar las metaclases a las necesidades concretas de una plataforma (como .NET o J2EE) o de un dominio de aplicación (modelado de procesos de negocio, aplicación de tiempo real, etc.).

En la especificación de UML 2.0 se señalan varias razones por las que un diseñador puede querer extender y adaptar un metamodelo existente, ya sea el propio UML, el de CWM, o incluso un Perfil UML. Entre ellas se destacan:

- Disponer de una terminología o vocabulario propio de un dominio de aplicación o de una plataforma de implementación concreta (por ejemplo, poder manejar dentro del modelo del sistema terminología propia de EJB como "Home interface", "entity bean", "archive", etc.).
- Definir una sintaxis para construcciones que no cuentan con una notación propia (como sucede con las acciones).
- Definir una nueva notación para símbolos ya existentes, más acorde con el dominio de la aplicación objetivo (poder usar, por ejemplo, una figura con un ordenador en lugar del símbolo para representar un nodo que por defecto ofrece UML para representar ordenadores en una red).
- Añadir cierta semántica que no aparece determinada de forma precisa en el metamodelo (por ejemplo, la incorporación de prioridades en la recepción de señales en una máquina de estados de UML).
- Añadir cierta semántica que directamente no existe en el metamodelo (por ejemplo relojes, tiempo continuo, etc.).

- Añadir restricciones a las existentes en el metamodelo, restringiendo su forma de utilización (por ejemplo, impidiendo que ciertas acciones se ejecuten en paralelo dentro de una transición, o forzando la existencia de ciertas asociaciones entre las clases de un modelo).
- Añadir información que puede ser útil a la hora de transformar el modelo a otros modelos, o a código

Un Perfil se define en un paquete UML, estereotipado «profile», que extiende a un metamodelo o a otro Perfil. Los mecanismos que se utilizan para definir Perfiles son tres: estereotipos (stereotypes), restricciones (constraints), y valores etiquetados (tagged values). Para ilustrar estos conceptos utilizaremos un ejemplo simple de Perfil UML. Este perfil va a definir dos nuevos elementos para que puedan ser añadidos a cualquier modelo UML: colores y pesos.

- En primer lugar, un estereotipo viene definido por un nombre, y por una serie de elementos del metamodelo sobre los que puede asociarse. Gráficamente, los estereotipos se definen dentro de cajas, estereotipadas «stereotype». Obsérvese cómo el Perfil especifica los elementos del metamodelo de UML sobre los que se pueden asociar los estereotipos estereotipados «metaclass», mediante flechas continuas de punta triangular en negrita.
- A los estereotipos es posible asociarles restricciones, que imponen condiciones sobre los elementos del metamodelo que han sido estereotipados. De esta forma pueden describirse, entre otras, las condiciones que ha de verificar un modelo "bien formado" de un sistema en un dominio de aplicación. Por ejemplo, supongamos que el metamodelo de nuestro dominio de aplicación impone la restricción de que si dos o más clases están unidas por una asociación coloreada, el color de las clases debe coincidir con el de la asociación. Dicha restricción se traduce en la siguiente restricción del Perfil UML, en el lenguaje OCL (Object Constraint Language).Las restricciones pueden expresarse tanto en lenguaje natural como en OCL. OCL es un lenguaje para consultar y restringir los elementos de un modelo, definido por OMG y es parte integrante de UML. OCL permite escribir expresiones sobre modelos, como por ejemplo invariantes, pre- y post-condiciones, reglas de derivación para los atributos y las asociaciones, el cuerpo de las operaciones de consulta, etc. El término "Constraint" que aparece en el nombre OCL perdura de la primera versión de OCL, que sólo permitía definir restricciones. Sin embargo, la versión 2.0 de OCL proporciona un lenguaje de consultas mucho más general, con una expresividad similar a la de SQL.
- Finalmente, un valor etiquetado es un meta-atributo adicional que se asocia a una metaclase del metamodelo extendido por un Perfil. Todo valor etiquetado ha de contar con un nombre y un tipo, y se asocia un determinado estereotipo. Los valores etiquetados se representan de forma gráfica como atributos de la clase que define el estereotipo.

Es importante señalar que estos tres mecanismos de extensión no son de primer nivel, es decir, no permiten modificar metamodelos existentes, sólo añadirles elementos y restricciones, pero respetando su sintaxis y semántica original. Sin embargo, sí que son muy adecuados para particularizar un metamodelo para uno o varios dominios o plataformas existentes. Cada una de estas particularizaciones o adaptaciones viene definida por un Perfil, que agrupa los estereotipos, restricciones, y valores etiquetados propios de tal adaptación. Actualmente ya hay definidos varios Perfiles UML, algunos de los cuales han sido incluso estandarizados por la OMG: los Perfiles UML para CORBA y para CCM (CORBA Component Model), el Perfil UML para EDOC (Enterprise Distributed Object Computing), el Perfil UML para EAI (Enterprise Application Integration), y el Perfil UML para Planificación, Prestaciones, y Tiempo (Scheduling, Performance, and Time). Otros Perfiles UML se encuentran actualmente en proceso de definición y estandarización por parte de la OMG, y se espera que vean la luz muy pronto. También hay Perfiles UML definidos por otras organizaciones y empresas fabricantes de lenguajes de programación y herramientas que, aun no siendo estándares oficiales, están disponibles de forma pública y son comúnmente utilizadas (convirtiéndose por tanto en estándares "de facto". Un ejemplo de tales Perfiles es el "UML/EJB Mapping Specification", que ha sido definido por JCP (Java Community Process). También se dispone de Perfiles UML para determinados lenguajes de programación, como pueden ser Java o C++. Cada uno de estos

Perfiles define una forma concreta de usar UML en un entorno particular. Así por ejemplo, el Perfil UML para CORBA define una forma de usar UML para modelar interfaces y artefactos de CORBA, mientras que el Perfil UML para Java define una forma concreta de modelar código Java usando UML.

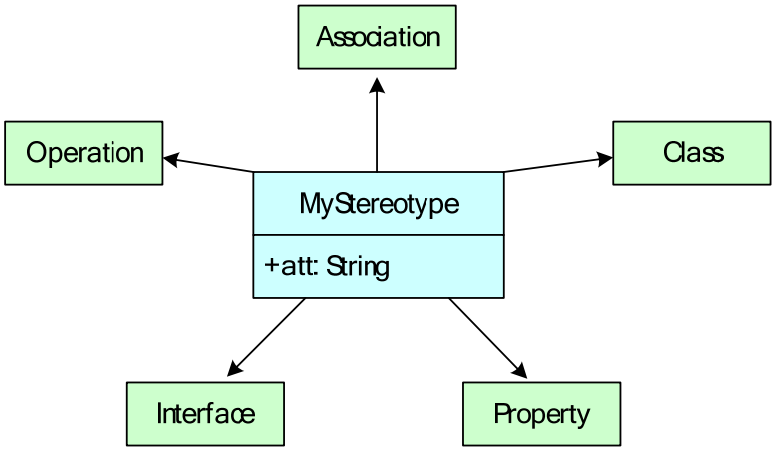

Figura 4 - Modelo de estereotipos

3.4.1.1 Diagrama de clases y diagramas UML

Es importante recordar que para esta tesis trabajaremos principalmente sobre los diagramas de clases, por lo tanto, creemos conveniente profundizar sobre ellos.

Un diagrama de clases es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargaran del funcionamiento y la relación entre uno y otro.

Las propiedades, también llamadas atributos son valores que corresponden a un objeto, como color, edad, cantidad, fecha, etc. Generalmente se conoce como la información detallada del objeto. Suponiendo que el objeto es una puerta, sus propiedades serían: la marca, tamaño, color y peso.

Las operaciones son aquellas actividades o verbos que se pueden realizar con/para este objeto, como por ejemplo abrir, cerrar, buscar, cancelar, debitar, etc.

Las relaciones definen la manera en que se unen los diferentes elementos del sistema. Las relaciones pueden ser:

- **Herencia:** Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la super clase.
- **Agregación:** Son relaciones utilizadas cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación.
- **Asociación:** La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre sí. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.
- **Uso:** El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo una aplicación grafica que instancia una ventana (la creación del Objeto Ventana está condicionado a la instanciación proveniente desde el objeto Aplicación).

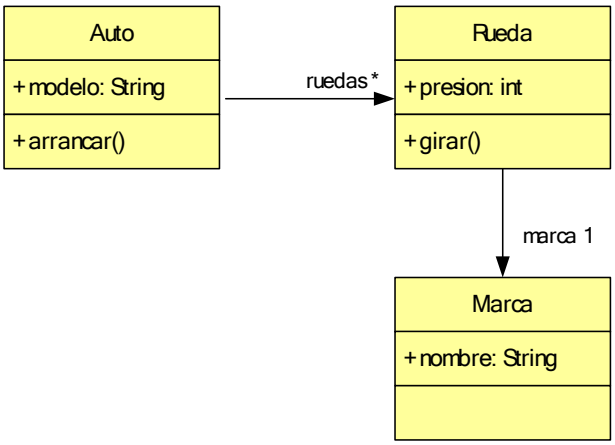

Figura 5 - Diagramas de clase de UML

Como mencionamos anteriormente, los perfiles permiten extender modelos con conceptos específicos de un dominio. En particular, para el desarrollo de esta tesis se extenderán los diagramas de clases, agregando estereotipos, restricciones y valores etiquetados. El siguiente es un ejemplo de un diagrama de clases extendido usando perfiles:

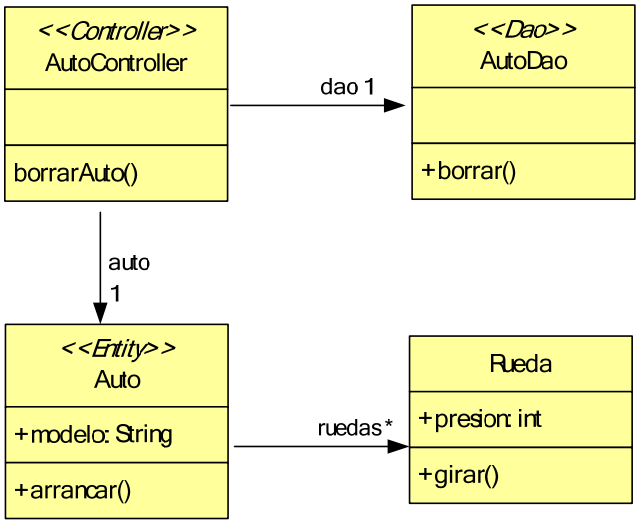

Figura 6 - Diagramas de clase de UML con estereotipos

3.4.2 UML 2 Testing Profile

UML Testing Profile (U2TP) define un lenguaje para diseñar, visualizar, especificar, analizar, construir, y documentar los artefactos de sistemas de pruebas de caja negra.

U2TP se define utilizando el enfoque de meta modelado de UML y se ha diseñado teniendo en cuenta los siguientes principios de diseño:

- **Integración con UML:** como un verdadero perfil de UML, U2TP se define sobre la base del metamodelo de UML y sigue los principios de perfiles UML.
- **Reutilización y minimalidad:** siempre que sea posible, U2TP hace uso directo de los conceptos, y agrega nuevos conceptos sólo en caso de necesidad.

3.4.2.1 Estructura de U2TP

U2TP está organizado en cuatro grupos conceptuales:

- **Test Architecture:** define los conceptos relacionados con la estructura y configuración de los test. Provee la base de elementos que participarán para llevar a cabo el comportamiento de los test. Test Architecture es descripto utilizando principalmente diagramas de clases. Bajo este paquete, uno o varios objetos son identificados como SUT (Sistema bajo test). Los SUT se comunican con otros objetos llamados Test Components para realizar el comportamiento de la prueba. Los Test Context permiten a los usuarios agrupar los casos de test para definir Test Configurations y los Test Controls. Los Test Configuration definen la conexión entre los SUT y los Test Components, y los Test Controls definen el flujo de ejecución de los casos de prueba. Cada TestContext produce un resultado llamado Veredict(veredicto) a partir de un proceso de evaluación llevado a cabo por el Arbiter(árbitro). La ejecución de los Test es llevada a cabo por el Scheduller(planificador) el cual tiene la responsabilidad de de controlar la ejecución de las pruebas y de los componentes de pruebas. El planificador es el que comienza y determina el fin de las pruebas.
- **Test Behavior**: define los conceptos relacionados con los aspectos dinámicos de los casos de test. Se utiliza especificar los casos de test y sus respectivos comportamientos. Test Behavior es descripto utilizando diagramas de secuencia, diagramas de actividades y máquina de estados. Bajo este paquete, un Test Objetive(objetivo de la prueba) define el objetivo de la prueba. El comportamiento de la prueba es especificado el Test Case (caso de prueba), el cual es una operación del TestContext que indica cómo un conjunto de operaciones interactúan para cumplir con el objetivo de la prueba. Cuando ciertas acciones inesperadas ocurren en la prueba, se utilizan los Defaults para determinar cómo continua la ejecución del caso de prueba. Una ValidationAction(acción de validación) es ejecutada para resolver el veredicto local del caso de prueba. Luego el árbitro toma todos los veredictospara decidir el veredicto del contexto de prueba en su totalidad.
- **Test Data:** define los conceptos sobre los datos utilizados para los test por la lógica de los mismos. Refiere a los tipos y valores que son enviados o recibidos por el SUT. Bajo este paquete, los Wildcard(comodines) son utilizados representar valores, rango de valores y la existencia de valores. Los DataPools(pool de datos) son usados como medio para proporcinar all contexto de prueba particiones de datos o valores explícitos asociados a los contextos de prueba y adicionan los datos concretos de la prueba. Las particiones de datos son clase de equivalecias de un conjunto de valores, por ejemplo (nombre de ciudades pertenecientes a la provincia de Cordoba). Estos datos son seleccionados por el DataSelector a través de una operación.
- **Test Time:** define los conceptos para restringir y controlar el comportamiento de los test con respecto al tiempo.Bajo este grupo, los Timers son utilizados para manipular y controlar el comportamiento de la prueba, además de asegurar la terminación de los casos de prueba. En ciertos casos, cuando el sistema es distribuido, se utilizan los TimeZone(zonas horarias) para permitir una correcta comparación de los venetos de tiempo.

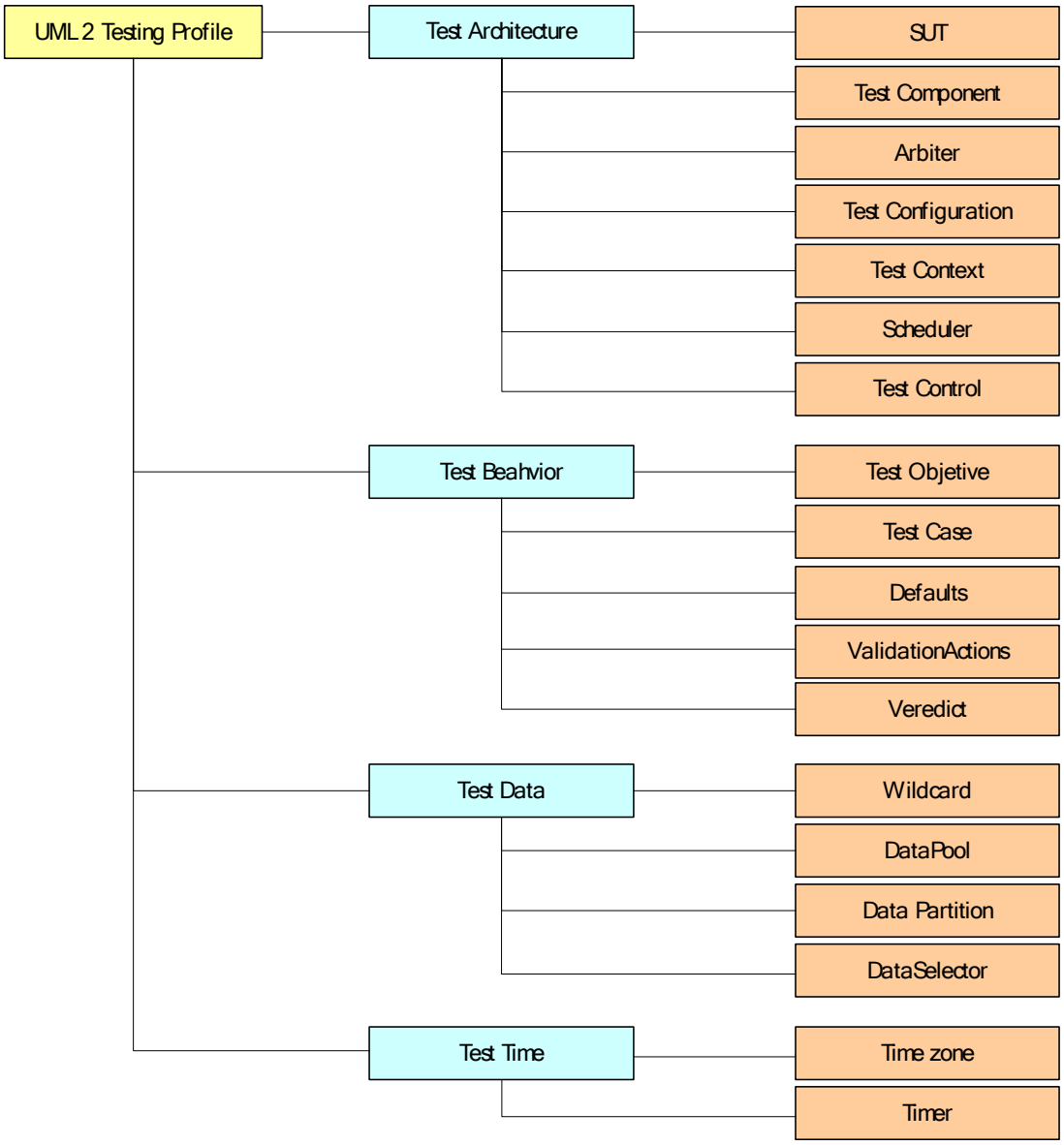

Figura 7 - Paquetes de U2TP

3.4.2.1.1 Test Architecture

Como nuestra tesis derivará código sólo para la parte estructural del perfil, detallaremos en profundidad los elementos de Test Architecture. Esta sección contiene los conceptos necesarios para describir los elementos que son definidos usando el perfil para los casos de prueba.

3.4.2.1.1.1 Arbiter

El Arbiter es una interfaz predefinida provista por el perfil. El propósito de una implementación de un árbitro es determinar el veredicto final de un caso de test. Esta decisión está hecha acorde a una estrategia de arbitraje particular, la cual está provista en la implementación de la interface Arbiter.

Cada acción de validación causa la invocación de la operación setVeredict sobre la implementación del Arbiter.

Cada Test Context debe tener una implementación de la interface Arbiter, y las herramientas de construcción basadas en el perfil de prueba proveerán un Arbiter por defecto para ser usado si alguno no está definido explícitamente en el contexto de prueba.

Operaciones

- **getVerdict():** Devuelve el veredicto actua. Es utilizada por un componente de prueba para proveer al árbitro información actualizada con respecto a la situación actual de un caso de prueba.
- **setVerdict(v : Verdict):** Configura un nuevo valor de veredicto. Esta operación es utilizada por las acciones de validación.

Semántica

La configuración de la semántica del veredicto es definida por la implementación del árbitro. Un ejemplo de cómo implementar la operación del setVeredict:

- Si el veredicto correcto (pass), este puede ser cambiado solamente a el estado inconcluso, falla, o error.
- Si el veredicto es inconcluso, este puede ser cambiado solamente a el estado falla o error.
- Si un veredicto es falla, este puede ser cambiado solamente al estado error.
- Si el veredicto es error, este no puede ser cambiado.

3.4.2.1.1.2 Scheduler

El Scheduler es una interfaz predefinida en el perfil de prueba. El propósito de una implementación del Scheduler es controlar la ejecución de los diferentes componentes de prueba. El Scheduler deberá mantener información acerca de cuáles componentes existen en cualquier punto en el tiempo, y colaborar con el Arbiter para que le informe cuando sea el momento de emitir el resultado final.

Este mantiene el control sobre la creación y destrucción de los componentes de test y conoce cuales son los componentes de test que constituyen cada caso de test.

Cada Test Context deberá tener una implementación del Scheduler. Las herramientas deben proporcionar una implementación que deberá ser usada si no hay una implementación explícita definida en el contexto.

Operaciones

- **Scheduler():** El constructor del Scheduler. Este deberá comenzar el SUT y el Arbiter.
- **startTestCase():** El Scheduler comenzará el caso de test notificando a todos los componentes involucrados.
- **finishTestCase(t:TestComponent):** Almacena que el test component t ha terminado su ejecución para el caso de test. Si todos los Test Components involucrados en el caso de test terminaron, el Arbiter será notificado.
- **createTestComponent(t:TestComponent):** Almacena que el Test Component t ha sido creado por otro test Test Component.

Semántica

La implementación de la interface predefinida determinará los detalles semánticos. La implementación deberá asegurar que el Scheduler tenga la suficiente información para realizar el seguimiento de la participación de los Test Components en caso de prueba. El Test Context por si mismo deberá asegurar la creación de un Scheduler.

3.4.2.1.1.3 SUT

El SUT es el sistema bajo test. Determina el conjunto de elementos que se someterán a la prueba. El perfil sólo aborda pruebas de caja negra. Los SUTs son accedidos sólo desde sus operaciones públicas y, por lo tanto, no se puede utilizar información interna del SUT en la especificación del caso de prueba.

3.4.2.1.1.4 Test Elements

Los Test Elements son definidos a partir de dos elementos del paquete de arquitectura: Test Context y Test Components. Un Test Context contiene una colección de casos de prueba, una implementación de la interface de Arbiter, normalmente una o varias instancias de SUT, y posiblemente el comportamiento de alto nivel usado para controlar la ejecución de los casos de prueba. Test Components son elementos que interactúan con el SUT para realizar los casos de prueba definidos en el contexto de Test.

3.4.2.1.1.5 Test Component

Un Test Component es una clase del sistema de prueba. Los objetos Test Components son utilizados para especificar los casos de pruebas. Esta especificación se realiza a partir de un conjunto de interfaces a través de las cuales se interactúa con otros componentes de test o con el SUT.

3.4.2.1.1.6 TestContext

Un Test Context es un elemento que tiene la función de agrupar un conjunto de casos de prueba. La estructura que forma un Test Context es denominada TestConfiguration (configuración de la prueba) debido a que estos son los que almacenan el árbitro y planificador que serán utilizados para un conjunto de casos de prueba.

Atributos

- **arbiter:** Arbiter [1] Implementación de la interface Arbiter
- **scheduler:** Scheduler [1] Implementación de la interface Scheduler.

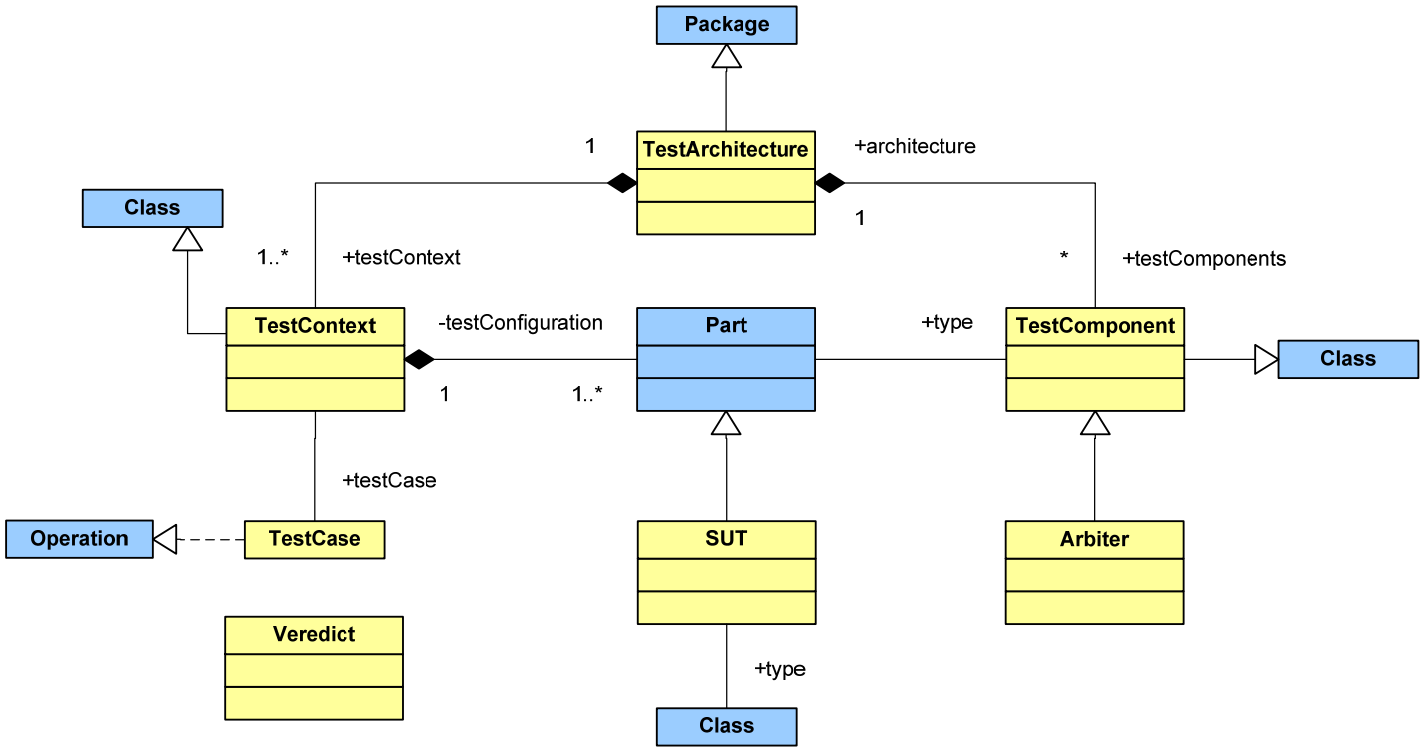

Figura 8 - Paquete de TestArchitecture de U2TP

4 Descripción de tecnologías

4.1 Eclipse

4.1.1 Plataforma

Eclipse es una plataforma de software de código abierto independiente de una plataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos integrados de desarrollo (del inglés IDE).

La plataforma eclipse, está construida sobre un mecanismo para descubrir, integrar y correr módulos llamados plugins. Un proveedor de herramientas escribe una herramienta como un plugin separado, que opera sobre archivos en el workspace y su interfaz de usuario específica trabaja sobre la superficie del workbench. Cuando la plataforma es cargada, el usuario es presentado con un ambiente de desarrollo integrado (IDE) compuesto de un conjunto de plugins habilitados.

La Plataforma Eclipse está diseñada y construida para cumplir con los siguientes requerimientos:

- Soportar la construcción de una variedad de herramientas para el desarrollo de aplicaciones.
- Soportar un irrestricto conjunto de proveedores de herramientas, incluyendo vendedores de software independientes.
- Soportar herramientas para manipular tipos de contenido arbitrario (por ej. HTML, Java, C, JSP, EJB, XML, UML, GIF, ETC).
- Permitir una fácil integración de las herramientas entre sí, a través de los diferentes tipos de contenidos y sus proveedores.
- Soportar el desarrollo de aplicaciones basadas y no, en interfaz gráfica de usuario (en inglés Graphical User Interface, GUI y non-GUI-based).
- Correr en una gran cantidad de sistemas operativos.

El principal objetivo es el de proveer facilidades a los proveedores de herramientas para desarrollar las mismas con mecanismos de uso y reglas para seguir. Estos mecanismos son provistos por una API (Application Programming Interface - Interfaz de Programación de Aplicaciones) de interfaces bien definida, clases y métodos.

4.1.2 Plataforma Runtime y la Arquitectura de Plugin

Un Plugin es la mínima unidad de la Plataforma Eclipse 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 kernel conocido como Plataforma Runtime, toda la funcionalidad de la Plataforma Eclipse 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 llamado *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.

Un punto de extensión puede ser extendido por otro plugin. Por ejemplo, el *workbench* plugin declara un punto de extensión para las preferencias del usuario. Cualquier plugin puede contribuir con sus propias preferencias para el usuario definiendo extensiones a este punto de extensión.

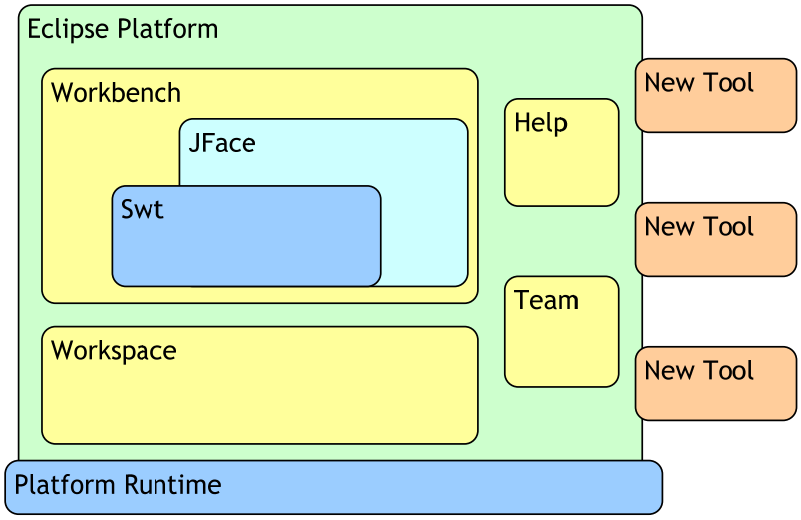

Figura 9 - Arquitectura de Eclipse

4.1.3 La Arquitectura Eclipse

Eclipse está formado por el núcleo, el entorno de trabajo (Workspace), el área de desarrollo (Workbench), la ayuda al equipo (Team support) y la ayuda o documentación (Help).

- **Núcleo:** su tarea es determinar cuáles son los plugins disponibles en el directorio de plugins de Eclipse. Cada plugin tiene un fichero XML manifest 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 y recursos.
- **Entorno 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, y contienen archivos y carpetas.
- **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 tiene el aspecto 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, 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.
- **Ayuda al grupo:** 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. Eclipse incluye un cliente para CVS.
- **Documentación:** al igual que el propio Eclipse, 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.

4.2 EMF (Eclipse Modeling Framework)

El proyecto EMF es un Framework de modelado y generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurados. A partir de una especificación de modelo descripta en XMI, EMF provee herramientas y soporte Runtime para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y edición vía comandos del modelo, y un editor básico.

Los modelos pueden ser especificados usando anotación Java, documentos XML, o herramientas de modelado como Rational Rose, y después ser exportados a EMF. Lo más importante de todo, EMF suministra las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.

El EMF incluye XML Schema Infoset Model (XSD), un nuevo componente del proyecto Model Development Tools (MDT), y una implementación basada en EMF de Service Data Objects (SDO). XSD provee un modelo y una API para manipular componentes de un esquema XML, con acceso a representación DOM del documento del esquema.

EMF consiste de tres piezas fundamentales:

- **EMF-Core**: El core del Framework EMF incluye un meta modelo (ECore) para describir modelos y soporte runtime para los modelos incluyendo notificaciones de cambio, soporte de persistencia con serialización XMI (XML Metadata Interchange) por defecto, y una muy eficiente API para manipular objetos EMF genéricamente.
- **EMF-Edit**: El Framework EMF.Edit incluye clases reutilizables genéricas para construir editores para modelos EMF. Esto provee:

  Clases proveedoras de contenido y etiquetas, y otras clases que permiten a los modelos EMF ser mostrados usando visualizadores de escritorio estándar.

  Un Framework de comandos, incluyendo un conjunto de clases de implementación de comandos genéricos para editores de construcción que soporten completamente el “rehacer” y “deshacer” automáticos.
- **EMF-Codegen**: La generación de código EMF es capaz de generar todo lo necesitario para construir un editor completo para un modelo EMF. Esto incluye un GUI desde el cual pueden ser especificadas las opciones de generación, y los generadores a ser invocados. La generación se basa en JDT (Java Development Tooling), un componente de Eclipse. Hay tres niveles de generación de código que son soportadas:
  - **Modelo**: Proporciona clases Java de implementación e interfaz para todas las clases del modelo, además una clase de implementación de factory y package (meta data).
  - **Adaptadores**: Genera clases de implementación (llamadas ItemsProviders) que adaptan las clases del modelo para ser editadas y mostradas.
  - **Editor**: Produce un editor estructurado adecuadamente que se ajusta al estilo recomendado por los editores de modelos EMF Eclipse y sirve como punto de partida al comienzo de la personalización.

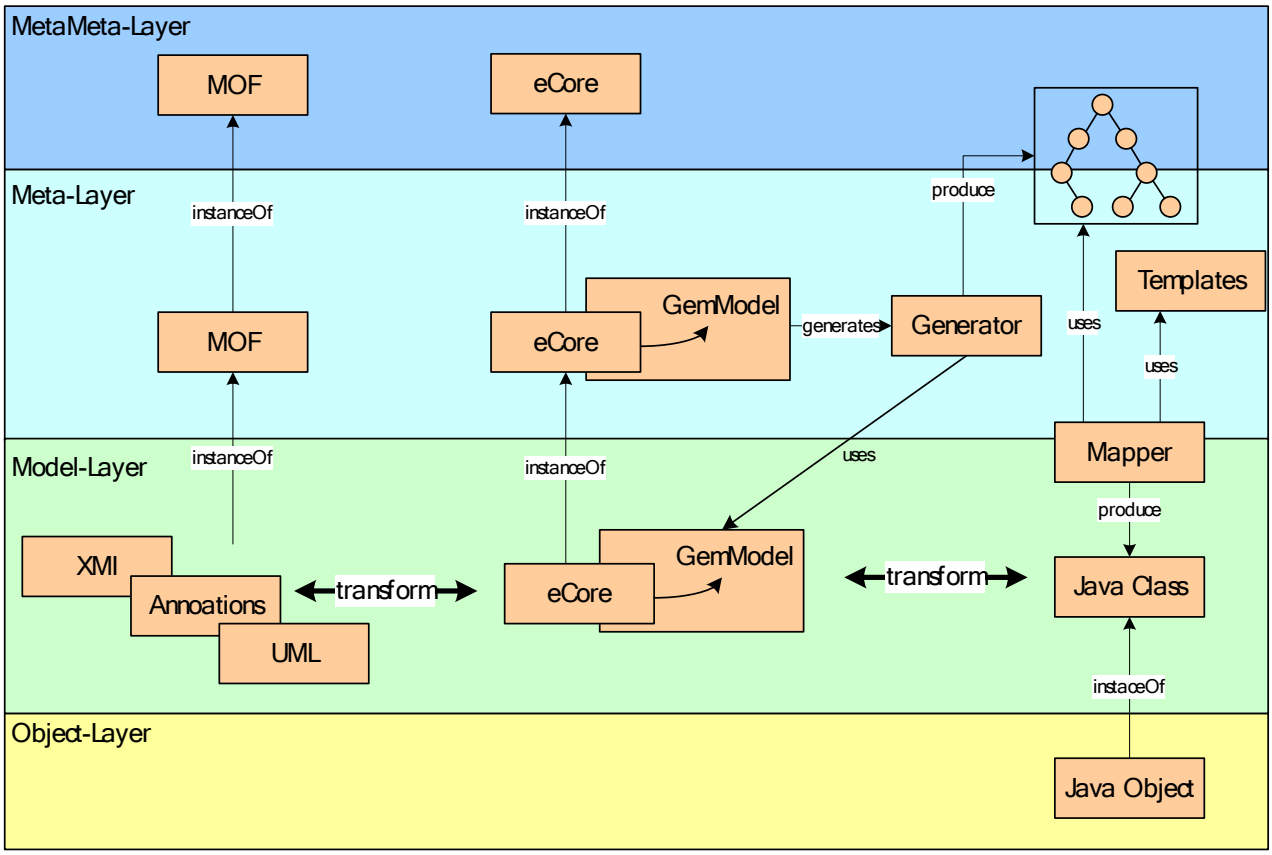

Figura 10 - Modelo de EMF

4.2.1 Transformaciones QVT

La transformación de modelos es el proceso de convertir un modelo *Ma* instancia del metamodelo *MMa* en un modelo *Mb* instancia de un metamodelo *MMb*.

La transformación de modelos es un componente crítico de MDA; por esta razón, ya existe una propuesta para que QVT sea parte del estándar de MDA.

QVT define una manera estándar para transformar los modelos de origen en modelo destinos. Actualmente hay varios productos (comerciales o de código abierto) que intentan convertirse en el standard.

El lenguaje QVT integra el estándar OCL 2.0 y también extiende a la parte iterativa de OCL. También, define 3 DLS (domain-specific languages) llamados *Relations*, *Core* y *Operational Mapping*. Relations y Core son lenguajes declarativos en dos niveles de abstracción diferentes. El lenguaje *Relations* tiene una sintaxis textual y gráfica concreta. El lenguaje *Operational Mapping* es un lenguaje imperativo que extiende Relations y a Core. La sintaxis de *Operational Mapping* provee constructores como los comúnmente encontrados los lenguajes imperativos como loops, condiciones, etc.

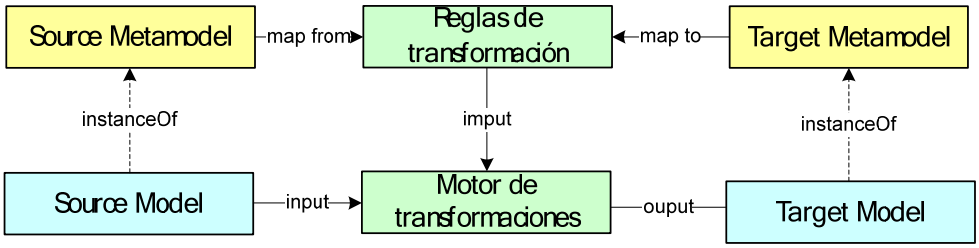

Figura 11 - Esquema de transformación de QVT

4.3 GEF

Permite a los desarrolladores tomar un modelo de aplicación existente y crear fácilmente un editor gráfico rico. 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 especifico.

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

- Constructores GUI.
- Editores de diagramas UML (como por ejemplo workflow y diagramas de modelado de clases).
- Editores de texto WYSIWYG 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.

4.3.1 TopCased

TopCased es una herramienta visual que funciona como plugin de Eclipse la cual es utilizada principalmente para manipular modelos UML. Las siguientes son sus funciones principales:

- Crear y editar los diagramas de estructura, comportamiento y secuencia y tiempo de UML de manera visual utilizando como modelo de datos Eclipse UML2.
- Definición de templates los cuales pueden ser utilizados posteriormente para crear modelos UML con mayor facilidad.
- Dar soporte completo para la documentación de los modelos a partir de editores RTF asociados a los mismos.
- Exportación de diagramas a diferentes formatos.
- Capacidad de refactoring de los elementos UML. Por ejemplo, permite renombrar o mover elementos sin alterar la validez de los modelos.

Otra de las características importante que será la que utilizaremos en este trabajo es la capacidad de crear y manipular perfiles UML. La herramienta permite crear estereotipos e indicar a que elementos de la metaclase de UML pueden aplicarse de manera gráfica. Además permite manipular las propiedades de los estereotipos y definir herencia entre los mismos.

Una vez creado el perfil, la herramienta permite definir diagramas UML aplicando sobre ellos los perfiles definidos por UML o los definidos por el usuario. Cuando un perfil es aplicado, se adiciona a la vista de propiedades los estereotipos definidos por el perfil para que puedan ser aplicados a los elementos UML correspondientes.

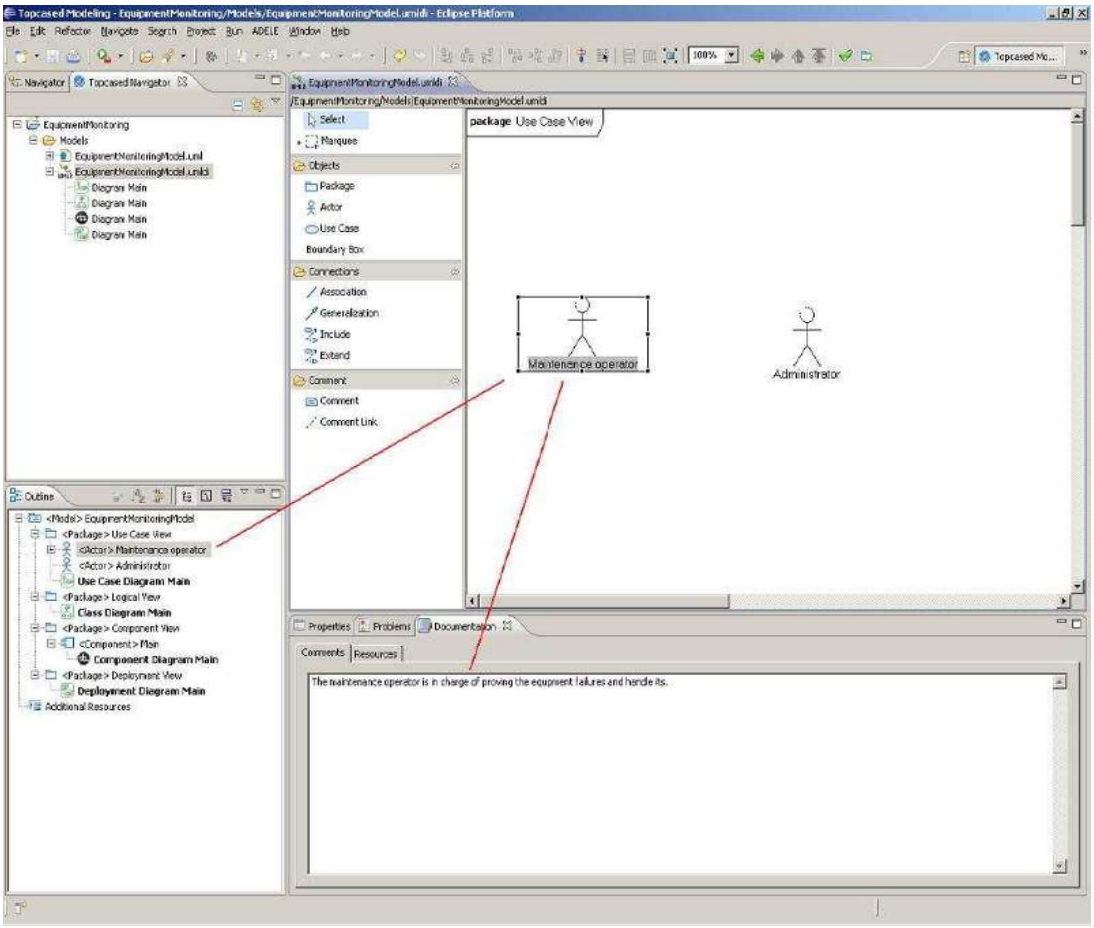

Figura 12 - Pantalla de TopCased

4.4 Java

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

Las aplicaciones Java están típicamente compiladas en un *bytecode*, aunque la compilación en código máquina nativo también es posible. En el tiempo de ejecución, el *bytecode* es normalmente interpretado o compilado a código nativo para la ejecución, aunque la ejecución directa por hardware del *bytecode* por un procesador Java también es posible.

El lenguaje Java se creó con cinco objetivos principales:

- Debería usar la metodología de la programación orientada a objetos.
- Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
- Debería incluir por defecto soporte para trabajo en red.
- Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
- Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

El siguiente código es un ejemplo de cómo se implementa un clásico “Hola mundo” utilizando Java.

```
public class Example {

      public static void main(String[] args) {
            System.out.println("¡Hola, mundo!");
       }
     }
```

Figura 13 - Java "Hola Mundo"

4.4.1 JUnit

JUnit es un Framework creado por Erich Gamma y Kent Beck que es utilizado para realizar pruebas de unidad en el lenguaje de programación Java. Es un conjunto de clases que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

También, el Framework es un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

Más allá de que JUnit está en su versión 4, para esta tesis decidimos el uso de su versión 3 debido a que es más intuitiva al momento de derivar código fuente.

La clase central de JUnit es TestCase. Un TestCase define el fixture para ejecutar múltiples test. Para definir un TestCase debemos:

- Implementar una subclase de TestCase.
- Definir variables de instancia para almacenar el estado del fixture.
- Inicializar el conjunto de datos del test redefiniendo el método setUp();
- Hacer una limpieza de los datos redefiniendo el método tearDown();

```
public class PersonaTest extends TestCase {

private Persona persona;

    protected void setUp() {
       persona = new Persona();
    }

    protected void tearDown() {
       persona = null;
    }

 }
```

Figura 14 - Ejemplo de TestCase

Para cada test que se desea implementar se debe definir un método cuyo nombre comienza con *test* el cual interactuará contra el conjunto de datos y realizará aserciones sobre los mismos.

```
public void testCambiarNombre() {
    persona.setName("Anibal");
    assertEquals("Anibal", persona.getName());
}
```

Figura 15 - Ejemplo de método de test

Una vez definidos los métodos, podemos ejecutar el los test. Para ello crearemos una instancia de la subclase creada y le indicaremos qué test ejecutar.

```
TestCase test= new MathTest("testAdd");
test.run();
```

Figura 16 - Ejemplo de la ejecución de un test

JUnit permite crear ejecuciones de varios test utilizando el concepto de TestSuite.

```
public static Test suite() {
    suite.addTest(new MathTest("testAdd"));
    suite.addTest(new MathTest("testDivideByZero"));
    return suite;
 }
```

Figura 17 - Ejemplo de la ejecución de un Test Suite

4.4.2 JMock

En la programación orientada a objetos se llaman objetos simulados (pseudoobjetos, *mock object*, objetos de pega) a los objetos que imitan el comportamiento de objetos reales de una forma controlada. Se usan para probar a otros objetos en pruebas de unidad que esperan mensajes de una clase en particular para sus métodos, al igual que los diseñadores de autos usan un *crash dummy* cuando simulan un accidente.

En los test de unidad, los *objetos simulados* se usan para simular el comportamiento de objetos complejos cuando es imposible o impracticable usar al objeto real en la prueba. De paso nos resuelve el problema del caso de objetos interdependientes, que para testear el primero debe ser usado un objeto no testeado aún, lo que invalida la prueba: los *objetos simulados* son muy simples de construir y devuelven un resultado determinado y de implementación directa, independientemente de los complejos procesos o interacciones que el objeto real pueda tener.

Los *objetos simulados* se usan en lugar de objetos reales que tengan algunas de estas características:

- Devuelven resultados no determinísticos.
- Su estado es difícil de crear o reproducir (por ejemplo errores de conexión)
- Es lento (por ejemplo el resultado de un cálculo intensivo o una búsqueda en una DB)
- El objeto todavía no existe o su comportamiento puede cambiar.

- Debería incluir atributos o métodos exclusivamente para el testeo.

Los objetos simulados para imitar al objeto real deben imitar su misma interfaz.

JMock es una librería que es utilizada para implementar objetos *mock* de manera declarativa.

```
final GreetingTime gt = context.mock(GreetingTime.class);
Greeting g = new Greeting();
g.setGreetingTime(gt);

      context.checking(new Expectations(){{

      one(gt).getGreeting();

      will(returnValue("Good afternoon"));
}});
```

Figura 18 - Ejemplo de la configuración de JMock

5 Diseño de la solución

En este capítulo describiremos el diseño de una solución que permitirá generar código fuente a partir de modelos UML desarrollados con el perfil U2TP. En primer lugar se presenta una explicación conceptual de la solución propuesta. Luego se detalla la arquitectura que dará soporte a dicha solución.

5.1 Solución propuesta

Para poder definir correctamente el diseño de la solución hemos restringido el problema a resolver con mayor precisión. Luego de haber analizado en detalle la especificación del perfil U2TP hemos decidido considerar sólo la parte estructural del mismo. El motivo para tomar esta decisión fue que, con la complejidad provista por la parte estructural del perfil, existe una gran variedad de elementos para el desarrollo de esta tesis. Esto nos permite capitalizar mejor nuestros esfuerzos y brindar una solución concreta y funcional que represente un verdadero aporte a la comunidad MDE.

Otro punto importante es agregar esfuerzo en la definición de una solución modular, permitiendo determinar con claridad los diferentes puntos de trabajo y facilitar las futuras extensiones de la herramienta para nuevas características. Estas posibles extensiones serán detalladas luego en el capítulo Trabajos futuros.

Teniendo en cuenta estos aspectos, la solución propuesta se divide en cinco etapas.

5.1.1 Especificación gráfica de modelos UML+U2TP

Esta etapa tiene como objetivo permitir al usuario definir de manera gráfica modelos UML incorporando elementos del perfil U2TP. Para realizar esta tarea debemos extender el Framework Eclipse implementando el perfil U2TP. De esta forma, el perfil puede ser utilizado a través de plugins para edición gráfica de diagramas UML. En particular, proponemos utilizar el plugin TopCased.

5.1.2 Definición del metamodelo U2TP

Como resultado de esta etapa se espera obtener un metamodelo independiente a UML, que denominaremos metamodelo U2TP. En base a éste se podrán crear modelos que sinteticen los conceptos de testing contenidos en los modelos creados por el usuario. Es decir, a partir de un modelo UML+U2TP producido por el usuario se crea otro modelo que sólo contiene los conceptos de testing del primero. Este modelo es creado como una instancia del metamodelo U2TP.

Obviamente, los elementos de un modelo basado en el metamodelo U2TP están organizados de manera de simplificar la interpretación de los conceptos de testing. Esto facilitará el proceso de generación de código para los casos de test.

5.1.3 Transformación PIM a PSM

Esta etapa tiene por objetivo recibir un modelo UML+U2TP, creado con la herramienta TopCased, y producir dos modelos intermedios. El primero es un modelo UML equivalente al modelo original pero sin ningún aspecto perteneciente al perfil U2TP. El segundo modelo, basado en el metamodelo U2TP, está creado a partir de los elementos del perfil U2TP aplicados sobre el UML entrante. Ambos modelos son creados mediante transformaciones M2M utilizando QVT Operational.

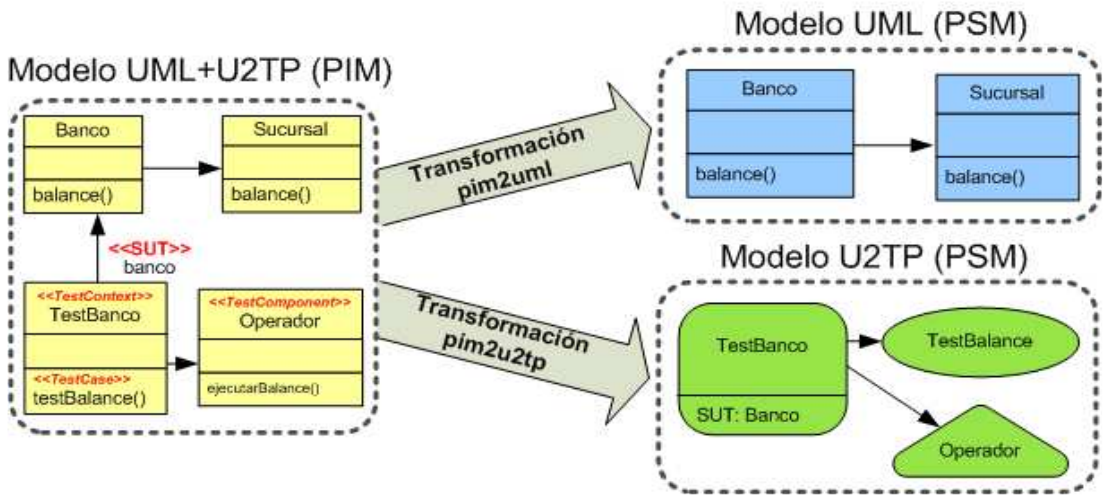

Figura 19 - Transformaciones PIM a PSM

Los modelos intermedios son elementos temporales que serán consumidos por las etapas posteriores de la generación de código. Cada uno tiene como objetivo representar de manera específica una parte de la información contenida en el modelo PIM entrante. Como se verá a continuación, esta división simplificará significativamente la derivación del código fuente. Cabe destacar que estos modelos son transparentes al usuario final.

5.1.4 Derivación de código fuente

Esta etapa tiene como objetivo tomar los modelos intermedios creados en la etapa anterior y generar código fuente utilizando una tecnología específica. Aquí volvemos a aplicar el concepto de modularización y resolvemos el problema en dos partes.

La modularización de la generación de código en dos etapas, además de simplificar el proceso, nos brinda un esquema flexible para derivar código en múltiples lenguajes de programación y Frameworks de testing. En particular, para la implementación de esta tesis, hemos elegido el lenguaje Java y los Frameworks JUnit y JMock.

5.1.4.1 Derivación de código fuente para las clases del dominio

Se toma como entrada el modelo intermedio UML generado en la etapa anterior. Al estar libre de conceptos de U2TP, sólo contiene información concerniente a clases del dominio del modelo entrante. De esta forma se produce el código fuente para las clases del dominio en un lenguaje de programación específico (Java, C++, PHP u otro). Esta tarea es delegada completamente a Acceleo debido a que es una de las funcionalidades provistas por dicho plugin. La derivación de modelos UML a código fuente es un tema ya solucionado por la comunidad MDA y en esta tesis será reutilizado para desarrollo.

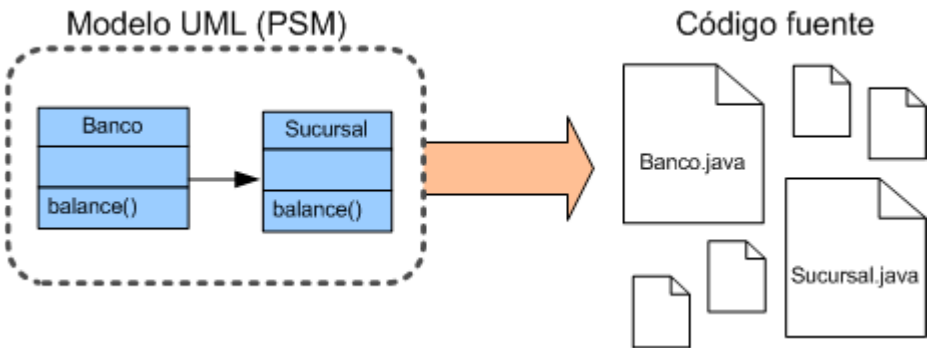

Figura 20 - Derivación de código para las clase de dominio

5.1.4.2 Derivación de código fuente para los casos de test

Se toma como entrada el modelo intermedio U2TP generado anteriormente. Dicho modelo sólo contiene información concerniente a aspectos de testing. De esta manera se genera el código fuente para los casos de test. Es necesario que el código fuente esté escrito en el mismo lenguaje que se utilizó en el punto anterior (para generar el código de las clases del dominio). De otro modo el código resultante sería inválido. Por ejemplo, si el código fuente para las clases del domino fue generado en Java, el código para los casos de test debería ser generado utilizando un Framework de testing Java, como JUnit o TTCN-3.

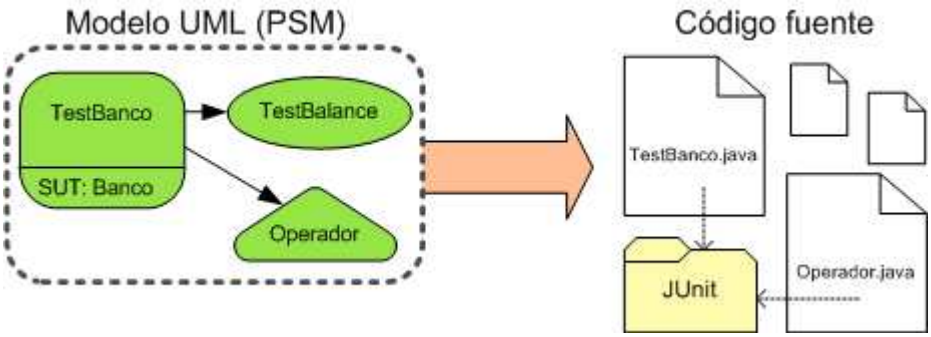

Figura 21 - Derivación de código para los casos de test

Para lograr esta tarea se utiliza el lenguaje de plantillas de Acceleo, determinando un mapeo entre los elementos del modelo U2TP y código fuente resultante. Cabe destacar que los Framework de testing existentes no siempre implementan todos los elementos del perfil U2TP. Es por esta razón que, al momento de desarrollar esta etapa, hay que adaptar el Framework de testing elegido para dar soporte a los elementos de U2TP.

La integración de las etapas anteriores se realiza mediante la utilización de cadenas Acceleo. Estas cadenas son un mecanismo provisto por el Framework Acceleo para la ejecución ordenada de un conjunto de acciones. Específicamente, se utilizan acciones de transformación M2M para la etapa de transformación PIM a PSM, y acciones de transformación M2T para la generación de código fuente.

A continuación se detallará la arquitectura para dar soporte a la solución propuesta.

5.2 Arquitectura

Como hemos visto en la sección anterior, la arquitectura de la solución debe proporcionar un esquema flexible para la generación de código fuente en múltiples lenguajes de programación y Frameworks de testing. Para lograr esto es necesario un esquema modular, donde cada componente se encargue de la generación para un lenguaje o Framework específico.

La arquitectura de nuestra solución está compuesta por módulos de software (plugins de Eclipse) que recaen en una de las siguientes capas:

- capa independiente del lenguaje de programación (capa ILP).
- capa específica del lenguaje de programación (capa ELP).
- capa específica del Framework de testing (capa EFT).

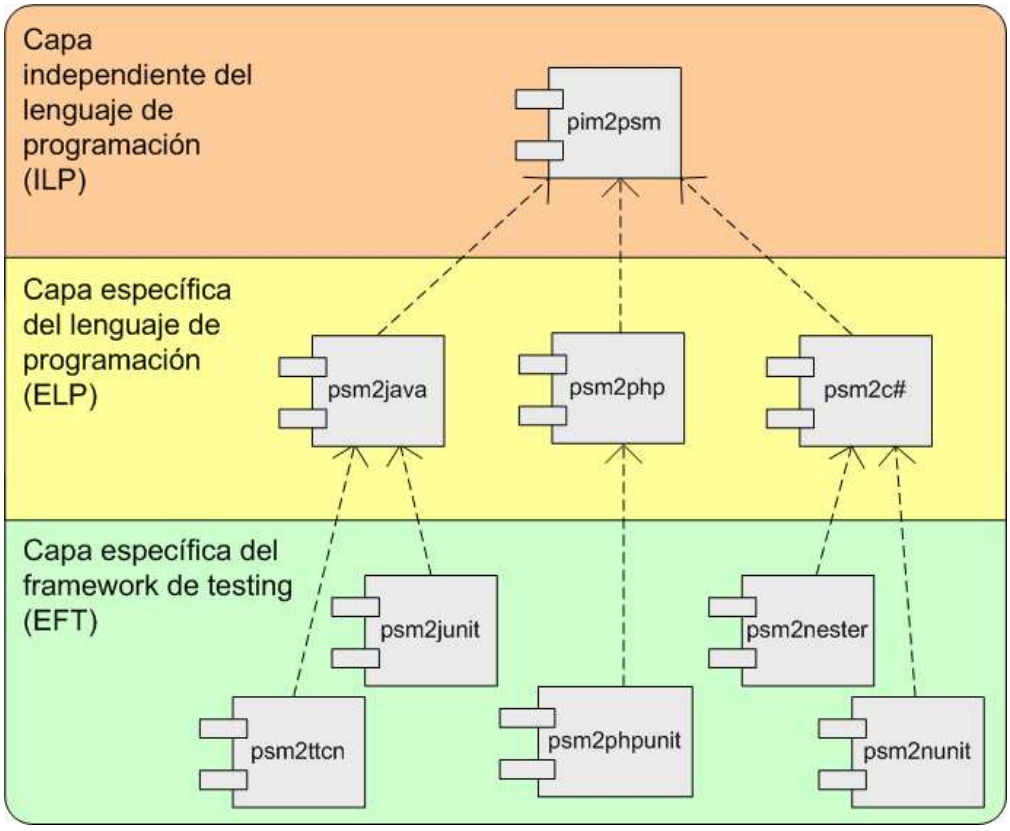

Figura 22 - Arquitectura del diseño de la solución

Los módulos de una capa determinada dependen de módulos que se encuentran en la capa superior. Por ejemplo, el módulo de generación de código JUnit (psm2junit) en la capa EFT depende del módulo de generación de código Java (psm2java) en la capa ELP. Más adelante se dará más detalle de este tema.

A continuación se describen cada una de las capas de la arquitectura.

5.2.1 Capa independiente del lenguaje de programación (ILP)

Para la primera, la capa independiente del lenguaje de programación, hay un solo módulo (pim2psm). Esto se debe a que dicho módulo contiene la funcionalidad más abstracta, la cual será utilizada por el resto de los módulos. Básicamente, este plugin tiene tres funciones:

- extender el Framework Eclipse con una implementación del perfil U2TP.
- dar una implementación para el metamodelo U2TP
- permitir transformar un modelo UML+U2TP en dos modelos intermedios: un modelo UML equivalente al modelo de entrada pero libre de conceptos de test, y un modelo U2TP que sólo contenga los elementos de test del modelo entrante.

Como se puede observar, estas funciones se corresponden con las tres primeras etapas de la solución propuesta anteriormente. Además, no están ligadas a ninguna plataforma en particular, de manera que pueden ser reutilizadas por los módulos de las capas inferiores.

Por último, es importante resaltar que para futuras extensiones de la herramienta (hacia nuevos lenguajes o Frameworks de test) los desarrolladores no incluirán módulos en esta capa, puesto que el único módulo es desarrollado como parte de esta tesis.

5.2.2 Capa específica del lenguaje de programación (ELP)

La capa específica del lenguaje de programación está compuesta por un conjunto de plugins, uno por cada lenguaje de programación. En otras palabras, para generar código fuente en un lenguaje determinado es necesario que exista un módulo que lo represente en esta capa. Estos módulos toman como entrada el modelo intermedio UML (libre de conceptos de test) generado por el módulo pim2psm. A partir del mismo, producen el código fuente para las clases del dominio en un lenguaje de programación en particular. Esta derivación de código es lograda mediante módulos preexistentes de Acceleo (disponibles en http://www.acceleo.org/pages/modules-repository).

Por lo anterior, podemos decir que los plugins de la segunda capa cumplen con el objetivo de la cuarta etapa de la solución propuesta. Además, si bien cada uno de ellos se dedica a generar código en un lenguaje determinado, ninguno está ligado a un Framework de testing en particular (de hecho no generan código de test). Esto permite reutilizar un módulo de generación de código específico de un lenguaje para múltiples Frameworks de testing.

Se sugiere, para los plugins de la capa ELP, el nombre psm2nombre_lenguaje. Por ejemplo, psm2java sería el nombre apropiado para el módulo de generación de código en lenguaje Java.

5.2.3 Capa específica del Framework de testing (EFT)

Hasta el momento tenemos resuelta la generación de código para las clases del dominio. Sólo nos resta completar la quinta etapa de la solución propuesta, es decir, la generación de código para los casos de test. Así surge la tercera capa de la arquitectura: la capa específica del Framework de testing. Esta capa también está compuesta por un conjunto de plugins: uno por cada Framework de testing. En este caso, para generar código fuente en un Framework de testing en particular es necesario que exista un módulo que lo represente en esta capa. Estos módulos toman como entrada el modelo intermedio U2TP (que sólo contiene elementos de test) generado por el módulo pim2psm. A partir del mismo, produce el código correspondiente para los casos de test en un Framework de testing específico.

Como se mencionó anteriormente, la generación de código fuente para los casos de test es lograda con plantillas de Acceleo. Para esto, se debe crear una plantilla por cada elemento del metamodelo U2TP. Dicha plantilla determina cómo va a ser derivado el código fuente para ese elemento. Luego son procesados uno a uno los elementos del modelo intermedio U2TP, aplicando las plantillas correspondientes para generar el código de testing.

Para los plugins de la capa EFT, se sugiere el nombre psm2nombre_fmk_testing. Por ejemplo, psm2junit sería el nombre apropiado para el módulo de generación de código en JUnit.

5.2.4 Dependencia entre capas

Por último, es importante destacar la dependencia que existe entre los plugins que componen la arquitectura. En primer lugar, todos los plugins de la segunda capa dependen del plugin pim2psm de la primera capa:

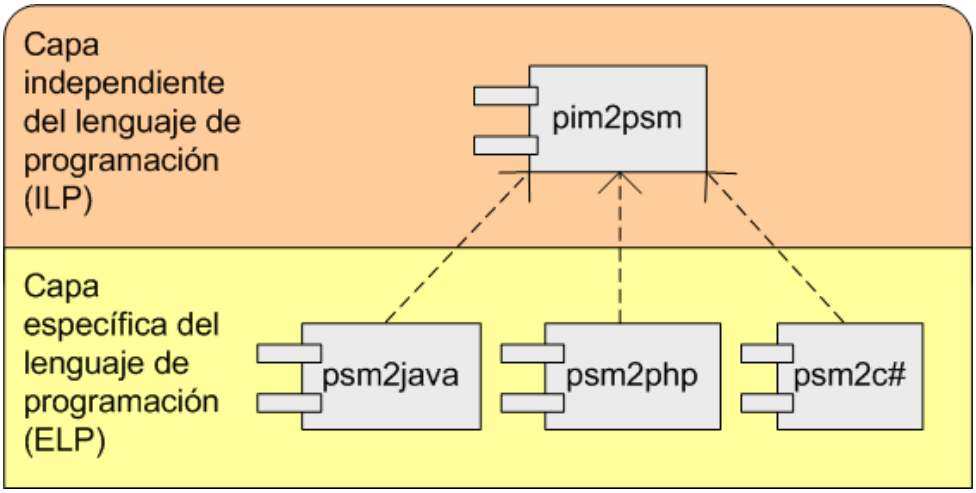

Figura 23 - Dependencias entre capas ILP y ELP

Los plugins de la capa ELP necesitan disponer del modelo intermedio U2TP para poder generar el código fuente de las clases de dominio. Puesto que el encargado de producir los modelos intermedios es el plugin pim2psm, quedan justificadas dichas dependencias.

Por otra parte, todos los plugins de la tercera capa dependen de plugins de la segunda capa. En particular, un plugin de la capa EFT, dedicado a generar código en un Framework de testing, depende del plugin de la capa ELP que representa al lenguaje en que está escrito dicho Framework:

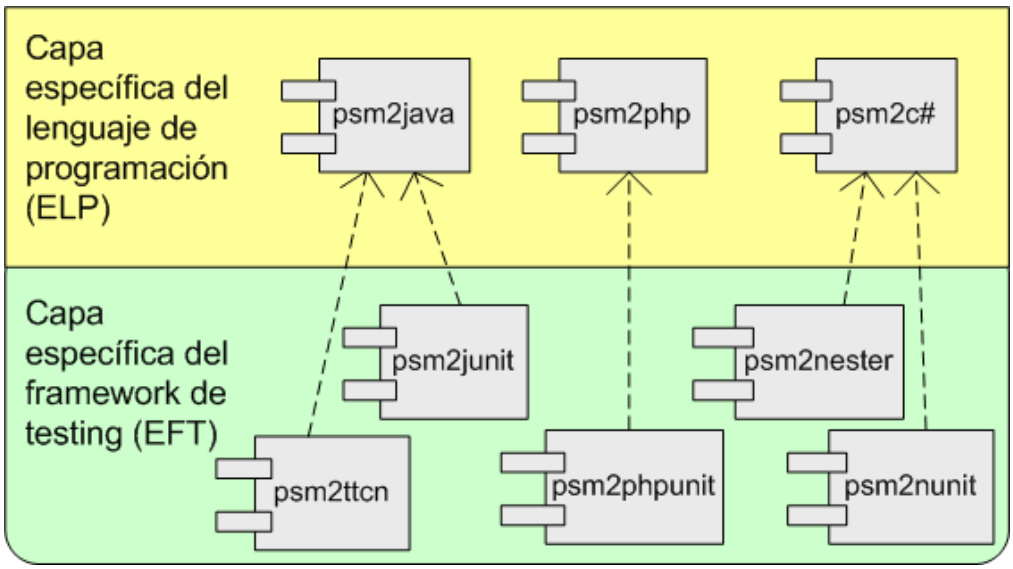

Figura 24 - Dependencias entre la capa ELP y EFT

Como sabemos, los casos de test son creados para verificar el funcionamiento de las clases de dominio. A partir de esto, es evidente que el código generado para las clases del dominio y el código generado para los casos de test deben estar escritos en el mismo lenguaje. Así quedan justificadas las dependencias desde la capa EFT hacia la capa ELP.

6 Implementación de la solución

En este capítulo se detalla la implementación concreta de los módulos que componen la arquitectura de la solución. Como resultado obtendremos un conjunto de plugins de Eclipse que juntos brindan a los usuarios una herramienta visual para crear modelos UML+U2TP y derivar código fuente en múltiples lenguajes de programación y Frameworks de testing.

Para ejemplificar la arquitectura de la solución se implementarán los plugins necesarios para derivar código en lenguaje Java, utilizando el Framework de testing JUnit. Estos módulos servirán como punto de partida en las futuras extensiones para otros lenguajes de programación y Frameworks de testing.

Puesto que para la primera capa existe un solo módulo (pim2psm) y dicho módulo es desarrollado como parte de esta tesis, aquellos desarrolladores que deseen extender esta herramienta, incorporando nuevos módulos, deberán enfocar sus esfuerzos en la segunda y en la tercera capa de la arquitectura. Es decir, suponiendo que se desee extender la herramienta para un Framework de testing X escrito en el lenguaje Y, se deberá:

- implementar, si no existe, un plugin para el lenguaje de programación Y en la capa especifica del lenguaje (ELP).
- implementar el plugin para el Framework de testing X en la capa específica del Framework de testing (EFT).

Como hemos visto cada capa tiene objetivos bien definidos. Es por esto que la implementación de un módulo depende de la capa a la que pertenece. A continuación se detalla la implementación de módulos en las tres capas de la arquitectura.

6.1 Implementación del módulo para la capa ILP

En la capa independiente del lenguaje de programación existe un solo módulo: el plugin pim2psm. Como resultado de este apartado obtendremos una implementación del mismo. Es importante remarcar que este punto de la implementación es realizado una sola vez y se incluye como parte del desarrollo de esta tesis. Para las futuras extensiones a esta arquitectura sólo es necesario desarrollar módulos para las dos capas siguientes.

Como se mencionó en el capítulo anterior, las funciones del módulo pim2psm se corresponden con los tres primeros puntos de la solución propuesta. Es decir, el plugin debe:

- Permitir la especificación gráfica de modelos UML+U2TP.
- Brindar una definición del metamodelo U2TP.
- Permitir la transformación de modelos PIM a PSM.

El primer paso en la implementación del plugin pim2psm es la creación del plugin en sí. Esta es una tarea relativamente sencilla, simplificada por el plugin PDE (Plugin Development Environment) incluido en el Framework estándar de Eclipse.

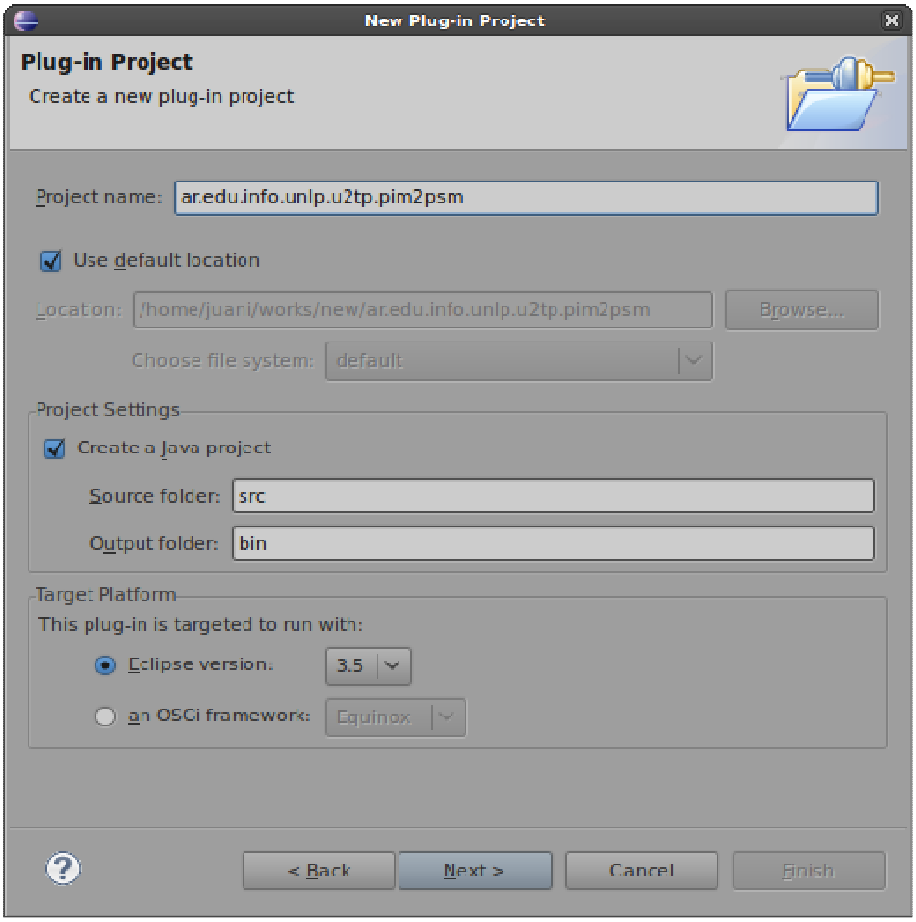

Figura 25 - Creación del plugin pim2psm

Una vez que el plugin ha sido creado, la estructura del proyecto es similar a la siguiente:

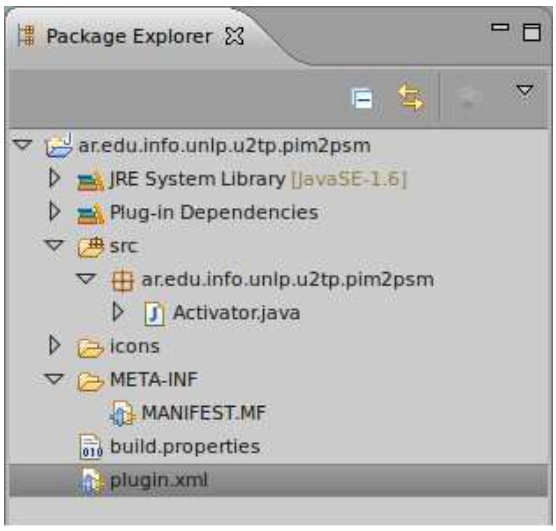

Figura 26 - Estructura del plugin pim2psm

La tarea siguiente es implementar la funcionalidad del plugin. Para esto debemos:

- crear una implementación del perfil u2tp.
- crear una implementación para el metamodelo u2tp.
- crear las trasformaciones PIM a PSM.

Los siguientes apartados detallan cada uno de estos pasos.

6.1.1 Implementación del perfil U2TP

La implementación de un perfil UML dentro del Framework Eclipse consta de dos etapas: la creación del perfil y su posterior publicación

Para crear un perfil, se puede utilizar el editor de modelos provisto por el plugin UML2 o bien otra herramienta que facilite dicha tarea. En nuestro desarrollo, la creación del perfil se realizó utilizando el plugin TopCased. Esta herramienta simplificó significativamente el trabajo, gracias a que dispone de un template específico para la creación de perfiles UML:

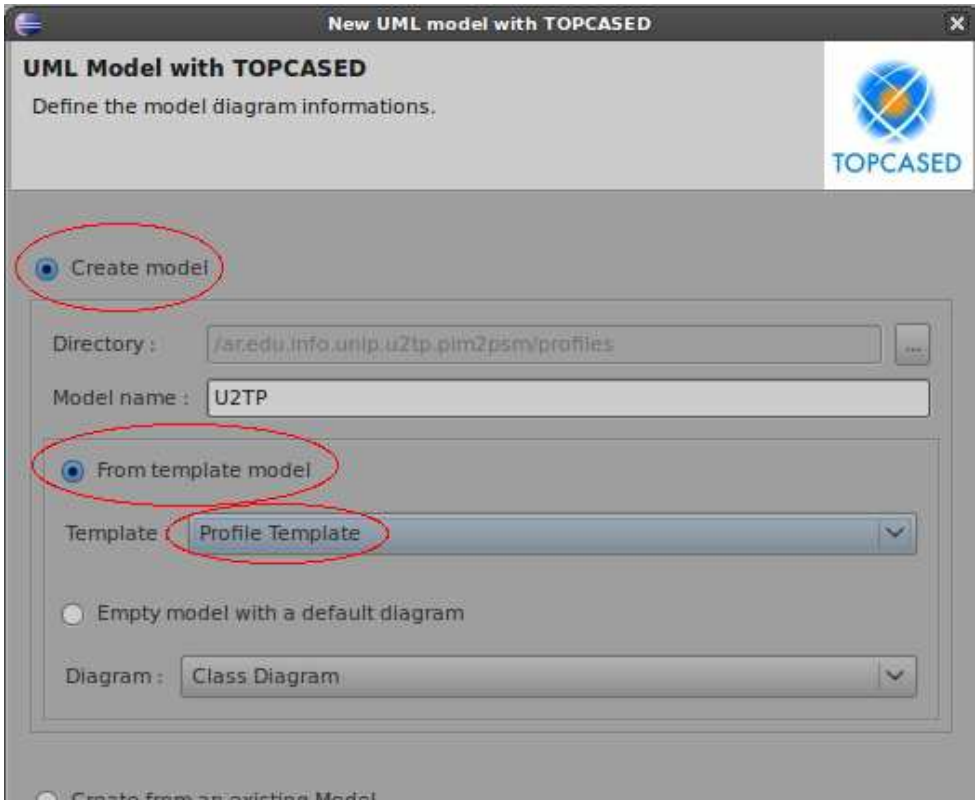

Figura 27 - Wizard de creación el perfil

La siguiente tarea es definir los elementos que componen al perfil. Cabe aclarar que la implementación del perfil U2TP dentro del entorno Eclipse se realiza directamente, sin necesidad de interpretaciones a su especificación. La creación de los elementos del perfil se realizó a través del entorno gráfico que provee TopCased, arrastrando los componentes desde una paleta, ubicándolos en el modelo del perfil y configurando sus propiedades:

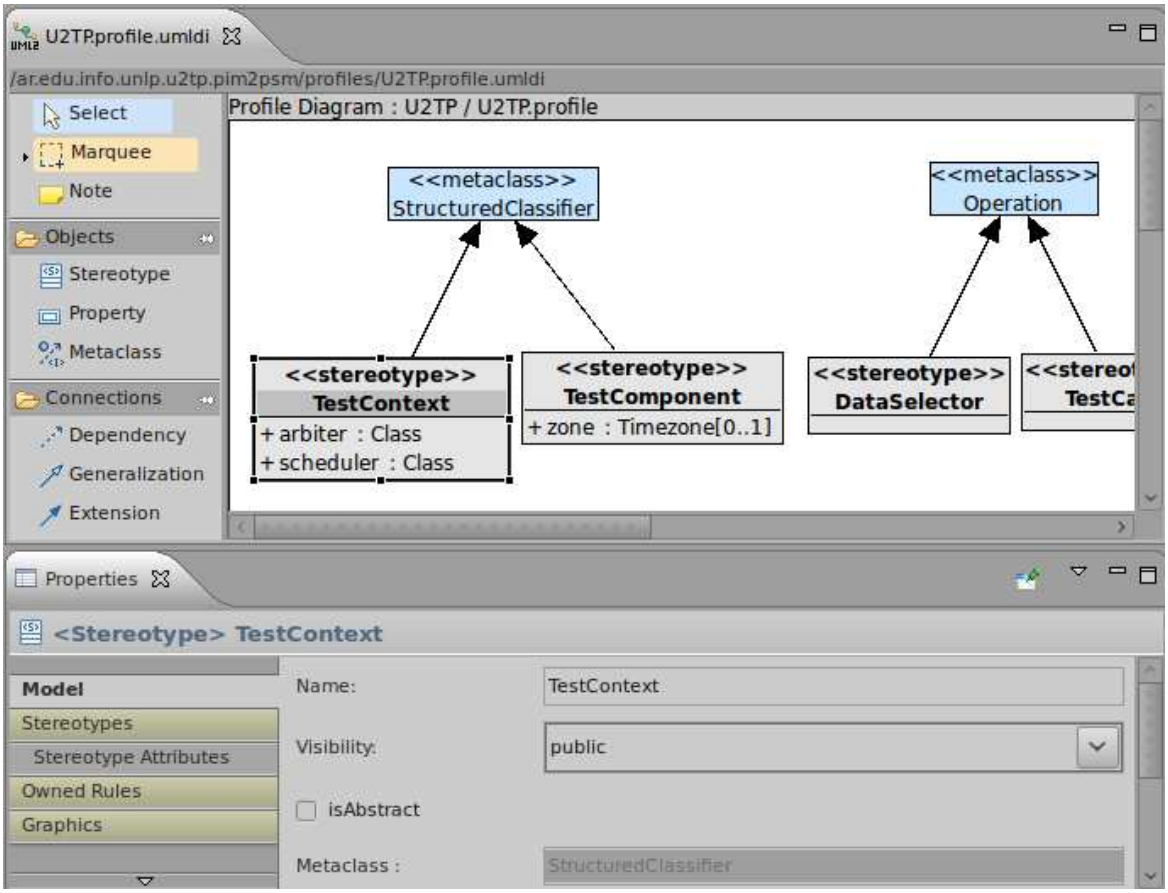

Figura 28 - Definición del perfil U2TP utilizando TopCased

Una vez completado el perfil, debemos publicarlo dentro de Eclipse para que pueda ser utilizado por el resto de los plugins. Para esta tarea modificamos el archivo ***plugin.xml*** agregando dos extensiones:

*ar.edu.info.unlp.u2tp.pim2psm

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
     point="org.eclipse.emf.ecore.uri_mapping">
    <mapping
       source="pathmap://UML2_PROFILES/"
       target="platform:/plugin/ar.edu.info.unlp.u2tp.pim2psm/profiles/">
    </mapping>
  </extension>
  <extension
     point="org.eclipse.uml2.uml.dynamic_package">
    <profile uri="http://www.omg.org/U2TP"
             location="pathmap://UML2_PROFILES/U2TP.profile.uml#_0"/>
  </extension>
```

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | plugin.xml | buil

Figura 29 - Extensión para agregar el perfil en Eclipse

Ambas son realizadas a puntos de extensión del Framework Eclipse. La primer extensión asigna una URI a la ruta (path) donde se encuentra el perfil; la segunda realiza la publicación en sí del perfil, asignándole una URI. Esa URI será utilizada por las herramientas de modelado para hacer referencia al perfil, permitiendo a los usuarios crear modelos UML e incorporar elementos del perfil.

En este punto tenemos una implementación funcional del perfil U2TP dentro del entorno Eclipse. El siguiente paso en la implementación del módulo pim2psm es la creación del metamodelo U2TP. Este metamodelo será utilizado para la creación de modelos intermedios que simplifican la generación de código fuente.

6.1.2 Implementación del metamodelo U2TP

Como se mencionó en capítulos anteriores, dentro de Eclipse los metamodelos se especifican usando el meta-metamodelo ECore. Además, la definición de un metamodelo puede hacerse de cuatro maneras diferentes, obteniendo el mismo resultado:

- usando un editor gráfico para metamodelos ECore.
- a partir de un documento XML.
- a partir de un modelo UML.
- como interfaces de Java con Anotaciones.

En nuestra implementación, la creación del metamodelo U2TP se hizo a partir de un documento XML. Más precisamente, se tomó como base el documento XML contenido en la especificación de U2TP. En este documento, la OMG propone una definición aproximada del metamodelo U2TP. Dicha definición fue completada y refactorizada para simplificar el procesamiento de las transformaciones, siempre procurando no cambiar la semántica original.

Para crear un modelo ECore a partir de un documento XML se debe:

- Abrir el diálogo "File/New/Other..."
- Expandir "Eclipse Modeling Framework" y seleccionar "EMF Model". Luego hacer click en el botón "Next".

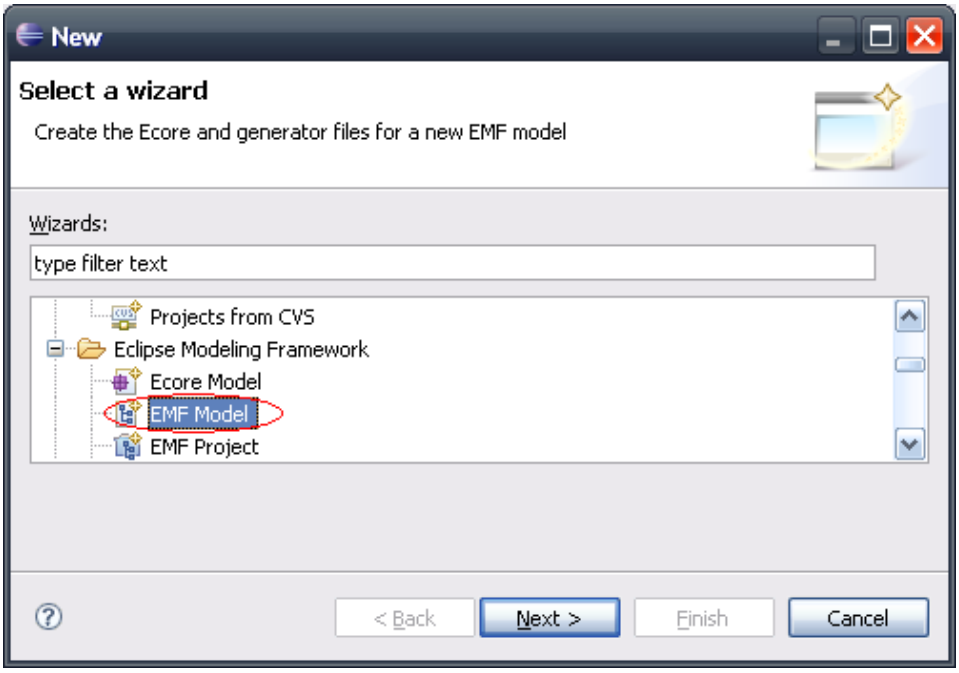

Figura 30 - Wizard EMF Model

- Seleccionar una ubicación y dar un nombre para el archivo del modelo. Luego hacer click en "Next".

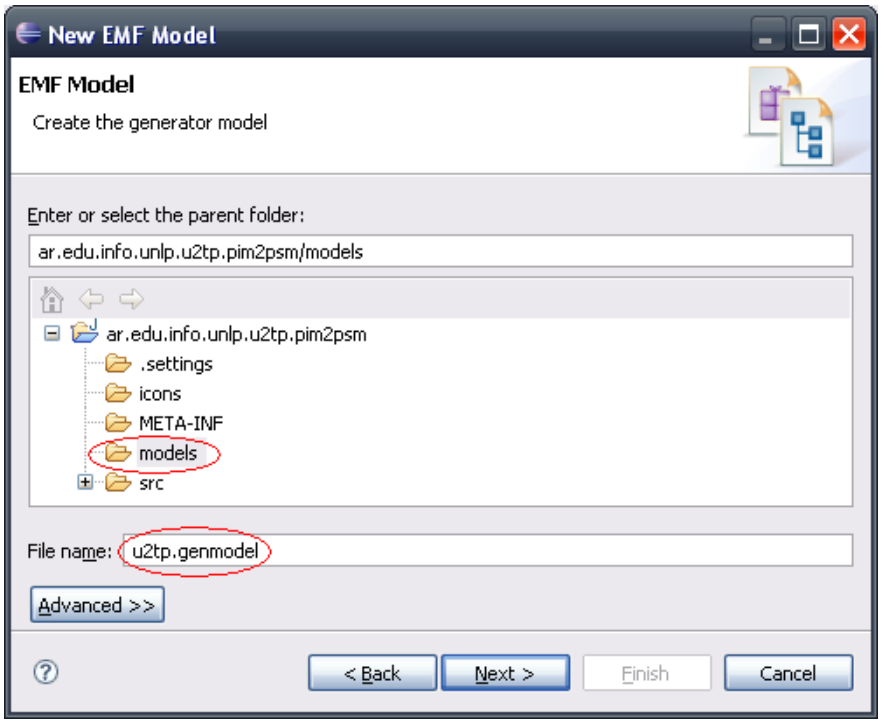

Figura 31 - Establecer nombre del modelo EMF

- Seleccionar "XML Schema" y luego hacer click en el botón "Next".

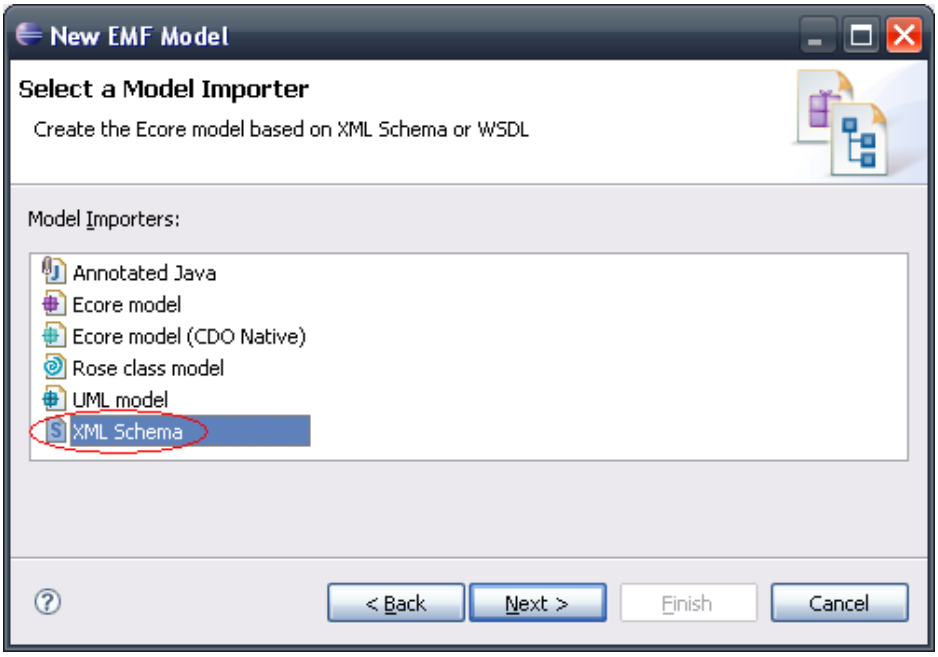

Figura 32 - Selección de XML Schema

- Hacer click en el botón "Browse", localizar el archivo XML y hacer click en "Next"

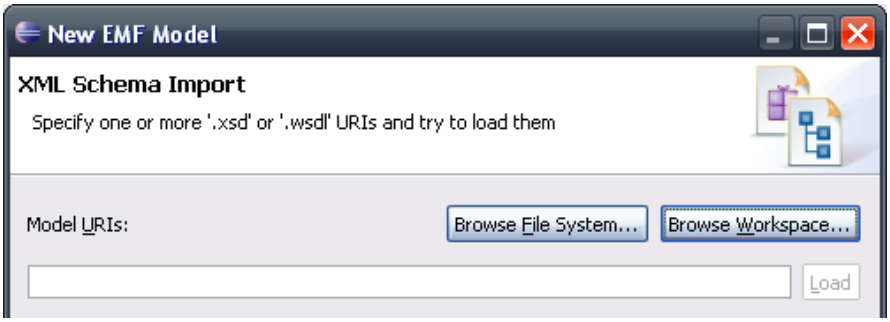

Figura 33 - Ubicación del XML Schema para importar

- En general, puede ser generado más de un paquete desde un simple archivo XML. Aquí se debe seleccionar el paquete para el cual se desea generar el modelo ECore. En este caso, el único paquete es "org.omg.u2TPSA". Luego hacer click en el botón "Finish".

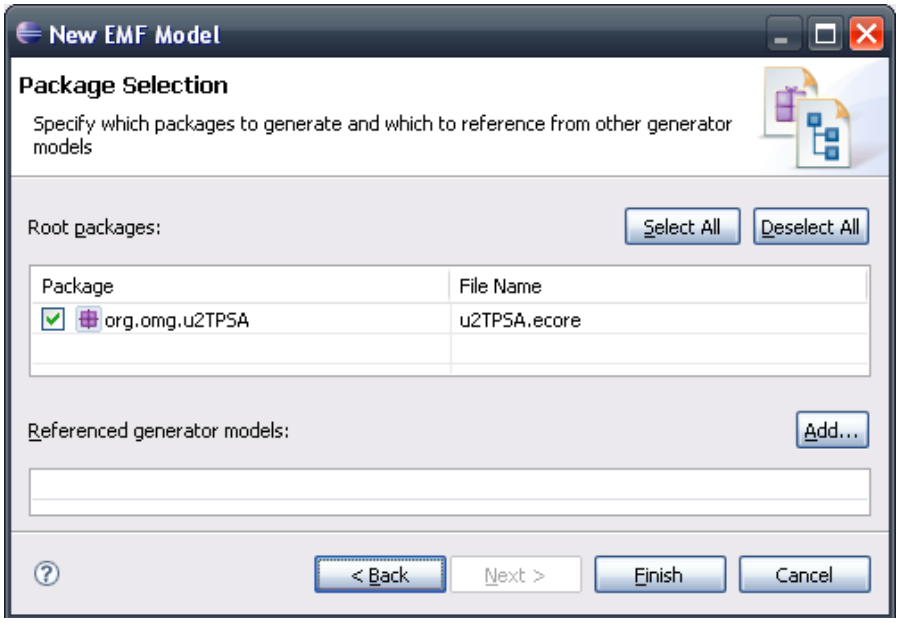

Figura 34 - Paquete que se usará para generar código

- A continuación serán creados un modelo ECore (u2TPSA.ecore) y un modelo generador (u2tp.genmodel). El modelo generador es abierto automáticamente en la vista principal.

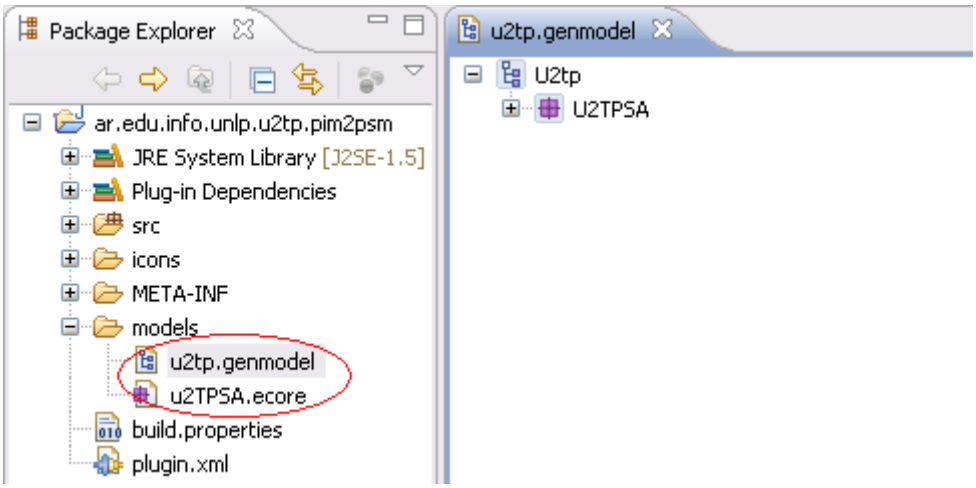

Figura 35 - Vista del modelo generador de U2TP

De manera similar a un perfil, es necesario que un metamodelo ECore sea publicado para que pueda ser utilizado por otras herramientas de Eclipse. En nuestro caso necesitamos usar el metamodelo U2TP en transformaciones QVTO. Para lograr esto, primero debemos generar el código fuente del metamodelo, haciendo click derecho en un elemento del modelo generador y eligiendo "Generate Model Code".

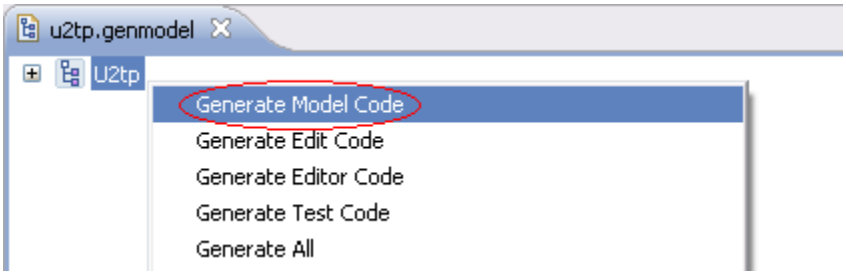

Figura 36 - Generación del modelo de código

A continuación puede observarse el código generado para las clases del metamodelo:

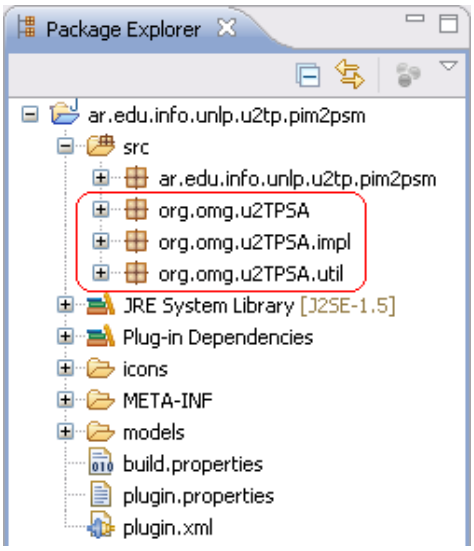

Figura 37 - Código generado a partir del modelo U2TP

Por último, se debe agregar la siguiente extensión en el archivo ***plugin.xml***:

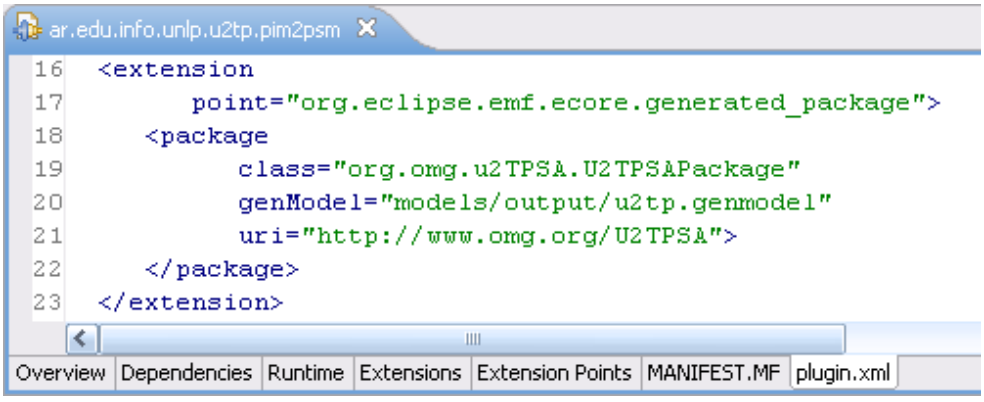

Figura 38 - Extensión que permite adicionar el modelo U2TP

Como se puede ver, se debe especificar la interface del paquete generado anteriormente, el archivo del modelo generador (genmodel) y una URI. Dicha URI permite hacer referencia al metamodelo y será utilizada más tarde en las transformaciones PIM a PSM y en la generación de código fuente.

Hasta el momento hemos implementado el perfil y el metamodelo U2TP. A continuación se explicará la creación de las transformaciones PIM a PSM, las cuales harán uso de ambos elementos para generar los modelos intermedios.

6.1.3 Implementación de las transformaciones PIM a PSM

Existen dos transformaciones PIM a PSM. Ambas toman como entrada un modelo UML enriquecido con elementos del perfil U2TP y producen como salida un modelo intermedio que será consumido más tarde en la generación de código fuente.

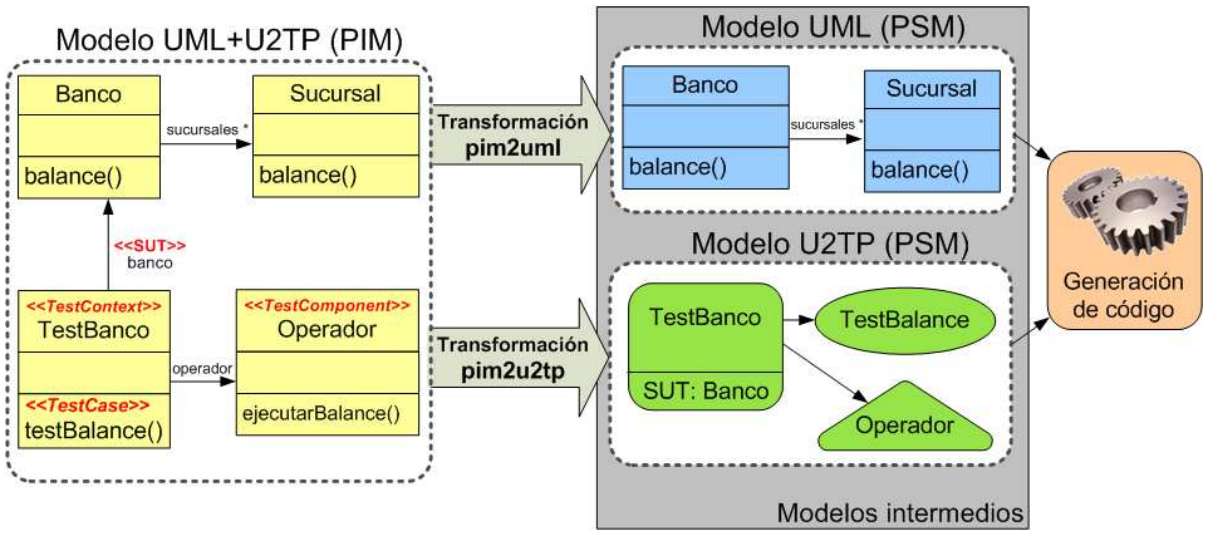

Figura 39 - Esquema de derivación de código fuente

La primera transformación, llamada **pim2uml**, produce un modelo UML equivalente al modelo de entrada pero sin conceptos de test. Para esto deberá procesar cada elemento del modelo de entrada replicándolo en el modelo de salida, ignorando aquellos elementos que estén enriquecidos con elementos de U2TP. Por ejemplo, deberá ignorar todas las clases estereotipadas con <<*TestContext*>>. Gráficamente, la transformación pim2uml tiene la siguiente forma:

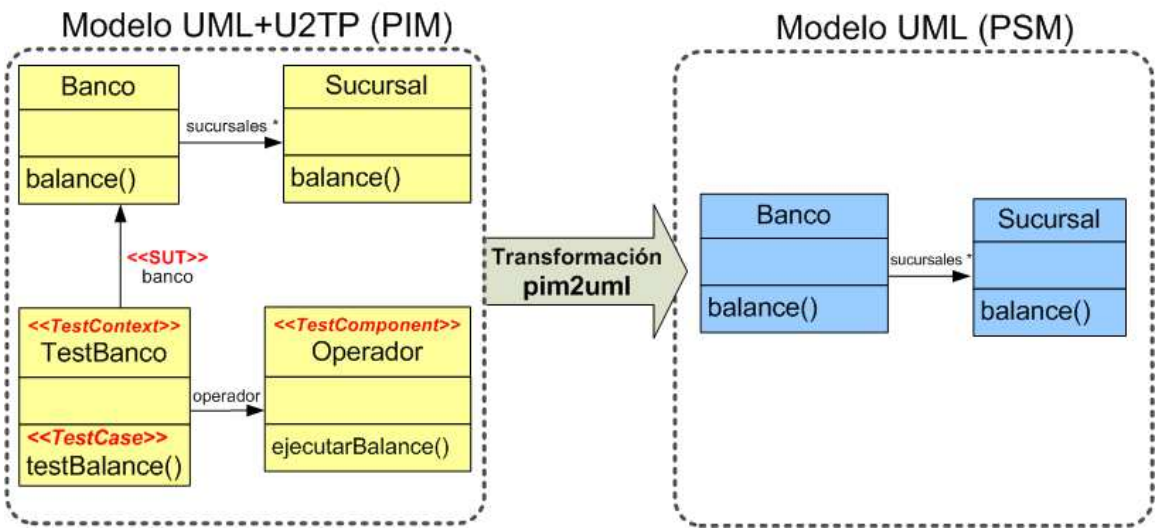

Figura 40 - Transformación al modelo UML

La segunda transformación, llamada **pim2u2tp**, produce un modelo basado en el metamodelo U2TP, creado en el apartado anterior. Dicho modelo contiene una especificación de los elementos de U2TP aplicados a los elementos del modelo entrante. En otras palabras, esta transformación produce un modelo intermedio que sólo contiene conceptos de test. Para esto deberá procesar los elementos del modelo entrante considerando sólo aquellos que estén enriquecidos con elementos de U2TP y generando para éstos una representación adecuada en el modelo de salida. Por ejemplo, para una clase del modelo UML entrante estereotipada con <<*TestComponent*>> deberá crear un elemento *TestComponent* en el modelo U2TP de salida.

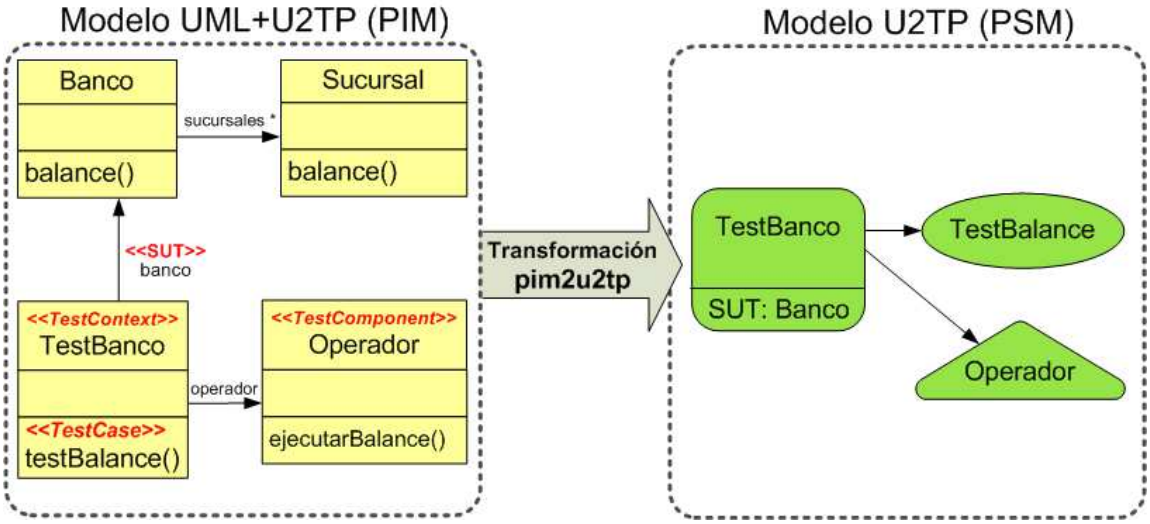

Figura 41 - Transformación al modelo U2TP

Como se mencionó antes, ambas transformaciones son implementadas con QVT Operational (QTVO). Sin entrar en detalles, cada transformación es un archivo (con extensión .qvto) que define un proceso de conversión de un modelo de entrada a un modelo de salida.

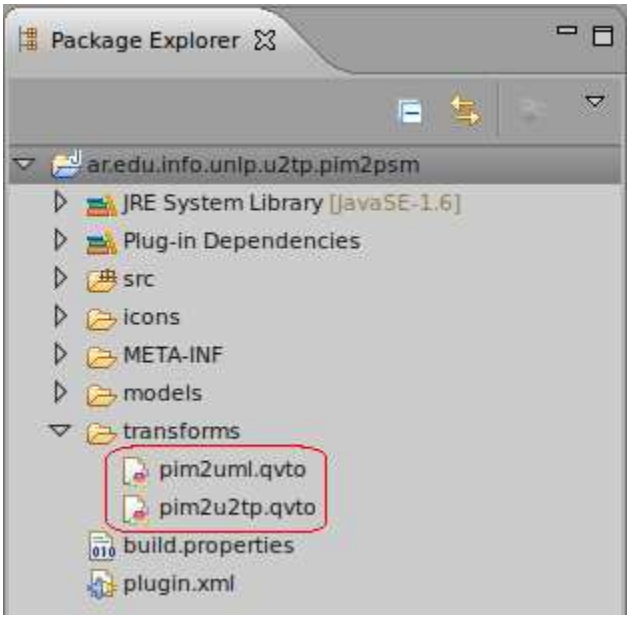

Figura 42 - Transformadores QVTO

En nuestro caso tenemos dos archivos .qvto, uno por cada transformación. Estos archivos definen en su encabezado los metamodelos involucrados en la transformación. Por ejemplo, el encabezado del segundo tiene la siguiente forma:

Figura 43 - Cabecera del archivo de QVTO

Como se puede observar, en primer lugar son declarados los metamodelos que serán utilizados en la transformación. Para esto se usa la sentencia *modeltype*, la cual define un identificador para un metamodelo y la URI del mismo. Notar que la URI del metamodelo U2TP es la misma que se utilizó anteriormente en su publicación. Más abajo se encuentra la sentencia *transformation* que define la signatura de la transformación. En ella se especifican los metamodelos de los modelos de entrada y salida, utilizando sus identificadores. Por ejemplo, en la signatura de la transformación anterior se indica que la entrada es un modelo UML y la salida es un modelo U2TP.

Luego del encabezado, una transformación QVTO define su punto de entrada principal mediante la operación *main*:

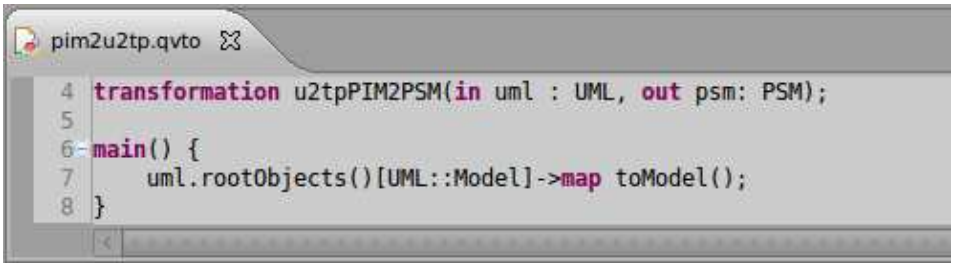

Figura 44 - Punto de entrada principal del archivo de QVTO

Esta operación no recibe parámetros y debe existir solamente una por transformación. Su ejecución se produce automáticamente después de que la transformación es instanciada e invocada. Generalmente la operación main procesa el modelo de entrada, representado por el parámetro *in*, iterando sobre sus elementos. En el ejemplo anterior, por cada elemento del modelo entrante se llama a una operación de mapeo.

Una operación de mapeo es definida con la sentencia *mapping* y establece la relación entre un elemento del modelo de entrada y un elemento del modelo de salida. Ambos tipos, los de entrada y salida, son declarados en la signatura de la operación de mapeo:

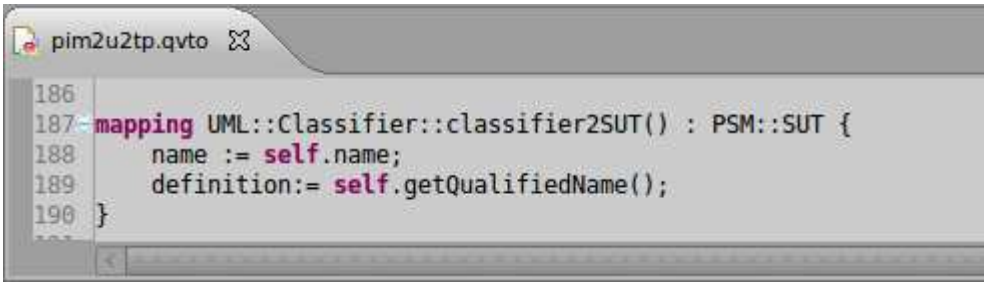

Figura 45 - Operación de mapeo de QVTO

En el caso anterior se declara la operación de mapeo *classifier2SUT* que a partir de un elemento *Classifier* de UML genera un elemento *SUT* de U2TP. En su cuerpo se incluyen una serie de asignaciones que definen las propiedades del elemento de salida en base a las propiedades del elemento entrante (referido con la palabra reservada *self*). Las propiedades del elemento de salida también pueden ser completadas a partir del resultado de la invocación a otra operación de mapeo. Por ejemplo:

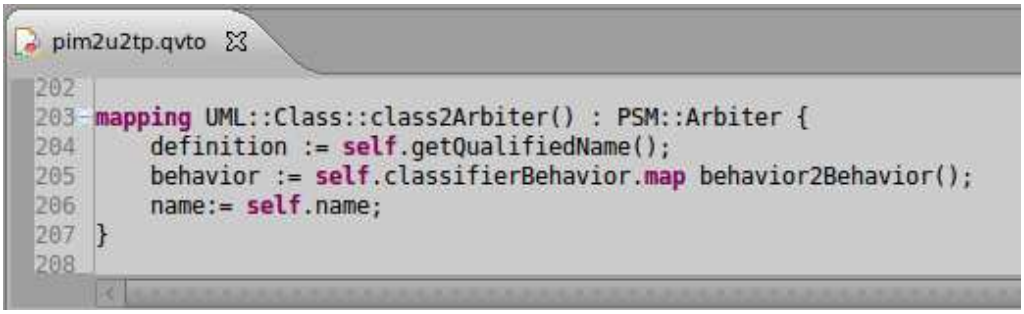

Figura 46 - Operación que deriva a otra operación de QVTO

Para invocar a una operación de mapeo se utiliza la sentencia *map.* En la invocación se definen el elemento que será mapeado y el nombre de la operación de mapeo. Como se puede observar, en el ejemplo

anterior se invoca a la operación de mapeo *behavior2Behavior* a partir de la propiedad *classifierBehavior* del elemento entrante y el resultado se almacena en la propiedad *behavior* del modelo de salida.

Las sucesivas invocaciones a operaciones de mapeo son las que permiten a una transformación cumplir con su objetivo, es decir, le permiten ir generando un modelo de salida a partir de transformaciones de elementos del modelo entrante. Por último es importante destacar que no necesariamente todos los elementos del modelo entrante deben ser mapeados al modelo de salida. Eso depende de los metamodelos involucrados y del propósito de la transformación. Por ejemplo, en nuestra implementación ambas transformaciones procesan sólo una parte de los elementos del modelo UML de entrada: la primera sólo procesa aquellos elementos que no tengan aplicados elementos del perfil U2TP, y la segunda lo contrario, es decir, sólo procesa los elementos que tengan aplicados elementos de U2TP.

Una vez creadas ambas transformaciones es necesario integrarlas de algún modo para que puedan ser accedidas desde otros plugins. Para esto se utiliza una cadena Acceleo. Como se mencionó anteriormente, una cadena permite la ejecución ordenada de un conjunto de acciones con el fin de realizar un procesamiento sobre uno o más modelos de entrada. En nuestro caso, la cadena toma como entrada un modelo UML enriquecido con el perfil U2TP, lo validará y le aplicará las transformaciones para obtener los modelos intermedios. Gráficamente la cadena tiene la siguiente forma:

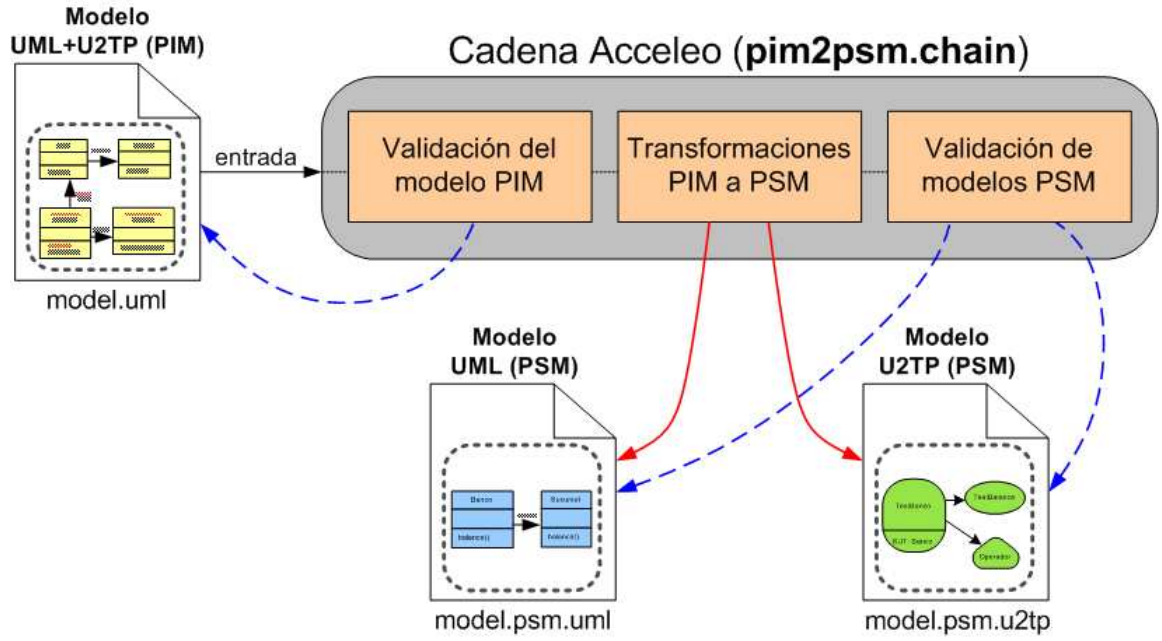

Figura 47 - Cadena de Acceleo para la transformación pim2psm

Como se puede observar son realizadas dos validaciones. Una se hace sobre el modelo PIM de entrada y la otra sobre los modelos PSM producidos por las transformaciones. Estas validaciones certifican que los modelos sean apropiados para el procesamiento, de manera de evitar errores durante el mismo. La primer validación asegura que el modelo de entrada sea correcto para la ejecución de las transformaciones PIM a PSM. Por ejemplo, verifica que aquellos elementos *Property* estereotipados con <<*SUT*>> tengan asignado un tipo. La segunda validación asegura que los modelos PSM sean apropiados para la generación de código. Por ejemplo, verifica que los elementos *TestContext* contengan al menos un elemento *SUT*.

Si bien hay muchas técnicas para validar modelos, en el desarrollo de esta tesis la validación se realiza programáticamente utilizando las librerías de EMF. Es decir, se toma el modelo de entrada como un objeto de tipo *EObject*. A partir de ese objeto se procesan recursivamente sus hijos, validando para cada uno que se cumplan las restricciones específicas de su tipo. Por ejemplo, la validación de los elementos *Property* estereotipados con <<*SUT*>> se realiza de la siguiente forma:

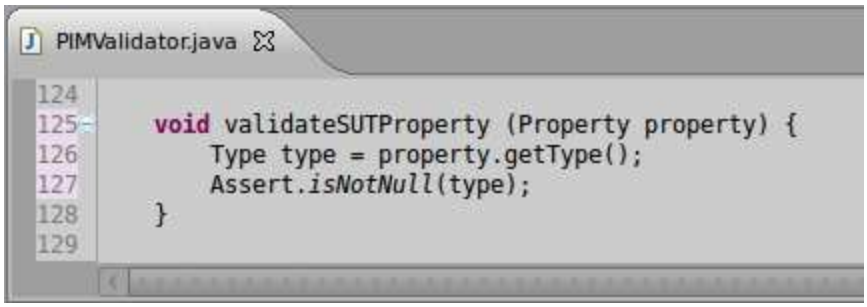

Figura 48 - Validador de QVTO

El método anterior recibe por parámetro un elemento *Property*. Para obtener el tipo de ese elemento se usa el método *getType()*. Luego se utiliza la clase *Assert* para verificar que el tipo sea distinto de *null*. En caso de que el tipo sea *null*, el método *isNotNull* lanzará una excepción abortará el procesamiento inmediatamente.

Para crear la cadena Acceleo se debe:

- Abrir el diálogo "File/New/Other...".
- Expandir "Acceleo" y seleccionar "Empty Chain". Luego hacer click en el botón "Next".

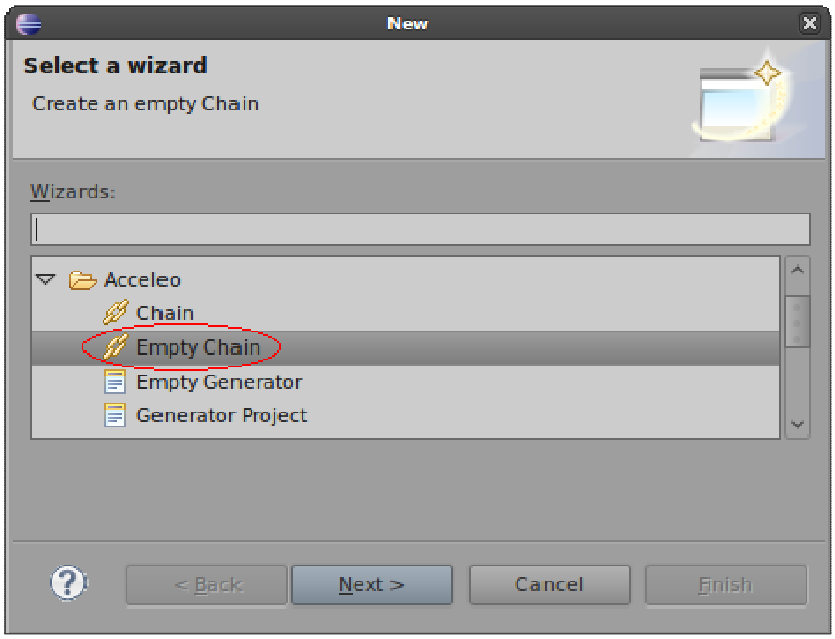

Figura 49 - Creación de una cadena de Acceleo

- Seleccionar una ubicación y dar un nombre para el archivo de la cadena. Luego hacer click en "Finish".

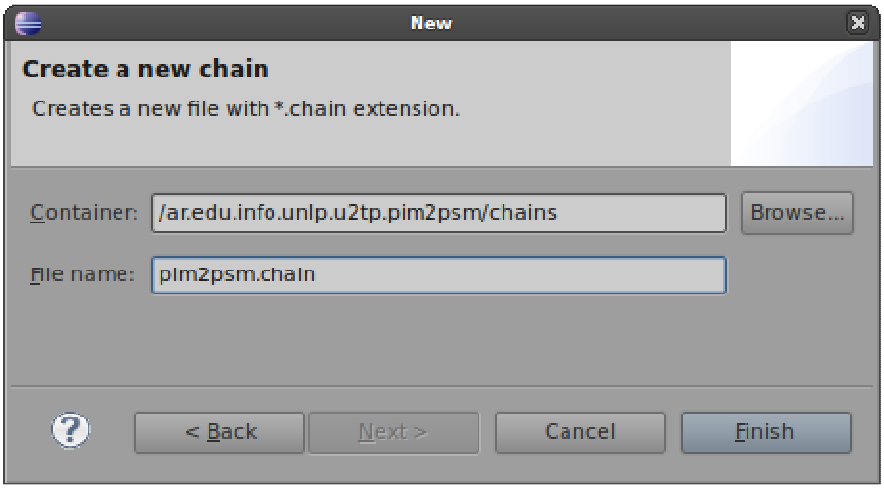

Figura 50 - Nombre de la cadena de Acceleo

- Una vez creada la cadena se abrirá el editor de la misma. Hacer click derecho en "Repository" y seleccionar "New Child/Model".

Figura 51 - Agregamos a la cadena un modelo

- En la vista de propiedades completar la propiedad "Path" con "<%model%>".

Repetir el procedimiento para crear:

- un elemento "Folder" con path <%src%>.
- un elemento "Log" con path "<%error.log%>".
- un elemento "Emf Metamodel" con path "http://www.eclipse.org/uml2/2.1.0/UML".

- Seleccionar la cadena y en la vista de propiedades editar la propiedad "Parameters files" agregando los tres elementos creados anteriormente: <%model%>, <%src%> y <%error.log%>. Luego hacer click en "OK".

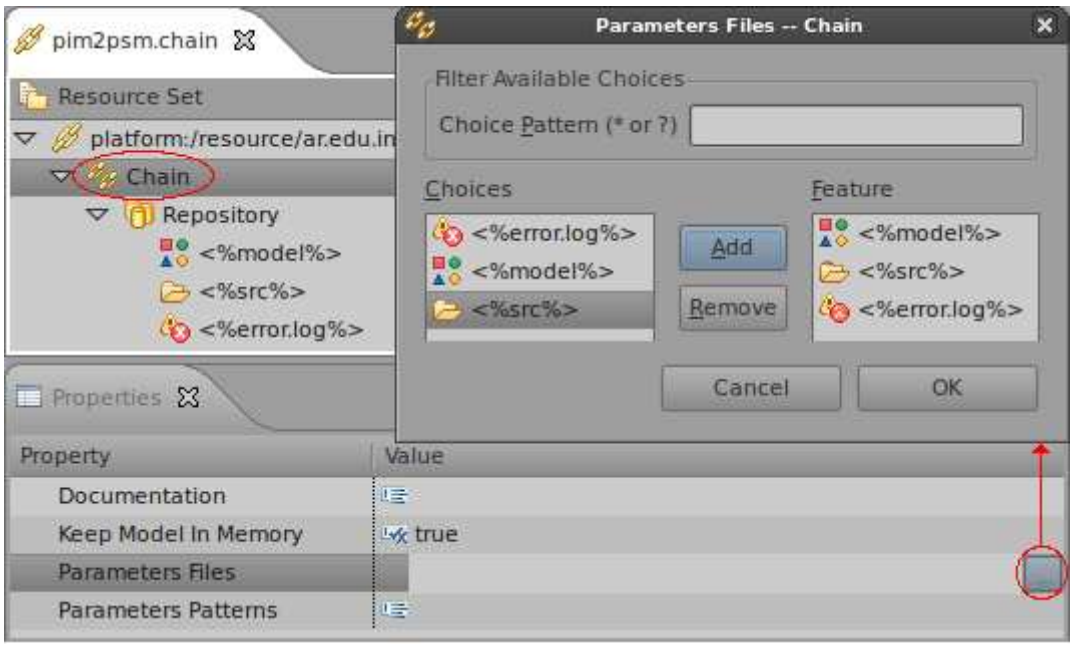

Figura 52 - Parámetros e la cadena Acceleo

Hasta el momento hemos creado una cadena que recibe tres parámetros: un modelo, una carpeta y un archivo de log. A continuación vamos a crear la primera acción de la cadena (la que realiza la validación del modelo PIM). Para esto debemos:

Crear una clase que implemente la interface *fr.obeo.acceleo.chain.tools.IChainCustomAction* como la siguiente:

ValidatePIMCustomAction.java

```
package ar.edu.info.unlp.u2tp.pim2psm.actions;

import org.eclipse.core.runtime.CoreException;

public class ValidatePIMCustomAction implements IChainCustomAction {

    @Override
    public void run(Data[] arg0, CChain arg1, IGenFilter arg2,
            IProgressMonitor arg3, LaunchManager arg4) throws CoreException {
        // TODO validacion
    }

}
```

Figura 53 - Acción de Aceeleo

Dentro del método *run* se debe escribir el código necesario para validar el modelo de entrada, el cual se recibe por el parámetro *arg0*. Recordemos que para la validación se deben usar las clases de EMF.

Dentro del archivo plugin.xml agregar la una extensión a *fr.obeo.acceleo.chain.custom* con la siguiente forma:

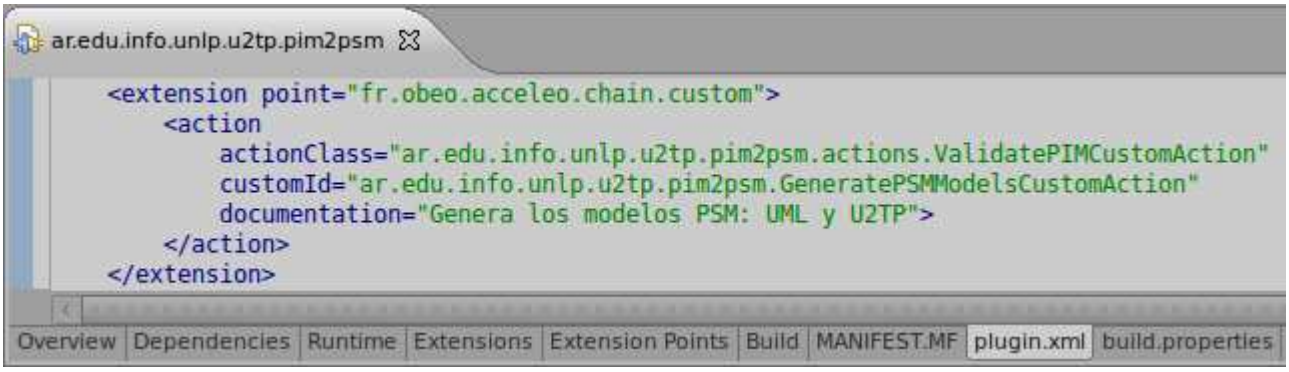

Figura 54 - Acción de validación

En la extensión se debe indicar el nombre de la clase creada en el punto anterior, un identificador y una descripción para la acción.

Por último se debe agregar la acción a la cadena. Para esto, en el editor de la cadena hacer click derecho en "Action Set" y seleccionar "New Child/Custom Action".

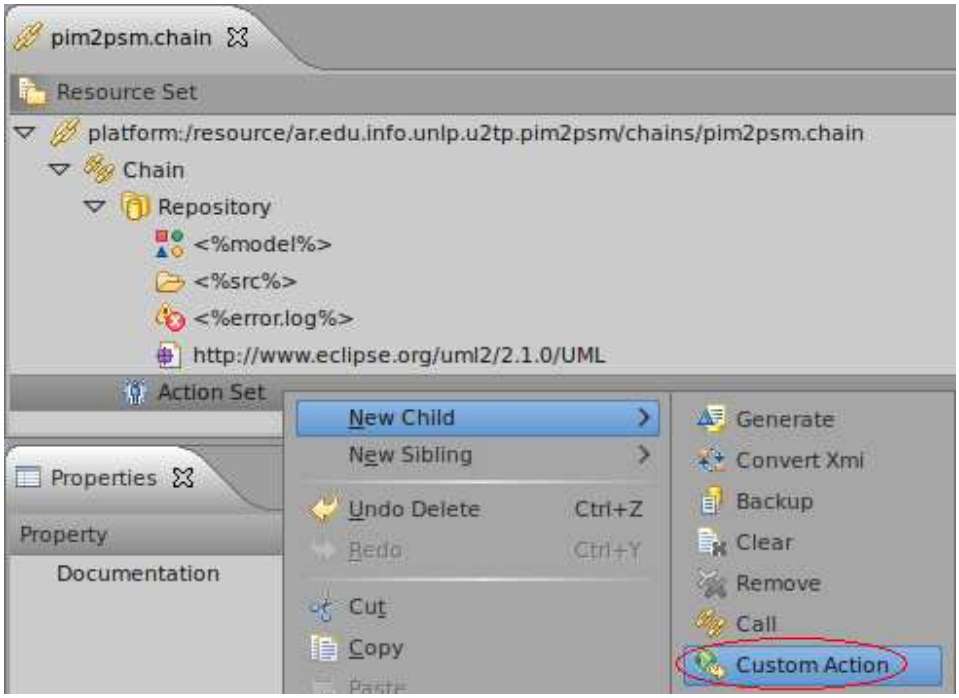

Figura 55 - Adición de la acción a la cadena de Acceleo

Seleccionar la acción creada y completar los siguientes valores en la vista de propiedades:

- *Documentation:* con una descripción de lo que hace la acción
- *ID:* con el identificador de la acción (el que se definió antes en la extensión a *fr.obeo.acceleo.chain.custom*)
- *Log:* con el valor "<%error.log%>"
- *Resources:* con el valor "<%model%>"

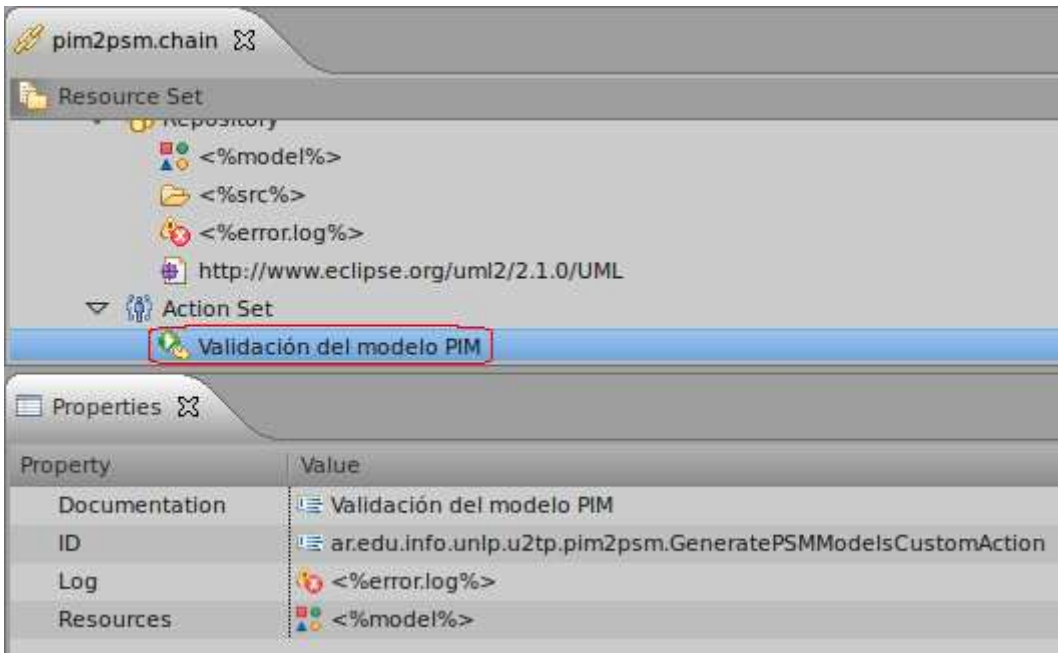

Figura 56 - Acción agregada a la cadena Acceleo

De la misma forma que se creó la acción anterior, se deben crear dos acciones más:

- una acción para invocar a las transformaciones PIM A PSM.
- una acción para validar los modelos intermedios generados por la acción anterior.

Puesto que el procedimiento para crear ambas acciones es muy similar al que se describió anteriormente, no se entrará en detalle al respecto. Sólo vale destacar dos cuestiones. En primer lugar, la acción que invoca a las transformaciones deberá generar dos archivos (uno para cada modelo intermedio). El nombre elegido para estos archivos es importante ya que será necesario referenciarlos más adelante. En nuestra implementación, si el archivo del modelo PSM de entrada tiene el nombre *modelo.uml*, los archivos para los modelos intermedios UML y U2TP se llamarán *modelo.psm.uml* y *modelo.psm.u2tp* respectivamente. La segunda cuestión a destacar es que para la invocación a las transformaciones QVTO se debe utilizar la clase *QvtoTransformationHelper*, a través del método *executeTransformation*.

En este punto hemos finalizado la implementación del plugin pim2psm. Resumiendo, este módulo contiene una implementación del perfil U2TP, dos transformaciones QVTO que generan los modelos intermedios, un conjunto de clases para validación de los modelos, y una cadena Acceleo que integra los conceptos anteriores. A continuación se detallará la implementación de los módulos para la capa independiente del lenguaje de programación (ELP).

6.2 Implementación de un módulo para la capa ELP

Un módulo de la capa específica del lenguaje de programación (ELP) tiene como objetivo generar el código fuente en un lenguaje de programación determinado (como Java, PHP, o C++). El código producido corresponde a las clases del dominio de un modelo UML enriquecido con el perfil U2TP. Para simplificar su tarea, utiliza un modelo intermedio UML generado a partir del modelo entrante. El modelo intermedio sólo contiene información de las clases del dominio del modelo de entrada.

Como hemos visto, la generación de los modelos intermedios está soportada por el módulo *pim2psm* (por medio de transformaciones). Además, para acceder a esta funcionalidad se debe utilizar la cadena Acceleo *pim2psm.chain* contenida en el mismo. De esta forma, un módulo de la capa ELP podrá reutilizar dicha funcionalidad y dedicarse exclusivamente a la generación del código fuente. Esto implica una dependencia entre los módulos ELP y el módulo *pim2psm*. Es decir, un módulo ELP primero deberá invocar

a la cadena *pim2psm.chain* para generar los modelos intermedios y luego deberá procesar uno de esos modelos (el UML) generando el código fuente. Para lograr esto, nuevamente utilizaremos el mecanismo de cadenas Acceleo:

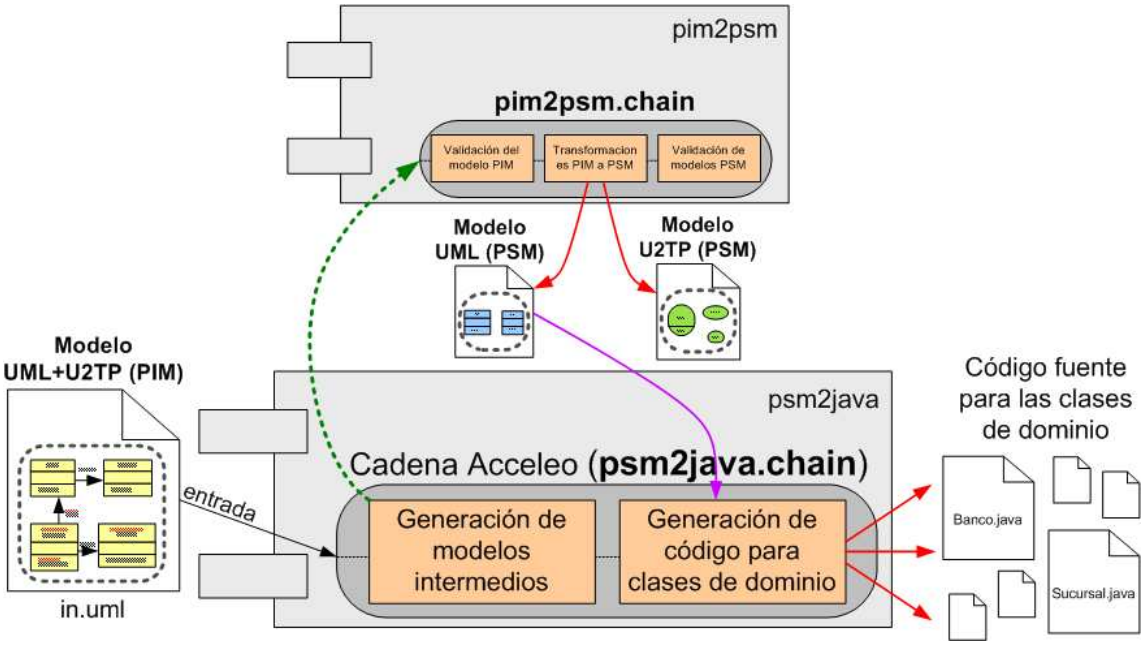

Figura 57 - Cadena de Acceleo para la transformación psm a Java

A modo de ejemplo se ha desarrollado el módulo ELP correspondiente al lenguaje Java, llamado *psm2java*. Como se puede observar, dicho módulo contiene una cadena Acceleo (*psm2java.chain*). Esta cadena recibe como entrada un modelo UML enriquecido con el perfil U2TP. A su vez contiene dos acciones: la primera es la encargada de invocar a la cadena *pim2psm.chain* del módulo *pim2psm*; la segunda es la responsable de la generación de código a partir del modelo intermedio UML.

Como hemos mencionado anteriormente, la generación de código fuente a partir de un modelo UML es un tema ya resuelto por la comunidad MDA. En particular, el Framework Acceleo dispone de un conjunto de módulos, cada uno dedicado a la generación de código en un lenguaje en particular. Estos módulos están disponibles en http://www.acceleo.org/pages/modules-repository. En la implementación de módulos para la capa ELP se hará reuso de estos módulos.

El primer paso en la implementación de un módulo para la capa ELP es la creación del plugin en sí. Esta tarea es análoga a la detallada en la implementación del módulo de la capa ILP, por lo cual no se entrará en detalle al respecto. Sólo se sugiere como nombre del plugin *psm2nombre_lenguaje*. En nuestro ejemplo se utiliza el nombre *psm2java*.

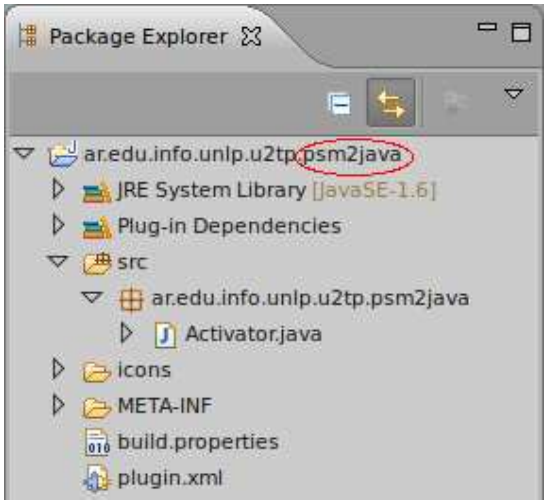

Figura 58 - Estructura del plugin psm2java

Una vez creado el plugin, debemos construir una cadena Acceleo con la siguiente forma:

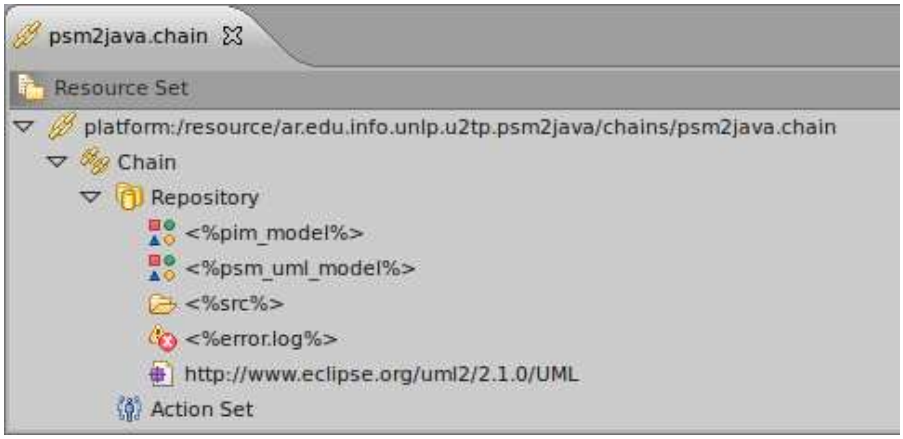

Figura 59 - Cadena de Acceelo para transformar de PSM a Java

Como se puede observar, dentro de Repository se deben crear 5 elementos:

- un elemento "Model" con path "<%pim_model%>"
- un elemento "Model" con path "<%psm_uml_model%>"
- un elemento "Folder" con path "<%src%>"
- un elemento "Log" con path "<%error.log%>"
- un elemento "Emf Metamodel" con path "http://www.eclipse.org/uml2/2.1.0/UML"

El elemento <%pim_model%> representa la ubicación del archivo del modelo PIM entrante. Por otra parte, el elemento <%psm_uml_model%> representa la ubicación del archivo del modelo intermedio UML, el cual será creado durante la ejecución de la cadena. El elemento <%src%> representa el directorio donde se va a ubicar el código generado. El elemento <%error.log%> representa el archivo de log donde se registrarán los errores producidos durante el proceso de generación de código. El último elemento representa el metamodelo del modelo PIM. Todos estos elementos, excepto el metamodelo, son recibidos por parámetros de la cadena:

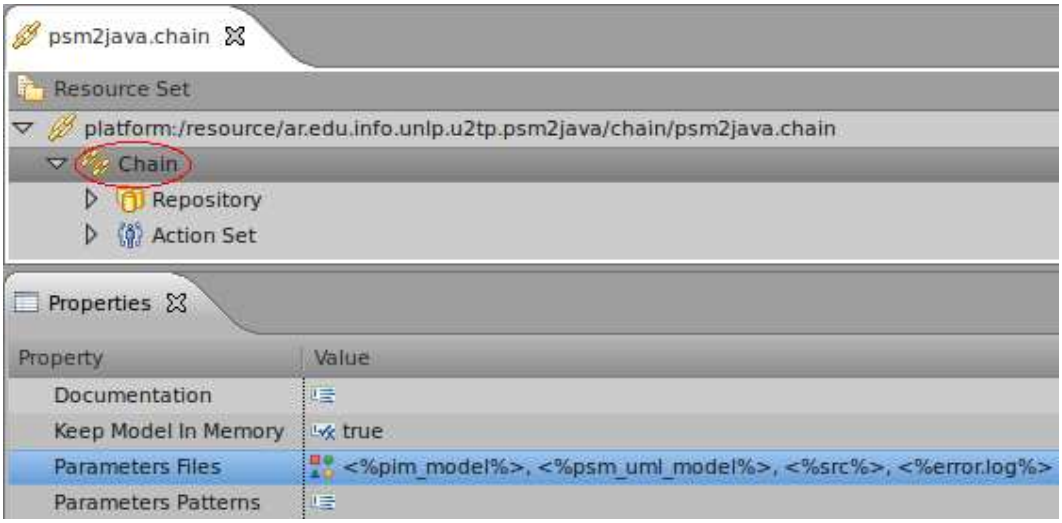

Figura 60 - Parámetros de la cadena PSM a Java

Como se puede ver, la propiedad "Parameters Files" debe ser completada con los elementos <%pim_model%>, <%psm_uml_model%>, <%src%> y <%error.log%> (en ese orden).

A continuación se deben crear dos acciones. La primera debe realizar una invocación a la cadena *pim2psm.chain*, enviando como parámetro la ubicación del archivo del modelo PIM. Para esto:

En el editor de la cadena hacer click derecho en "Action Set" y seleccionar "New Child/Call"

Figura 61 - Llamada a otra cadena de Acceleo

Seleccionar la acción creada en el punto anterior y completar los siguientes valores en la vista de propiedades:

- *Arguments Files:* con los valores "<%pim_model%>", "<%src%>" y <%model%> (en ese orden).
- *Chain path:* con la ubicación de la cadena *pim2psm.chain (ar.edu.info.unlp.u2tp.pim2psm/chain/pim2psm.chain*)
- *Documentation:* con una descripción de lo que realiza la acción
- *Log:* con el valor "<%error.log%>"

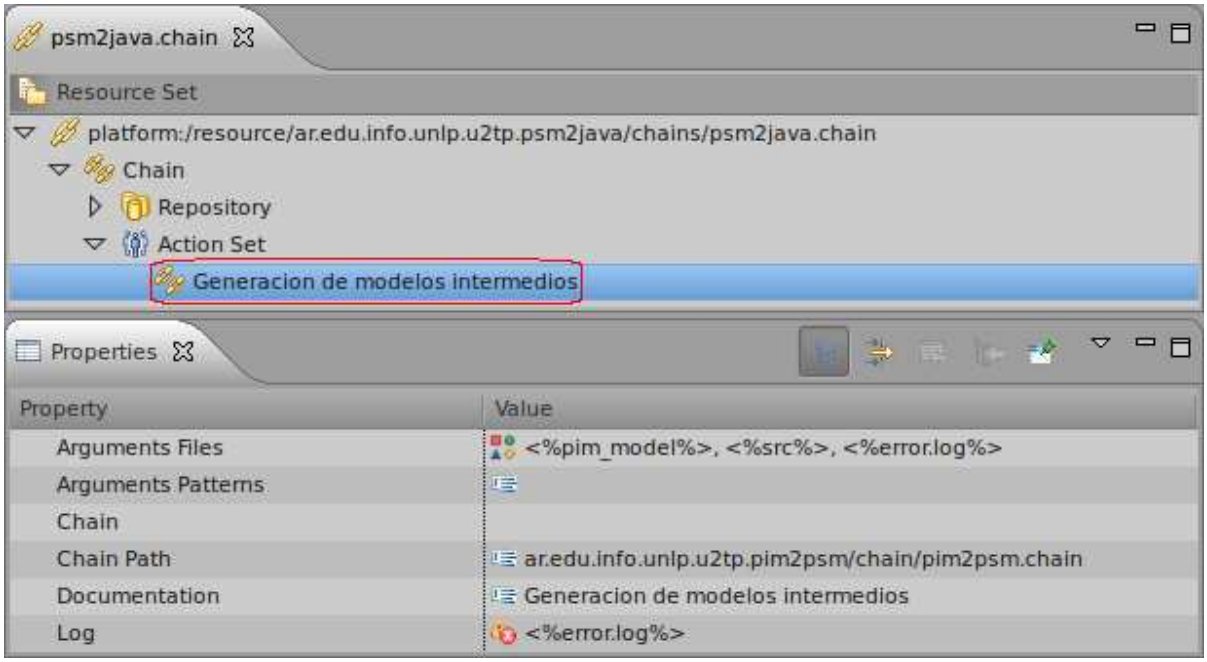

Figura 62 - Estructura de la cadena Acceleo de PSM a Java invocando a otra cadena

La segunda acción de la cadena se debe encargar de la generación de código fuente para las clases del dominio, tomando como entrada el modelo intermedio UML. Para esto se reutilizarán los módulos preexistentes de Acceleo. Cada módulo es capaz de generar código fuente en un lenguaje específico, a partir de un modelo UML. Esta funcionalidad es provista a través de una cadena contenida en dicho módulo.

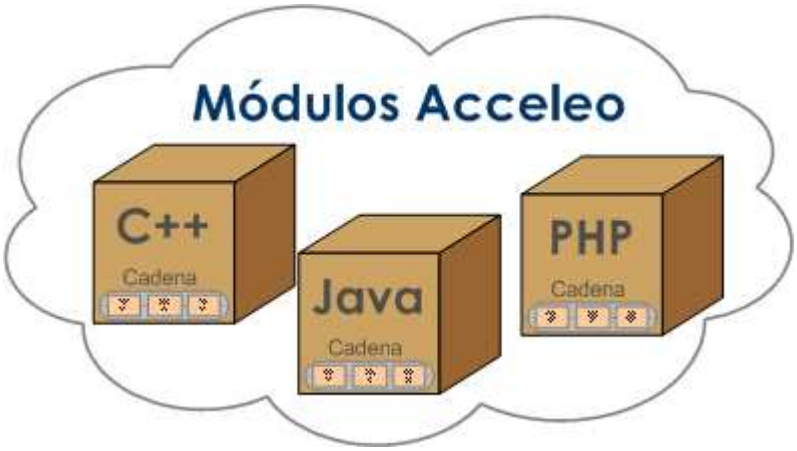

Figura 63 - Módulos de Acceleo para derivar código fuente

Es por esto que la segunda acción de la cadena será similar a la primera, excepto que debe invocar a una cadena contenida en un módulo Acceleo. La acción resultante tiene la siguiente forma:

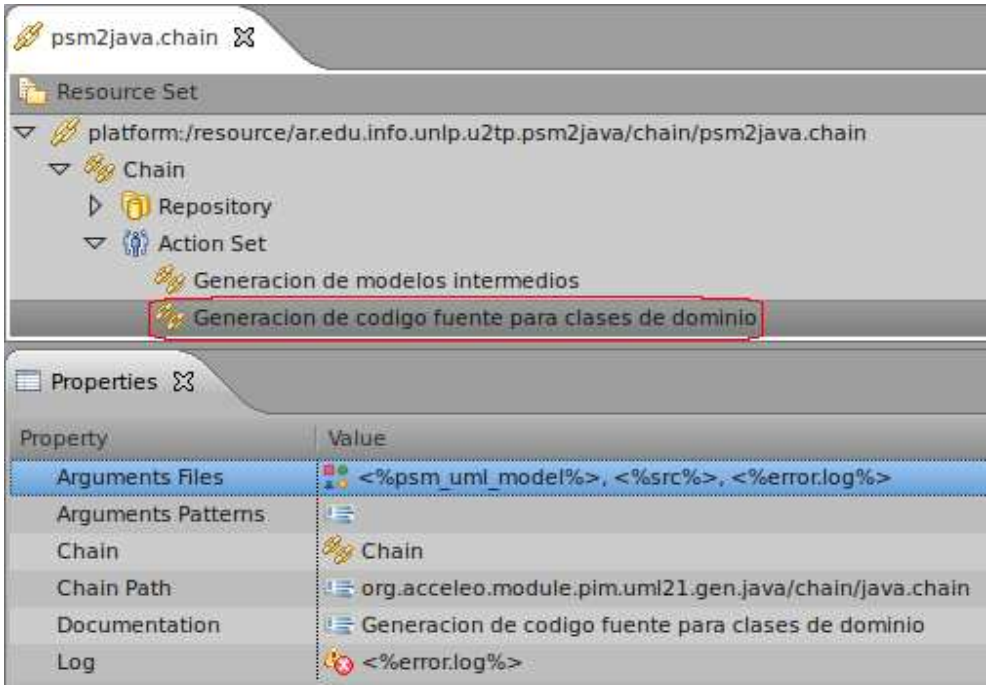

Figura 64 - Llamada a una cadena contenida en el módulo Acceleo

Notar que la diferencia más significativa respecto a la primera acción está en la propiedad "ChainPath". En este caso, dicha propiedad hace referencia a la cadena *java.chain* contenida en un módulo de Acceleo. Dicha cadena es la encargada de generar código fuente Java a partir de un modelo UML.

Aquí hemos finalizado con la construcción de un módulo para a capa ELP. Resumiendo, un módulo ELP es un plugin de Eclipse que posee una cadena Acceleo con dos acciones. La primera invoca a la cadena *pim2psm.chain* del módulo *pim2psm* para generar los modelos intermedios. La segunda invoca a una cadena de un módulo Acceleo para generar el código correspondiente a las clases del dominio. A continuación se detalla la implementación de los módulos de la capa EFT.

6.3 Implementación de un módulo para la capa EFT

La capa específica del Framework de testing está compuesta por un conjunto de plugins. Cada uno de ellos tiene como objetivo generar código fuente para un Framework de testing en particular (como JUnit, TTCN-3, PHPUnit, etc). El código generado corresponde a los casos de test para las clases dominio de un modelo UML. Estos test son especificados aplicando al modelo elementos del perfil U2TP. Para simplificar la derivación de código se utiliza un modelo intermedio U2TP, generado a partir del modelo UML entrante. El modelo intermedio sólo contiene información sobre los casos de test especificados en el modelo de entrada.

Para poder generar el código de los casos de test, un módulo EFT necesita contar con dos cosas. Por una parte, necesita disponer del modelo intermedio U2TP para la derivación de código en sí. Por orto lado, necesita que sea generado el código fuente correspondiente a las clases del dominio, puesto que los casos de test deben hacer referencia a las mismas. Además, tanto el código de las clases del dominio como el de los casos test deben estar escritos en el mismo lenguaje de programación. Como se mostrará a continuación, ambas tareas son delegadas a un módulo de la capa ELP.

A lo largo de este apartado se detallará la implementación del módulo EFT para el Framework de testing JUnit. Este ejemplo podrá ser utilizado para el desarrollo de otros módulos EFT ya que el procedimiento es análogo para todos.

Como hemos mencionado, JUnit es un Framework de testing para el lenguaje de programación Java. Su diseño está pensado para capturar eficientemente las expectativas de un desarrollador acerca de su código para luego verificar rápidamente que el código cumpla con esas intenciones. El hecho de que JUnit esté escrito en Java implica que el código a verificar también esté escrito en Java (recordemos que los casos de test, en su definición, hacen referencia a las clases del dominio):

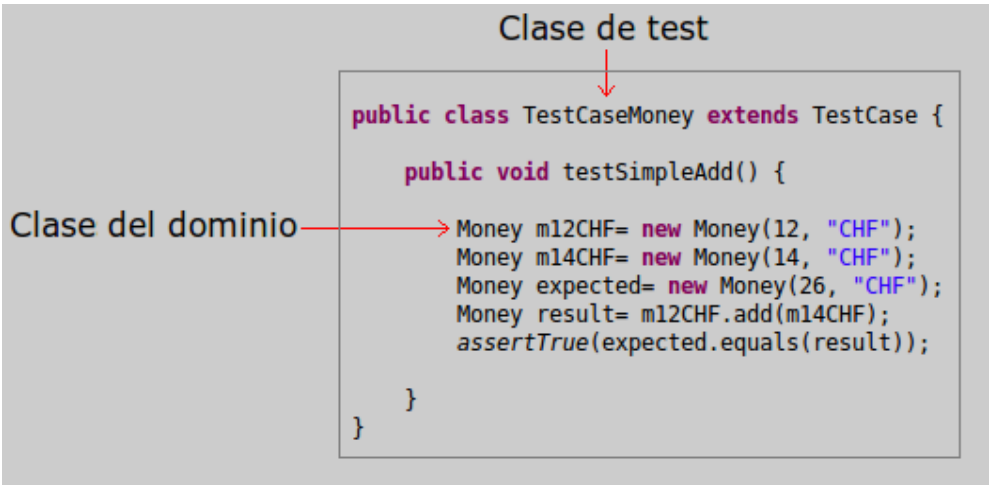

Figura 65 - Ejemplo de JUnit

Puesto que la generación de código fuente en un lenguaje de programación específico, en este caso Java, está soportada por un módulo de la capa EFT, se delegará dicha tarea al mismo. Por ejemplo, el módulo JUnit que desarrollaremos a continuación deberá invocar al módulo Java (*psm2java*) de la capa ELP. Para lograr esto, nuevamente apelamos al concepto de cadenas de Acceleo:

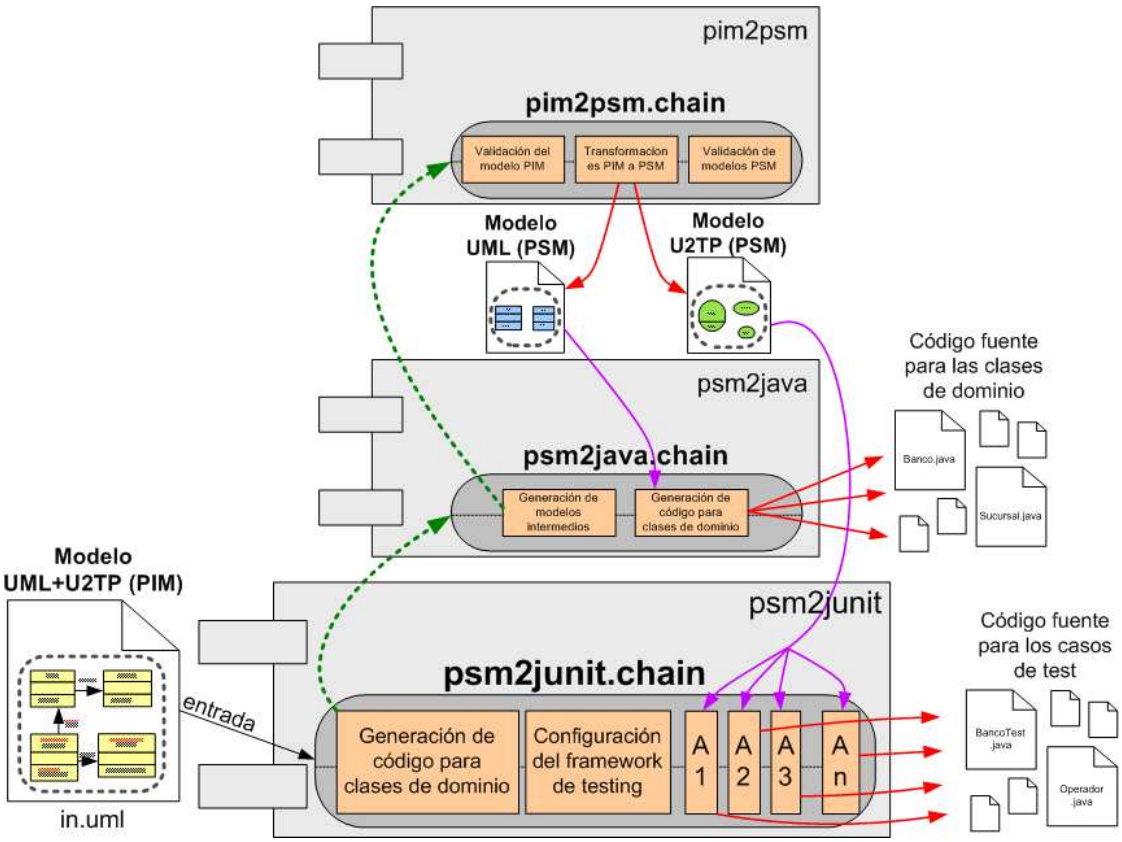

Figura 66 - Modelo general de la solución

Como se puede observar, el módulo *psm2junit* contiene una cadena Acceleo llamada *psm2junit.chain*. Esta cadena recibe como entrada un modelo UML enriquecido con elementos de perfil U2TP (modelo PIM) y realiza su procesamiento en tres pasos:

- Primero genera el código fuente para las clases de dominio. Para lograr esto invoca a la cadena *psm2java.chain* contenida en el módulo *psm2java*. La invocación a esta cadena produce dos resultados: por un lado se generan los modelos intermedios UML y U2TP (modelos PSM), y por otro, se genera el código correspondiente a las clases del dominio.
- Luego lleva a cabo la configuración del Framework de testing. Es decir, generan aquellos archivos de configuración necesarios para el funcionamiento del Framework de testing, a si como también la inclusión de las librerías del mismo modelo.
- Por último genera el código fuente para los casos de test. Esta tarea es lograda mediante la ejecución de un conjunto de acciones (en el diagrama anterior son llamadas A^1, A^2, A^3, ., A^n). Cada una de estas acciones toma como entrada el modelo intermedio U2TP y genera el código fuente para un tipo determinado de elementos. Por ejemplo, la primera acción genera el código para los elementos *TestContext* del modelo entrante, la segunda genera el código para los elementos *TestControl*, y así siguiendo.

Una vez más, el primer paso en la implementación de un módulo es la creación del plugin en sí. Para los módulos de la capa EFT se sugiere el nombre *psm2nombre_fmk_testing*. En nuestro caso, el plugin se llama *psm2junit*:

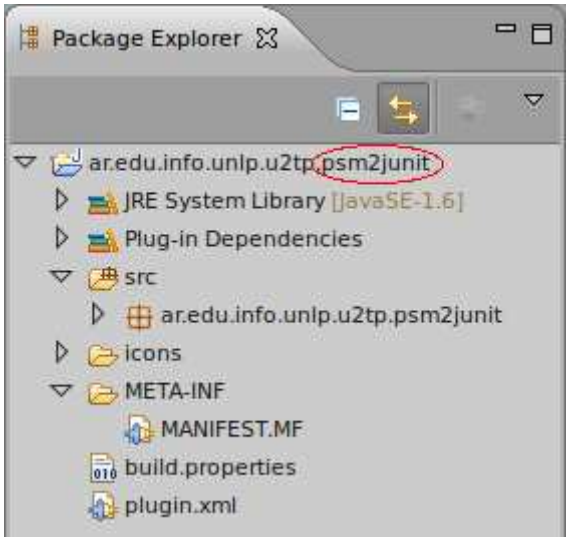

Figura 67 - Estructura del plugin PSM a JUnit

Una vez creado el plugin, debemos agregar una dependencia al plugin ELP, en nuestro caso *psm2java.* Para esto debemos abrir el archivo *plugin.xml,* en la parte inferior seleccionar la solapa *Dependencies* y hacer click en *Add...*:

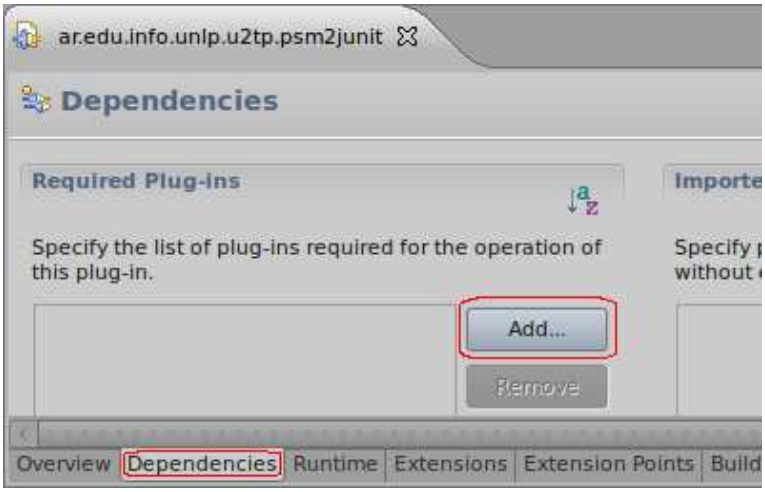

Figura 68 - Agregar dependencia al plugin psm2java

- Luego selecionar el plugin *ar.edu.info.unlp.u2tp.psm2java* y hacer click en "OK".
- A continuación debemos crear una cadena Acceleo con la siguiente forma:

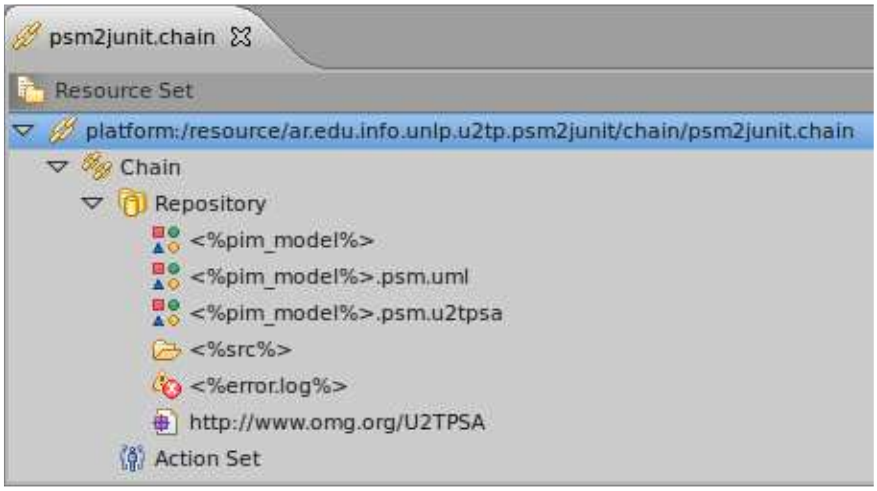

Figura 69- Cadena de Acceleo del plugin PSM a JUnit

Como se puede ver, dentro de Repository se deben crear 6 elementos:

- un elemento *Model* con path *<%pim_model%>*
- un elemento *Model* con path *<%pim_model%>.psm.uml*
- un elemento *Model* con path *<%pim_model%>.psm.u2tp*
- un elemento *Folder* con path *<%src%>*
- un elemento *Log* con path *<%error.log%>*
- un elemento *Emf Metamodel con path http://www.omg.org/U2TPSA*

El elemento *<%pim_model%>* representa la ubicación del archivo del modelo PIM entrante. Los dos elementos siguientes representan los archivos de los modelos intermedios UML y U2TP respectivamente. El elemento *<%src%>* representa el directorio donde se va a ubicar el código generado. Por su parte, el elemento *<%error.log%>* representa el archivo de log donde se registrarán los errores producidos durante el proceso de generación de código. El último elemento representa el metamodelo U2TP, el cual será utilizado para generación de código de los casos de test. Todos estos elementos, excepto el metamodelo, son recibidos como parámetros de la cadena:

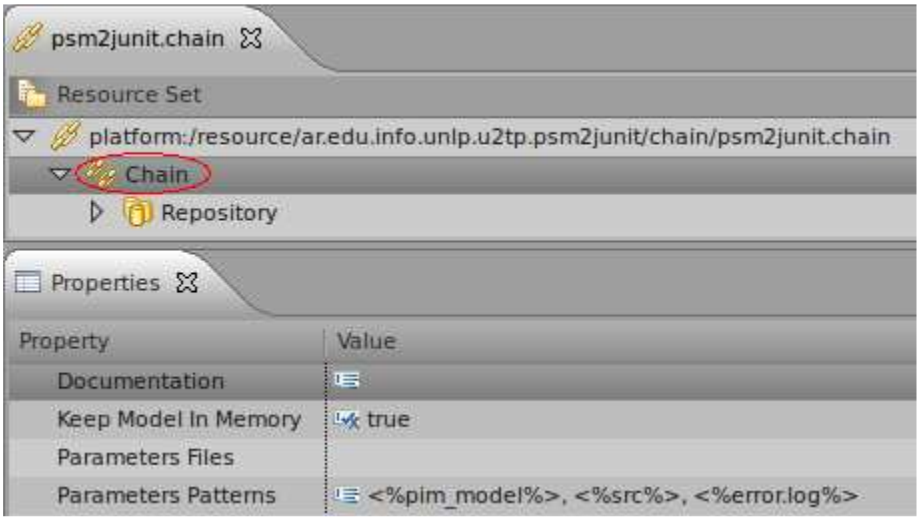

Figura 70 - Parámetros de la cedana Acceleo del plugin PSM a JUnit

Notar que, a diferencia de las cadenas creadas anteriormente, aquí los parámetros son especificados mediante la propiedad *Parameters Patterns*. Esto significa que, en el momento que se invoca la cadena, todos los patrones contenidos en los *paths* de los elementos del Repository serán reemplazados por los argumentos especificados en la invocación. En este caso, la propiedad *Parameters Patterns* debe ser completada con los valores *<%pim_model%>*, *<%src%>* y <%error.log%> (en ese orden).

A continuación debemos crear las acciones para la cadena. Como hemos mencionado, la primer acción deberá invocar al módulo *pim2java* para generar los modelos intermedios y el código fuente de las clases del dominio. Para esto:

- En el editor de la cadena hacer click derecho en "Action Set" y seleccionar "New Child/Call"

Figura 71 - Llamada a la cadena del plugin PIM a Java

- Seleccionar la acción creada en el punto anterior y completar los siguientes valores en la vista de propiedades:
  - *Arguments Files:* con los valores "<%pim_model%>", "<%pim_model%>.psm.uml", "<%src%>" y "<%model%>" (en ese orden).
  - *Chain path:* con la ubicación de la cadena *psm2java.chain (ar.edu.info.unlp.u2tp.psm2java/chain/psm2java.chain*)
  - *Documentation:* con una descripción de lo que realiza la acción
  - *Log:* con el valor "<%error.log%>"

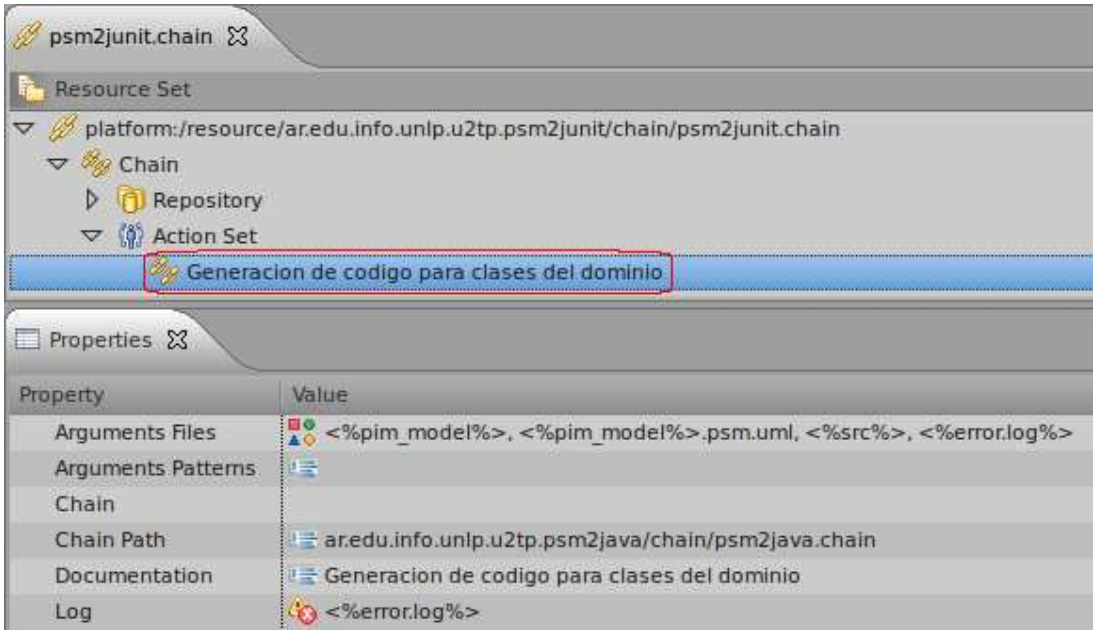

Figura 72 - Propiedades de la cadena del plugin PSM a JUnit

La segunda acción de la cadena *pim2junit.chain* será la encargada de la configuración del Framework de testing. Como mínimo deberá incluir las librerías del Framework para que el código de los casos de test pueda funcionar. En nuestro caso, debemos incluir la librería JUnit (empaquetada en un archivo JAR) al *classpath* del proyecto Java destino. Para esto debemos:

- Crear una clase que implemente la interface *fr.obeo.acceleo.chain.tools.IChainCustomAction* como la siguiente:

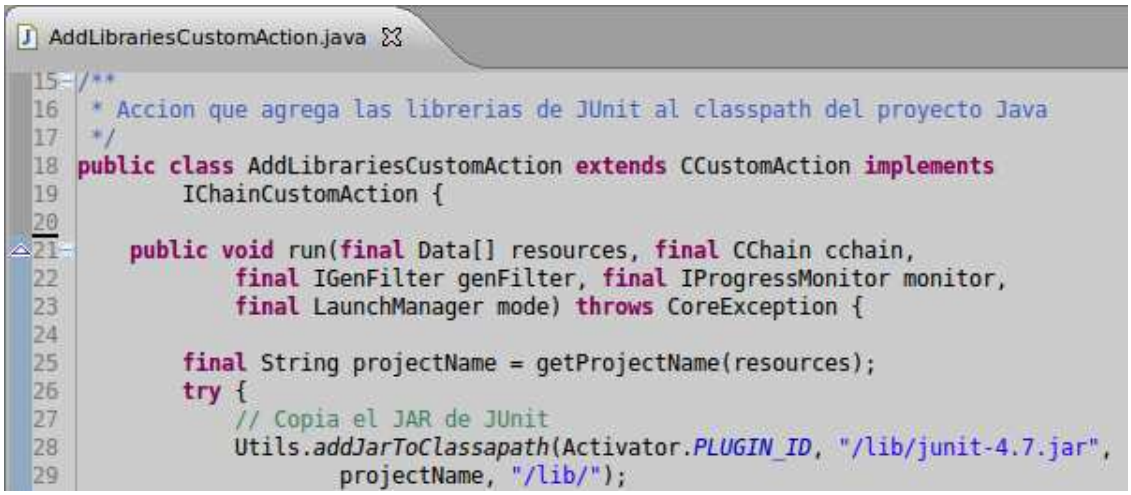

Figura 73 - Acción de Acceleo que agrega las librerias de JUnit

Dentro del método *run* se debe escribir el código necesario para la configuración del Framework de testing. En el caso anterior se agrega la librería de JUnit al *classpath* del proyecto Java destino. Como veremos más adelante, la acción recibe por parámetro el elemento *<%src%>*, el cual representa la ubicación

donde se va a ubicar el código fuente generado. A partir de dicho parámetro se calcula la ubicación del proyecto destino.

- Luego, dentro del archivo plugin.xml se debe agregar una extensión a *fr.obeo.acceleo.chain.custom* con la siguiente forma:

ar.edu.info.unlp.u2tp.psm2junit

```
<extension
      point="fr.obeo.acceleo.chain.custom">
   <action
      actionClass="ar.edu.info.unlp.u2tp.psm2junit.actions.AddLibrariesCustomAction"
      customId="ar.edu.info.unlp.u2tp.psm2junit.actions.AddLibrariesCustomAction"
      documentation="Configuracion del framework JUnit">
   </action>
</extension>
```

Figura 74 - Registro de la acción con un punto de extensión

En la extensión se debe indicar el nombre de la clase creada en el punto anterior, un identificador y una descripción para la acción.

- Por último se debe agregar la acción a la cadena *psm2junit.chain*. Para esto, en el editor de la cadena hacer click derecho en "Action Set" y seleccionar "New Child/Custom Action". A continuación seleccionar la acción creada y completar los siguientes valores en la vista de propiedades:
  - *Documentation:* con una descripción de lo que hace la acción
  - *ID:* con el identificador de la acción (el que se definió en la extensión a *fr.obeo.acceleo.chain.custom*)
  - *Resources:* con los valores *"<%src%>"* y *"<%error.log%>"* (en ese orden)

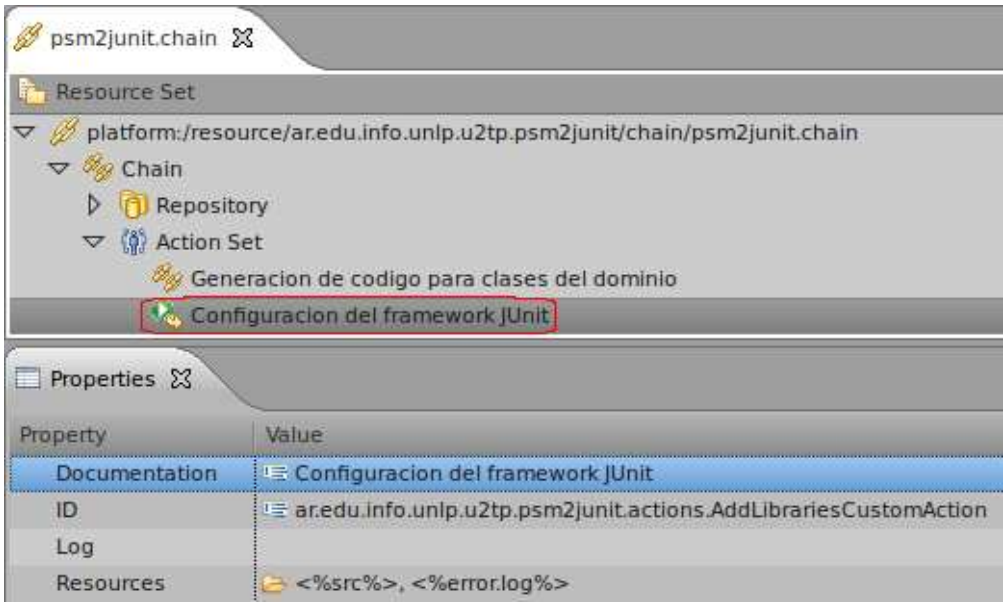

Figura 75 - Propiedades de la acción de Acceleo del plugin PSM a JUnit

Hasta el momento hemos completado la cadena *pim2junit.chain* con dos acciones. Una para generar los modelos intermedios y el código de las clases del dominio, y otra para realizar la configuración del Framework de testing. A continuación vamos a crear las acciones para la generación del código fuente de los casos de test.

Como hemos mencionado, la generación del código para los casos de test es lograda mediante la utilización de plantillas de Acceleo. Este mecanismo permite definir un esquema de correspondencia entre los elementos de un modelo y el código fuente que debe ser generado por cada uno de ellos. En nuestro caso, debemos definir la relación entre cada elemento del modelo intermedio U2TP y el código JUnit correspondiente. Cada relación se especifica con una plantilla como la siguiente:

datapartition.mt

```
<%
metamodel /ar.edu.info.unlp.u2tp.pim2psm/models/output/u2TPSA.ecore

import ar.edu.info.unlp.u2tp.psm2junit.gen.services.Common
%>

<%script type="DataPartition" name="genDataPartition"
         file="<%getDefinableFullFilePath()%>" %>
<% genPackageDeclaration %>
<% genImports %>
<% genComment %>
public class <% name %> {
    <% genFields %>

    <% genConstructor %>

    <% genAccesors %>

    <% genInitialize %>

    <% genMethodsUserCode %>
}
```

Figura 76 - Plantilla de Acceleo Para la generación de código

Una plantilla Acceleo es un archivo, con extensión .mt, que contiene una serie de directivas que determinan la correspondencia entre un elemento de un modelo y el código fuente que debe ser generado. En el ejemplo anterior, la plantilla *datapartition.mt* define la correspondencia entre un elemento *DataPartition* de un modelo U2TP y la clase Java que será generada. Sin dudas, un plantilla Acceleo constituye una forma de transformación M2T (modelo a texto).

Antes de proceder con la creación de plantillas debemos configurar el plugin EFT, en nuestro caso el plugin *psm2junit*, para convertirlo en proyecto "generador". Para esto, se debe hacer click derecho sobre el plugin y seleccionar "Acceleo/Convert Project to Generator Project":

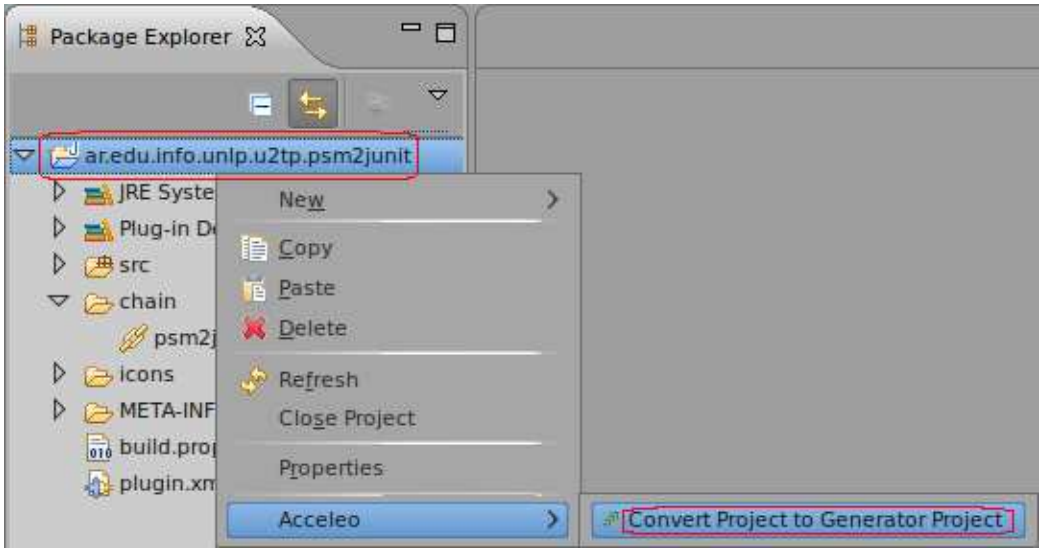

Figura 77 - Conversión del proyecto en un proyecto Acceleo

Ahora sí, para crear una plantilla Acceleo se debe:

- Abrir el diálogo *"File/New/Other..."*.
- Expandir "Acceleo" y seleccionar *"Empty Generator"*. Luego hacer click en el botón "Next".

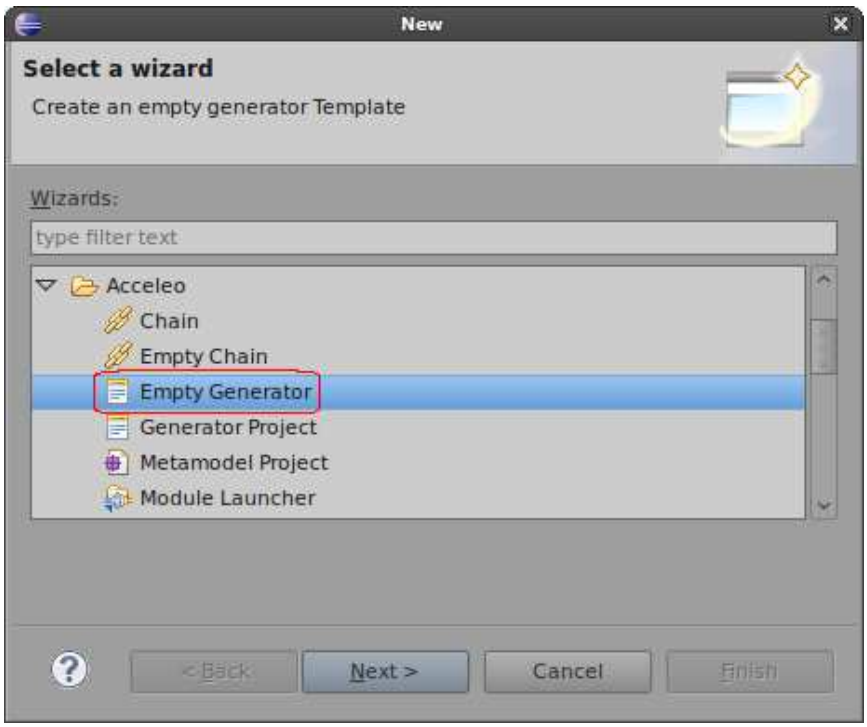

Figura 78 - Selección de una plantilla vacía de Acceleo

- En la siguiente página se deben completar los campos:
  - Metamodel URI: con la URI del metamodelo U2TP.

- Type values: con el tipo de elementos que procesará la plantilla. En este caso se creará una plantilla para los elementos *TestContext*.

- Luego hacer click en "Next".

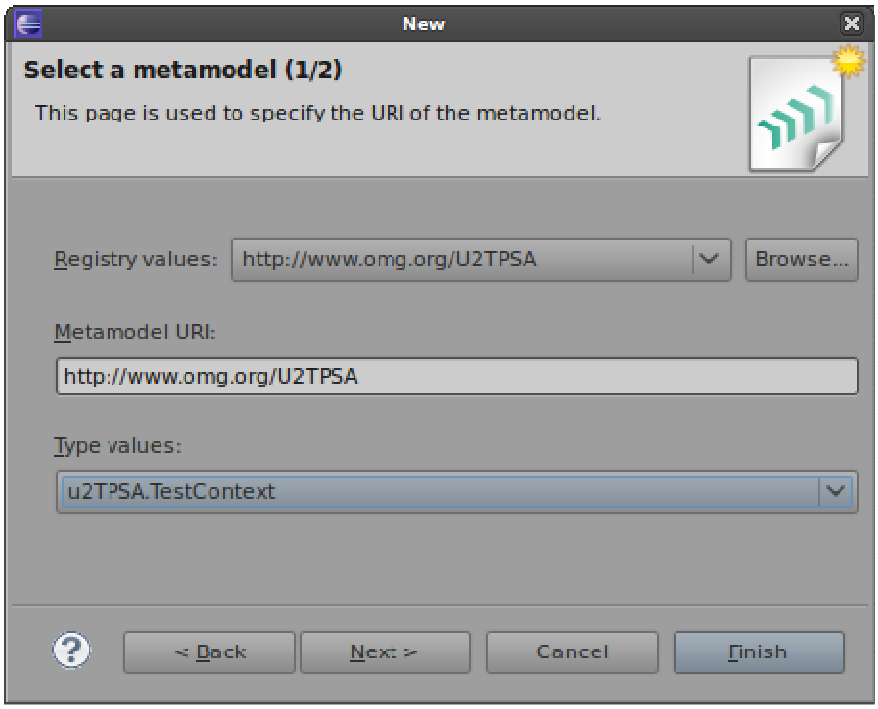

Figura 79 - Propiedades de la plantilla de Acceleo

- En la última página se deben completar los campos:
  - Container: con el directorio donde se va a crear la plantilla.
  - File name: con un nombre para el archivo de la plantilla (debe tener extensión *.mt*).
- Luego hacer click en "Finish".

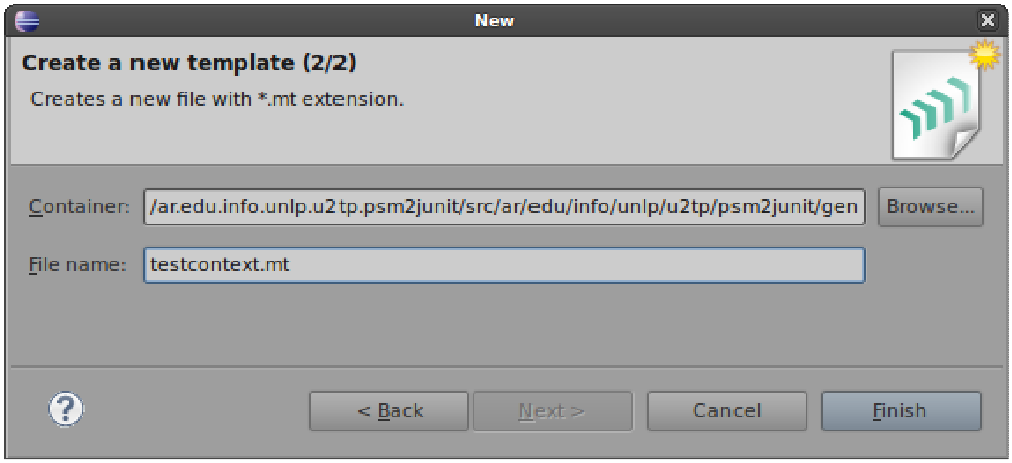

Figura 80 - Nombre del archivo de la plantilla de Acceleo

A continuación es creado el archivo de la plantilla y su editor es abierto en la ventana principal:

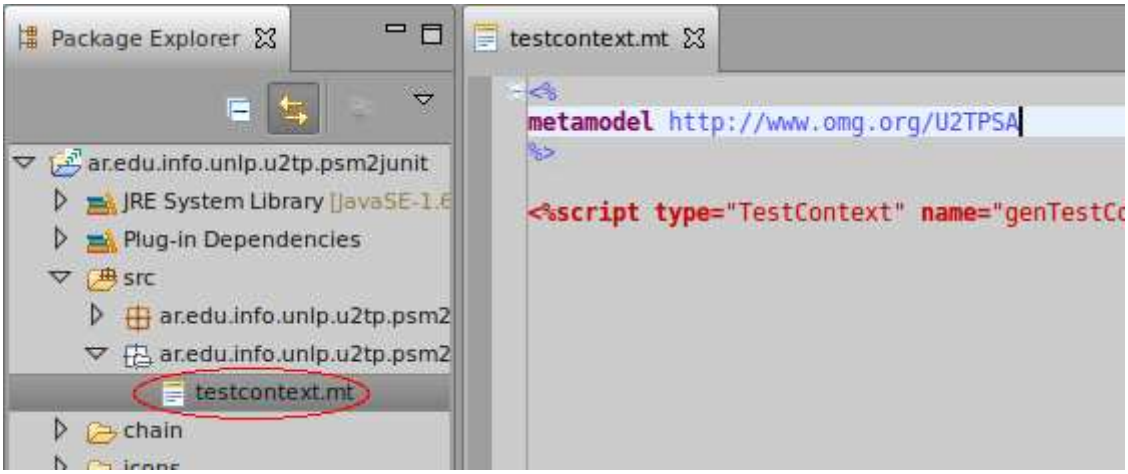

Figura 81 - Editor de plantillas de Acceleo

Como se puede observar, en el encabezado de una plantilla se declara la URI de un metamodelo, utilizando la sentencia *metamodel***.** Esto permite determinar sobre qué modelos puede ser aplicada la plantilla. En nuestro caso, todas las plantillas van a ser aplicadas sobre modelos U2TP. Luego del encabezado se declara una serie de scripts, los cuales especifican de qué forma debe ser generado el código fuente.

Cada script se declara con el tag *<%script ...%>*. Su contenido es el texto que comienza después del tag script y finaliza cuando comienza otro script o cuando termina el archivo de la plantilla. Para cada script se debe definir el tipo de elementos del metamodelo sobre el cual se aplicará el script y un nombre, mediante los atributos *type* y *name* respectivamente:

```
<%script type="TestContext" name="genTestContext" file="<% getFullFilePath() %>"%>
```

Figura 82 - Script para transformar elementos del metamodelo

El parámetro *file* es opcional. Con este parámetro se indica la ubicación del archivo que será generado a partir de la ejecución del script. En el ejemplo anterior, el valor del atributo file es el resultado de la evaluación de otro script (*getFullFilePath*). En caso de que dicho parámetro no esté definido, o su valor sea *null*, no se generará ningún archivo.

Luego, se debe completar el contenido del script para determinar el texto que será generado. Para esto se puede usar tanto texto estático como variables. Estas útlimas se especifican entre <% %>, como por ejemplo:

```
<%script type="TestContext" name="genTestContext" file="<% getFullFilePath() %>"%>

public class <% name %> extends AbstractTestContext {


}
```

Figura 83 - Template Acceleo 1

En el ejemplo anterior muestra la utilización de la variable *<%name%>*. Durante la ejecución del script la declaración de una variable es reemplazada por su valor. Las variables que se pueden utilizar dentro

de un script están determinadas por el tipo del elemento al que se aplica dicho script (declarado con el atributo type). La utilización de variables es una de las formas para generación de contenido dinámico. Otra forma es la navegación entre scripts, como por ejemplo:

```
<%script type="TestContext" name="genTestContext" file="<% getFullFilePath() %>"%>
<% genComment %>
public class <% name %> extends AbstractTestContext {


}

<%script type="TestContext" name="genComment" post="trim" %>
/**
 * TestContext implementation for <% name %>
 */
```

Figura 84 - Template Acceleo 2

En el ejemplo anterior el script *genTestContext* invoca al script *genComment* para generar el comentario de la clase. Es importante destacar que la navegación entre scripts está restringida por los tipos de elementos a los que se aplican los mismos. Es decir, desde un script que se aplica a un tipo de elementos X se puede navegar hacia cualquier script que tenga como tipo X o una superclase de X.

Además, dentro de un script se pueden utilizar las sentencias de control de flujo if y for, como por ejemplo:

```
<%script type="TestContext" name="genDataPoolAccesors" %>
<%for (dataPool){%>
/**
 * @return the DataPool <%name%>
 */
public <% definition.getClassName() %> get<% name %>() {
    return <% name %>;
}

<%}%>
```

Figura 85 - Condicionales en el template de Acceleo

En el ejemplo anterior se muestra el uso de la sentencia for. El script se aplica a elementos *TestContext* e itera sobre los valores *DataPool*, accesibles a través de la variable *dataPool*, generando un método Java por cada uno de ellos. Notar que las variables declaradas en el interior de un bloque *for* son relativas a los elementos que se están iterando. La utilización de la sentencia *if* es análoga al *for*, por eso omitimos su explicación.

Hay casos en los que es necesario que ciertos bloques de código sean completados por el usuario. Esto se debe a que no siempre es posible generar el código automáticamente. En estos casos los cambios deben ser preservados a través de las sucesivas generaciones de código. Para esto se debe utilizar un tag especial de Acceleo:

```
<%script type="DataPartition" name="genInitialize"%>
/**
 * Initializes this DataPartition
 */
private void initialize() {
    //<%startUserCode%> to initialize this data partition

    //<%endUserCode%>
}
```

Figura 86 - Espacios reservados en la plantilla para código no automatizado

El código modificado por el usuario entre los tags **<%startUserCode%>** y **<%endUserCode%>** se protege para no ser sobrescrito durante la siguiente generación de código. De esta forma se permite la especialización del código generado.

Una vez creada la plantilla, con los scripts correspondientes, debemos agregar una acción a la cadena *psm2junit.chain*. Dicha acción va a ser la encargada de invocar a la plantilla con los elementos del modelo intermedio U2TP. Para esto:

- En el editor de la cadena hacer click derecho en "Action Set" y seleccionar "New Child/Generate".

Figura 87 - Incorporación de la plantilla a la cadena Acceleo

- Seleccionar la acción creada en el punto anterior y completar los siguientes valores en la vista de propiedades:
    - *Documentation:* con una descripción de lo que realiza la acción
    - *Folder*: con el valor "<%src%>"
    - Generator: con la ubicación de la plantilla creada anteriormente
    - *Log:* con el valor "<%error.log%>"
    - *Metamodel:* con el valor "http://www.omg.org/U2TPSA"
    - *Model:* con el valor "<%pim_model%>.psm.u2tpsa"

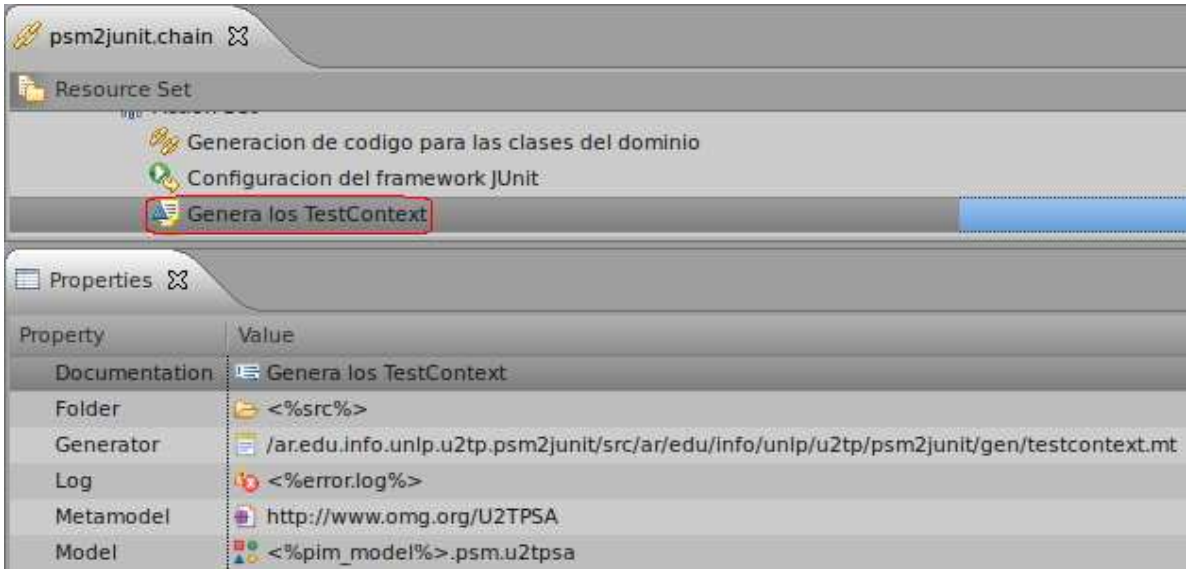

Figura 88 - Propiedades de la plantilla en la cadena Acceleo

El proceso de creación de plantillas debe repetirse por cada tipo de elementos del metamodelo U2TP. Vale destacar que no todos los tipos van a producir como resultado un archivo de código fuente. Por ejemplo, en el metamodelo U2TP el tipo de elementos *TestCase* no genera un nuevo archivo .java, sino que genera un método dentro de una clase de tipo *TestContext*. Obviamente, esto depende del tipo de elementos, del lenguaje de programación y del Framework de testing correspondientes al módulo EFT que se esté implementando.

De esta forma, mediante la creación de plantillas Acceleo con sus respectivos scripts, se logra la generación de código fuente para los casos de test. Es importante tener en cuenta que los Frameworks de testing existentes no siempre implementan todos los conceptos de U2TP. Por ejemplo, en JUnit no existen los conceptos *TestContext, Arbiter*, *Scheduler*, etc. Por este motivo, en ciertos casos es necesario implementar una adaptación al Framework de testing para dar soporte a todos los conceptos de U2TP. En nuestra implementación del módulo EFT para el Framework JUnit fue necesario una adaptación del mismo puesto que, como hemos dicho, no soporta la mayoría de los conceptos de U2TP. Sin entrar en detalles, esta adaptación se materializa en una librería de Java (un archivo JAR) que contiene un conjunto de clases. Estas clases extienden a las clases de JUnit para dar soporte a los elementos de U2TP. Por ejemplo, se implementaron las clases *AbstractTestContext*, *AbstractArbiter*, *AbstractScheduler*, etc.

El último paso en la implementación de un módulo para la capa EFT es la creación de un módulo Acceleo. Como hemos mencionado, los módulos de Acceleo constituyen un mecanismo que permite a los usuarios generar código fuente para distintas plataformas tecnológicas a partir de un mismo modelo. En nuestro caso utilizaremos un módulo Acceleo por cada Framework de testing. Para esto debemos modificar el archivo plugin.xml del módulo EFT agregando una extensión como la siguiente:

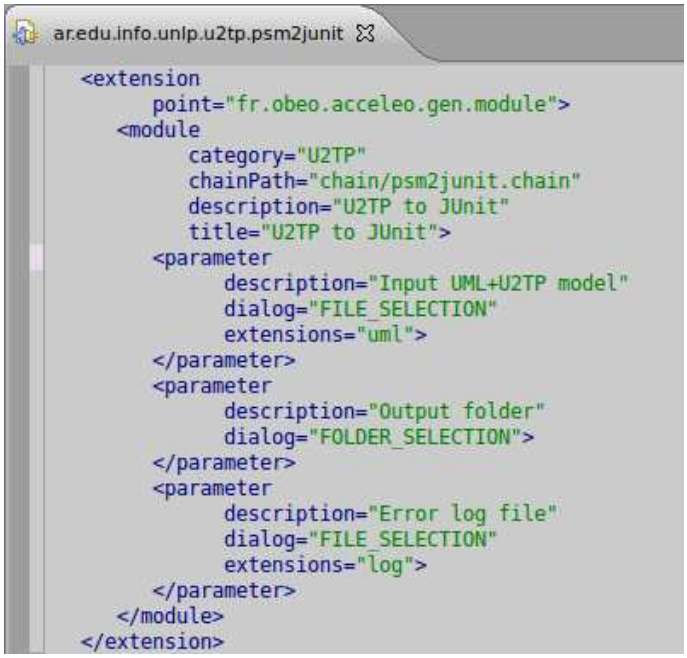

Figura 89 - Extensión para agregar el módulo PSM a JUnit

Como se puede observar la extensión se hace al punto de extensión "*fr.obeo.acceleo.gen.module*". En la misma se declara un elemento *module* cuyos atributos deben completarse de la siguiente forma:

- *category*: con la categoría donde se va a ubicar el módulo.
- *chainPath*: con la ubicación de la cadena del módulo EFT, en este caso la cadena *psm2junit.chain.*
- *description*: con una descripción del módulo.
- *title*: con el nombre del módulo.

Además, dentro del módulo se deben especificar 3 parámetros. Como se verá, sus valores van a ser completados por el usuario de la herramienta y respectivamente se corresponden con:

- el archivo PIM de entrada.
- el directorio donde se va a generar el código fuente.
- el archivo de log donde se van a registrar los errores producidos durante la generación.

La creación de un módulo Acceleo da como resultado una nueva entrada en el dialogo "File/New/Other..." dentro de la categoría "Acceleo/Module Launcher":

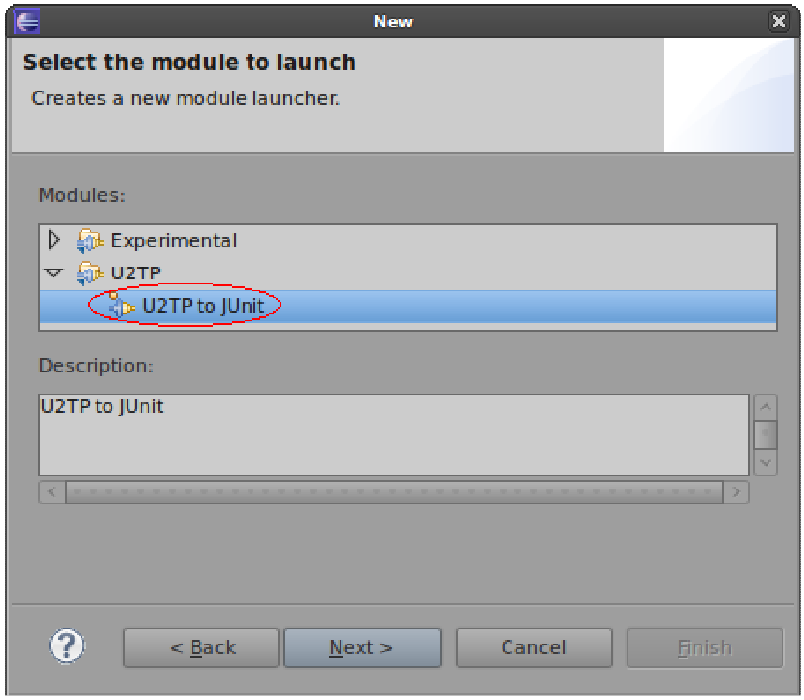

Figura 90 - Módulo PSM a JUnit incorporado a Eclipse

Seleccionando el módulo y avanzando en las páginas del wizard podremos ver cómo son solicitados los parámetros definidos para el mismo:

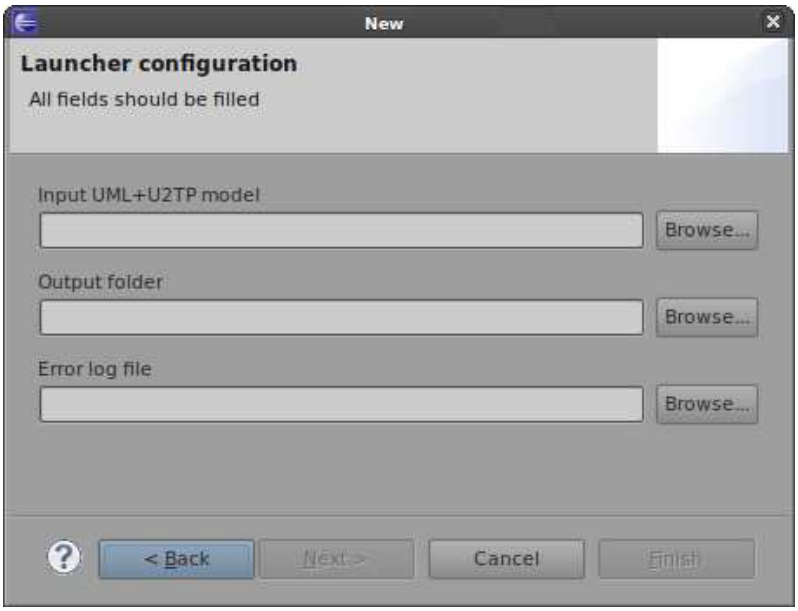

Figura 91 - Pantalla para generar código a partir de un modelo de entrada

Con la construcción del módulo Acceleo finaliza la construcción del módulo EFT. En resumen, para crear un módulo de la capa específica del Framework de testing se debe:

- crear un plugin de Eclipse.

- agregarle la dependencia un módulo ELP.
- dentro del plugin, crear una cadena Acceleo.
- agregar a la cadena una acción que invoque al módulo ELP para generar los modelos intermedios y el código fuente de las clases del dominio.
- agregar a la cadena una acción para configuración del Framework de testing
- convertir el plugin a proyecto "generador".
- crear las plantillas Acceleo para generar el código de los casos de test, con sus respectivas acciones dentro de la cadena.
- modificar el archivo plugin.xml agregando el módulo Acceleo correspondiente al Framework de testing.

7 Caso de estudio

En este capítulo se mostrará el funcionamiento de la herramienta que hemos implementado en los capítulos anteriores a través de un caso de ejemplo concreto. La idea es especificar gráficamente un modelo UML que defina la estructura de un sistema medianamente complejo y luego aplicarle el perfil U2TP para definir casos de tests en los niveles de integración y sistema. Posteriormente se tomará el modelo para generar código fuente en lenguaje Java, utilizando la librería JUnit en la implementación de los casos de test. Ambos, el código para las clases del dominio y el de los casos de test, serán generados automáticamente por la herramienta que estamos presentando en esta tesis.

Se utilizará como ejemplo el desarrollo de un conjunto de test para verificar la lógica de un cajero automático (ATM). Se pretende asegurar su correcto funcionamiento cuando un usuario inicia una transacción para depositar y transferir dinero desde una cuenta hacia otra. Tanto el hardware, el banco y las conecciones de red son emulados, ya que sólamente vamos a verificar la lógica del cajero automático.

7.1 Definición del modelo UML

A lo largo de esta sección se creará un modelo UML para definir la estructura del sistema ATM. Como se mencionó anteriormente, la especificación del modelo se hará gráficamente, utilizando el plugin TopCased.

En primer lugar debemos crear un proyecto para alojar al modelo UML. Para esto se debe:

- Abrir el diálogo "File/New/Other...", expandir "General" y seleccionar "Project". A continuación hacer click en el botón "Next".
- Darle un nombre al proyecto y hacer click en "Finish"

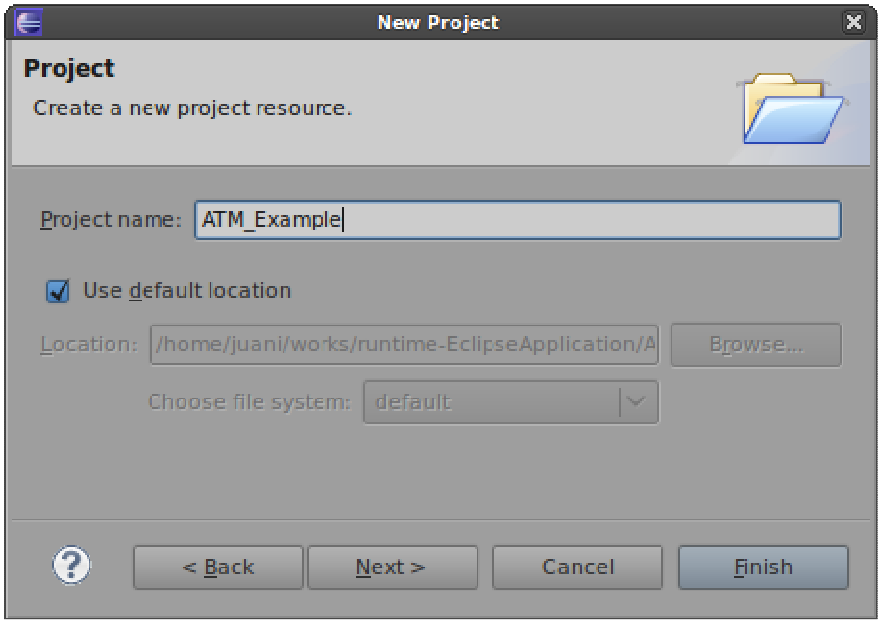

Figura 92 - Creación del proyecto ATM

Ahora estamos en condiciones de crear el modelo. Para esto:

- Nuevamente abrir el diálogo"File/New/Other...". Luego expandir la categoría "Topcased/Topcased Diagrams", seleccionar "UML Model with TOPCased" y hacer click en "Next".

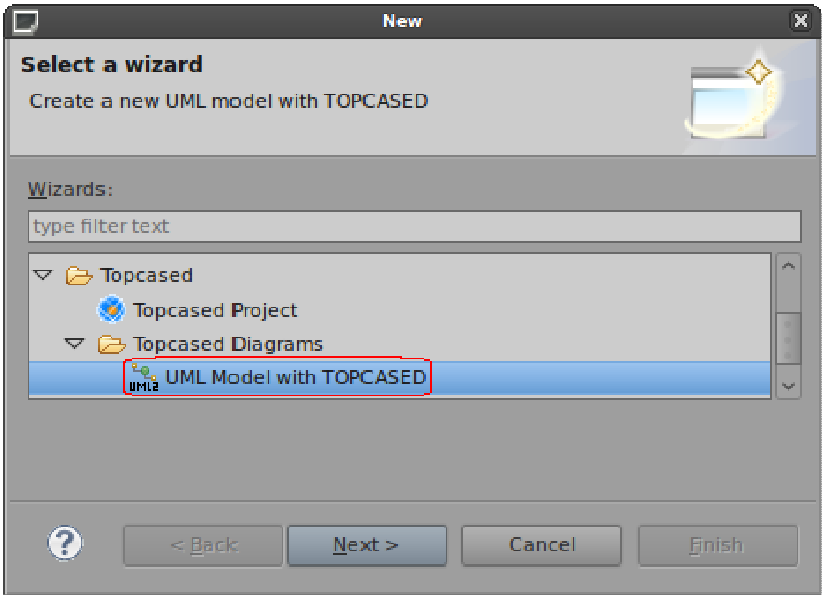

Figura 93 - Wizard de creación de UML con TopCased

- Luego seleccionar "Create model", "Empty model with a default diagram" y "Class Diagram". Completar el campo "Directory" con la ubicación del proyecto creado en el punto anterior y "Model name" con el nombre para el primer paquete del modelo. A continuación hacer click en "Finish":

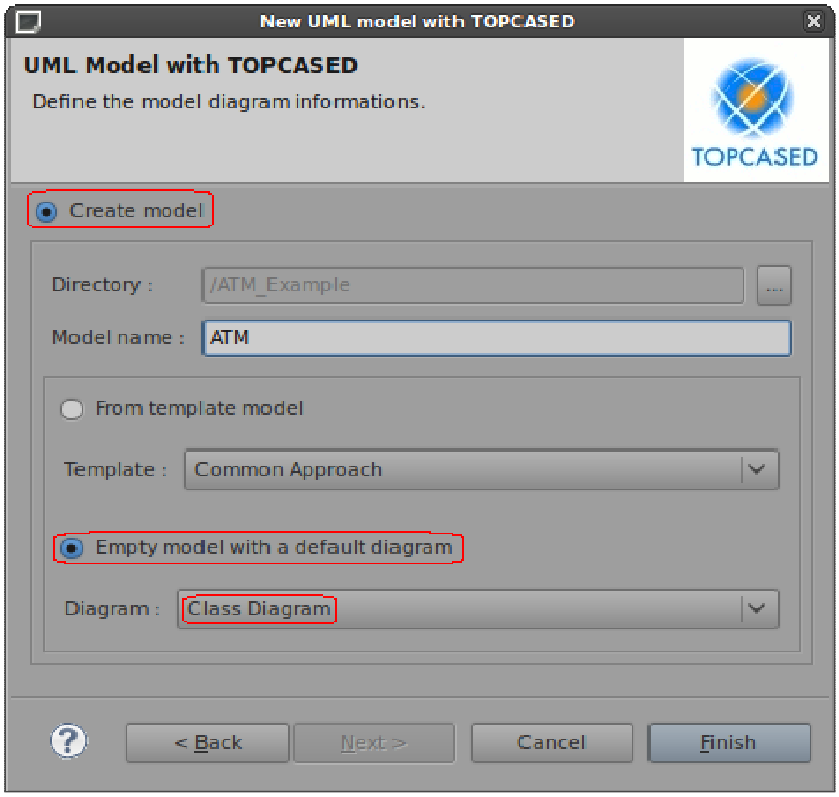

Figura 94 - Wizard para la creación del diagrama de clases

A continuación será creado el modelo UML y su editor se abrirá en la ventana principal:

- Abriendo la vista "Outline" podremos ver el contenido del modelo en forma de árbol. Para esto, hacer click en "Window/Show view/Outline". Luego se abrirá una ventana como la siguiente:

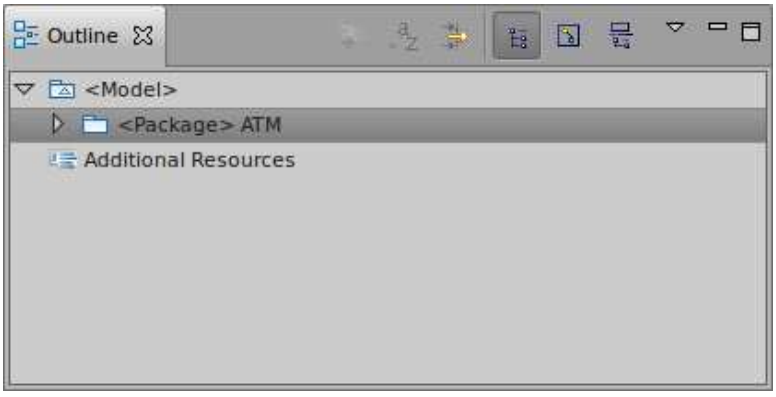

Figura 95 - Resumen del modelo de clases

Ahora vamos a crear los paquetes del modelo:

- En la vista "Outline" hacer click derecho en el elemento "Model" y seleccionar "Add diagram/Class diagram".
- Arrastrar el paquete ATM al editor del modelo. Será creado un elemento gráfico para dicho paquete:

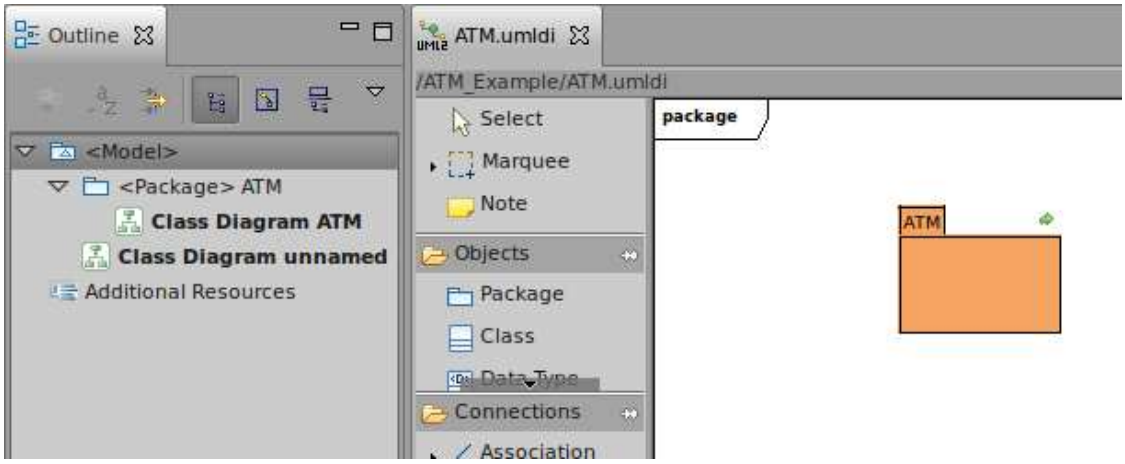

Figura 96 - Paquete ATM en el modelo de clases

- Seleccionar "Package" en la paleta de elementos. Luego hacer click en algún lugar vacío del editor. Se creará un nuevo paquete y se solicitará un nombre para el mismo. Repetir proceso hasta obtener cuatro paquetes como los siguientes:

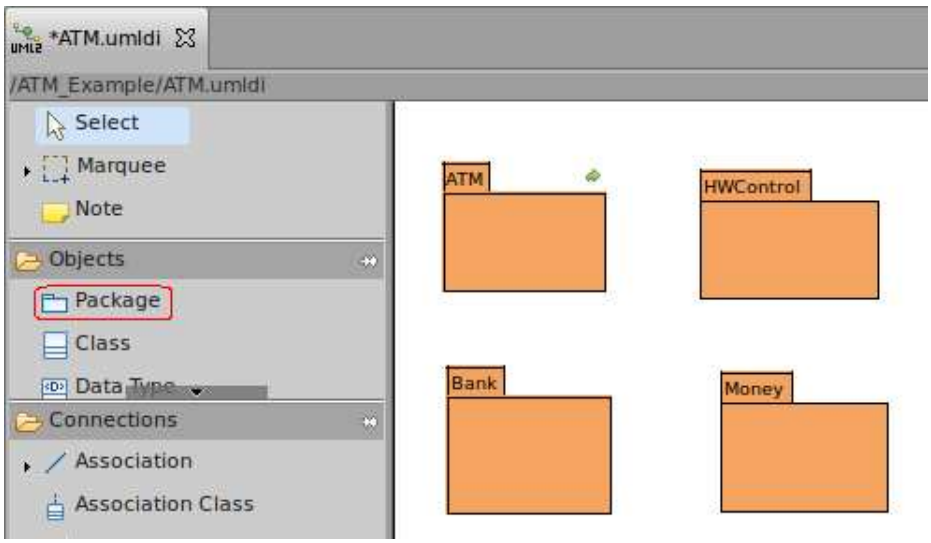

Figura 97 - Paquetes para el caso de prueba

- Seleccionar "Package Import" en la paleta de elementos. Luego hacer click sobre los paquetes de manera de crear las siguientes dependencias:

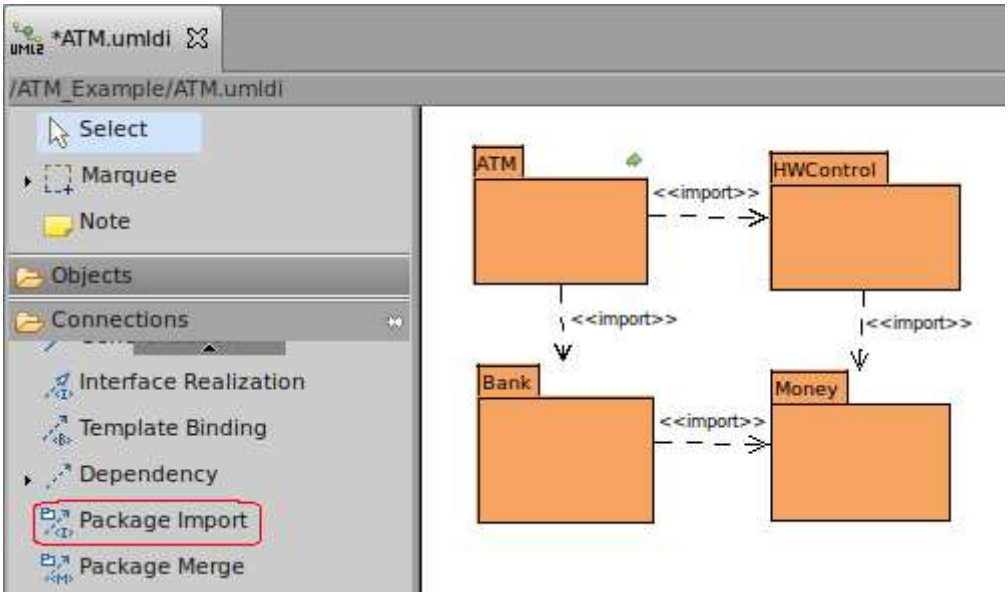

Figura 98 - Dependencias de paquetes

La lógica del cajero automático va a estar contenida en el paquete ATM. Dicho paquete importa el paquete HWControl, donde son especificadas las interfaces del hardware, y el paquete Bank donde es especificada la interface del banco.

A continuación vamos a completar los paquetes con los elementos correspondientes. Comenzaremos con el paquete Money. Para modificar el contenido de un paquete se debe hacer doble click sobre el mismo. Aparecerá un diálogo que nos pide seleccionar un tipo de diagrama. Allí seleccionamos "Class Diagram".

Seguidamente se creará un diagrama de clases vacío y será mostrado en el editor del modelo.

El diagrama de clases anterior se corresponde con el paquete Money. Es decir, todos los elementos creados dentro del diagrama estarán contenidos en el paquete en cuestión.

A continuación vamos a crear una Interface. Para esto se debemos:

- Seleccionar "Interface" en la paleta del editor. Luego hacer click en el interior del paquete. Será creada una interface y se solicitará un nombre para la misma.
- Por cada operación de la interface, seleccionar en la paleta "Operation" y hacer click en la interface. Será creada una operación y se solicitará un nombre para la misma.
- Luego, seleccionar la operación creada y en la vista de propiedades elegir la solapa "Parameters". Hacer click en "Add" para agregar los parámetros de la operación. Por cada uno de ellos se debe establecer: nombre, tipo, visibilidad y dirección. En el siguiente ejemplo se agrega un sólo parámetro que representa el valor de retorno de la operación.
- Repetir el proceso de creación de operaciones hasta lograr la siguiente interface:

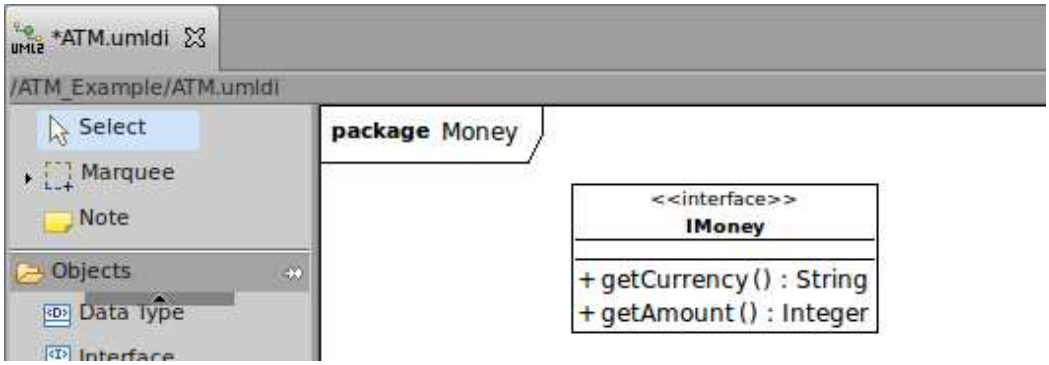

Figura 99 - Interfaz IMoney creada por Topcased

A continuación vamos a crear una Clase. Para esto se debemos:

- Seleccionar "Class" en la paleta del editor. Luego hacer click en el interior del paquete. Será creada una clase y se solicitará un nombre para la misma:

Por cada operación de la clase, seleccionar en la paleta "Operation" y hacer click en la clase. Será creada una operación y se solicitará un nombre para la misma.

- Seleccionar la operación creada en el punto anterior, abrir la vista de propiedades y elegir la solapa "Parameters". Luego hacer click en "Add" para agregar los parámetros de la operación. Por cada uno de ellos establecer: nombre, tipo, visibilidad y dirección.

Figura 100 - Parámetros de una operación de una clase

- Repetir el proceso de creación de operaciones hasta lograr la siguiente clase:

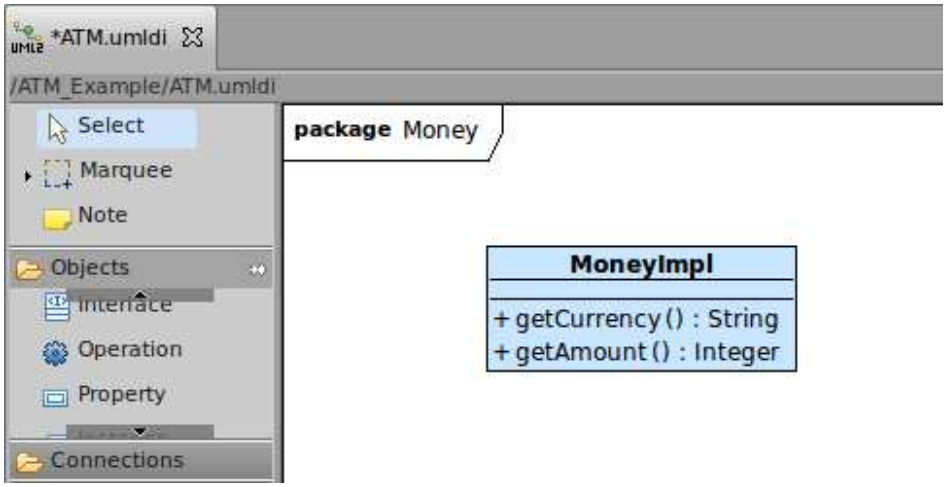

Figura 101 - Clase MoneyImpl

En particular, esta clase debe implementar la interface que hemos creado anteriormente. Para esto, en la paleta del editor seleccionamos "Interface realization". Luego hacemos click en la clase y después en la interface.

Si observamos la vista "Outline" veremos que a medida que creamos elementos gráficamente se irán creando los elementos en el modelo UML. Puesto que hasta ahora hemos creado los paquetes, una interface, una clase y una realización de interface, el contenido de la vista "Outline" se ve así:

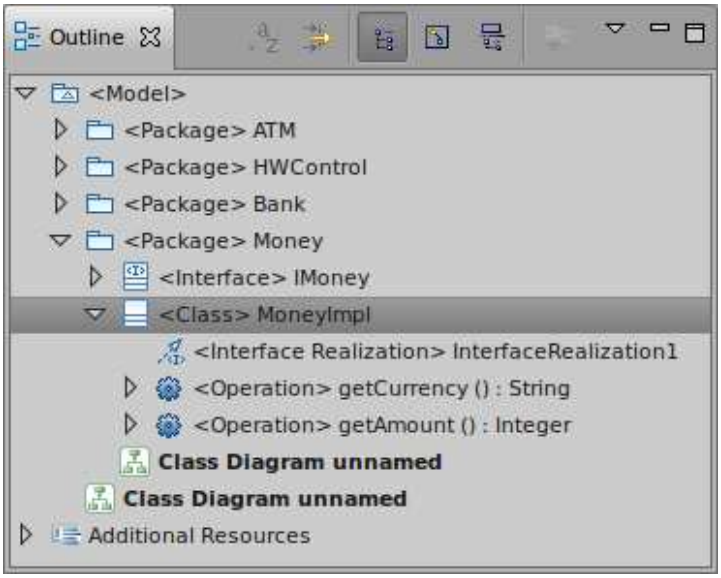

Figura 102 - Resumen del diagrama de clases

En este punto hemos finalizado con la creación de elementos del paquete Money. Ahora nos resta completar el contenido de los demás paquetes. El precedimiento es análogo en todos los paquetes, es decir, se deben crear clases e interfaces con sus respectivas operaciones utilizando el editor, la paleta y la vista de propiedades. En lo que sigue no vamos a entrar en detalle en esos conceptos, pero sí en otros nuevos.

Ahora vamos a completar el contenido del paquete Bank. Este paquete contiene un elemento "tipo de datos" y una interface. Aquí el nuevo concepto es el "tipo de datos", puesto que ya hemos creado una interface antes. Para crear un tipo de datos se sebe:

- Hacer click en la paleta sobre el elemento "Data Type". Luego hacer click en el interior del paquete. Será creado un elemento visual que representa un tipo de datos, solicitando un nombre para el mismo.

Figura 103 - Creación de tipo de datos

De ser necesario agregar operaciones y/o propiedades al tipo creado, haciendo click en el correspondiente elemento de la paleta y luego en el tipo de datos. Luego establecer las propiedades utilizando la vista de propiedades. En nuestro caso, el tipo no contiene operaciones ni propiedades.

Para completar el paquete Money se debe crear una interface como la siguiente:

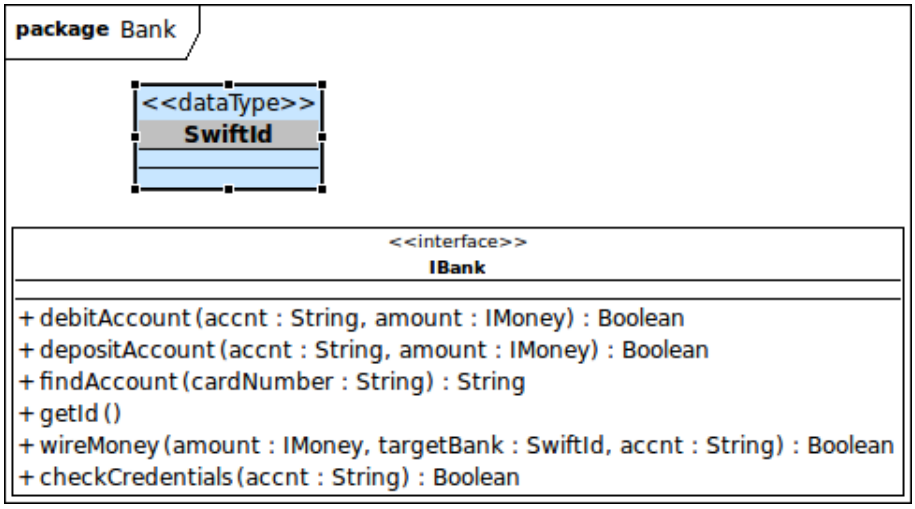

Figura 104 - Paquete Bank

Notar que la interface IBank contiene 6 operaciones y algunas de ellas utilizan como parámetro el tipo IMoney y el tipo de datos SwiftId, ambos importados desde el paquete Bank.

Por su parte, el paquete HWControl sólamente contiene una interface:

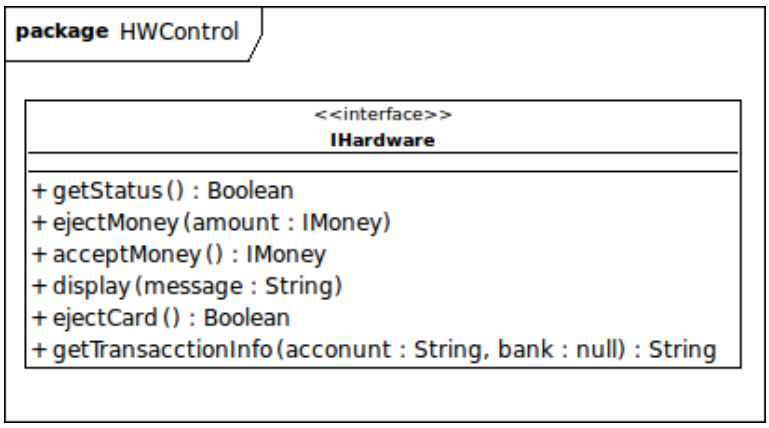

Figura 105 - Paquete HWControl

Puesto que en la creación de este paquete no se utilizan nuevos conceptos, su explicación su explicación es omitida.

Por último nos resta completar el paquete ATM. Este paquete contiene dos clases, una interface y una enumeración. Tanto la creación de clases como interfaces han sido explicadas anteriormente. Por lo tanto, nos concentraremos en la creación del tipo enumerativo. Para esto:

- En la paleta del editor seleccionar el elemento "Enumeration". Luego hacer click en el interior del paquete. Será creado un elemento gráfico para el enumerativo y se solicitará un nombre para el mismo:

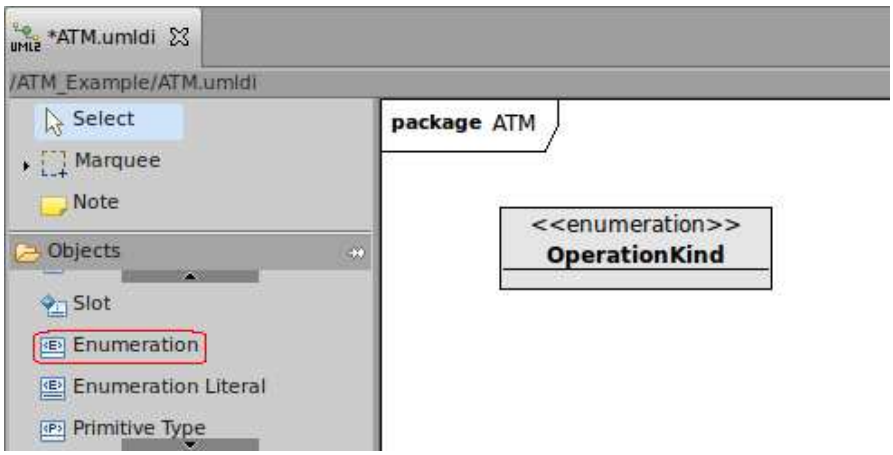

Figura 106 - Creación del elemento gráfico enumerativo

- Por cada valor literal del enumerativo seleccionar en la paleta "Enumeration Literal" y luego en el enumerativo creado anteriomente. Será creado un literal y se solicitará un nombre para el mismo:

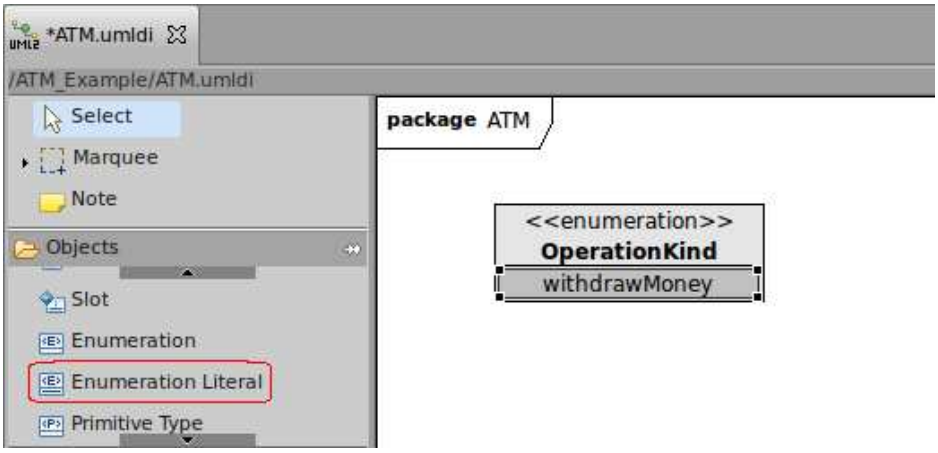

Figura 107 - Creación del literal enumerativo

- Completar el diagrama anterior hasta obtener uno similar al siguiente:

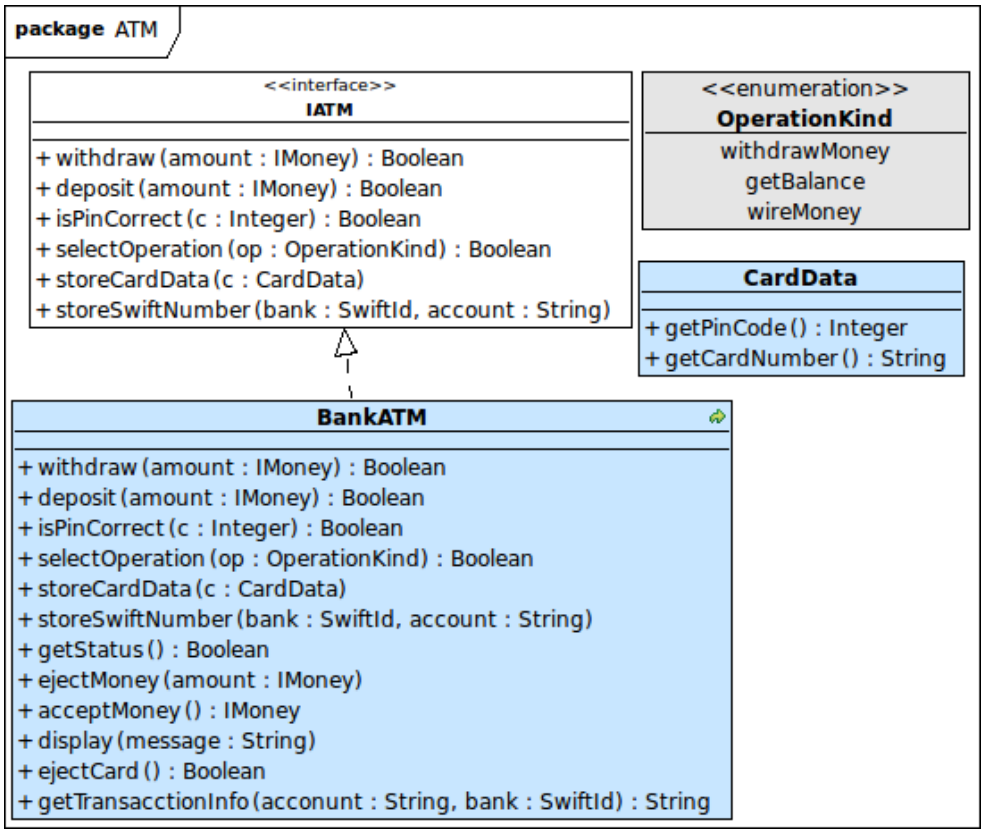

Figura 108 - Paquete ATM

Como se puede observar, el tipo enumerativo debe ser completado con 3 literales. Luego deben ser creadas dos clases: CardData y BankATM. Esta última debe implementar la interface IATM y, puesto que es la encargada de controlar la lógica del cajero automático, será el foco de los casos de test.

En este punto hemos finalizado con la definición del modelo UML para las clases del dominio. En resumen, se han creado cuatro paquetes y sus respectivos contenidos. Entre otras cosas, hemos definido

interfaces, clases, enumerativos, y tipos de datos. A continuación vamos a extender este modelo aplicando el perfil U2TP para especificar los casos de test correspondientes.

7.2 Aplicación del perfil U2TP

Una vez completada la definición del modelo UML para las clases del dominio, es necesario definir los casos de test para verificar el correcto funcionamiento de las mismas. En este apartado vamos a agregar al modelo un paquete que contenga todos los elementos necesarios para especificar completamente los casos de test.

- Como primer paso, debemos abrir el editor del modelo UML que define a las clases del dominio (el que construimos en el apartado anterior). Luego crear un paquete llamado ATMTest. Este paquete debe importar al paquete ATM para tener acceso a los elementos a ser testeados:

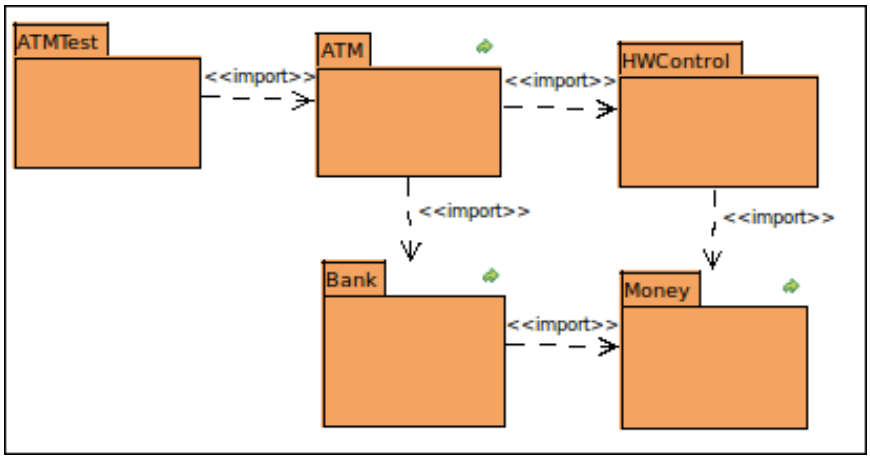

Figura 109 - Creación de los paquetes de test

- Ahora debemos aplicar el perfil U2TP al paquete. Esto nos permitirá utilizar los elementos de perfil dentro del mismo. Para esto, seleccionar el paquete ATMTest, abrir la vista de propiedades y elegir la solapa "Profiles". En la lista de perfiles disponibles marcar el perfil U2TP y luego hacer click en "Add". El pefil pasará a la lista de los perfiles aplicados:

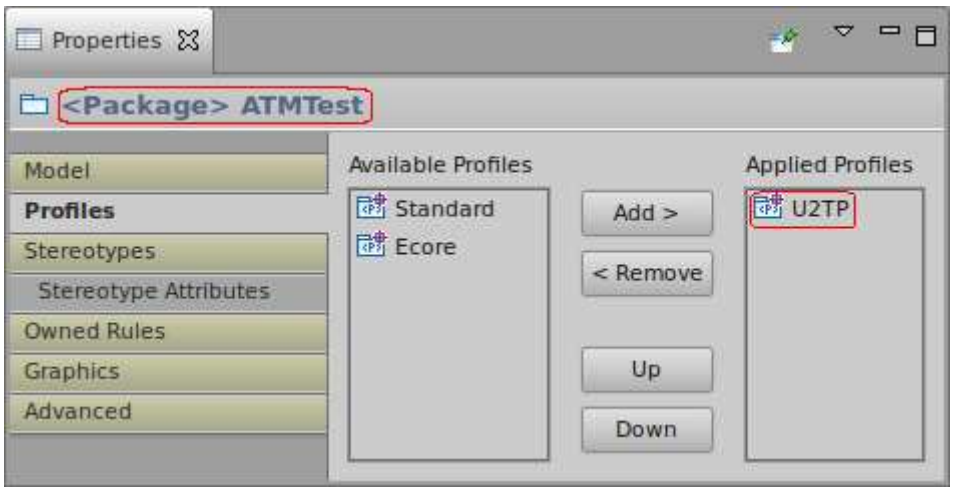

Figura 110 - Aplicación del perfil U2TP

Dentro del paquete ATMTest vamos a crear varios elementos de test. Comenzaremos con los elementos TestComponent. Estos implementarán las interfaces de los paquetes Bank y HWControl, y se utilizarán como emuladores de dichos paquetes. Para crear los elementos TestComponent:

- Hacer doble click sobre el paquete ATMTest. Recordemos que cuando el paquete se abre por primera vez, se solicitará el tipo de diagrama que se desea crear para el mismo. Nuevamente seleccionamos "Class diagram".
- Una vez dentro del paquete, debemos agregar las referencias a las interfaces que serán emuladas. Por cada interface, seleccionarla en la vista "Outline" y arrastrarla hacia el paquete. Será creada un elemento gráfico para la misma. Repetir el proceso para la interfaces IBank e IHardware:

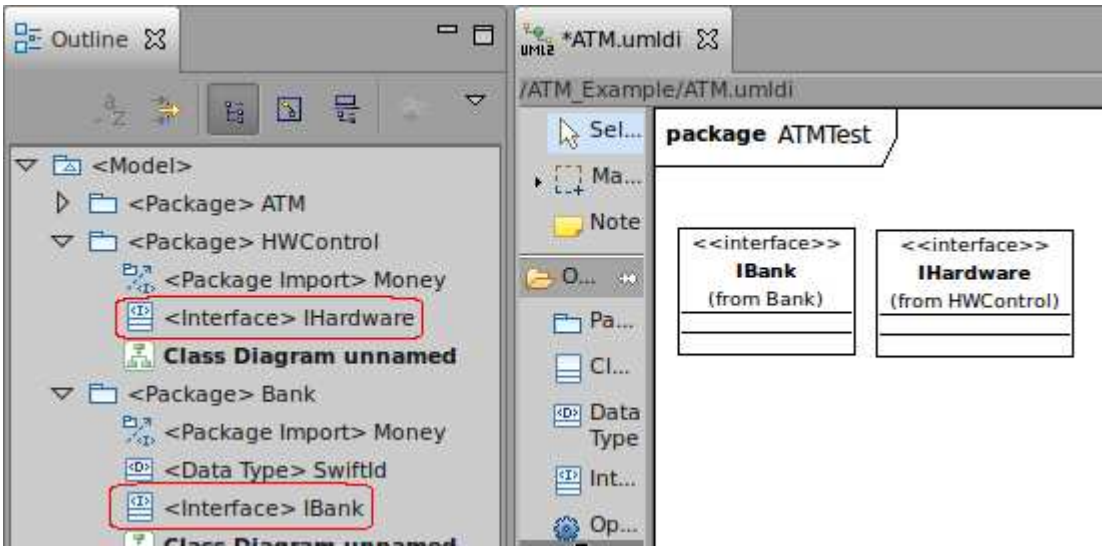

Figura 111 - Referencias a las interfaces emuladas

- Por cada TestComponent, crear una clase con sus respectivas operaciones y propiedades. A su vez, cada clase debe implementar una de las interfaces agregadas anteriormente:

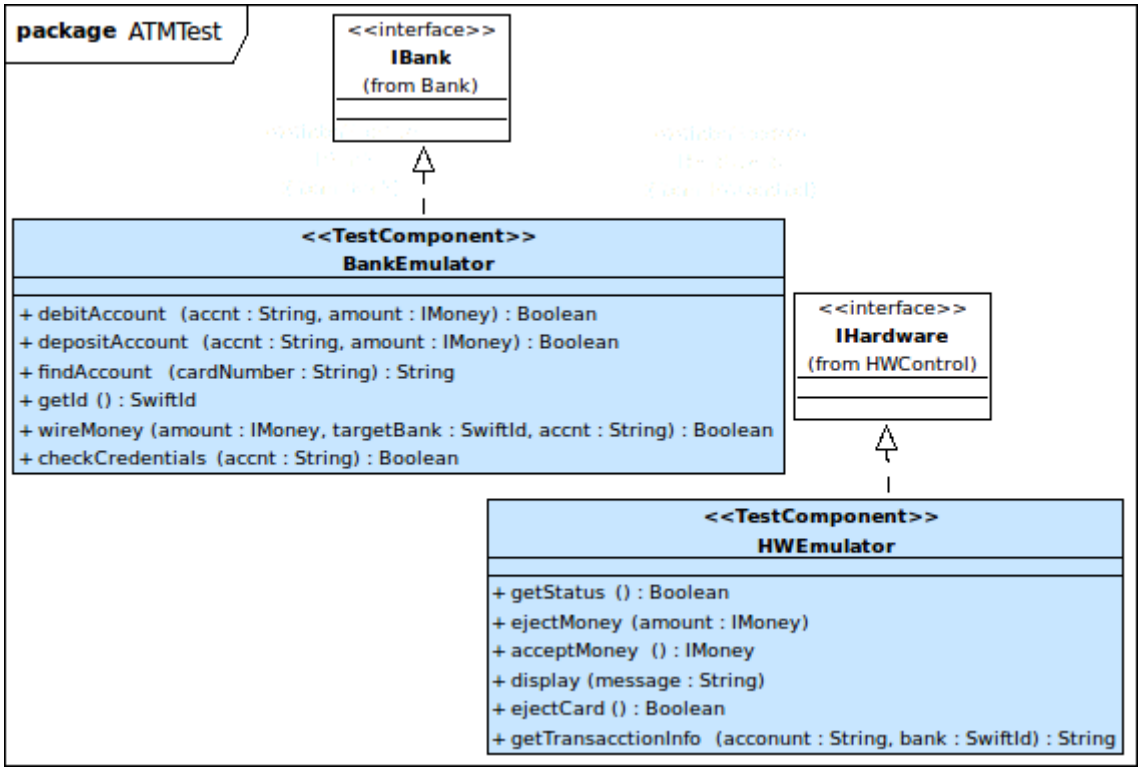

Figura 112 - Implementación de las clases emuladas

En el caso anterior, el TestComponent BankEmulator implementa la interface IBank y HWEmulator implementa la interface IHardware.

- Por cada clase TestComponent, seleccionarla en el editor, abrir la vista de propiedades y elegir la solapa "Stereotypes". En la lista de estereotipos disponibles seleccionar "U2TP::TestComponent" y luego hacer click en "Add". El estereotipo pasará a la lista de estereotipos aplicados:

Figura 113 - Aplicación del estereotipo TestComponent

A continuación vamos a crear los elementos Arbiter y Scheduler, los cuales serán utilizados más tarde por el TestContext. Recordemos que el Arbiter es el encargado de determinar el veredicto final de la ejecución de los casos de test, y el Scheduler es el responsable de crear y ejecutar los TestComponent. Ambos elementos son especificados con simples clases que implementan las interfaces Arbiter y Scheduler

respectivamente. Estas interfaces están definidas dentro del perfil U2TP. Por lo tanto, para utilizarlas hay que crear una referencia para cada una de ellas dentro del paquete de test.

- En la vista "Outline" se deben localizar dichas interfaces (se pueden encontrar bajo la categoría "Additional resources"). Una vez encontradas, se deben arrastrar hacia el paquete ATMTest. Será creada una referencia hacia las mismas:

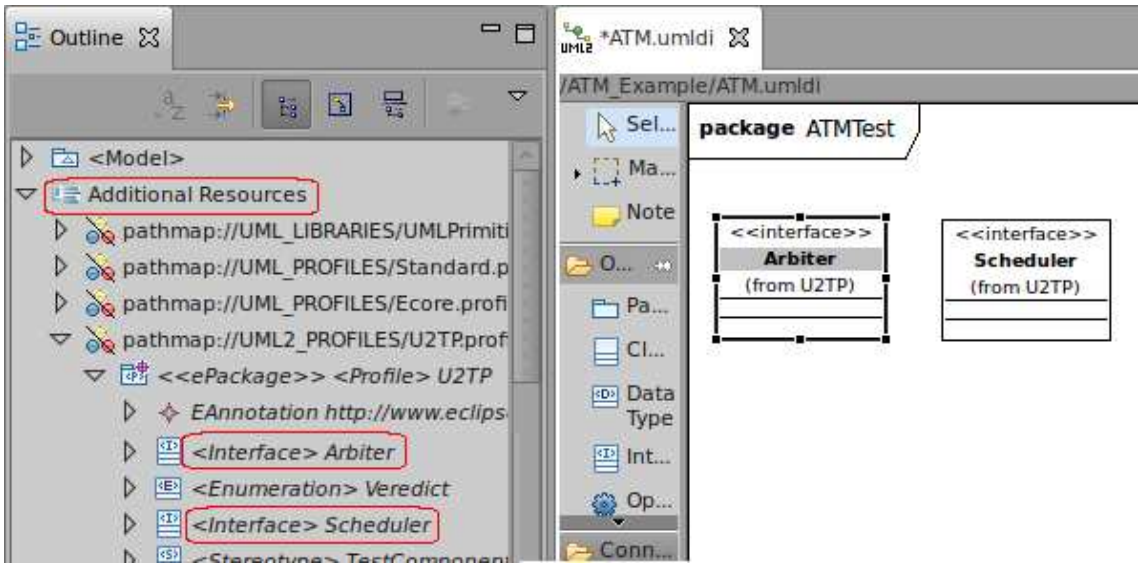

Figura 114 - Referencia a los elementos Arbiter y Scheduler

- Luego se deben crear dos clases: una que implemente la interface Arbiter y la otra Scheduler:

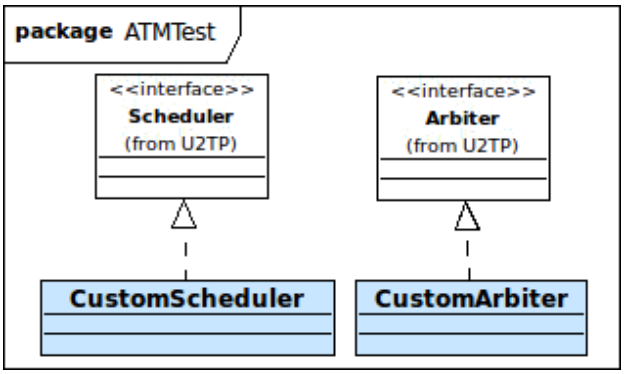

Figura 115 - Implementación del Scheduler y el Arbiter

Seguidamente vamos a crear el elemento más importante del paquete de test. Nos referimos al TestContext, el cual tiene como objetivo agrupar los casos de test y mantener asociaciones hacia los elementos de test, entre ellos, los TestComponent, los SUT, el Arbiter y el Scheduler.

- Para crear un elemento TestContext primero se debe crear una clase (utilizando el elemento "Class" de la paleta):

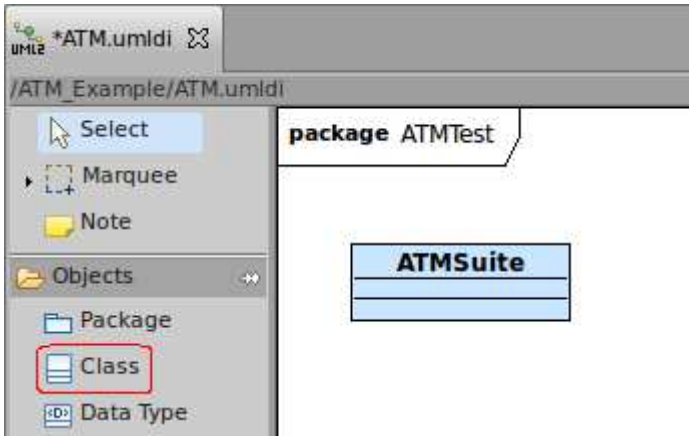

Figura 116 - Creación del TestContext

- Una vez creada la clase, seleccionarla en editor, abrir la vista de propiedades y elegir la solapa "Stereotypes". En la lista de estereotipos disponibles seleccionar "U2TP::TestContext" y luego hacer click en "Add". El estereotipo pasará a la lista de estereotipos aplicados:

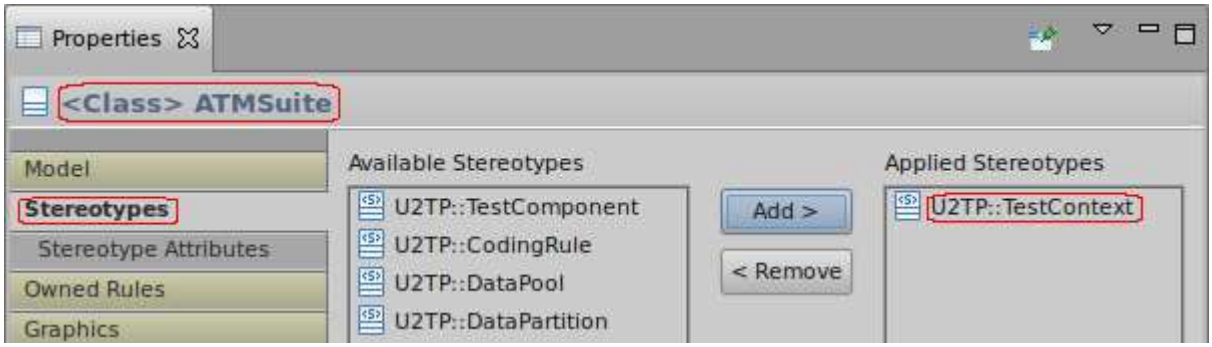

Figura 117 - Aplicación del estereotipo TestContext

- Seleccionar la solapa "Stereotype Attributes" y completar los atributos "Arbiter" y "Scheduler" con las clases que hemos creado anteriormente:

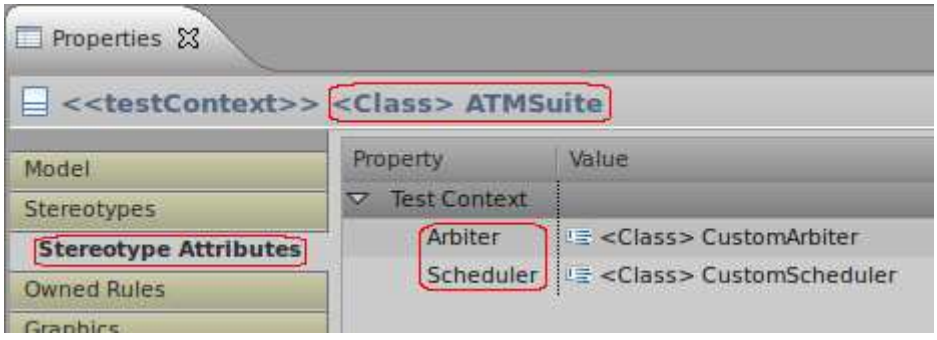

Figura 118 - Asignación del Arbiter y del Scheduler al TestContext

Ahora vamos a crear las asociaciones hacia los elementos TestComponent.

- Para esto, por cada TestComponent seleccionar en la paleta del editor "Association (Composite)", hacer click en el TestContext y luego en TestComponent. Será creado el elemento gráfico que representa a la asociación:

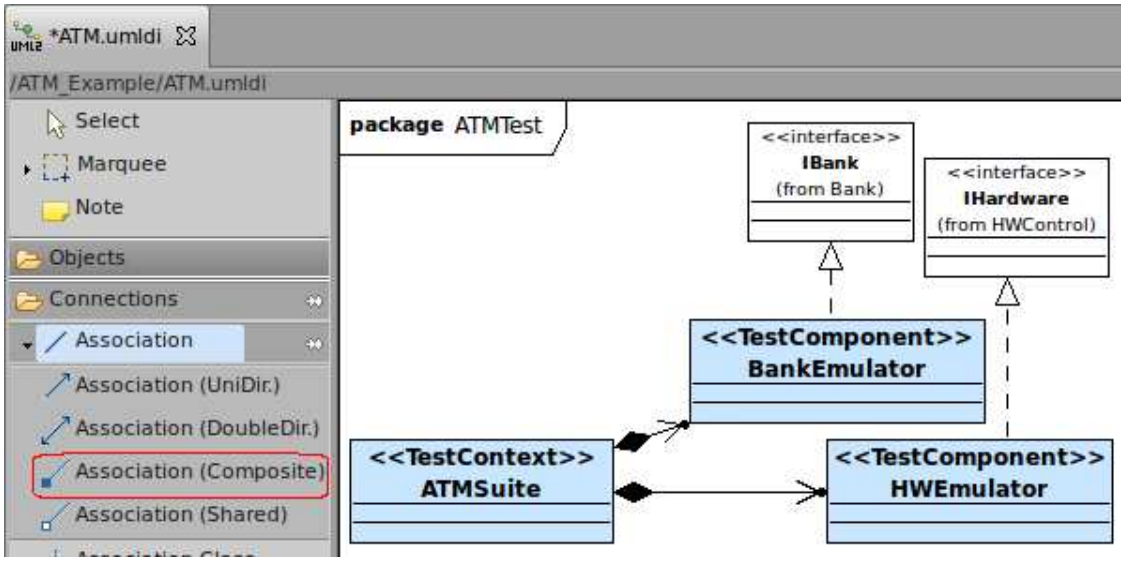

Figura 119 - Asociaciones para los TestComponents

A continución vamos a crear la asociación hacia el elemento SUT. Recordemos que el SUT es el elemento que será sujeto a test. En nuestro caso es el objeto BankATM. Por cada elemento SUT se debe:

- Crear una referencia hacia el elemento SUT en el paquete de test, seleccionándolo en la vista "Outline" y arrastrándolo hacia el paquete de test:
- Seleccionar en la paleta "Association (Composite)", hacer click en el TestContext y luego en el elemento SUT. Será creado el elemento gráfico que representa a la asociación:

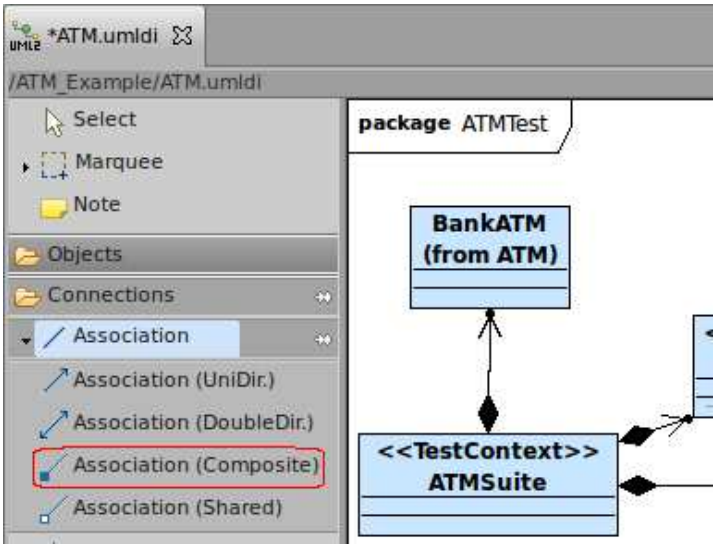

Figura 120 - Asociación con el elemento SUT

Si abrimos la vista "Outline" y expandimos la clase ATMSuite, veremos que dentro de ésta hay una propiedad por cada asociación que hemos creado. En particular, una de esas propiedades es la que representa al elemento SUT:

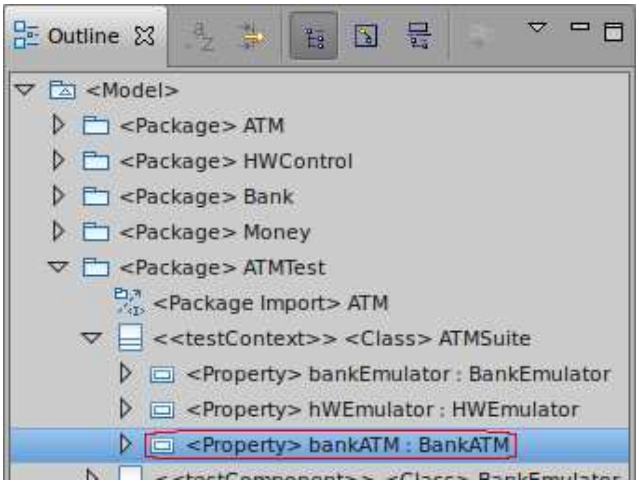

Figura 121 - Refefencia al SUT

- Seleccionar la propiedad que representa al elemento SUT, abrir la vista de propiedades y elegir la solapa "Stereotypes". En la lista de estereotipos disponibles seleccionar el estereotipo "U2TP::SUT", luego hacer click en "Add". El estereotipo pasará a la lista de estereotipos aplicados:

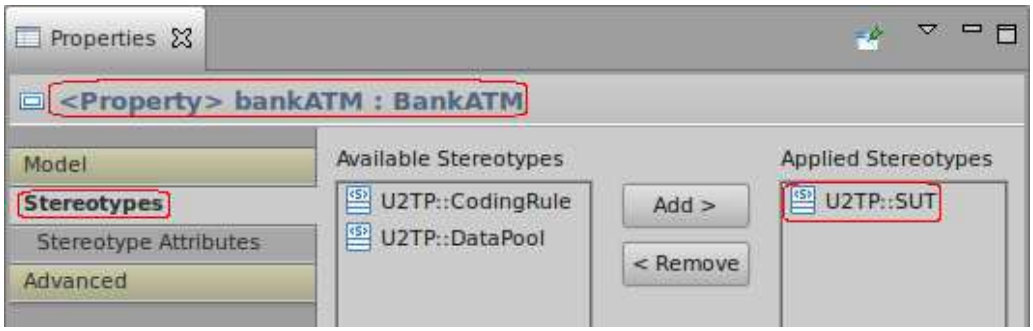

Figura 122 - Aplicación del estereotipo SUT

Por último, vamos a crear los elementos TestCase. Estos elementos están contenidos dentro de un TestContext y especifican cómo los TestComponent interactúan con el SUT para realizar los casos de test y retornar un veredicto. Para implementar un TestCase:

- Dentro de la clase TestContext, crear una operación que tenga como tipo de resultado Veredict:

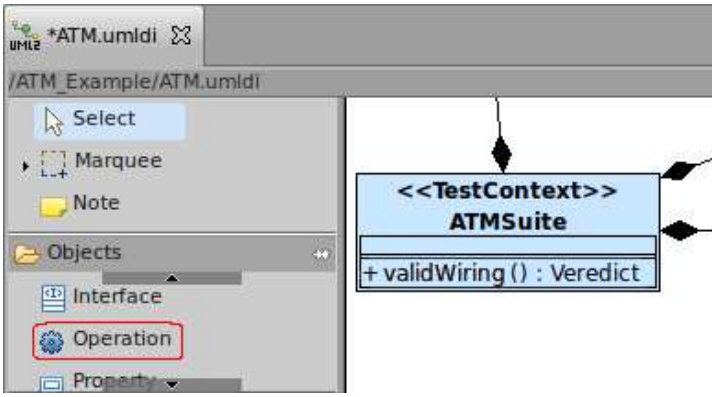

Figura 123 – Operación que devuelve el veredicto

- Seleccionar la operación creada, abrir la vista de propiedades y elegir la solapa "Stereotypes". En la lista de estereotipos disponibles seleccionar "U2TP::TestCase" y luego hacer click en "Add". El estereotipo pasará a la lista de estereotipos aplicados:

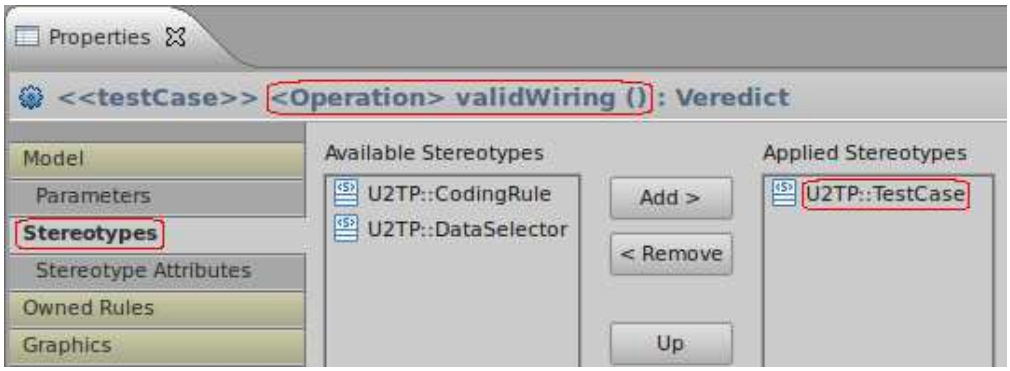

Figura 124 - Aplicación del estereotipo TestCase

- Repetir el procedimiento anterior hasta conseguir tres TestCase como los siguientes:

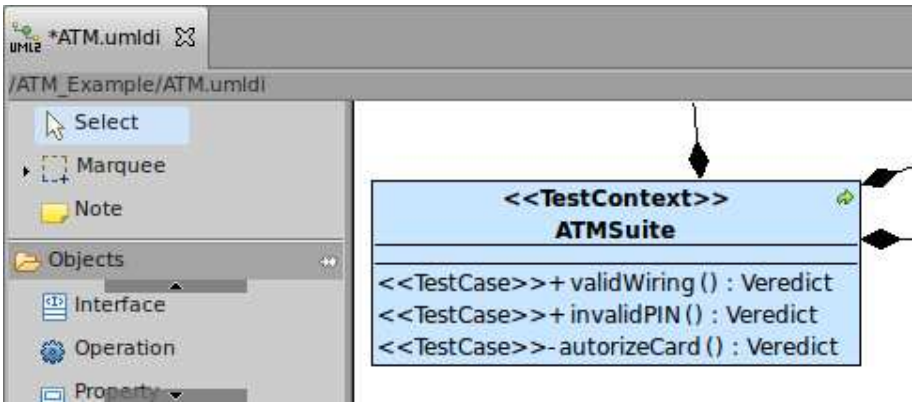

Figura 125 - TestCases del ATMSuite

Aquí hemos finalizado con la especificación de los casos de test. Para simplificar el caso de ejemplo, se utilizaron los elementos de test más significativos de U2TP, entre ellos: TestContext, TestComponent,

TestCase, Arbiter, Scheduler y SUT. La aplicación del resto de los elementos es muy similar. Es decir, la mayoría de los conceptos de test se especifican aplicando estereotipos del perfil U2TP sobre clases y propiedades, o bien creando clases que implementen interfaces predefinidas del perfil.

Además, recordemos que en el desarrollo de esta tesis hemos decidido focalizarnos en la parte estructural del perfil U2TP. Es decir, a la hora de generar el código fuente sólo serán consideradas las estructuras de las clases y no así el comportamiento de las mismas. Por lo tanto, a la hora de especificar los elementos del modelo la definición sus comportamientos es indistinta.

A continuación procedemos a la generación de código fuente para las clases del dominio y los casos de test del modelo que hemos especificado hasta el momento.

7.3 Generación de código fuente Java + JUnit

Como era de esperar, esta es la parte más sencilla del proceso. Esto se debe a que la generación de código fuente es, en gran parte, automática y corre por cuenta de la herramienta que estamos presentando en esta tesis. Sólo es necesario determinar cuál es el modelo que se utilizará y para qué tecnología se desea generar el código fuente. En un futuro se espera que la generaración de código sea totalmente automática. Esta proposición será presentada más adelante en el capítulo de trabajos futuros.

En este ejemplo se utilizarán las tecnologías Java y JUnit puesto que fueron desarrolladas anteriormente durante el capítulo de implementación. Recordemos que el lenguaje Java se utiliza para la generación del código correspondiente a las clases del dominio, y el Framework JUnit se utiliza para la generación de los casos de test. Aún así, el proceso es análogo para cualquier tecnología.

El primer paso en la generación de código fuente es crear un proyecto donde se pueda alojar el mismo. En nuestro caso, se debe crear un proyecto Java. Para esto:

- Abrir el diálogo "File/New/Project...", expandir "Java" y seleccionar "Java Project". A continuación hacer click en el botón "Next".
- Dar un nombre para el proyecto y hacer click en "Finish".

Seguidamente vamos a crear un archivo ejecutor (en inglés launcher) que más tarde nos permitirá iniciar la generación del código fuente. Este archivo debe ser creado sólo una vez. En las sucesivas generaciones de código sólo es ejecutado. Para crearlo se debe:

- Abrir el diálogo "File/New/Other...", expandir "Acceleo", seleccionar "Module launcher" y hacer click en "Next".

En la lista de módulos, seleccionar el módulo correspondiente a la tecnología para la que deseamos generar el código fuente.

- En nuestro caso, seleccionar "U2TP to JUnit". Luego hacer click en "Next".
- Completar el campo "Container" con la ubicación del proyecto que hemos creado anteriormente y "File name" con un nombre para el archivo ejecutor. Después hacer click en "Next":

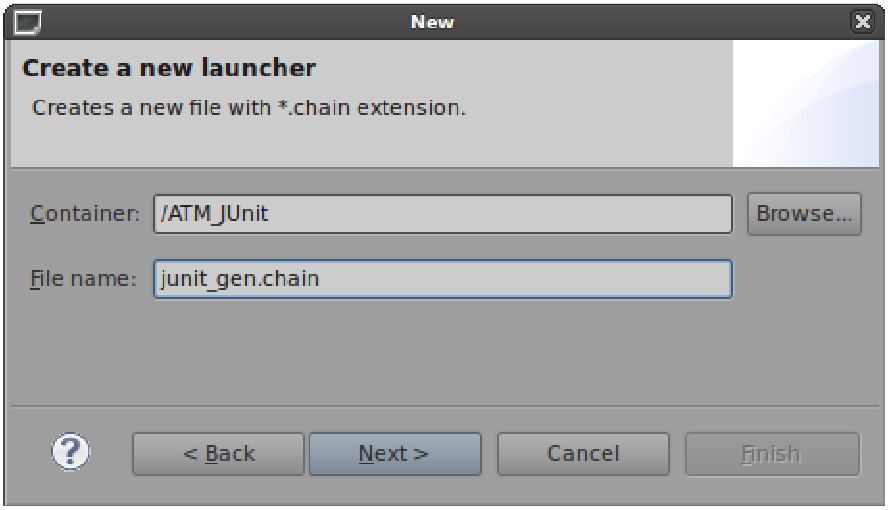

Figura 126 - Creación de la cadena de ejecución

- Seleccionar el archivo del modelo UML+U2TP a partir del cual se generará el código fuente, el directorio donde se va a alojar el código fuente generado y la ubicación para el archivo de log. En este último se registrarán los errores producidos durante el proceso de generación de código. Luego hacer click en "Finish".

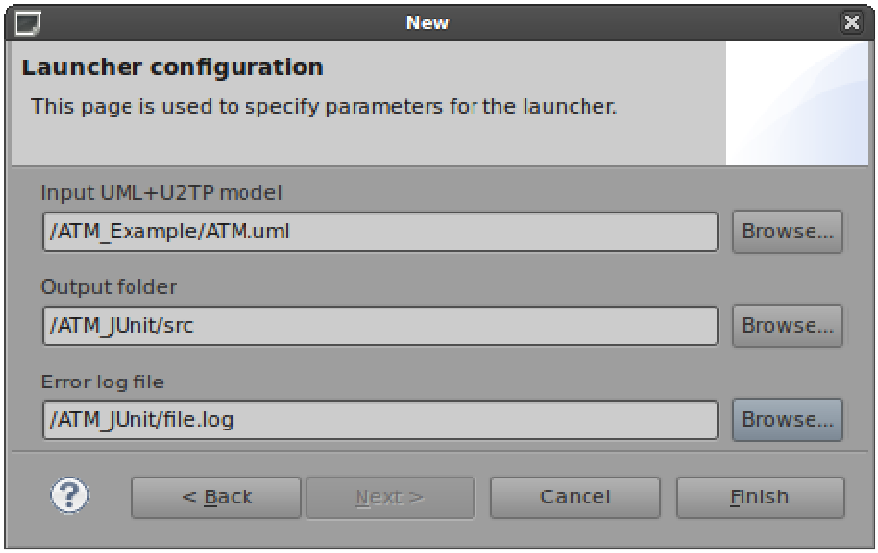

Figura 127 - Configuración de la ejecución de la cadena

Si observamos nuevamente el proyecto, veremos que ha sido creado un archivo ejecutor (con el nombre que le asignamos en el diálogo de creación). Para iniciar la generación de código, simplemente hacer click derecho sobre el mismo y seleccionar "Launch":

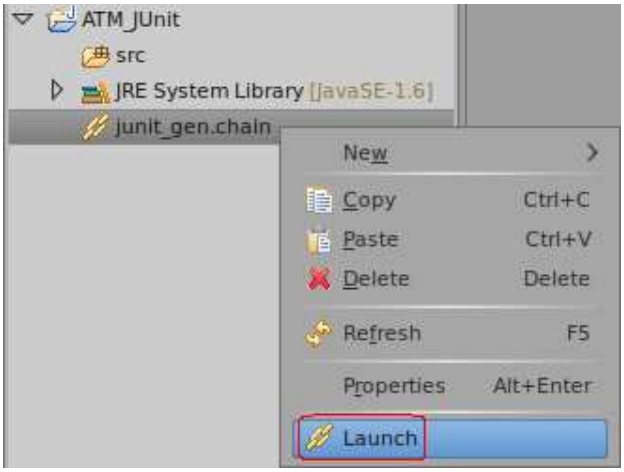

Figura 128 - Ejecución de la cadena

Así se iniciará el proceso de generación de código y un diálogo de progreso mostrará el avance del mismo. Una vez terminada la generación de código, el proyecto tendrá la siguiente forma:

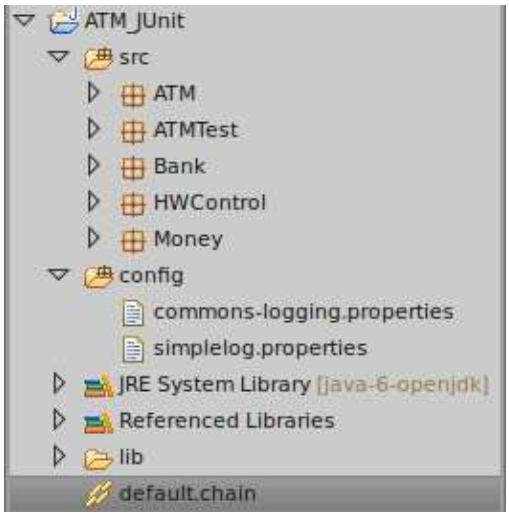

Figura 129 - Estructura del código generado

Como se puede observar, dentro del directorio "src" es creado un paquete Java por cada paquete del modelo UML+U2TP origen. Por su parte, en el directorio "config" son creados los archivos de configuración del Framework de testing. Por último, en el directorio "lib" son agregadas todas las librerías (archivos JAR) correspondientes al Framework de testing, en este caso JUnit.

Una vez generado el código es posible modificar ciertas partes del mismo para especializar su funcionamiento. En particular, se debe completar el comportamiento de los elementos de test (recordemos que en la generación de código sólo se considera la parte estructural de éstos). Inicialmente, todos los elementos de test tienen asignado un comportamiento por defecto, el cual permite que la ejecución de los casos de test sea satisfactoria. Por ejemplo, los TestCase son generados de la siguiente manera:

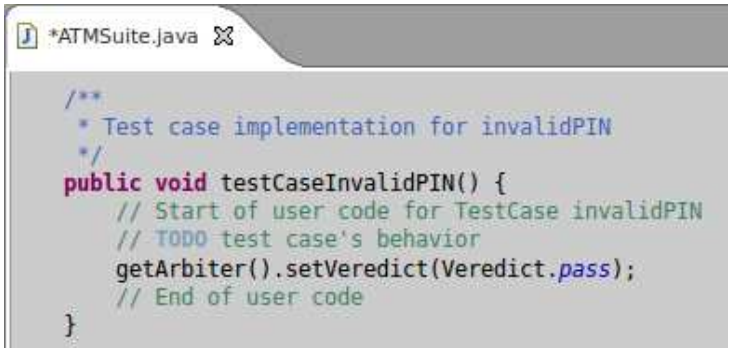

Figura 130 - TestCase generado

Notar que, por defecto los TestCase asignan al Arbiter el veredicto Pass, indicando que la ejecución del test fue satisfactoria. Además, dentro del método hay una etiqueta TODO. Este tipo de etiquetas se utilizan para sugerir a los desarrolladores completar ciertas partes del código. Luego es posible ver todas las etiquetas de este tipo en la vista de tareas ("Window/Show view/Tasks"):

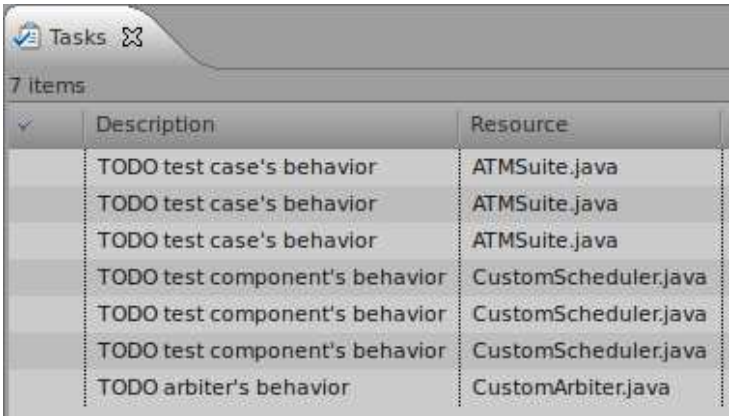

Figura 131 - TODO de tareas a realizar

Cada vez que se completa una porción de código debe eliminarse la etiqueta TODO. De esta forma se elimina la tarea de la vista y el desarrollador puede saber qué partes del código le restan completar. Por ejemplo, el TestCase anterior podría completarse de la siguiente forma:

Figura 132 - Test Case completado manualmente

El TestCase invalidPin verifica que si son ingresados una tarjeta correcta y un PIN incorrecto, el PIN será rechazado y el usuario deberá ingresarlo nuevamente. Como se puede observar, la etiqueta TODO ha sido eliminada y ha sido completado el comportamiento del TestCase. Vale destacar que el código modificado debe estar entre las etiquetas "Start of user code" y "End of user code". Recordemos que el código incluído entre dichas etiquetas será conservado durante las sucesivas generaciones de código.

De esta forma, completando el comportamiento de los elementos de test, damos por finalizada la tarea de generación de código fuente. En resumen, se debe crear (sólo la primera vez) el archivo ejecutor, luego accionarlo (con la opción "Launch") y posteriormente completar los comportamientos de los elementos de test sugeridos en la vista de tareas.

8 Conclusiones

MDE es actualmente uno de los paradigmas más prometedores en su género. Propone mejorar el proceso de construcción de software basándose en un proceso guiado por modelos soportado por potentes herramientas. Los modelos, orientados al dominio, constituyen el foco principal en el desarrollo de software. El código, orientado a la plataforma, es generado automáticamente aplicando una serie de transformaciones sobre el modelo. De esta forma, MDE permite reducir el tiempo y los costos del desarrollo de software, y adaptarlo rápidamente a los cambios tecnológicos y a modificaciones en los requisitos, siempre manteniendo la consistencia entre los modelos y el código del software.

Como hemos mencionado, UML es el lenguaje de modelado más utilizado en el desarrollo con MDE. Los perfiles UML, provistos a partir de la versión 2.0 de UML, se suman a éste para lograr definiciones más específicas de los dominios particulares. En especial, el perfil U2TP (UML 2.0 Testing Profile) es un estándar de la OMG que aún está en desarrollo que permite la especificación de casos prueba (tests) sobre un modelo UML. Debido a su reciente y no definitiva especificación, son pocas las herramientas de modelado que permiten la definición de casos de prueba con U2TP, más aún, ninguna de ellas permite la derivación automática del código fuente para los mismos.

A lo largo de esta tesis hemos desarrollado una herramienta de software que permite definir gráficamente modelos UML, aplicando el perfil U2TP para especificar los casos de prueba, y a partir de éste derivar el código fuente para las clases del dominio y los casos de prueba. La herramienta, basada en el Framework Eclipse, posee una estructura modular que posibilita la generación de código en múltiples lenguajes de programación y Frameworks de testing. Además hace uso de los Frameworks TopCased y Acceleo para valerse de cuestiones ya resueltas por la comunidad MDE, como son la edición gráfica de modelos UML y la generación de código fuente a partir de éstos.

Después de analizar en detalle los conceptos incluidos en el perfil U2TP y al momento de diseñar la herramienta, hemos decidido considerar sólo la parte estructural del perfil. Esta decisión nos ha permitido capitalizar nuestros esfuerzos y lograr una solución concreta y funcional al problema planteado. Sumado a esto, consideramos muy importante habernos esforzado en una definición modular de la herramienta, ya que nos permitió determinar con claridad los diferentes puntos de trabajo y, en un futuro, facilitará significativamente a las extensiones de la misma.

Para lograr la generación de código en múltiples lenguajes y Frameworks de testing era necesario pensar en un esquema modular, donde cada componente se encargue de la generación en un lenguaje o Framework específico. Es por esto que la arquitectura de nuestra solución se dividió en tres capas. Cada capa está formada por un conjunto de plugins de Eclipse. La funcionalidad de la primer capa es independiente de cualquier lenguaje o Framework de testing. En cambio, las funcionalidades de la segunda y tercer son más específicas. Cada plugin de la segunda capa se encarga de generar código fuente para un lenguaje de programación en particular (como Java, C++ o PHP), mientras que cada plugin de la tercera se dedica a un Framework de testing específico (como JUnit, TTCN o PHPUnit).

Como parte de esta tesis hemos presentado la implementación de tres plugins de Eclipse, uno para cada capa de la arquitectura. Estos permiten la definición gráfica de modelos UML aplicando el perfil U2TP, y su posterior derivación de código fuente en lenguaje Java, utilizando el Framework de testing JUnit para la generación de los casos de prueba. Estos módulos podrán ser utilizados como referencia en las futuras extensiones de la herramienta para otros lenguajes de programación y Frameworks de testing.

Con el desarrollo de nuestra herramienta hemos comprobado que el paradigma MDE es un enfoque muy poderoso y puede llevarse a la realidad con muy buenos resultados. Además, consideramos que brinda un esquema flexible y potencialmente puede ser aplicado en cualquier ámbito del desarrollo del software.

9 Trabajos futuros

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

- Extender los plugins que hemos desarrollado para soportar la generación de código fuente de los conceptos faltantes del perfil U2TP que no fueron tratados. Esto implica extender los plugins desarrollados agregando soporte para los paquetes Test Behavior, Test Data y Test Time.
- Implementar nuevos plugins para la segunda capa de la arquitectura, de manera de permitir la derivación de código fuente para diversos lenguajes de programación y Frameworks de test, como por ejemplo C++, C#, PHP o Smalltalk entre otros.
- Analizar las posibilidades para realizar ingeniería inversa de manera que los modelos UML se mantengan actualizados luego de que los casos de prueba sean modificados desde el código fuente.

10 Bibliografía

- **Acceleo** [Online] / auth. Acceleo. - http://www.acceleo.org.
- **Desarrollo de software dirigido por modelos, Conceptos teóricos y su aplicación práctica** / auth. Pons Claudia, Giandini Roxana and Perez Gabriela.
- **Eclipse** [Online] / auth. Eclipse. - http://www.eclipse.org/.
- **Eclipse EMF** [Online] / auth. Eclipse. - Eclipse EMF http://www.eclipse.org/modeling/emf/.
- **Eclipse GEF** [Online] / auth. Eclipse. - http://www.eclipse.org/gef/.
- **Eclipse UML2** [Online] / auth. Eclipse. - http://www.eclipse.org/modeling/mdt/?project=uml2.
- **Generación de Micromundos para Transformaciones de Modelos** / auth. Castro Manacero Noel, Saporiti Martín
- **JMock** [Online] / auth. JMock. - http://www.jmock.org.
- **JUnit** [Online] / auth. JUnit. - http://www.junit.org.
- **Manejo visual de transformaciones entre modelos MOF** / auth. Rivera Conrado, Rodriguez Nicolás
- **MDA Guide** / auth. OMG.
- **Meta Object Facility (MOF) 2.0** [Book] / auth. OMG. - 2003.
- **Model-Driven Testing with UML 2.0** / auth. Dai Zhen Ru.
- **MOF 2.0 Query/View/Transformations (QVT)** [Book] / auth. OMG. - 2005.
- **OMG** [Online] / auth. OMG. - http://www.omg.org.
- **The Unified Modeling Language Infrastructure 2.0** [Book] / auth. OMG. - 2005.
- **The UML Testing Profile** / auth. Alessandra Cavarra
- **TopCased** [Online] / auth. TopCased. - http://www.topcased.org.
- **UML 2 Testing Profile** [Book] / auth. OMG. - 2007.
- **Una Introducción a los Perfiles UML** / auth. Fuentes Lidia, Vallecillo Antonio .