Perfiles de testing aplicados a modelos de software

Luis Fernando Palacios

Director: Claudia Pons

Tesis presentada a la Facultad de Informática de la Universidad Nacional de La Plata como parte de los requisitos para la obtención del título de Magíster en Ingeniería de Software.

La Plata, Octubre de 2009
Facultad de Informática
Universidad Nacional de La Plata
Argentina

Índice general

Índice de figuras

Índice de Tablas

Perfiles de testing aplicados a modelos de software

INTRODUCCIÓN

Actualmente, la complejidad de los sistemas de software se ha incrementado. El software sufre cambios y evoluciona durante todo el ciclo de vida del desarrollo, por lo tanto es fundamental contar con un proceso de pruebas que detecte errores y fallas en la implementación en todas las etapas garantizando además la calidad del producto final. Las técnicas de validación y verificación también se pueden aplicar a los modelos de pruebas de software permitiendo automatizar la creación y ejecución de los casos de pruebas, aumentando la productividad y reduciendo los costos.

El Desarrollo de software Dirigido por Modelos (en inglés Model Driven software Development, MDD) propone un nuevo mecanismo de construcción de software a través de un proceso guiado por modelos que van desde los más abstractos (en inglés Platform Independent Model, PIM) a los más concretos (en inglés Platform Specific Model, PSM) realizando transformaciones y/o refinamientos sucesivos que permitan llegar al código aplicando una última transformación. Dentro del contexto de MDD, las Pruebas de software Dirigidas por Modelos (en inglés Model-Driven Testing, MDT) son una forma de prueba de caja negra [Bei 95] que utiliza modelos estructurales y de comportamiento para automatizar el proceso de generación de casos de prueba.

Para ello, MDT utiliza un lenguaje definido con mecanismos de perfiles basado en el Perfil de Pruebas UML [U2TP 04] (en inglés UML 2.0 Testing Profile, U2TP). Este lenguaje permite diseñar los artefactos de los sistemas de pruebas e identificar los conceptos esenciales del dominio en cuestión adaptados a plataformas tecnológicas y a dominios específicos. La especificación del Perfil de Pruebas UML proporciona además un marco formal para la definición de un modelo de prueba bajo la propuesta de caja negra que incluye las reglas que se deben aplicar para transformar dicho modelo a código ejecutable.

Actualmente existen herramientas basadas en técnicas de validación y verificación formal de programas y de chequeo de modelos que se enfocan principalmente en cómo expresar las transformaciones. Sin embargo, la validación y verificación en forma automática a través de una alternativa práctica como es el testing dirigido por modelos lo hacen en menor medida. El testing consiste en el proceso de ejercitar un producto para verificar que satisface los requerimientos e identificar diferencias entre el comportamiento real y el comportamiento esperado (IEEE Standard for Software Test Documentation, 1983), lo cual es más simple y no requiere tener experiencia en métodos formales comparadas con las técnicas mencionadas anteriormente.

Tanto UML y sus extensiones, como el Perfil de Pruebas UML, están definidos a través de una especificación de tecnología estandarizada por OMG (en inglés Object Management Group) denominada MOF [MOF] (en inglés Meta-Object Facility).

MOF es un meta-metamodelo utilizado para crear metamodelos que pueden ser transformados a texto a través de herramientas que soporten la definición MOF. MOFScript [Oldevik 06] es un lenguaje textual basado en QVT [QVT] (en inglés "Queries, Views and Transformations") que puede ser utilizado para realizar transformaciones de metamodelos MOF a texto.

El objetivo de esta tesis es desarrollar una herramienta que permita realizar las transformaciones en forma automática de los modelos de pruebas estructurales y de comportamiento a código JUnit [JUnit]. Para lograr dicho objetivo, definimos el

lenguaje para modelar dominios de pruebas utilizando el Perfil de Pruebas UML y las reglas formales de transformación de modelos U2TP a código de testing JUnit basadas en el lenguaje MOFScript.

Esta tesis está organizada de la siguiente manera. En el capítulo 2 se introducen los conceptos del desarrollo de software dirigido por modelos. En el capítulo 3 se describen las pruebas de software dirigidas por modelos. En el capítulo 4 se definen las reglas de transformación de modelos de prueba a código JUnit. En el capítulo 5 se describe la implementación de la herramienta que permite transformar en forma automática modelos definidos con el Perfil de Pruebas UML a código JUnit, además de describir la arquitectura utilizada en el proyecto. El capitulo 6 muestra un caso de estudio del trabajo realizado desde la perspectiva del usuario final. En el capítulo 7 se detallan los trabajos relacionados. En el capítulo 8 se exponen las conclusiones finales y se citan futuros trabajos.

DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

Este capítulo presenta los conceptos básicos del modelado de software a través de la creación de modelos. Primero, definimos el concepto y los tipos de modelos. Luego, describimos la Arquitectura Dirigida por Modelos, los conceptos de meta-metamodelo y metamodelo para finalmente hablar sobre UML, su infraestructura y la importancia de sus extensiones por medio de perfiles y estereotipos.

1. Modelos

El Desarrollo de software Dirigido por Modelos (en inglés Model Driven Software Development, MDD) se ha convertido en un nuevo paradigma de desarrollo de software basado en un proceso guiado por modelos que se van generando desde los más abstractos a los más concretos a través de pasos de transformación y/o refinamientos, hasta llegar al código aplicando una última transformación. Los puntos claves de la iniciativa MDD fueron identificados en [Booch 04] de la siguiente forma:

1. El uso de un mayor nivel de abstracción en la especificación tanto del problema a resolver como de la solución correspondiente, en relación con los métodos tradicionales de desarrollo de software.

2. El aumento de confianza en la automatización asistida por computadora para soportar el análisis, el diseño y la ejecución.

3. El uso de estándares industriales como medio para facilitar las comunicaciones, la interacción entre diferentes aplicaciones y productos, y la especialización tecnológica.

2. Tipos de modelos

La siguiente es una clasificación que identifica los tipos de modelos que MDD considera yendo desde los más abstractos e independientes de la plataforma de software hasta los más específicos de una tecnología que además involucran su implementación.

- **El modelo independiente de la computación** (en inglés Computation Independent Model, CIM).

  Un CIM es una vista del sistema desde un punto de vista independiente de la computación que no muestra detalles de la estructura del sistema. Usualmente al CIM se lo llama modelo del dominio y en su construcción se utiliza un vocabulario que resulta familiar para los expertos en el dominio en cuestión;

juega un papel muy importante en reducir la brecha entre los expertos en el dominio y sus requisitos por un lado, y los expertos en diseñar y construir artefactos de software por el otro.

- **El modelo independiente de la plataforma** (en inglés, Platform Independent Model, PIM).

  Un PIM es un modelo con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación. Dentro del PIM el sistema se modela desde el punto de vista de cómo se soporta mejor al negocio, sin tener en cuenta cómo va a ser implementado.

- **El modelo específico de la plataforma** (en inglés, Platform Specific Model, PSM).

  Un PIM puede generar uno o múltiples PSMs, cada uno para una tecnología en particular. Generalmente, los PSMs deben colaborar entre sí para una solución completa y consistente.

- **El modelo de la implementación (Código).**

  El paso final en el desarrollo es la transformación de cada PSM a código fuente. Ya que el PSM está orientado al dominio tecnológico específico, esta transformación es bastante directa. El tipo de sistema descripto por un modelo es relevante para las transformaciones de modelos. La derivación completamente automática de PIMs a partir de un CIM no es posible, dado que la decisión acerca de qué partes del CIM será soportada por un sistema de software debe ser tomada por un humano.

3. Arquitectura Dirigida por Modelos

La Arquitectura Dirigida por Modelos (en inglés Model Driven Architecture, MDA) [MDAG] proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos, siguiendo el proceso MDD. En MDA, la funcionalidad del sistema es definida en primer lugar como un modelo independiente de la plataforma (PIM) a través de un lenguaje específico para el dominio del que se trate. En este punto aparece además un tipo de modelo existente en MDA, no mencionado por MDD, el modelo de definición de la plataforma (en inglés Platform Definition Model, PDM) que especifica el metamodelo de la plataforma destino.

Entonces, dado un PDM correspondiente a una tecnología, el modelo PIM puede traducirse a uno o más modelos específicos de la plataforma (Platform Specific Model, PSM) para la implementación correspondiente, usando diferentes lenguajes específicos del dominio, o lenguajes de propósito general como JAVA, C#, y Python entre otros. MDA está relacionada con múltiples estándares, tales como el Lenguaje Unificado de Modelado (en inglés Unified Modeling Language, UML), el Meta-Object Facility (MOF), XML Metadata Interchange (XMI) [XMI], Enterprise Distributed Object Computing (EDOC), el Software Process Engineering Metamodel (SPEM) y el Common Warehouse Metamodel (CWM) [CWM].

El OMG (en inglés Object Management Group) mantiene la marca registrada sobre MDA, así como sobre varios términos similares incluyendo Model Driven Development

(MDD), Model Driven Application Development, Model Based Application Developtment, Model Based Programming y otros similares. El acrónimo principal que aún no ha sido registrado por el OMG hasta el presente es MDE, que significa Model Driven Software Engineering. A consecuencia de esto, el acrónimo MDE es usado actualmente por la comunidad investigadora internacional cuando se refieren a ideas relacionadas con la ingeniería de modelos sin centrarse exclusivamente en los estándares del OMG.

Los principales objetivos de MDA son:

- Interoperabilidad (independencia de los fabricantes a través de estandarizaciones).
- Portabilidad (independencia de la plataforma) de los sistemas de software.
- Separación del diseño del sistema tanto de la arquitectura como de las tecnologías de construcción, facilitando así que el diseño y la arquitectura puedan ser alterados independientemente.

El diseño alberga los requisitos funcionales (casos de uso, por ejemplo) mientras que la arquitectura proporciona la infraestructura a través de la cual se hacen efectivos los requisitos no funcionales como la escalabilidad, fiabilidad o rendimiento. MDA se asegura que el PIM, el cual representa un diseño conceptual que plasma los requisitos funcionales, sobreviva a los cambios que se produzcan en las tecnologías de fabricación y en las arquitecturas de software. La traducción entre modelos se realiza normalmente utilizando herramientas automatizadas que soportan MDA, algunas de las cuales permiten al usuario definir sus propias transformaciones. Una iniciativa del OMG es la definición de un lenguaje de transformaciones estándar denominado QVT [QVT] que aún no ha sido masivamente adoptado por las herramientas, por lo que la mayoría de ellas aun definen su propio lenguaje de transformación, y sólo algunos de éstos se basan en QVT. Actualmente existe un amplio conjunto de herramientas que brindan soporte para MDA.

3.1. MOF

El lenguaje MOF (en inglés Meta-Object Facility) [MOF] es un estándar del OMG para la ingeniería conducida por modelos. La especificación de MOF proporciona los siguientes conceptos:

- Una definición formal del meta-metamodelo MOF.

- Un conjunto de reglas para el mapeo de metamodelos MOF a interfaces independientes del lenguaje de programación definidos por medio del estándar de CORBA IDL para manejar cualquier tipo de metadato.

- Una jerarquía de interfaces reflexivas para manipular metadatos independientes del metamodelo.

- Un formato XMI para el intercambio de modelo MOF.

Los metamodelos de UML y UML2 con perfiles entre otros están especificados mediante dicho meta-metamodelo.

3.2. Arquitectura de 4 capas

MOF define una arquitectura en la que existen cuatro niveles o capas denominadas M3, M2, M1, M0 que se utilizan para estandarizar los conceptos relacionados al modelado, desde los más abstractos a los más concretos detalladas a continuación:

1. Meta-metamodelo (capa M3)

   La capa de meta-metamodelo (OMG, 2003) define el lenguaje para especificar un metamodelo, siendo el nivel más abstracto, que permite definir metamodelos concretos.

   Dentro del OMG, MOF es el lenguaje estándar de la capa M3 por lo tanto todos los metamodelos de la capa M2, son instancias de MOF. No existe otro meta nivel por encima de MOF, básicamente MOF se define a sí mismo.

2. Metamodelo (capa M2)

   Un metamodelo es una instancia de un meta-metamodelo y significa que cada elemento del metamodelo es una instancia de un elemento del meta-metamodelo. El objetivo de la capa del metamodelo es definir un lenguaje para especificar modelos, ejemplo: UML y OCL.

3. Modelo del usuario (capa M1)

   Un modelo es una instancia de un metamodelo que representa el modelo de un sistema de software. La capa M1 define un lenguaje para describir los dominios semánticos que permiten a los usuarios modelar una variedad de dominios diferentes. Los conceptos del nivel M1 representan categorías de las instancias de M0.

4. Instancias en tiempo de ejecución (capa M0)

   La capa M0 representa las instancias reales de los elementos del modelo en tiempo de ejecución, es decir, entidades físicas que existen en el sistema. Todas estas entidades son instancias pertenecientes a la capa M0.

La figura 1 es una vista general que muestra las cuatro capas de la arquitectura de modelado, indicando las relaciones entre los elementos en los diferentes niveles. Puede verse que en la capa M3 se encuentra el meta-metamodelo MOF, a partir del cual se pueden definir distintos metamodelos en el nivel M2, como UML y OCL. Instancias de estos metamodelos serán los elementos del nivel M1, como modelos UML que a su vez serán los objetos que cobran vida en las corridas del sistema en la capa M0.

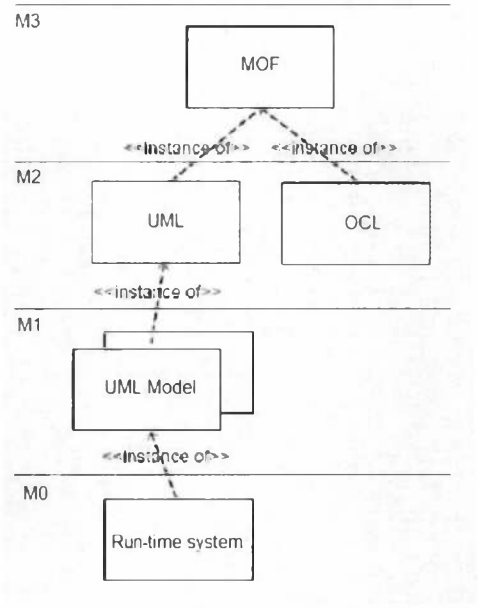


Figura 1 - Arquitectura MOF de 4 capas

La definición de MOF está separada en tres paquetes denominados EMOF (en inglés Essential MOF), CMOF (en inglés Complete MOF) y SMOF (en inglés Semantic MOF). El propósito de los paquetes EMOF y CMOF es utilizar constructores básicos y extenderlos con los elementos necesarios para definir metamodelos simples, en el caso de EMOF y metamodelos más sofisticados, en el caso de CMOF como UML2. El paquete SMOF se encarga de la semántica de dominio para CMOF, opcionalmente para EMOF también, que describe las capacidades funcionales del sistema modelado y la forma en que se relacionan a los elementos del modelo. De acuerdo a cada semántica de dominio describe el modo en que los elementos MOF son inicializados.

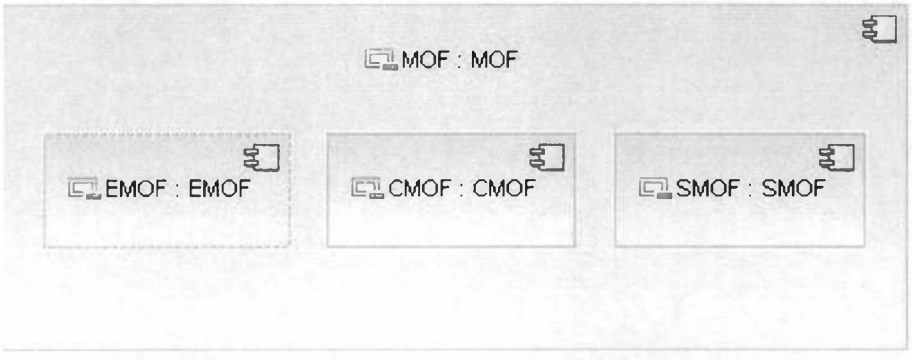
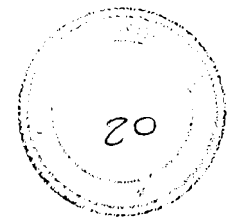


Figura 2 - Paquete MOF y sus subpaquetes



3.3. UML, Perfiles y Estereotipos

Generalmente UML es el lenguaje elegido para definir los modelos en MDD ya que es un estándar abierto y es el estándar de facto para el modelado de software. UML es un lenguaje de modelado de software de propósito general que podemos aplicar de varias formas con una arquitectura que define los siguientes principios de diseño:

- Modularidad
- Separación de Capas
- Flexibilidad
- Extensibilidad
- Reuso

Extendiendo UML mediante estereotipos y perfiles se pueden agregar nuevos conceptos específicos al contexto que estamos modelando.

Los estereotipos [Intro U2TP] permiten extender el vocabulario de UML para crear nuevos elementos del modelo, derivados de unos existentes, pero que tengan propiedades específicas del dominio. Son usados para la clasificación o marcado de bloques de construcción de UML con el fin de introducir nuevos bloques de construcción que hablen el lenguaje del dominio y que puedan mirar elementos primitivos o básicos del modelo. El uso de un estereotipo transmite información semántica adicional que resulta entendible tanto para el modelador humano como para las herramientas automáticas que procesan a los modelos.

Un perfil [Intro U2TP] de UML es un conjunto de extensiones que especializan a UML para su uso en un dominio o contexto particular. Los perfiles están basados en estereotipos adicionales y valores etiquetados que son aplicados a los elementos, atributos y métodos entre otros.

El catálogo de perfiles UML se encuentra disponible en:

http://www.omg.org/technology/documents/profile_catalog.htm

Algunos de los ejemplos que se pueden encontrar allí son los siguientes:

- El perfil para Testing:

  www.omg.org/cgi-bin/doc?formal/05-07-07

- El perfil para Servicios de Software:

  www.ibm.com/developerworks/rational/library/05/419_soa

- El perfil para schedulability, desempeño y tiempo real:

  www.omg.org/technology/documents/formal/schedulability.htm

- El perfil para Objetos Empresariales Distribuidos (EDOC)

  www.omg.org/technology/documents/formal/edoc.htm

3.4. Infraestructura UML 2.0

La Librería de Infraestructura UML (en inglés UML Infrastructure Library) [UML IL] es la especificación más simplificada que define los constructores básicos y conceptos comunes para lenguajes de modelado. El objetivo de la Infraestructura es establecer los siguientes requerimientos de diseño:

- Definir elementos de metalenguaje comunes que se puedan reutilizar definiendo otros metamodelos como pueden serlo MOF, UML y CWM.
- Alinear la arquitectura de UML, MOF y XMI para soportar el intercambio de modelos.
- Permitir el refinamiento de UML a través de los perfiles, creando nuevos lenguajes basados en el mismo metalenguaje principal de UML.

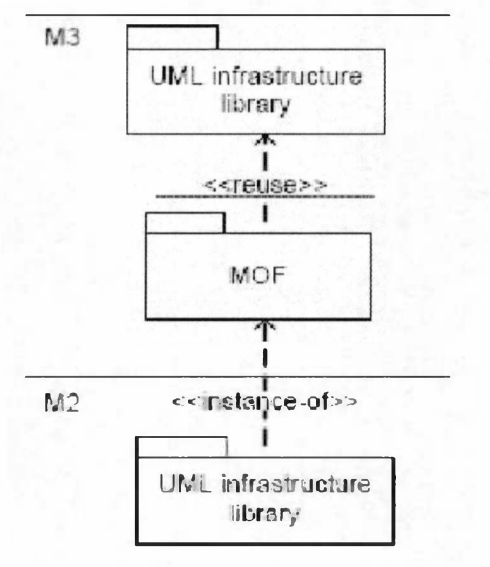


Figura 3 - Dependencias MOF y la librería de infraestructura UML

La jerarquía de paquetes se muestra en la figura 4 (limitada a dos niveles). El paquete Core es considerado el núcleo arquitectural de MDA. La intención es reusar todo o parte de este paquete al definir otros metamodelos, brindando el beneficio de contar con sintaxis abstracta y semántica que ya han sido definidas.

Con el fin de facilitar su reuso, el paquete Core está subdividido en 4 paquetes, a su vez también divididos en paquetes más refinados: *PrimitiveTypes, Abstractions, Basic, Constructors*.

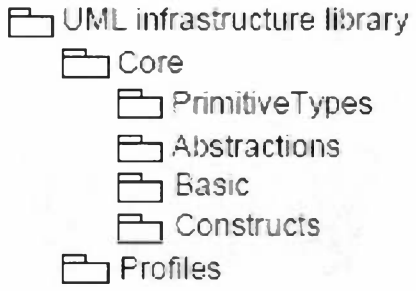


Figura 4 - Árbol de jerarquía de la librería de infraestructura UML

4. Resumen

En este capítulo hemos enunciado los temas fundamentales del Desarrollo de software Dirigido por Modelos. Describimos el rol central de los modelos, mostramos el proceso de construcción de sus distintos niveles, sus características y los beneficios que ofrecen en cada etapa dentro del ciclo de desarrollo del software. Estos tipos de modelos nos permiten construir software que sobreviva a la evolución de sus requisitos funcionales desacoplando además su tecnología de implementación.

La arquitectura de este paradigma soporta la utilización de lenguajes formales para expresar los modelos permitiendo que otras herramientas puedan interpretarlos. UML es el lenguaje de modelado estándar utilizado por la industria del software que brinda editores y notación gráfica para modelar un sistema que facilita la comunicación visual con los usuarios y su interpretación a través de herramientas automáticas de transformaciones. Además, UML nos proporciona un mecanismo flexible para crear nuevos estereotipos y así poder definir perfiles más específicos al dominio que estamos modelando. En los siguientes capítulos vamos a mostrar como todos estos elementos colaboran para lograr el desarrollo de nuestra propuesta permitiendo generar código ejecutable a partir de modelos de pruebas.

PRUEBAS DE SOFTWARE DIRIGIDAS POR MODELOS

En este capítulo presentamos una introducción a la verificación y validación del software enfocados en el testing dirigido por modelos. Luego explicamos los distintos tipos de pruebas que se pueden aplicar para garantizar la calidad del software. A continuación introducimos el perfil de pruebas UML, su arquitectura y como los elementos se utilizan para especificar los modelos de prueba. Por último describimos los frameworks JUnit y EasyMock para la elaboración de pruebas y justificamos su uso dentro del proceso de pruebas.

1. Verificación y validación de software

Los términos verificación y validación se refieren a los procesos de comprobación y análisis que aseguran que el software esté acorde con su especificación y cumpla las necesidades de los clientes [PGP 08].

La validación determina si estamos haciendo el software correcto, por lo tanto sólo se puede hacer con la activa participación del usuario, en cambio la verificación establece si estamos haciendo el software correctamente.

Técnicas de verificación y validación:

- Verificación formal de programas y modelos

  Este es un método de verificación estática (se analiza a través del propio código del programa, a partir de una abstracción o de una representación simbólica), en el que partiendo de un conjunto axiomático, reglas de inferencia y algún lenguaje lógico (como la lógica de primer orden), se puede encontrar una demostración o prueba de corrección de un programa. Entre las herramientas para la verificación formal de programas podemos citar a ESC/Java (Extended Static Checker for Java) [ESC 2], desarrollada por el Compaq Systems Research Center para encontrar errores en tiempo de ejecución de programas Java por análisis estático del texto del programa. Las recientes versiones de ESC/Java están basadas alrededor de Java Modeling Language (JML) [JML 2].

- Chequeo de modelos (en inglés model checking)

  Es un método automático de verificación de un sistema formal, descripto mediante un modelo que debe satisfacer una especificación formal definida mediante una fórmula, a menudo escrita en alguna variedad de lógica temporal. Entre las herramientas de model checking se encuentra Java PathFinder [JPF 2], empleada para el bytecode ejecutable de Java.

- Testing

  El testing es una alternativa práctica de Verificación y Validación de menor complejidad que no requiere tener experiencias en métodos formales. Esta técnica consiste en el proceso de ejercitar un producto para verificar que satisface los requerimientos e identificar diferencias entre el comportamiento real y el comportamiento esperado *(IEEE Standard for Software Test Documentation, 1983)*. El testing no puede garantizar la ausencia de errores, sólo puede asegurar la existencia de errores [DDH 72], su éxito depende fuertemente de la capacidad para crear pruebas útiles para descubrir y eliminar problemas en todas las etapas del desarrollo [Bei 90]; esto requiere diseñarlas sistemáticamente desde la fase de análisis. Los casos de prueba determinan el tipo y alcance de la prueba, dada la complejidad del proceso, su automatización (aunque sea parcial) es indispensable.

En el contexto de MDD, el Model-Driven Testing (MDT) [UL 07] es una de las propuestas más recientes que enfrentan este problema. El MDT es una forma de prueba de caja negra [Bei 95] que utiliza modelos estructurales y de comportamiento descritos por ejemplo, con el lenguaje estándar UML, para automatizar el proceso de generación de casos de prueba.

MDT reduce el costo de mantenimiento de los tests, ya que los desarrolladores sólo tienen que regenerarlos para reflejar los cambios del modelo que afectan a todas las pruebas. Las herramientas de MDT refuerzan la comunicación del equipo porque los modelos proporcionan una vista clara y unificada del sistema y de las pruebas.

2. Pruebas de software

Las pruebas de software [PS] son los procesos que permiten verificar y validar la calidad del mismo permitiendo identificar posibles fallos de implementación. Dichas pruebas se integran durante todo el proceso de desarrollo de software cubriendo los distintos aspectos de la aplicación. Para determinar el nivel de calidad se deben efectuar unas medidas o pruebas [Art Testing] que permitan comprobar el grado de cumplimiento respecto de las especificaciones iniciales del sistema.

2.1. Tipos de pruebas

- Prueba de unidad: es una forma de probar el correcto funcionamiento de un módulo de código. Su objetivo es asegurar que cada uno de los módulos funcione correctamente por separado. La idea es escribir casos de prueba para cada función no trivial o método en el módulo de forma que cada caso sea independiente del resto.

- Prueba de integración: se realizan una vez que se han aprobado las pruebas unitarias, y asegura la correcta interacción, compatibilidad y funcionalidad de las distintas partes que componen un sistema.

- Prueba de validación: es el proceso de revisión que el sistema de software producido cumple con las especificaciones y que cumple su cometido, además de comprobar que lo que se ha especificado es lo que el usuario realmente quería. Dentro de este grupo encontramos la siguiente sub-clasificación:

    - Prueba de aceptación: realizadas por el cliente para verificar si el sistema cumple con sus requerimientos con el fin de determinar si acepta la aplicación.

    - Pruebas alfa: realizadas por el usuario con el desarrollador como observador en un entorno controlado.

    - Pruebas beta: realizadas por el usuario en su entorno de trabajo y sin observadores.

    - Prueba funcional: se basa en la ejecución, revisión y retroalimentación de las funcionalidades previamente diseñadas para el software. Las pruebas funcionales se hacen mediante el diseño de modelos de prueba que buscan evaluar cada una de las opciones con las que cuenta el paquete informático.

- Caja Blanca: se realiza sobre las funciones internas de un módulo. Las pruebas de caja blanca se llevan a cabo en primer lugar, sobre un módulo concreto, para luego realizar las de caja negra sobre varios subsistemas (integración).

- Caja Negra: verifica el software a partir de las entradas que recibe y las salidas o respuestas que produce, sin tener en cuenta su funcionamiento interno. Se interesa en **qué es lo que hace**, pero sin dar importancia a **cómo lo hace**.

- Prueba no funcional: valida los requerimientos no funcionales para la arquitectura designada, como seguridad, robustez, instalación o como el sistema se recupera de las caídas, fallas del hardware, u otros problemas. Debajo mostramos una clasificación de los tipos de pruebas funcionales que encontramos:

    - Prueba de carga: se ocupa de verificar las métricas de rendimiento (tales como tiempo de respuesta y flujo de datos) y escalabilidad del sistema en la carga. Determina hasta donde puede soportar el sistema determinadas condiciones extremas.

    - Prueba GUI o Usabilidad: prueba que la interfaz de usuario (GUI) sea intuitiva, amigable, accesible y funcione correctamente. Determina la calidad de la experiencia de un usuario en la forma en la que se interactúa con el sistema.

- Prueba de regresión: prueba la aplicación en conjunto, debido a cambios (adición o modificación) producidos en cualquier módulo o funcionalidad del sistema y puede implicar la re-ejecución de otros tipos de prueba. Generalmente, las pruebas de regresión se llevan a cabo durante cada interacción, ejecutando otra vez las pruebas de la interacción anterior con el objetivo de asegurar que:

    1. Los defectos identificados en la ejecución anterior de la prueba se hayan corregido.

2. Los cambios realizados no hayan introducido nuevos defectos o re-introducido defectos anteriores.

3. El Perfil de Pruebas UML

El Perfil de Pruebas UML (en inglés UML 2.0 Testing Profile, U2TP) [U2TP 04] es un lenguaje que se ha implementado utilizando el mecanismo de perfiles de UML 2.0. Este perfil proporciona un marco formal para la definición de un modelo de prueba bajo el acercamiento de caja negra [Bei 95]. Aplicando este enfoque las pruebas se derivan de la especificación del artefacto que se somete a la verificación, donde este es una "caja negra" cuyo comportamiento sólo se puede determinar estudiando sus entradas y salidas.

Al ser un perfil basado en UML hereda las mismas características:

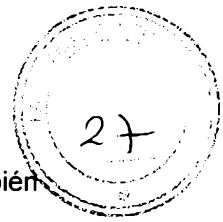
- División de capas, estableciendo una arquitectura de metamodelo de 4 niveles.
- Particionamiento al utilizar paquetes para refinar las estructuras.
- Extensibilidad que permite adaptar los perfiles de pruebas a plataformas tecnológicas y dominios específicos.

El perfil se organiza en cuatro grupos lógicos que cubren los aspectos de pruebas de: *arquitectura (*en inglés *test architecture*), *comportamiento* (en inglés *test behavior*), *datos* (en inglés *test data*) y *tiempo (*en inglés *test time*). Estos conceptos definen el lenguaje de modelado para diseñar, visualizar, especificar, analizar, construir y documentar los artefactos de los sistemas de pruebas. En la siguiente sección, introducimos los conceptos principales de U2TP (tabla 1) [MDT-Dai].

3.1. Prueba de Arquitectura

La Prueba de Arquitectura define un conjunto de conceptos agregados a los estructurales de UML 2.0 que permiten especificar los aspectos estructurales de un contexto de pruebas cubriendo los componentes y el sistema bajo pruebas, su configuración, los elementos y sus relaciones establecidas en una prueba.

Uno o más objetos pueden identificarse como Sistema Bajo Pruebas (en inglés *System Under Test, SUT*). Los *Componentes de Pruebas* (en inglés *Test Component*) son objetos dentro de un sistema de pruebas que pueden comunicarse con el *SUT* u otros componentes para verificar el comportamiento de las pruebas. El *Contexto de Pruebas* (en inglés *Test Context*) permite a los usuarios agrupar los casos de pruebas (en inglés *test cases*), y describir su correspondiente *Configuración* (en inglés *Test Configuration*) por ejemplo para establecer la conexión entre los componentes de pruebas y el SUT. Además define el Control de Pruebas (en inglés Test Control) que posibilita definir el orden de ejecución de los casos de pruebas. El *Árbitro* (en inglés *Arbiter*) evalúa el resultado del veredicto final de un contexto de prueba. Los valores definidos son *Pass (Pasa)*, *Fail (Falla)*, *Inconclusive (Incierto)* y *Error* en ese orden, aunque se pueden redefinir implementando la interface *Arbiter*. El *Scheduler* controla la ejecución de la prueba y los componentes de pruebas; mantiene la información acerca de qué componentes existen en todo momento y es



responsable de la creación de los componentes de pruebas, como así también de que inicien y terminen en forma sincronizada.

3.2. Prueba de Comportamiento

Adicionalmente a los conceptos de comportamiento descriptos en UML 2.0, el Perfil de Pruebas agrega nuevos conceptos para especificar el comportamiento de las pruebas, sus objetivos y la evaluación de los SUT.

El Objetivo de Prueba (en inglés *Test Objective*) es un elemento que describe lo que se debería probar y está asociado al Caso de Prueba (en inglés *test case*). Los diagramas UML de comportamiento como por ejemplo el diagrama de máquinas de estado o secuencia pueden utilizarlo para definir un Estímulo de Prueba (en inglés *Test Stimuli*), observaciones, controles, invocaciones, coordinación y acciones de pruebas. La prueba de comportamiento se especifica en un caso de prueba, que es una operación del contexto de prueba especificando cómo el conjunto de componentes interactúan con el SUT para la prueba. La Acción de Validación (en inglés *validation action*) se ejecuta por un componente de prueba local que informa al Arbiter sobre el veredicto final. El veredicto (en inglés *Verdict*) de prueba muestra el resultado de la ejecución de la prueba, los posibles resultados son: *pasa* (en inglés pass), *incierto* (en inglés *inconclusive*), *falla* (en inglés *fail*) y *error*.

3.3. Prueba de Datos

Las pruebas de datos utilizan símbolos especiales (en inglés *wildcards*) para manejar eventos inesperados, o eventos con distintos valores. Existen tres tipos de estos elementos: los que se refieren a cualquier valor, los que especifican un valor o ninguno y aquellos para los que se omiten los valores. Los Pool de Datos (en inglés *Data Pool*) se utilizan con los contextos o componente de prueba. Los selectores de datos (en inglés Data Selectors) son operaciones para obtener datos de las pruebas desde los data pool o particiones de datos (en inglés data partitions). Las reglas de codificación (en inglés *coding rules*) permiten definir la codificación y decodificación de los datos de pruebas cuando se comunica con el SUT.

3.4. Prueba de Tiempo

Los conceptos de tiempo son esenciales para proveer una descripción precisa y completa al especificar casos de pruebas. Los conceptos de tiempo ya descriptos por UML 2.0 no cubren todos los requerimientos para especificarlos, por lo tanto el Perfil de Pruebas agrega un conjunto adicional de conceptos de tiempo que ayudan a describir restricciones y observaciones de tiempo, además de cronómetros (en inglés timers) para cuantificar la ejecución de las pruebas. Los cronómetros son necesarios para manipular y controlar el comportamiento de las pruebas además de utilizarse para asegurar la finalización de los casos de pruebas. Los husos horarios (en inglés time zones) agrupan y sincronizan los componentes de pruebas dentro de un sistema distribuido donde cada uno de

ellos pertenece a determinado grupo de tiempo permitiendo comparar los eventos de tiempo dentro del mismo huso horario.

Prueba de arquitectura **(test architecture)**	Prueba de comportamiento **(test behavior)**	Prueba de datos **(test data)**	Prueba de tiempo **(test time)**
SUT	Test objective	Wildcards	Timer
Test component	Test case	Data pool	Time zone
Test context	Defaults	Data partition	
Test configuration	Validation action	Data selector	
Test control	Verdicts	Coding rules	
Arbiter			
Scheduler			

Tabla 1 – Conceptos de la arquitectura U2TP

4. El framework de testing JUnit

En los últimos años se han desarrollado un conjunto de frameworks denominados XUnit que facilitan la elaboración de pruebas unitarias en diferentes lenguajes. La familia XUnit se compone, entre otros, de las siguientes herramientas:

- SUnit (sunit.sourceforge.net) para Smalltalk.

- JUnit (www.junit.org), TestNG (testng.org), JTest (www.parasoft.com), JExample (www.ohloh.net/p/jexample) son entornos de pruebas Java.

- CPPUnit (sourceforge.net/apps/mediawiki/cppunit) para C++.
- NUnit (www.nunit.org) para la plataforma .NET y Mono.
- Dunit (dunit.sourceforge.net) para Borland Delphi.
- PHPUnit (www.phpunit.de) para aplicaciones realizadas en PHP.

Estos frameworks resultan muy útiles para controlar las pruebas de regresión, validando que el nuevo código cumple con los requisitos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

JUnit es un framework de código abierto creado por Erich Gamma y Kent Beck que se utiliza para escribir y ejecutar los tests de aplicaciones escritas en Java. Este framework se ha popularizado por la comunidad de eXtreme Programming [XP] y es ampliamente utilizado por los desarrolladores que implementan casos de pruebas unitarias en Java. En los últimos años, se convirtió en el estándar de facto de los tests unitarios.

JUnit incluye formas para presentar los resultados de las pruebas que pueden ser en modo texto, gráfico (AWT o Swing) o como tarea en Ant [Ant]. Las pruebas se representan en JUnit como clases Java y pueden definir cualquier cantidad de métodos públicos de prueba. Por convención, los métodos utilizan el prefijo "test" delante del nombre del caso de la prueba, con el fin de que JUnit pueda identificarlos y ejecutarlos; los métodos de "test" no tienen parámetros. JUnit provee una librería (Assert) para poder comprobar el resultado esperado con el real. La prueba falla si la condición de la aserción no se cumple, generando un informe del fallo.

Los beneficios principales del framework JUnit se listan a continuación:

- Permite escribir y correr tests en forma automática y de manera repetitiva para validar la integridad del código.

- Código abierto.

- Incrementa la calidad y la estabilidad del software: con el uso de JUnit se incrementa la confianza en que los cambios del código realmente funcionan.

- JUnit chequea y muestra en forma visual los resultados en forma inmediata.

- Permite crear suites de tests.

Para crear un test con JUnit 3.x debemos seguir estos pasos:

- Crear una clase que herede de la clase *TestCase*

  public class SimpleTest extends TestCase

- Opcionalmente podemos crear un constructor con un parámetro de tipo String e invocar al constructor de la clase padre pasándole dicho String.

  public SimpleTest (String nombre) {

  super(nombre);

  }

- En cada clase de prueba se pueden sobrescribir opcionalmente los métodos setUp() y tearDown():
  - **setUp()** se ejecuta antes de cada caso de prueba, método "test", y se utiliza para inicializar el entorno de pruebas. Ejemplo: inicializar variables de instancias, establecer conexiones con la base de datos.
  - **tearDown()** se llama después de la ejecución de cada caso de prueba con el objetivo de liberar determinados recursos del sistema, como los recursos u objetos inicializados en el método setUp().

- Crear uno o varios métodos públicos que empiecen con "test", sin argumentos ni retorno de resultado.

  public void testMethodName() {

  // Código del caso de prueba.

  }

Aquí se listan las principales diferencias entre las versiones 3.x versus 4.5 o superior:

- Incluye anotaciones Java (Java 5 annotations):
  - @Test sustituye a la herencia de la clase *TestCase*.
  - @Before y @After como sustitutos de *setUp()* y *tearDown()*.
  - Se añade @Ignore para deshabilitar tests.

- Se elimina la distinción entre errores (errors) y fallos (failures).

La herramienta de transformación de modelos a código de pruebas Java basados en el framework JUnit está implementada sobre JUnit versión 3.x, no teniendo en cuenta las nuevas características introducidas en las versiones superiores listadas en la sección anterior. De todas maneras, las nuevas características de las versiones superiores se pueden agregar a la aplicación de transformación muy fácilmente.

5. EasyMock

EasyMock [EasyMock] es una librería de código abierto que extiende al framework JUnit permitiendo crear Objetos Fantasmas (en inglés Mock Objects) para simular parte del comportamiento del código de dominio. Los objetos fantasmas los crea dinámicamente, permitiendo además que devuelvan un resultado concreto para una entrada concreta.

EasyMock presenta las siguientes ventajas:

- Evita escribir manualmente los objetos que simulan determinados comportamientos para cada caso de prueba.
- Aumenta la confianza al realizar cambios en los nombres de métodos o reordenamiento de parámetros.
- Soporta el retorno de valores y excepciones.
- Soporta la ejecución de métodos en un orden dado para uno o más objetos mocks.

La restricción de EasyMock 2 es que únicamente trabaja con Java 2 versiones 5 o superior.

EasyMock soporta únicamente la creación por defecto de objetos fantasmas a partir de sus interfaces. De todas maneras, EasyMock Class Extension derivado de EasyMock genera objetos mocks para interfaces y clases.

En términos generales, estos son los pasos para usar EasyMock:

1. Crear el objeto *mock* a simular a partir de su interface o clase (utilizando EasyMock Class Extension):

   ```
   UserDaoImpl mockUserDao = EasyMock.createMock(UserDaoImpl.class);
   ```

2. Si se necesita grabar el comportamiento esperado se puede proporcionar los parámetros de entrada concretos y los objetos de salida concretos con *expect*:

   ```
   EasyMock.expect(mockUserDao.selectByLogin(
           EasyMock.anyObject(), EasyMock.anyObject())).
                   andReturn(EasyMock.anyObject());
   ```

3. Generar la implementación del *mock* con *replay*:

   ```
   EasyMock.replay(mockUserDao);
   ```

4. Si se especificó comportamiento, se podría verificar que el método del *mock* fue ejecutado con *verify*:

```
EasyMock.verify(mockUserDao);
```

De esta manera tenemos un objeto mock (fantasma) para el objeto UserDaoImpl. Ahora cuando se llame al método validateLogin de la clase LoginAuthenticator, este ejecutará el método selectByLogin del mock creado previamente:

```
LoginAuthenticator loginAuthenticator = new LoginAuthenticator();

loginAuthenticator.validateLogin(username, password);
```

Estos son algunas de las alternativas y agregados a EasyMock:

- Powermock (code.google.com/p/powermock) basado en EasyMock permite agregar comportamiento mock a métodos estáticos, constructores, clases y métodos finales y privados.
- EasyMock Property Utilities (www.stephenduncanjr.com/projects/easymock-propertyutils/index.html) es un agregado a EasyMock que permite usar propiedades del estilo JavaBeans para los argumentos cuando se utiliza EasyMock.
- Jmock (www.jmock.org) es otro framework para generar objetos fantasmas en forma dinámica. Su principal debilidad es la poca flexibilidad para readaptarse a los cambios en el código, pudiendo romper casos de pruebas anteriores.
- Mockito (www.mockito.org) muy similar a EasyMock aunque los pasos de uso son algo distintos. Mockito no soporta el concepto de configurar un comportamiento por lo tanto no existen las verificaciones de los objetos mocks.
- Hamcrest (code.google.com/p/hamcrest) es una librería que se puede utilizar con EasyMock.

6. Resumen

En este capítulo hemos descrito las distintas alternativas acerca de cómo verificar y validar el software desde la perspectiva del testing dirigido por modelos. En especial hemos mencionado el testing como una de las técnicas prácticas de verificación y validación aplicable sobre los modelos de pruebas de software. Esta técnica de prueba de caja negra permite identificar diferencias entre el comportamiento real y el esperado donde su proceso no requiere conocimientos sobre especificaciones formales.

A continuación hemos introducido el perfil de pruebas UML que utilizamos para modelar casos de pruebas. Dicho perfil nos brinda el marco formal para la definición de un modelo de pruebas que podemos aplicar sobre diagramas estructurales y de comportamiento permitiendo automatizar el proceso de generación de código ejecutable. Además, la especificación del perfil brinda las reglas de transformación para automatizar la generación de casos de pruebas. Es por eso que presentamos los frameworks JUnit y EasyMock, éste último trabaja como una extensión del primero posibilitando simular comportamiento creando objetos fantasmas. Ambos son frameworks de código abierto utilizados para crear casos de pruebas de unidad

basados en Java que permiten automatizar el proceso de testing utilizando el perfil de pruebas UML.

TRANSFORMANDO MODELOS DE PRUEBA A CÓDIGO

En este capítulo presentamos cómo transformar los modelos de pruebas a código JUnit. Inicialmente introducimos los pasos para especificar un perfil de pruebas. Luego mostramos el mapeo para convertir los estereotipos U2TP utilizados en la especificación del lenguaje a código JUnit. Explicamos qué tecnologías adicionales se pueden utilizar para traducir los componentes no soportados por el framework JUnit y el comportamiento dinámico de los métodos. Finalmente, describimos las reglas para transformar los modelos de prueba estructurales y de comportamiento a código ejecutable JUnit.

1. Cómo especificar un Perfil de Pruebas UML

A continuación se describen los pasos necesarios para especificar un Perfil de Pruebas UML:

- Brindar la terminología para un entendimiento básico de los conceptos del Perfil de Pruebas UML.
- Definir el metamodelo base UML 2.0 del Perfil de Pruebas UML.
- Definir un modelo MOF para el perfil de Pruebas UML habilitando el uso del Perfil de Testing independiente de UML.
- Dar ejemplos que demuestren el uso del Perfil de Pruebas UML para el nivel de componente y sistema de pruebas.
- Establecer un mapeo del Perfil de Pruebas UML al ambiente de ejecución de pruebas que se quiere implementar, como por ejemplo JUnit.

El Perfil de Pruebas UML puede aplicarse a los diagramas estructurales y de comportamiento que representan el sistema.

Los diagramas estructurales permiten describir la arquitectura y los datos de la prueba. La arquitectura define los conceptos relacionados con la estructura y configuración de las pruebas, es decir los elementos (y sus relaciones) que participan en esta. Los datos proporcionan las estructuras y la semántica de los valores procesados.

Los diagramas de comportamiento expresan conceptos relacionados con los aspectos dinámicos describiendo la manera en que colaboran grupos de objetos para un determinado caso de prueba. Además pueden mostrar restricciones de tiempo, que resultan útiles para especificarlas de manera precisa y completa.

El trabajo de tesis desarrolla el Perfil de Pruebas UML necesario para escribir diagramas estructurales y de comportamiento con el objetivo final de convertir los modelos creados con dicho perfil a código de prueba soportado por el framework JUnit.

2. Mapeo del Perfil de Pruebas UML a JUnit

En esta sección se describe el mapeo del Perfil de Pruebas UML a JUnit. Este mapeo considera primariamente al framework JUnit, y en los casos donde no exista una asociación hacia JUnit, se mencionan ejemplos de cómo el framework tiene que extenderse para soportar los conceptos incluidos en el Perfil de Testing UML.

El framework EasyMock se puede utilizar como mecanismo de extensión a JUnit, por ejemplo para traducir el estereotipo de prueba ***<<TestComponent>>***. Dado que el framework EasyMock fue seleccionado para extender algunos conceptos no soportados por JUnit, es que en la siguiente sección se introducirán sus conceptos y la forma de crear los objetos fantasmas que ayuden a generar un código de prueba más completo.

UML Testing Profile	JUnit
Test Behavior:	
Test Control	Sobrecargar el método "runTest" de JUnit TestCase. Con esto se especifica la secuencia de ejecución de los Test Cases.
Test Case	El Test Case se representa con una operación en JUnit. Este método público pertenece a la clase del Contexto de Prueba (Test Context), por convención comienza con el prefijo "test" y no tiene argumentos para no utilizar el Control de Prueba (Test Control).
Test Invocation	Un Test Invocation es una operación que se puede llamar desde otra operación Test Case o desde el Control de Prueba (Test Control).
Test Objective	Se puede usar haciendo una llamada al método "setName" del framework de pruebas.
Stimulus	No existe un concepto similar aplicable en JUnit. Los Estímulos son parte directa de la implementación de los Test Cases (métodos de pruebas).
Observation	No existe un concepto similar aplicable en JUnit. Las Observaciones son parte directa de la implementación de los Test Cases (métodos de pruebas).
Coordination	Puede implementarse a partir de cualquier mecanismo de sincronización disponible para los Componentes de Pruebas. Ejemplo utilizando semáforos.
Default	No existe un concepto similar aplicable en

UML Testing Profile	JUnit
	JUnit. El mecanismo de excepciones en Java se podría utilizar para implementar la jerarquía default del Perfil de Testing.
Verdict	En JUnit, los veredictos puede ser: *pass* (pasa), *fail* (falla), *error*. No existe un veredicto *inconclusive* en JUnit, por lo que se considera como un veredicto *fail*. Este último valor ya no existe en JUnit versión 4 o superior.
Validation Action	Se consideran las llamadas a la librería Assert de JUnit.
Log Action	No existe un mecanismo de log en JUnit.
Test Log	No existe un mecanismo de log en JUnit.
Test Architecture:	
Test Context	El Contexto de Prueba se representa a través de una clase que hereda de la clase TestCase de JUnit versión 3.8 (o bien utiliza la anotación @Test en versiones superiores).
Test Configuration	No existe un concepto similar aplicable en JUnit.
Test Component	No existe un Componente de Pruebas en JUnit. Para simularlo se utilizan extensiones a JUnit como Mock Objects (objetos fantasmas) como los mencionados anteriormente. En nuestra propuesta, seleccionamos EasyMock framework para complementar esta característica.
System Under Test (SUT)	El Sistema Bajo Prueba no necesita traducirse específicamente a JUnit ya que cualquier clase del sistema puede considerarse como una clase utilitaria o una clase SUT.
Arbiter	Se puede considerar como una propiedad del Test Context de un tipo de resultado de la prueba (TestResult). El algoritmo por defecto genera los valores *Pass*, *Fail* y *Error* similar a un Veredicto, aunque esos valores se pueden redefinir por el usuario.
Scheduler	Se puede representar como una propiedad del Test Context.

UML Testing Profile	JUnit
Utility Part	Cualquier clase disponible en el classpath de Java se puede considerar como una clase utilitaria o parte del SUT.
Test Data:	
Wildcards	No existe un concepto similar aplicable en JUnit.
Data pool	Agrupa todas las operaciones de acceso al pool de datos en una clase.
Data partition	Clase que hereda del Data Pool con métodos para acceder a la partición de datos.
Data Selector	Es un método del Data Pool o Data Partition.
Coding Rules	No existe un concepto similar aplicable en JUnit.
Time Concepts:	
Time zone Timer	JUnit no soporta conceptos de tiempo, aunque se podrían implementar a través de las APIs estándares disponibles para manipular el tiempo.
Test Deployment:	
Test Artifact Test Node	El despliegue está fuera del alcance de JUnit.

Tabla 2 - U2TP vs JUnit

Para ejemplificar su uso, vamos a describir una funcionalidad relacionada a la validación y autenticación de un usuario en un sistema foros que ayude a entender cómo interactúan los estereotipos agregados por nuestro perfil en la creación de un diagrama estructural. Los diagramas de comportamiento que definen conceptos relacionados con los aspectos dinámicos de la prueba serán evaluados y detallados más adelante.

La figura 5 muestra la especificación de la clase LoginAuthenticator del paquete ***forums*** y la clase de test LoginAuthenticator del paquete ***forumstest*** que prueba la funcionalidad del método *validateLogin*.

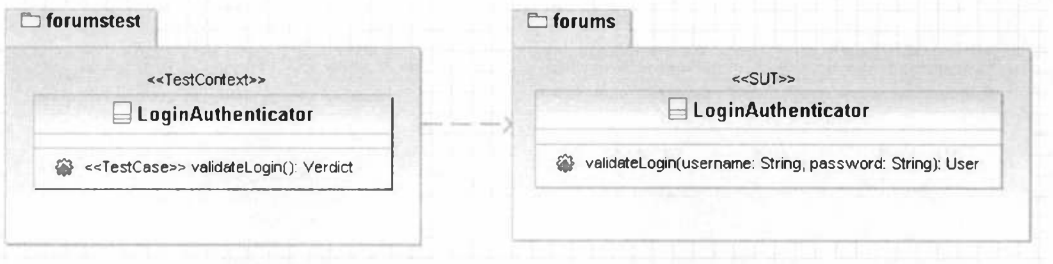


Figura 5 - Diagrama de clases de la autenticación de usuario

El objetivo de la prueba consiste en verificar las siguientes condiciones:

- ✓ El usuario esté registrado en el sistema.
- ✓ La clave proporcionada coincida con la identidad ingresada.

El diagrama de la figura 6 ilustra los paquetes ***forums*** y ***forumstest*** que agrupan las clases del sistema y de las pruebas respectivamente. La dependencia indica que hay clases del paquete ***forumstest*** que dependen de clases del paquete ***forums***. El diagrama de clases describe los objetos que participan de la prueba y las relaciones que existen entre ellos. Podemos ver el uso de los estereotipos del perfil de testing sobre operaciones, clases e interfaces UML que facilitan la comunicación y ayudan a comprender el diagrama de manera más clara. La figura 6 muestra también el diagrama de clases que representa el modelo completo donde aparecen los siguientes nuevos estereotipos aplicados a clases, interfaces, y métodos:

- **<<TestContext>>**
- **<<SUT>>**
- **<<TestCase>>**
- **<<Verdict>>**

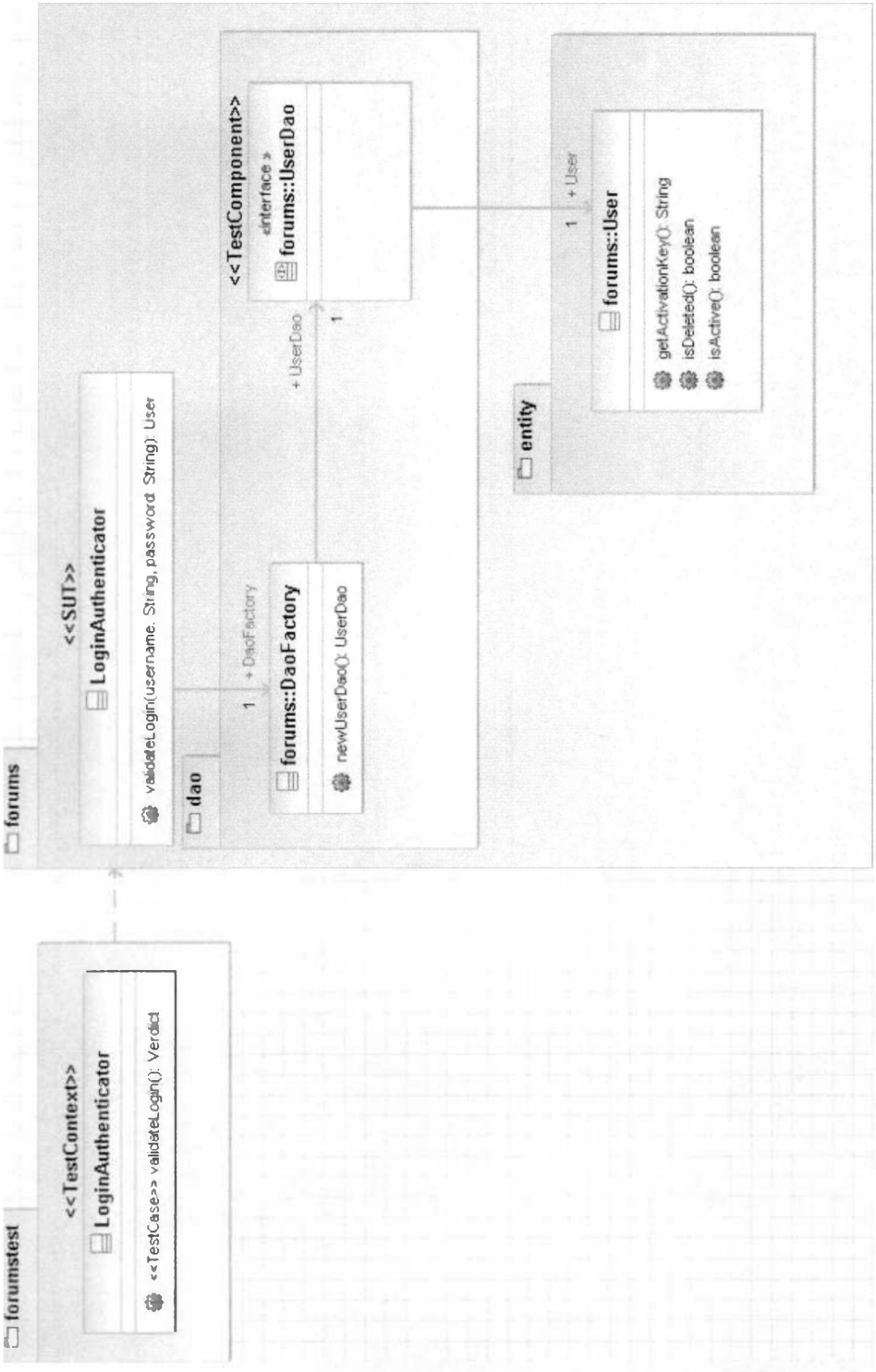


Figura 6 - Modelo de clases completo del modelo de prueba

El estereotipo ***<<TestContext>>*** (contexto de prueba) puede agrupar un conjunto de casos de prueba (test cases). En el ejemplo está aplicado a la clase LoginAuthenticator del paquete ***forumstest*** que contiene los casos de pruebas.

El estereotipo ***<<SUT>>*** es un acrónimo de System Under Test (Sistema Bajo Prueba), y puede consistir de varios objetos. En el ejemplo está aplicado a la case LoginAuthenticator del paquete ***forums***, y se utiliza para representar al sistema, subsistema, o componente que se somete a la prueba. Los diagramas de clases también muestran los casos de prueba efectuados sobre la aplicación (SUT).

En el ejemplo, la clase LoginAuthenticator del paquete ***forumstest*** de pruebas tiene una operación con el estereotipo ***<<TestCase>>***, indicando el caso de prueba que se debe verificar. Estos tienen acceso a todos los elementos del contexto de la prueba, incluso a los elementos del SUT. Estas operaciones retornan un veredicto (Verdict), los posibles resultados son:

- Pasa (pass): el sistema bajo la prueba cumple las expectativas.
- Incierto (inconclusive): no fue posible determinar un resultado.
- Falla (Fail): diferencia entre los resultados esperados y reales.
- Error: un error en el ambiente de prueba.

3. Reglas para transformar modelos de prueba a código JUnit

Aquí se describen las reglas de transformación para generar el código JUnit a partir del modelo de prueba escrito con el Perfil de Testing UML [PGP 08]. Estas reglas son bien conocidas [U2TP 04] y ya se detallaron anteriormente al indicar su correspondencia entre el Perfil de Pruebas UML y JUnit.

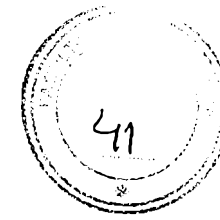
3.1. Reglas para generar el esqueleto del código de pruebas

Traduciendo los conceptos estructurales:

• Las clases raíces con el estereotipo **<<TestContext>>** (contexto de prueba) son traducidas como clases que heredan de **TestCase** en JUnit 3.8, para versiones superiores se utiliza la anotación **@Test**.

• Las clases no raíces con el estereotipo **<<TestContext>>** son traducidas como clases respetando la jerarquía establecida en el modelo de prueba.

• Las operaciones con el estereotipo **<<TestCase>>** son traducidas como operaciones de la clase que representa al contexto de prueba. Por convención, los nombres de estas operaciones deben comenzar con "test" y no deben tener parámetros; los posibles resultados son:



- Pasa (pass): el sistema bajo la prueba cumple las expectativas.
- Falla (Fail): diferencia entre los resultados esperados y reales.
- Error: un error (excepción) en el ambiente de prueba.

En JUnit no hay un veredicto incierto (inconclusive), por lo general éste es traducido como una falla (fail).

• Las operaciones sin estereotipos son traducidas como operaciones Java de la clase correspondiente.

• Las clases con el estereotipo **<<TestComponent>>** no pueden ser traducidas directamente a JUnit ya que éste no maneja este concepto. Sin embargo, se utilizará el framework EasyMock y EasyMock Class Extension como extensiones a JUnit para crear los componentes de prueba.

• Las clases con el estereotipo **<<SUT>>** (Sistema Bajo Prueba) no necesitan traducirse; cualquier clase en el classpath puede considerarse como una clase de utilidad o una clase de SUT. Con la herramienta también vamos a generar el código adicional que acompaña al modelado de pruebas.

A continuación se muestra el código resultante del modelo de la figura 7:

```
public class LoginAuthenticatorTest extends TestCase {

   /**
    * @param name
    */
   public LoginAuthenticatorTest(String name) {
     super(name);
   }

   /* (non-Javadoc)
    * @see junit.framework.TestCase#setUp()
    */
   @Override
   protected void setUp() throws Exception {
     super.setUp();
   }

   /* (non-Javadoc)
    * @see junit.framework.TestCase#tearDown()
    */
   @Override
   protected void tearDown() throws Exception {
     super.tearDown();
   }

   public void testValidateLogin() throws Exception {

   }
}
```

3.2. Reglas para generar comportamiento a los métodos de prueba

La traducción de los conceptos dinámicos de los modelos de pruebas expresados en los diagramas de secuencia por lo general no se puede automatizar en su totalidad por las herramientas de transformación de modelos a código. Por el contrario, requiere la intervención del programador para completar el código obtenido.

Para traducir y determinar el comportamiento para el caso de prueba *testValidateLogin* vamos a escribir el diagrama de secuencia del modelo de prueba que tomaremos como entrada en el proceso de transformación.

En el diagrama se muestra la interacción completa de todos los objetos. La instancia UserDao muestra el estereotipo ***<<TestComponent>>*** que JUnit no puede traducir por sí mismo. Por lo tanto, la utilización de una extensión al framework es utilizada para completar el comportamiento de la prueba. Para eso vamos a utilizar EasyMock y EasyMock Class Extension.

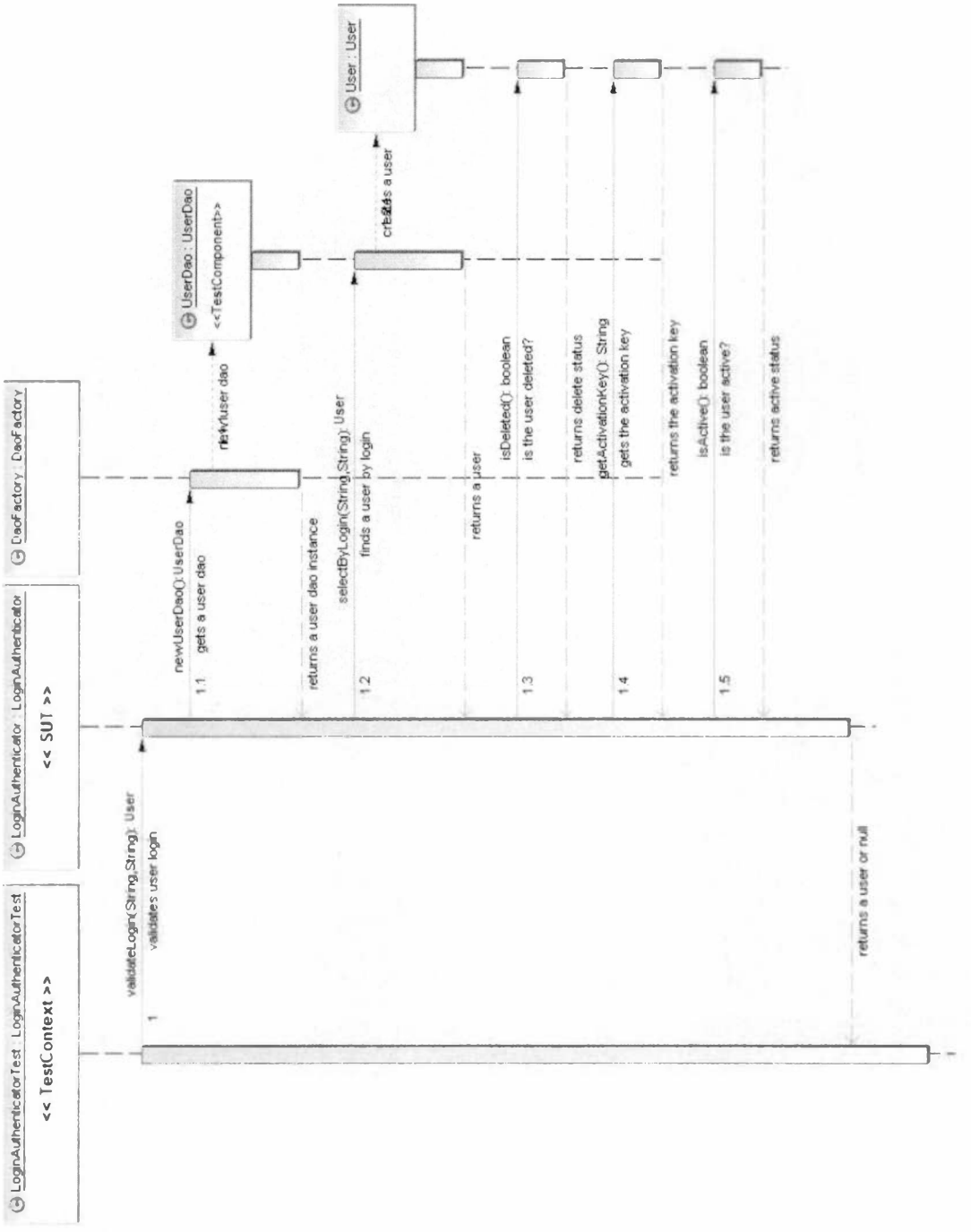


Figura 7 - Diagrama de secuencia del caso de pruebas de autenticación de usuario

Traducción del comportamiento del método *testValidateLogin*:

```
public void testValidateLogin() throws Exception {
  UserDaoImpl mockUserDao = EasyMock.createMock(UserDaoImpl.class);

  EasyMock.expect(mockUserDao.selectByLogin(EasyMock.anyObject(),
      EasyMock.anyObject())).andReturn(EasyMock.anyObject());

  String username = null;
  String password = null;

  LoginAuthenticator loginAuthenticator = new LoginAuthenticator();
  assertEquals("Verify the expected value",
   null, loginAuthenticator.validateLogin(username,password));

  EasyMock.replay(mockUserDao);
  EasyMock.verify(mockUserDao);
}
```

En el próximo capítulo se agregan nuevos ejemplos de modelos de prueba estructurales y de comportamiento que muestran cómo la herramienta desarrollada genera el código JUnit especificado en los diagramas.

4. Resumen

En este capítulo hemos mostrado cómo especificar un perfil de pruebas UML, los pasos para la creación de los metamodelos necesarios que permitan diseñar los casos de pruebas y como finalmente obtener el código ejecutable. Analizamos el mapeo correspondiente de todos los conceptos del perfil de pruebas UML al ambiente de ejecución de pruebas JUnit que se quiere implementar. Identificamos también que definiciones exceden la posibilidad de traducción por el framework JUnit e indicamos las alternativas para suplir dichas carencias con EasyMock que agregan comportamiento y aumentan significativamente el resultado final de las transformaciones obtenidas en forma automática. Por último, especificamos las reglas de transformación que se deben aplicar para traducir un modelo de pruebas a código JUnit, mostrando cómo es posible la automatización del esqueleto y el comportamiento asociado del caso de prueba.

IMPLEMENTACIÓN DE LA PROPUESTA

El paradigma MDA se puede aplicar tanto a los modelos de sistemas como de pruebas. Para este último OMG extendió a partir de marzo de 2004 el Lenguaje de Modelado Unificado (UML) creando un estándar que brinda el soporte necesario para modelar los distintos niveles de abstracciones denominado Perfil de Pruebas UML (U2TP). Este lenguaje permite modelar pruebas bajo el concepto de caja negra aplicable a modelos estructurales y de comportamiento.

En este capítulo introducimos la herramienta propuesta que permite automatizar la generación de código de prueba JUnit con Easy Mock a partir de recibir un modelo de pruebas especificado en UML y con los estereotipos U2TP. Para lograrlo, primero definimos el lenguaje de pruebas basado en el Perfil de Pruebas UML. Luego mostramos unos modelos de ejemplo especificados con los conceptos creados con U2TP y cómo se hacen las transformaciones utilizando las reglas especificadas anteriormente desde el PIM directamente a código de prueba ejecutable sin antes generar el PSM. Los diagramas de clases y secuencia, estructural y de comportamiento respectivamente, están soportados por la herramienta de transformación desarrollada. Esta aplicación está construida bajo la arquitectura plug-in de Eclipse utilizando el framework EMF y el lenguaje de transformación de modelos a texto denominado MOFScript. En las secciones siguientes explicamos el uso del proyecto Eclipse Modeling Framework y su importancia para el desarrollo de herramientas compatibles con MOF, además del lenguaje MOFScript utilizado para hacer las transformaciones de los modelos a código y cómo fueron utilizadas para la construcción de la herramienta. Por último, ilustramos la arquitectura de la implementación describiendo cómo utilizamos cada componente en el proceso de generación de código de prueba ejecutable a partir de los modelos de pruebas.

La figura 8 muestra la arquitectura general de Eclipse e indica cómo los plug-ins se agregan al framework para agregar nuevas características.

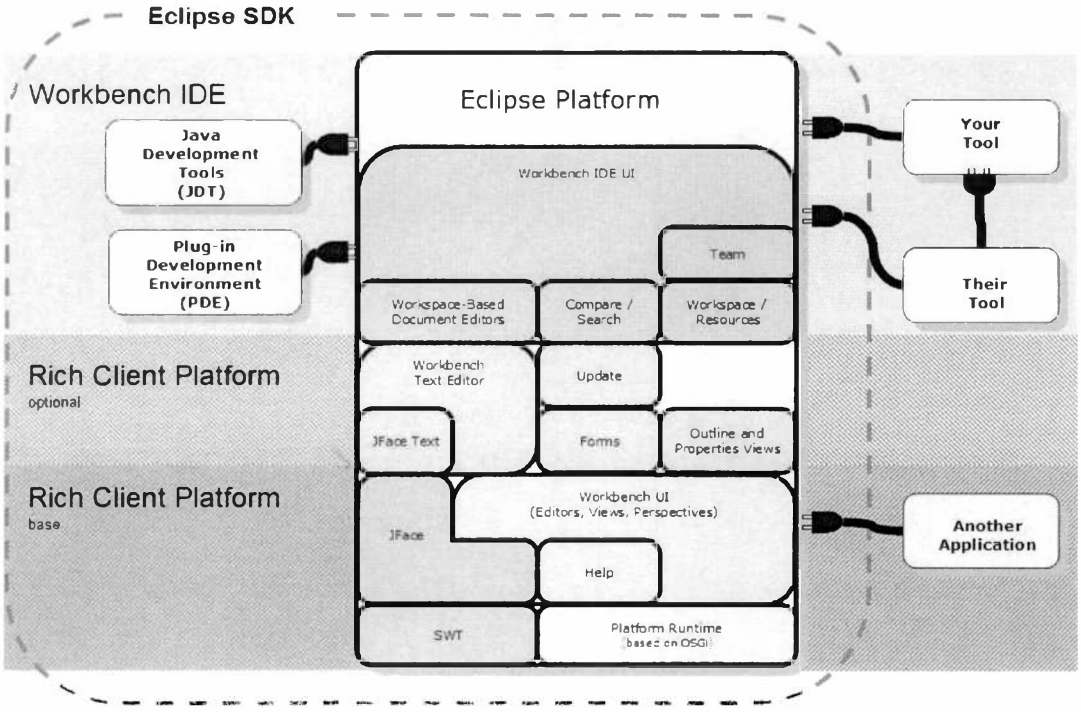


Figura 8 - Arquitectura de Eclipse para el desarrollo de plug-ins

1. Eclipse Modeling Framework

Eclipse Modeling Framework (Framework de Modelado Eclipse, EMF) [EMF] es un framework de modelado para Eclipse que facilita la generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurado.

EMF comenzó como una implementación de la especificación de MOF, pero luego evolucionó en base a la experiencia adquirida en la construcción de un conjunto de herramientas basadas en EMF. Sin embargo, soporta la lectura y escritura de serializaciones MOF, y está basado en un meta-metamodelo llamado Ecore (figura 9).

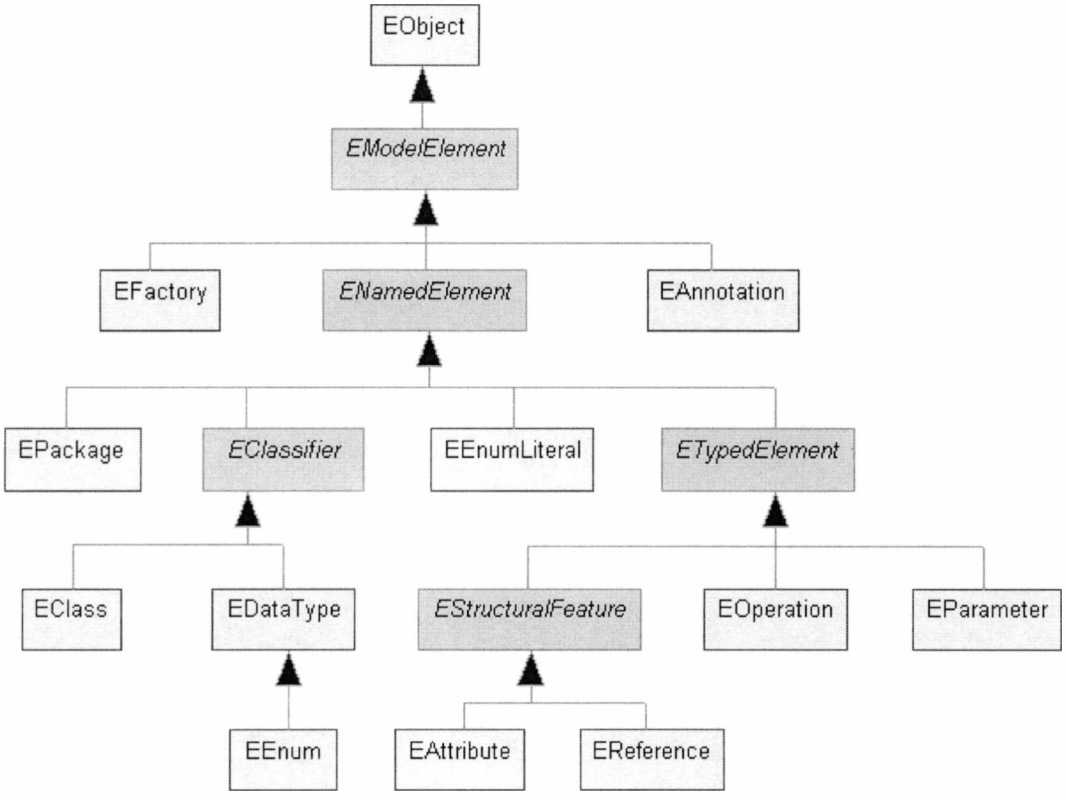


Figura 9 - Modelo Ecore

En EMF los modelos se especifican usando un meta metamodelo llamado Ecore. Ecore es una implementación de eMOF (en inglés Essential MOF), siendo éste un modelo EMF y su propio metamodelo. Si bien existen algunas diferencias entre Ecore y EMOF, EMF puede leer y escribir serializaciones de EMOF haciendo posible un intercambio de datos entre herramientas.

Los modelos son entonces, instancias del metamodelo Ecore y son guardados en formato XMI facilitando de este modo el intercambio de modelo en diferentes herramientas compatibles con MOF.

Un modelo Ecore entonces se puede generar a partir de:

1. Diagramas de clases UML
2. Interfaces Java
3. Esquemas XML

2. Definición del lenguaje de pruebas

El propósito de esta sección es explicar cómo crear y utilizar los estereotipos para describir la definición del modelo del lenguaje de pruebas utilizando la especificación del Perfil de Prueba de UML a partir del cual se va a generar el código JUnit.

Las especificaciones de UML 2 requieren que cualquier estereotipo sea definido en un elemento del perfil. Esto quiere decir que se necesita crear un perfil UML 2 para definir los estereotipos usados en la definición del lenguaje de pruebas. Como se describió anteriormente, el metamodelo puede especificarse con un editor gráfico para metamodelos Ecore, o a través de un documento XML [XML], un diagrama de clases UML o como interfaces de JAVA con Anotaciones. EMF provee asistentes para interpretar ese metamodelo y convertirlo en un modelo EMF, es decir, en una instancia meta-metamodelo Ecore.

La ventaja de partir de un modelo UML es que es un lenguaje conocido, por lo tanto no se requiere entrenamiento para usarlo, y que las herramientas de UML que suelen ser más amigables y proveen una mayor funcionalidad. Sólo hay que tener en cuenta que la herramienta permita exportar un modelo *core* que pueda ser usado como entrada para el framework EMF.

Para el desarrollo de nuestro perfil de pruebas basados en U2TP, vamos a utilizar el editor gráfico para metamodelos Ecore contenido dentro de un proyecto Eclipse.

Para comenzar la creación del perfil utilizando el editor UML, vamos a crear un archivo del tipo UML Model llamado *uml2testingprofile.profile.uml* seleccionando el tipo **Profile** para el objeto a modelar como muestra la figura 10. Por convención, los recursos que contienen perfiles utilizan el sufijo *.profile.uml* como extensión del archivo en UML2.

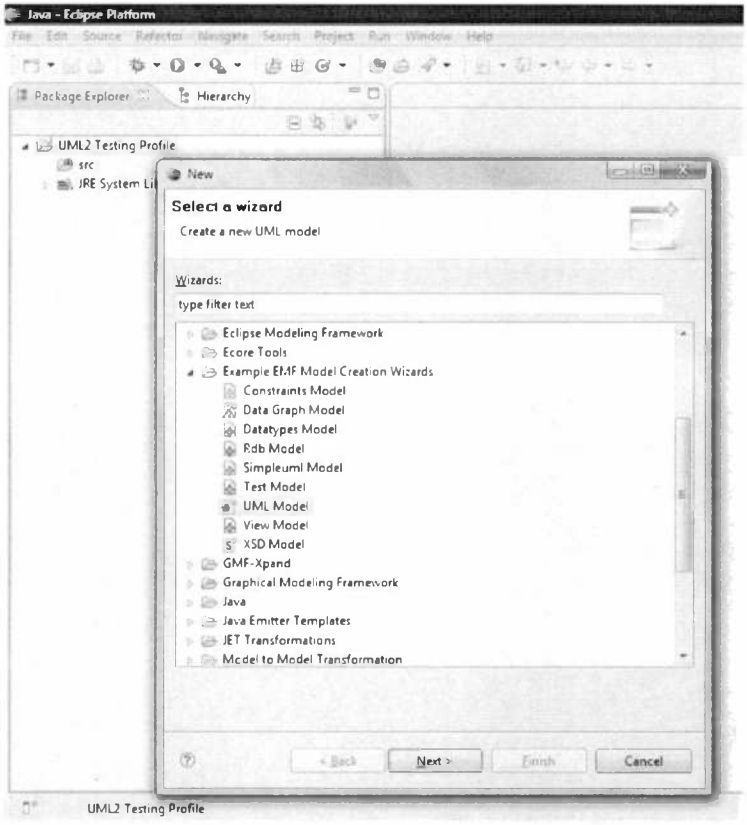


Figura 10 - Creando un perfil UML

2.1. Creando los estereotipos

Un estereotipo define como una metaclase existente se puede extender. Cada uno de los estereotipos puede extender una o más clases a través de las extensiones como parte de un perfil.

Para crear un estereotipo dentro del editor UML, hay que seleccionar el Perfil, en nuestro ejemplo llamado *"UML2 Testing Profile"*, y con un clic derecho seleccionar la opción del menú contextual: *New Child* > *Packaged Element* > *Stereotype* como muestra la figura 11. Para comenzar a definir los estereotipos necesarios en nuestro lenguaje es necesario crear los estereotipos <<SUT>>, <<TestContext>>, <<TestComponent>> y <<TestCase>>.

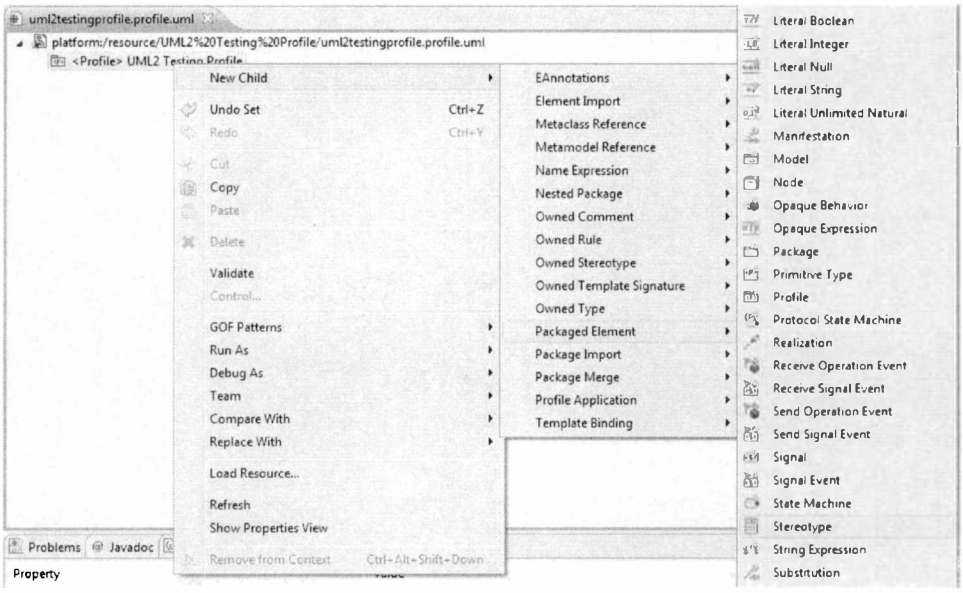


Figura 11 - Cómo crear un estereotipo

2.2. Referenciando metaclases

Un perfil siempre debe ser relativo a un metamodelo de referencia como por ejemplo UML y no se puede usar sin su metamodelo de referencia. Los perfiles pueden referenciar metaclases desde metamodelos creando una relación entre el perfil y la metaclase.

Desde el editor UML se logra la referencia a través de la opción del menú *UML Editor > Profile > Reference Metaclass*, donde seleccionaremos las metaclases *uml::Class, uml::Interface, uml::Operation, uml::Property*.

2.3. Creando extensiones

Las extensiones se utilizan para indicar que las propiedades de las metaclases son extendidas por los estereotipos y permiten flexiblemente aplicar estereotipos a los elementos. Una extensión es una clase de asociación, la cual indica que una instancia del estereotipo extendido debe ser creada si una instancia de la metaclase extendida se crea.

Los pasos para crear con el editor UML las extensiones son los siguientes:

1. Seleccionar el estereotipo (ejemplo <<SUT>>)
2. Desde el menú UML Editor > Stereotype > Create Extension...
3. Elegir la metaclase (ejemplo uml::Class)

2.4. Definiendo el Perfil de Pruebas UML

Un perfil representa efectivamente un aumento del metamodelo de referencia UML, por lo tanto antes de empezar a utilizar el perfil se necesita "definirlo" en el nivel de meta-metamodelo. La implementación del soporte de perfil en UML 2 hace esto para convertir el contenido del perfil a una representación Ecore equivalente que se guarda como una anotación en el perfil. Para definir el perfil utilizando el editor UML, se deben realizar los siguientes pasos:

1. Seleccionar el perfil en el editor UML (ejemplo "UML Testing Profile").
2. Seleccionar del menú la opción UML Editor > Profile > Define.

En este punto, nuestro perfil se encuentra definido de la siguiente manera:

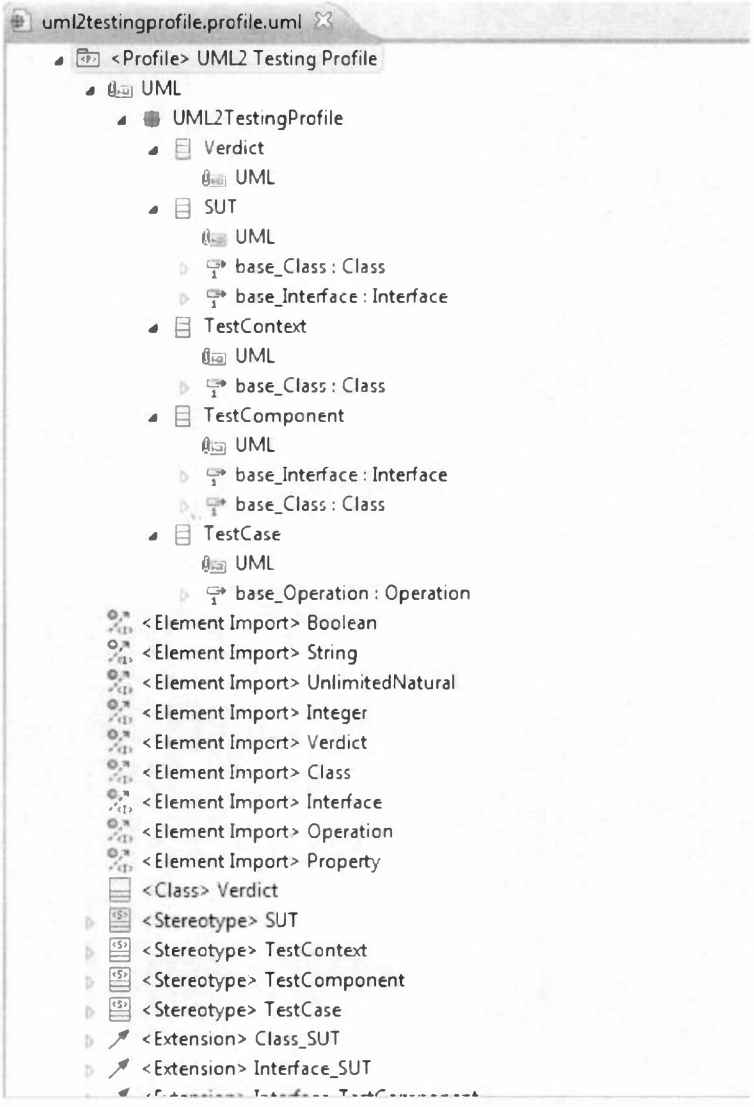


Figura 12 - Perfil de Pruebas UML

2.5. Herramientas de soporte para definir perfiles

Con el objetivo de conocer los detalles acerca de cómo funciona la definición de un Perfil de Pruebas UML, se utilizó el mecanismo manual de creación del perfil a través del editor UML de Eclipse. De todas maneras, existen herramientas de modelado basadas en UML2 para crear las definiciones de perfiles y exportarlos correctamente de manera sencilla a través de un editor gráfico. Estas herramientas ayudan a desarrollar los modelos de manera más eficiente mejorando la productividad durante el ciclo de vida de un proyecto.

Omondo UML Plug-in (www.eclipseuml.com) para Eclipse es un plug in de modelado UML 2.1 que soporta notación UML2.1 y metamodelo XMI 2.x disponible en Eclipse 3.4. Esta herramienta de modelado se puede utilizar tanto para crear nuevos modelos como para hacer ingeniería reversa de modelos existentes.

A modo de ejemplo, vamos a mostrar cómo crear un diagrama de Perfil de Pruebas UML con la ayuda de esta herramienta, evitando entonces conocer los detalles específicos de la definición de un perfil.

Primero hay que seleccionar el tipo de diagrama, en nuestro caso *UML Profile Diagram* como muestra la figura 13.

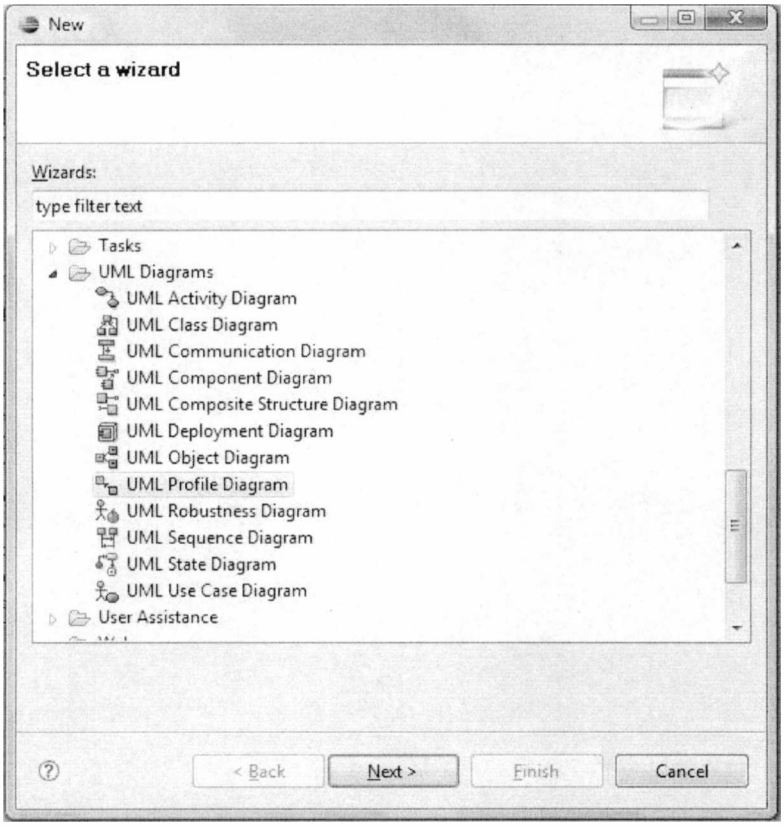


Figura 13 - Creando un diagrama de perfil UML

Darle un nombre al archivo del diagrama del perfil (ejemplo *TestProfileDiagram*), esta operación creará automáticamente otro archivo (*TestProfileDiagram.profile.uml*) con la extensión *.profile.uml* similar al generado manualmente en la sección anterior. Este archivo contiene la estructura del perfil mientras que el archivo *.prd* únicamente contiene información gráfica sobre el diseño del diagrama. La figura 14 muestra el editor gráfico para comenzar a crear los estereotipos del perfil.

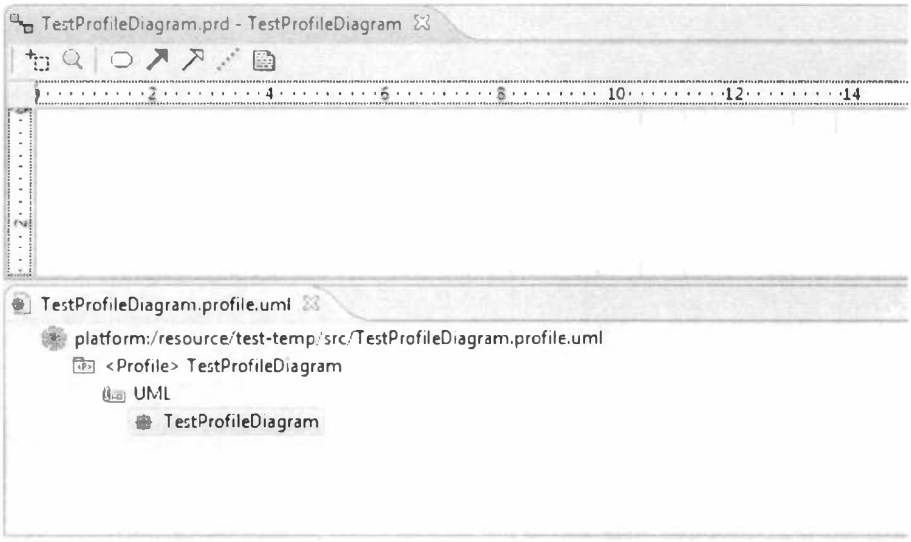


Figura 14 - Diagrama del Perfil UML inicial

En la figura 15 se muestra el perfil de testing modelado con la herramienta para dar soporte a la creación de los modelos estructurales y dinámicos generado.

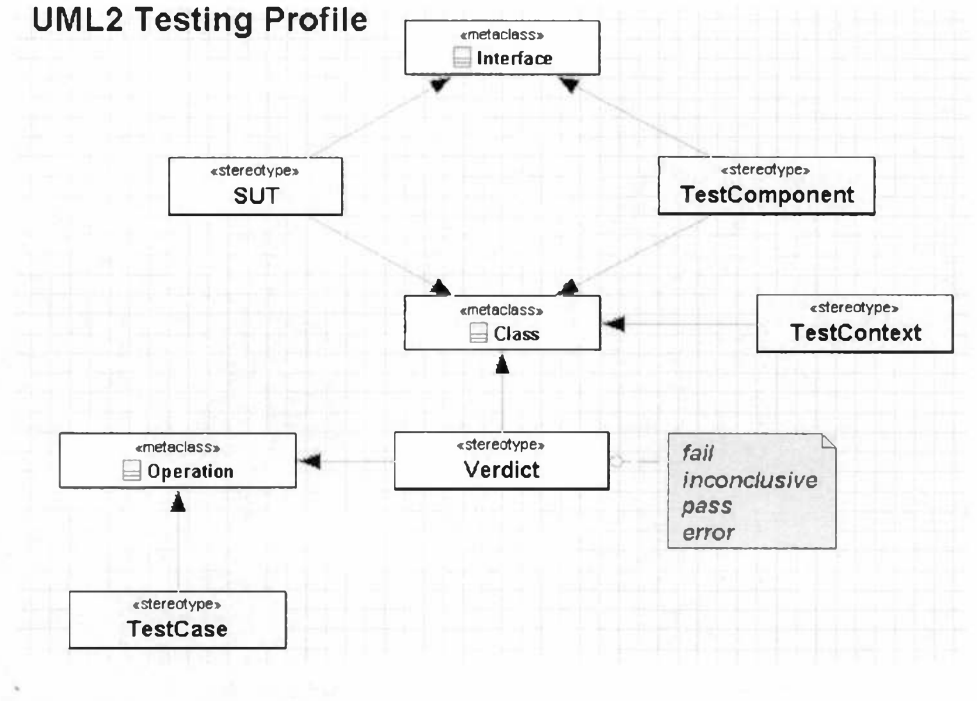


Figura 15 - Definición del Perfil de Pruebas UML

3. Aplicando el Perfil de Pruebas UML

Ahora que el Perfil de Pruebas está definido, necesitamos aplicar los estereotipos creados sobre los modelos estructurales y de comportamiento para demostrar como nuestra herramienta de transformación genera código JUnit a partir de estos modelos.

Para eso vamos a desarrollar dos ejemplos extraídos de un sistema de foros para las siguientes funciones:

1. Listar todos los foros, método *showAll()*.
2. Listar un foro a partir de su ID, método *showForum()*.

La figura 16 muestra el diagrama de clases modelado para soportar las funcionalidades descriptas, con los estereotipos U2TP sobre los objetos que nos interesan.

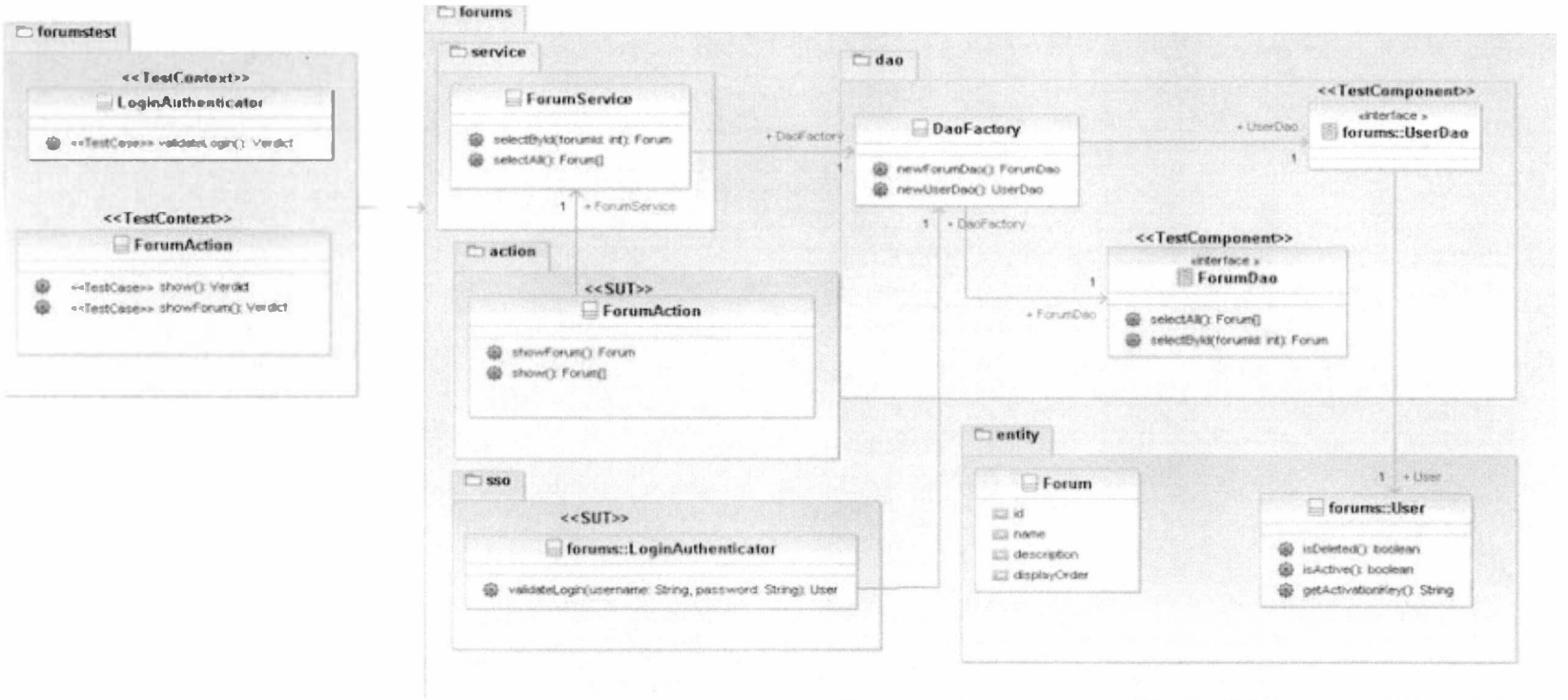

forumstest
<<TestContext>>
LoginAuthenticator
<<TestCase>> validateLogin(): Verdict
<<TestContext>>
ForumAction
<<TestCase>> show(): Verdict
<<TestCase>> showForum(): Verdict
forums
service
ForumService
selectById(forumId: int): Forum
selectAll(): Forum[]
+ DaoFactory
1
1
+ ForumService
dao
DaoFactory
newForumDao(): ForumDao
newUserDao(): UserDao
+ UserDao
1
<<TestComponent>>
«interface»
forums::UserDao
1
1
+ DaoFactory
<<TestComponent>>
«interface»
ForumDao
selectAll(): Forum[]
selectById(forumId: int): Forum
1
+ ForumDao
action
<<SUT>>
ForumAction
showForum(): Forum
show(): Forum[]
entity
Forum
id
name
description
displayOrder
1
+ User
forums::User
isDeleted(): boolean
isActive(): boolean
getActivationKey(): String
sso
<<SUT>>
forums::LoginAuthenticator
validateLogin(username: String, password: String): User

Figura 16 - Ejemplo modelado con U2TP

Además de crear el modelo estructural, vamos a especificar los diagramas de secuencias para los dos métodos del ejemplo que permitan darle comportamiento a nuestras pruebas.

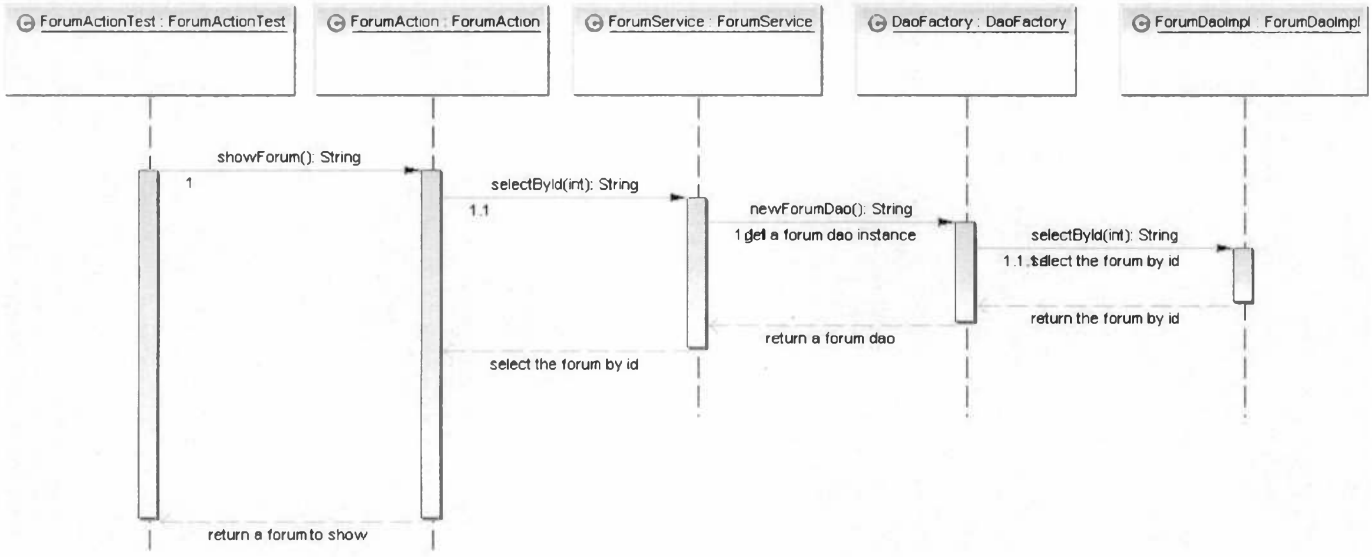


Figura 17 - Diagrama de secuencia del método showForum

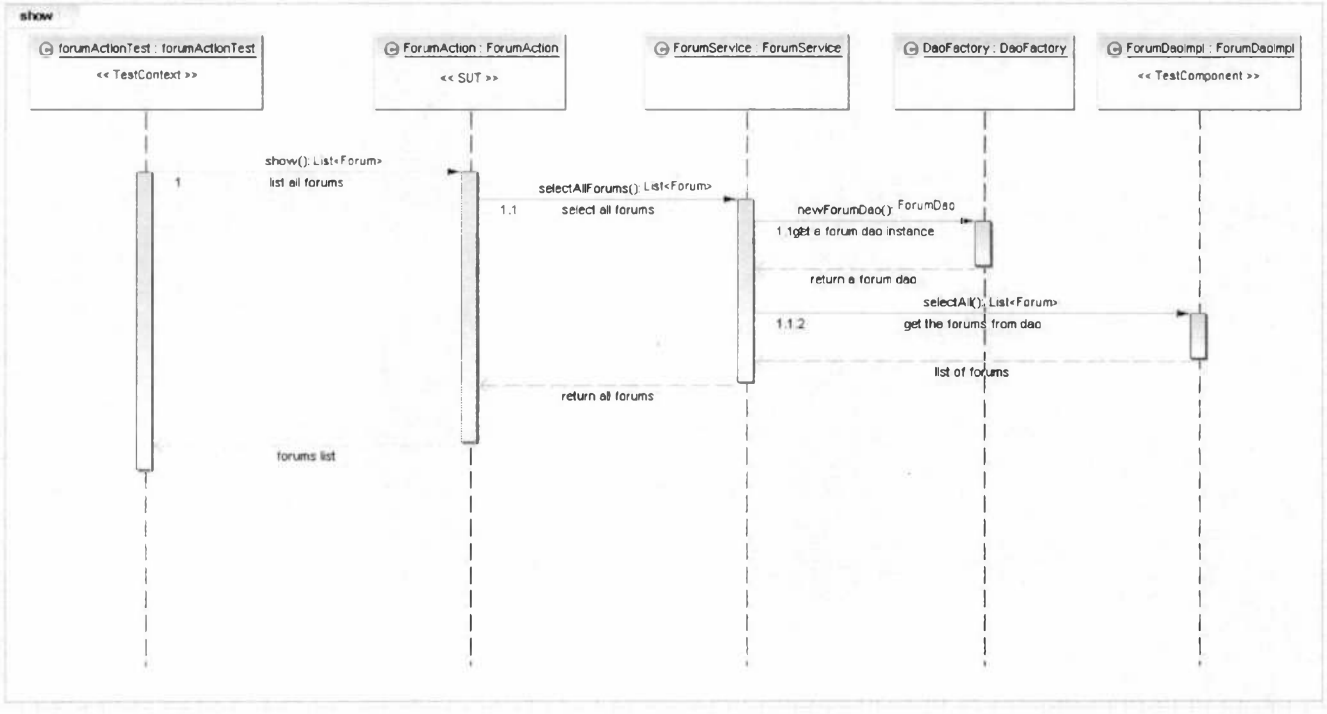


Figura 18 - Diagrama de secuencia del método showAll

Antes de poder hacer uso del Perfil de Pruebas con los nuevos estereotipos en el modelo estructural y de comportamiento definido (*example.uml*), hay que agregar el perfil dentro del modelo que vamos a modelar. La figura 19 muestra el Perfil de Pruebas UML definido para nuestro modelo.

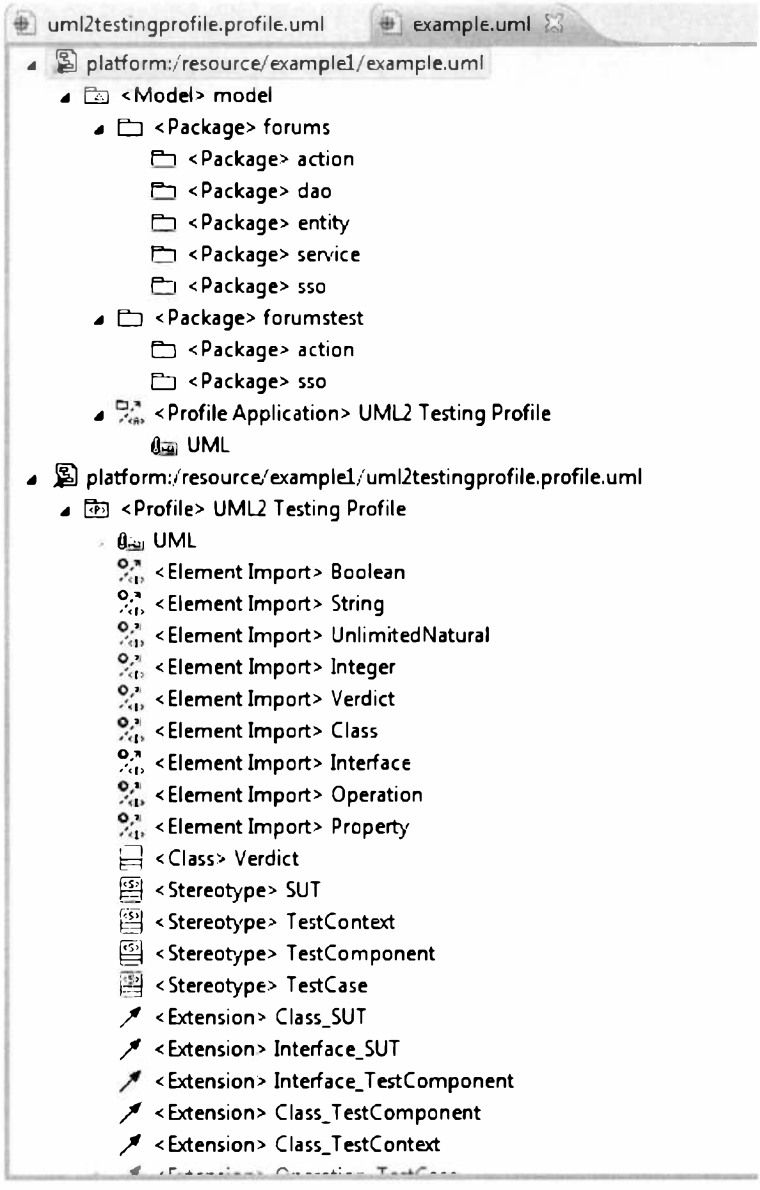


Figura 19 - Perfil de Pruebas UML definido

En estos momentos podemos comenzar a definir el perfil sobre el modelo como muestra la figura 20.

Figura 20 - Modelo de pruebas UML

Hasta aquí, hemos creado y aplicado el Perfil de Pruebas UML desarrollado sobre un modelo UML2 estructural y de comportamiento. Como paso a previo a mostrar el plug-in final que empaqueta a la aplicación de transformación de código, vamos a mostrar cómo es que la herramienta genera el código de prueba JUnit a partir del modelo UML2 de entrada.

4. Transformar modelos de pruebas a código JUnit

Las transformaciones modelo a texto (M2T) crean simplemente un documento en formato textual, generalmente de tipo String a partir de la definición de los metamodelos. En esta sección presentamos la definición formal de las transformaciones de modelos (PIM) utilizando el lenguaje MOFScript [Oldevik 06] para generar el código de pruebas JUnit. Las definiciones elaboradas permiten que las transformaciones puedan ser ejecutadas en forma automatizada. Para mostrar cómo se realizan las transformaciones vamos a utilizar los metamodelos del ejemplo del

sistema especificado en U2TP. Sólo mostraremos algunas partes del código de las transformaciones que consideramos relevantes.

4.1. Herramientas de transformación de modelos a texto

La herramienta MOFScript (www.eclipse.org/gmt/mofscript) permite la transformación de cualquier modelo MOF a texto, generando por ejemplo código Java, SQL Scripts, HTML o documentación a partir de los modelos. MOFScript provee un lenguaje de metamodelo independiente que permite el uso de cualquier clase de metamodelo y sus instancias para la generación de texto.

Las características principales de MOFScript son:

- Generación de texto a partir de cualquier modelo basado en un metamodelo definido en MOF, como por ejemplo, modelos UML o cualquier otro metamodelo.
- Manipulación de Strings
- Mecanismos de control básicos como por ejemplo loops y sentencias condicionales.

- Especificar el archivo salida para la generación del texto.

- Permite que una transformación pueda extender otra transformación, aunque sólo la herencia simple es permitida.
- Mantener las trazas entre los modelos y el texto generado.

- El editor de scripts posee ayuda para completar el código.

- Soporta operaciones UML2 como lo son los estereotipos utilizados en los perfiles UML.
- Define reglas (similar a los métodos) concretas y abstractas, además de permitir la sobre escritura de las reglas.
- Puede invocar código Java externo, además de contar con la posibilidad de integrar MOFScript con la API de Java.

Actualmente, la ingeniería inversa todavía no es parte de la herramienta.

El lenguaje de transformación MOFScript es un lenguaje que fue enviado al pedido de propuestas de lenguajes de transformación de modelo a texto lanzado por el OMG, pero que no resultó seleccionado. A pesar de eso, lo hemos seleccionado por que es muy similar al estándar elegido por OMG y además cuenta con un plug-in de Eclipse basado en EMF y el framework de metamodelos Ecore. La herramienta soporta el parseo, chequeo y ejecución de scripts escritos en MOFScript. Cualquier archivo con extensión **".m2t"** será tratado como un archivo MOFScript como muestra la figura 21.

MOFScript está basado en QVT, es un refinamiento del lenguaje operacional de QVT. Es un lenguaje textual, basado en objetos y usa OCL para la navegación de los elementos del metamodelo de entrada.

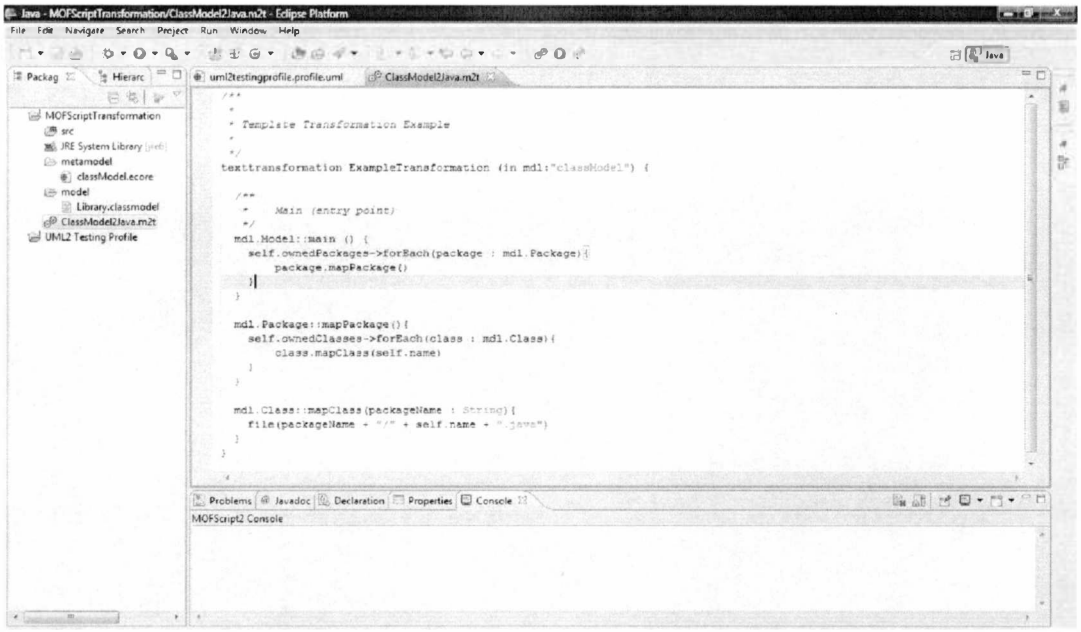


Figura 21 - Muestra un archivo MOFScript en Eclipse

4.1.1. La arquitectura de MOFScript

MOFScript está organizado en la herramienta y los servicios de componentes. La herramienta de componentes permite al usuario la edición de los scripts e interactuar con los servicios; en cambio los servicios facilitan el parseo, verificación y ejecución del lenguaje de transformación.

Los Servicios de Componentes consisten de las siguientes partes:

- ✓ El Model Manager que es un componente basado en EMF que se encarga de administrar los modelos MOFScript.
- ✓ Los componentes Parser y Lexer responsables de parsear las definiciones textuales de las transformaciones MOFScript, y popular un modelo MOFScript utilizando el Model Manager. El parseador está basado en Antlr [Antlr].
- ✓ Semantic Checker provee la funcionalidad de verificar si las transformaciones son correctas.
- ✓ El Execution Engine se encarga de la ejecución de la transformación, interpretando el modelo y generando el texto de salida.
- ✓ El Text Synchroniser que administra la traza entre los textos generados y el modelo original, manteniendo sincronizado el texto de acuerdo a los cambios en el modelo y vice versa.

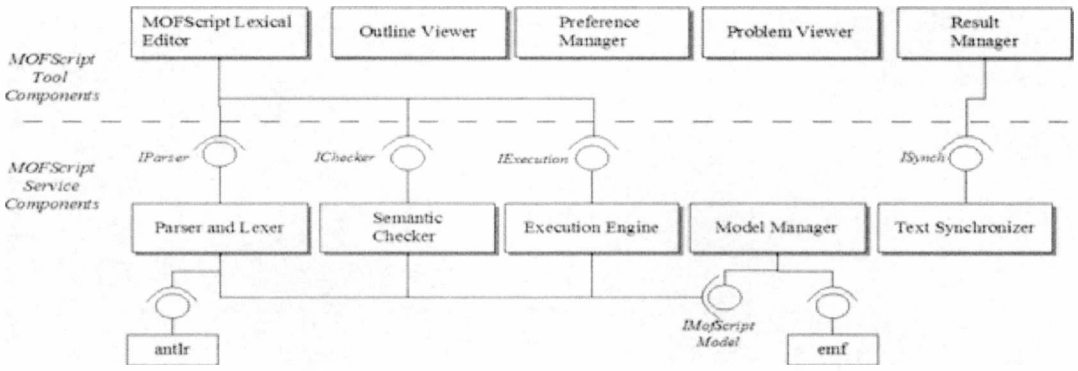


Figura 22 - Diseño de la arquitectura general MOFScript

4.2. Definiendo las transformaciones de modelos

En esta sección presentamos parte del código de definición de la transformación para convertir los modelos de pruebas (PIM) escrito en UML a código JUnit.

4.2.1. Transformando modelos estructurales

La primera regla define la transformación de un modelo UML. Esta regla se complementa con la definición de los estereotipos del Perfil de Pruebas UML, con las reglas que recorren los paquetes para transformar los elementos del modelo en clases JUnit, agregar el comportamiento en los métodos de pruebas definidos en los diagramas de comportamiento, y generar las clases Java de los objetos relacionados entre otras.

```
//Define la transformación de un modelo UML2
texttransformation umlmodelgenerator (in
    uml:"http://www.eclipse.org/uml2/2.1.0/UML") {

  ...

  //Propiedades de estereotipos U2TP.
  property U2TP_VERDICT      : String = "Verdict"
  property U2TP_TEST_CONTEXT : String = "TestContext"
  property U2TP_TEST_CASE    : String = "TestCase"

  /**
   * Módulo principal de entrada.
   */
  uml.Model::main() {
    //Recorre todos los paquetes del modelo y procesa
    //procesa sus objetos.
    self.ownedMember->forEach(p:uml.Package) {
        p.mapPackage()
      }
  }
```

```
/**
 * Mapea una instancia UML a una clase Java o
 * JUnit Test case.
 */
uml.Class::mapClass(packageName : String) {
  ...
  /**
   * Si la clase tiene estereotipo <<TestContext>> y no
   * super clase entonces extiende de TestCase, sino
   * extiende de su super clase.
   */
  if (!self.superClass.isEmpty()) {
    <% extends %>self.superClass.first().name <%%>
  } else if (self.superClass.isEmpty() &&
             self.hasStereotype(U2TP_TEST_CONTEXT)) {
    <% extends TestCase%>
  }
  ...
  if (self.hasStereotype(U2TP_TEST_CONTEXT)) {
<%  /**
     * @param name
     */
    public %> self.name<%(String name) {
      super(name);
    }
    /* (non-Javadoc)
     * @see junit.framework.TestCase#setUp()
     */
    @Override
    protected void setUp() throws Exception {
      super.setUp();
    }
    /* (non-Javadoc)
     * @see junit.framework.TestCase#tearDown()
     */
    @Override
    protected void tearDown() throws Exception {
      super.tearDown();
    }
%>
  }

/**
 * Mapea una instancia UML a una clase Java o
 * JUnit Test case.
 */
uml.Class::mapClass(packageName : String) {
  ...
  //Traduce las operaciones
  if (vReturn.equals("") or
      self.hasStereotype(U2TP_TEST_CASE)) {
    <%void%>
  } else {
    <%%>  + vReturn
  } <% %>
  /**
   * Si la operación tiene el estereotipo <<TestCase>> se
   * traduce como: public void test...()
   */
  if (self.hasStereotype(U2TP_TEST_CASE)) {
    <%test%> self.name.firstToUpper() <%{%>
  } else {  ...  }
  <%}%>
  if (withoutBody && self.isAbstract) {
    <%;%>
```


```
  } else {
    if (self.hasStereotype(U2TP_TEST_CASE)) {
      <% throws Exception%>
    }
  <% }%>
  //Verifica si el método tiene comportamiento implementado
  //en algún diagrama de secuencia.
  if (operationBody != null && !operationBody.equals("")) {
<%
%> operationBody<%%>
  } else {
<%
    // TODO should be implemented
%>
    if (self.hasStereotype(U2TP_TEST_CASE)) {
<%
      %> defaultAssertEquals<%%>
    }
  }<%
  }%>
...
}
```

El esqueleto del código JUnit generado se describe a continuación:

```
public class LoginAuthenticatorTest extends TestCase {

    public LoginAuthenticatorTest(String name) {
      super(name);
    }

    protected void setUp() throws Exception {
      super.setUp();
    }

    protected void tearDown() throws Exception {
      super.tearDown();
    }

    public void testValidateLogin() throws Exception {
      //TODO should be implemented
      assertEquals("Verify the expected value", null, null);
    }
}
```

4.2.2. Transformando modelos dinámicos

Primero se buscan los métodos con comportamiento dentro de los diagramas de secuencia del modelo de prueba recorriendo todos los paquetes en los que existen clases de tests para luego traducirlos. El código MOFScript muestra un breve resumen de cómo se realizan los pasos detallados.

```
/**
 * Mapea una instancia UML a una clase Java o
 * JUnit Test case.
 */
uml.Class::mapClass(packageName : String) {
  ...
```

```
  /**
   * Obtiene el comportamiento de los Diagramas de Secuencia.
   *
   */
  var testMethodBehaviorHash : Hashtable =
      self.getTestMethodBehavior()
  ...
}
  /**
   * @return Operaciones de TEST y objetos test components.
   */
  uml.Class::getTestMethodBehavior() : Hashtable {
    ...
    if (self.hasStereotype(U2TP_TEST_CONTEXT)) {
      //Recorre *todos* los paquetes buscando clases de test
      //utilizadas en Diagramas de Secuencia.
     self.package.ownedMember->forEach(col:uml.Collaboration) {
       col.ownedMember->forEach(interaction:uml.Interaction) {
         testComponentsHash = interaction.getTestComponents()
         var testOperationList:List = new List()
         interaction.ownedMember->forEach(lifeLine:uml.Lifeline) {
           var prop:uml.Property = lifeLine.represents
           var classReceiver:uml.Class = prop.type
           //Clase de Prueba para transformar y traducir el
           //comportamiento del método.
           if (classReceiver.hasStereotype(U2TP_TEST_CONTEXT)) {
             if (!lifeLine.coveredBy.isEmpty()) {
               ...
             }
           }
         }
       }
     }
    }
    ...
    result = testClassesHash
  }
```

Los métodos con comportamiento están indicados con el estereotipo TestComponent de U2TP. Para detectarlos con MOFScript se utiliza el soporte de UML2.

```
/**
 * Procesa todos los estereotipos <<TestComponent>>
 *
 * @return Clases de tests con sus casos de prueba.
 */
uml.Interaction::getTestComponents() : Hashtable {


    self.ownedMember->forEach(lifeLine:uml.Lifeline) {
      var prop:uml.Property = lifeLine.represents
      var classReceiver:uml.Class = prop.type
      //Estereotipo <<TestComponent>>
      if (classReceiver.hasStereotype(U2TP_TEST_COMPONENT)) {
        if (!lifeLine.coveredBy.isEmpty()) {
          var operationsToMockHash:Hashtable = new Hashtable()
          lifeLine.coveredBy->
          forEach(messageSpec:uml.MessageOccurrenceSpecification){
            if (messageSpec.message != null) {
              var msg:uml.Message = messageSpec.message
              var opToMock:String = msg.processTestComponentMessage()
              operationsToMockHash.put(messageSpec, opToMock)
            }
          }
          //Componentes de Tests y sus operaciones mocks.
```

```
            testComponentsHash.put(classReceiver, operationsToMockHash)
          }
        }
      }
      result = testComponentsHash
    }
}
```

Finalmente, mostramos el código JUnit obtenido por la herramienta de transformación que incluye además el comportamiento modelado en el diagrama de secuencia. De esta forma, logramos una mayor completitud del caso de caso de prueba en forma automática evitando así la intervención manual del desarrollador.

```
public class LoginAuthenticatorTest extends TestCase {

    ...

    public void testValidateLogin() throws Exception {
      UserDaoImpl mockUserDaoImpl =
          EasyMock.createMock(UserDaoImpl.class);
      EasyMock.expect(
          mockUserDao.selectByLogin(
            EasyMock.anyObject(),
            EasyMock.anyObject())).andReturn(
                  EasyMock.anyObject());
      String username = null;
      String password = null;
      assertEquals("Verify the expected value",
            null,
            loginAuthenticator.validateLogin(username,password));
      EasyMock.replay(mockUserDaoImpl);
      EasyMock.verify(mockUserDaoImpl);
    }
}
```

En resumen, la herramienta desarrollada brinda las siguientes características principales:

- Traduce a código JUnit diagramas de clases modelados con el perfil de pruebas U2TP.
- Permite modelar diagramas de secuencias para agregar comportamiento a los casos de pruebas en forma automática.
- Genera el esqueleto del caso de prueba si no existe un diagrama de secuencia asociado, o el diagrama modelado no utiliza objetos TestComponent que agreguen comportamiento al método.
- Traduce las clases SUT a código Java para que puedan ser utilizadas por el código de prueba y soportando además la traducción de datos no primitivos.

4.3. Integrando la herramienta en un plug-in de Eclipse

Los scripts de transformaciones MOFScript se pueden ejecutar en forma manual desde el entorno de Eclipse o en forma automática desde una aplicación utilizando la API de Java que provee. El siguiente código ilustra parte de la integración de MOFScript con la API de Java dentro de un plug-in desarrollado como producto final de la propuesta.

```
public class JUnitCodeGenAction
    implements IEditorActionDelegate, ExecutionMessageListener {


  @Override
  public void run(IAction action) {
      ...
    model2Text();
    ...
  }

  public void model2Text() {
    ...

    ParserUtil parserUtil = new ParserUtil();
    parserUtil.setMetaModelRepositoryURI(repository);
    parserUtil.setInputFileLocation(repository);

    //Busca los archivos .m2t files para utilizar en los imports.
    parserUtil.setCompilePath(repository);
    File f2 = new File (repository + "umlmodelgenerator.m2t");
    @SuppressWarnings("unused")
    MOFScriptSpecification spec2 = parserUtil.parse(f2, true);



    XMIResourceFactoryImpl _xmiFac = new XMIResourceFactoryImpl();
    ResourceSet rSet = new ResourceSetImpl();
    rSet.getResourceFactoryRegistry().getExtensionToFactoryMap()
        .put("*", _xmiFac);

    EObject selectedSourceModel = null;

    URI selectedFileURI = null;

    ...;

    execMgr.addSourceModel(selectedSourceModel);
    execMgr.setRootDirectory(ROOT_DIRECTORY);
    execMgr.setUseFileModel(false);
    execMgr.setUseLog(false);
    execMgr.getExecutionStack().addOutputMessageListener(this);
    try {
      execMgr.executeTransformation();
    } catch (MofScriptExecutionException mex) {
      MessageDialog.openError(
          shell,
          "U2TP Codegen Plug-in transformation error",
          mex.getMessage());
```

```
    }
  }
}
```

Aquí listamos el código resultado completo de las transformaciones de los ejemplos presentados al inicio de este capítulo para los casos de pruebas del paquete ***forumtests*** que se encarga de listar todos los foros o uno en particular. Esto incluye también, la parte dinámica modelada en los diagramas de secuencia.

```
/**
 * Generated class ForumAction
 *
 * @author MOFScript generator 'U2TP2Java'
 * @date 2/8/2009
 */
public class ForumAction extends TestCase {

  /**
   * @param name
   */
  public ForumAction(String name) {
    super(name);
  }

  /* (non-Javadoc)
   * @see junit.framework.TestCase#setUp()
   */
  @Override
  protected void setUp() throws Exception {
    super.setUp();
  }

  /* (non-Javadoc)
   * @see junit.framework.TestCase#tearDown()
   */
  @Override
  protected void tearDown() throws Exception {
    super.tearDown();
  }

  /**
   *
   */
  public void testShow() throws Exception {
    ForumDaoImpl mockForumDaoImpl =
        EasyMock.createMock(ForumDaoImpl.class);
    EasyMock.expect(
        mockForumDao.selectAll())
            .andReturn(EasyMock.anyObject());

    assertEquals(
        "Verify the expected value",
        null,
        forumAction.show());

    EasyMock.replay(mockForumDaoImpl);
```

```
        EasyMock.verify(mockForumDaoImpl);
    }

    /**
     *
     */
    public void testShowForum() throws Exception {
        ForumDaoImpl mockForumDaoImpl =
            EasyMock.createMock(ForumDaoImpl.class);
        EasyMock.expect(
            mockSelectByIdForumDao.selectById(EasyMock.anyObject()))
                .andReturn(EasyMock.anyObject());

        assertEquals("Verify the expected value",
            null,
            showForumAction.showForum());

        EasyMock.replay(mockForumDaoImpl);
        EasyMock.verify(mockForumDaoImpl);
    }
}
```

4.4. Arquitectura de la implementación propuesta

En la figura 23 finalmente mostramos la arquitectura de la implementación y describimos cómo sus distintos componentes y plug-ins interactúan entre sí permitiendo generar el código de pruebas JUnit a partir de modelos UML2 definidos con el perfil de pruebas creado anteriormente.

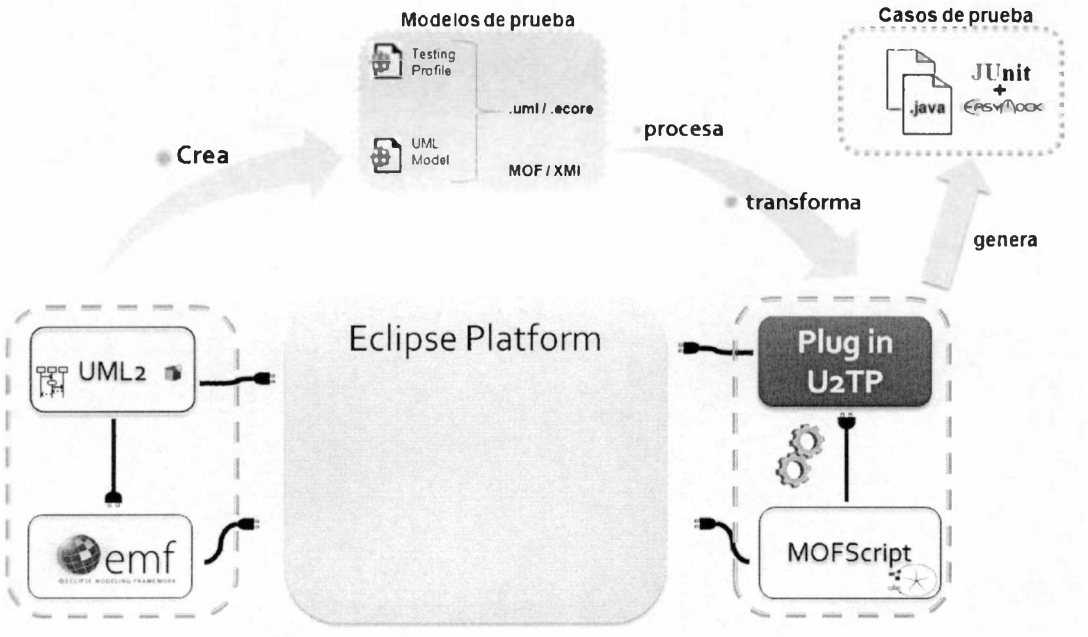


Figura 23 – Arquitectura de implementación

EMF es el framework de modelado y generación de código base que nos permite construir herramientas y otras aplicaciones basadas en su modelo de estructuras de datos.

UML2 es una implementación basada en EMF del metamodelo OMG del Lenguaje Unificado de Modelado (UML) 2.x para la plataforma Eclipse. A partir de estos dos componentes podemos construir metamodelos UML reusable a través de un esquema XMI común que facilita su intercambio entre distintas herramientas de modelado. UML2 provee un editor de modelos UML2 que nos permite diseñar los diagramas. En nuestro trabajo, lo hemos utilizado para diseñar el perfil de pruebas, los diagramas estructurales y de comportamiento, en nuestros casos clases y secuencias respectivamente y aplicar el perfil sobre el modelo generado. El editor de UML2 permite generar los metamodelos en archivos con extensiones .uml y/o .ecore.

MOFScript es la herramienta de transformación de modelo a textos utilizada para generar el código de implementación JUnit. Los ***templates*** están creados con el lenguaje de metamodelo independiente que permite usar cualquier clase de metamodelos y sus instancias para generación de texto. MOFScript está basado en MOF y Ecore como framework de metamodelo.

Por último presentamos el plug in U2TP desarrollado en el proyecto. Este plug in está integrado con MOFScript y recibe los metamodelos creados para obtener los casos de pruebas JUnit y Easy Mock. Actualmente soporta la generación de código de pruebas a partir de modelos UML2 de clases y secuencias.

5. Resumen

En este capítulo hemos presentado la implementación de la propuesta, tecnologías, frameworks y lenguajes utilizados para su desarrollo y las características soportadas por la herramienta de generación de código de prueba desarrollada. Esta herramienta está construida e integrada en un plug-in de Eclipse, utilizando como entrada modelos compatibles con MOF definidos por el framework EMF que establece un soporte de interoperabilidad con otras herramientas. Mostramos de qué manera definir el metamodelo del lenguaje de pruebas y cómo aplicarlo sobre un caso práctico.

Introducimos la herramienta de transformación de modelo a texto denominada MOFScript que ofrece compatibilidad con los estándares de OMG y brinda un lenguaje imperativo definido como una extensión del lenguaje QVT operacional y OCL con nuevas expresiones para generación de texto. Finalmente, describimos las partes principales de las plantillas construidas con MOFScript para obtener el código de prueba JUnit y EasyMock correspondiente a los modelos de pruebas diseñados a lo largo del capítulo.

CASO DE ESTUDIO

En esta sección vamos a mostrar la guía completa de cómo usar la herramienta que permite transformar modelos de prueba a código JUnit y EasyMock desde la perspectiva del usuario. Esto incluye los pasos iniciales necesarios para la generación y definición del modelo de prueba, además de indicar cómo aplicar el perfil U2TP diseñado y cómo utilizar el plug-in de código abierto desarrollado para la plataforma Eclipse.

A continuación partimos de un caso de ejemplo concreto, pasando por todas las etapas previas antes de llegar a la generación en forma automática del código JUnit con Easy Mock a través de la herramienta construida bajo nuestra propuesta.

1. Creando el modelo inicial

Para comenzar, creamos un nuevo modelo UML dentro de un proyecto de la plataforma Eclipse como muestra la figura 24. Este archivo contendrá el diseño del diagrama estructural de clases y los de comportamiento a través de los diagramas de secuencias. Más adelante mostramos el uso del perfil de pruebas aplicable a cada objeto del modelo.

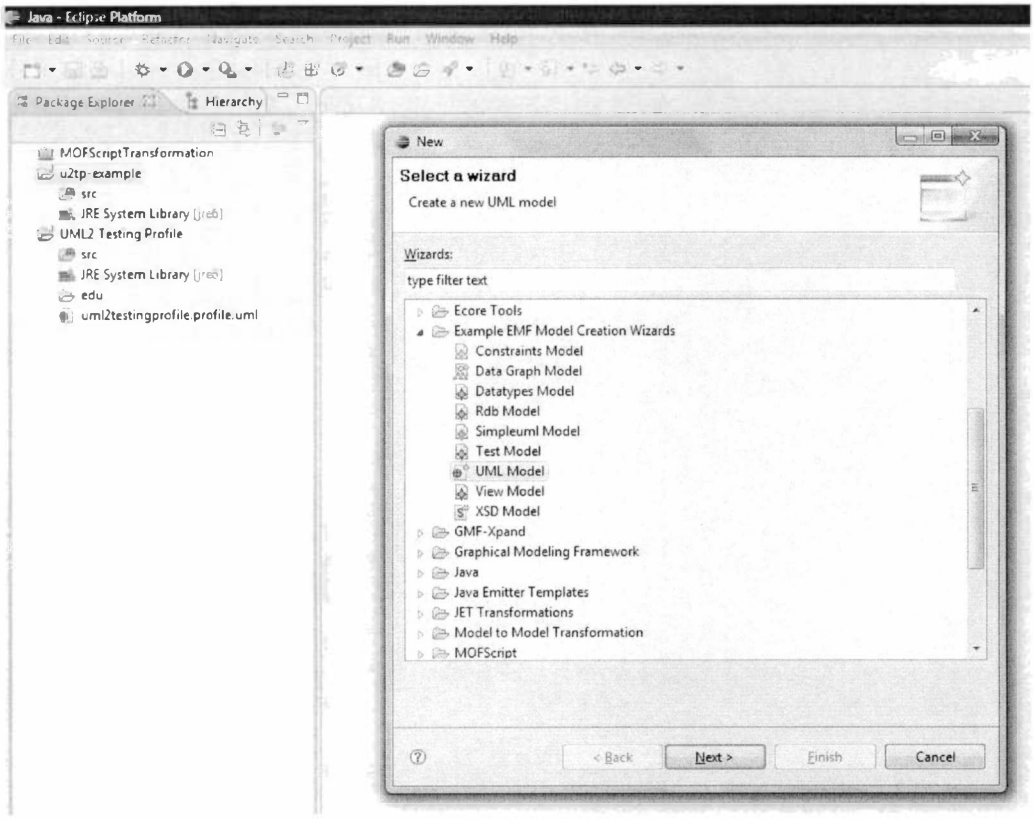


Figura 24 – Creando un modelo UML en Eclipse

Vamos a darle un nombre al archivo, ejemplo "*example.uml*". Los archivos con formato ***.uml*** son compatibles con los modelos Ecore basados en formato XMI que pueden ser leídos por herramientas con soporte para UML2.

Luego de identificar el archivo ***.uml***, debemos seleccionar el tipo de modelo a construir definido por el campo Model Object. Para nuestro caso seleccionamos el valor ***Model*** como indica la figura 25. Por último finalizamos la etapa inicial de creación del modelo UML.

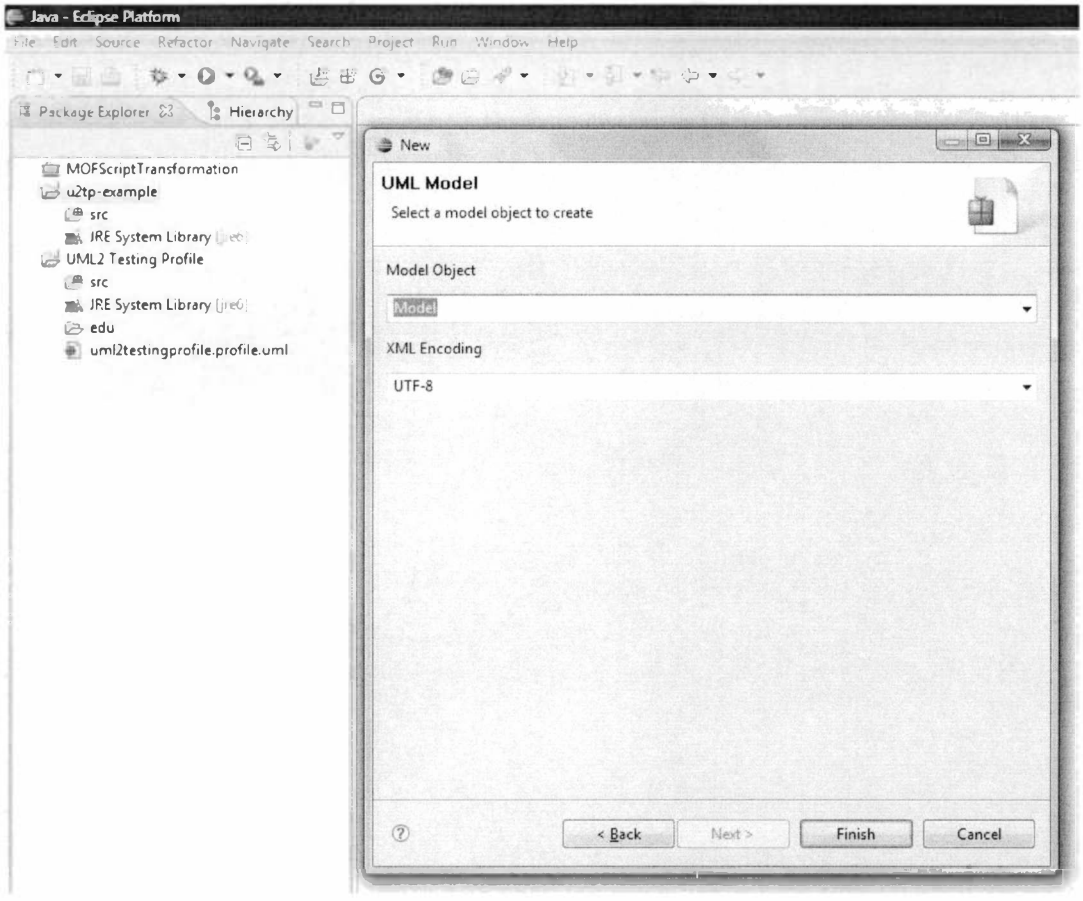


Figura 25 – Seleccionando el tipo de modelo de un diagrama UML

Ahora estamos en condiciones de modelar nuestro sistema. Para nuestro ejemplo, utilizamos el sistema de foros detallado en capítulos anteriores.

Si abrimos el archivo *example.uml* podemos observar el modelo inicial, sobre el cual iremos agregando los objetos necesarios que permitan modelar el sistema completo. Para eso, vamos a seleccionar el objeto ***Model*** y listar todas las opciones del menú contextual que el editor UML2 provee. En la figura 26 mostramos como crear el primer paquete de la aplicación denominado ***forums***.

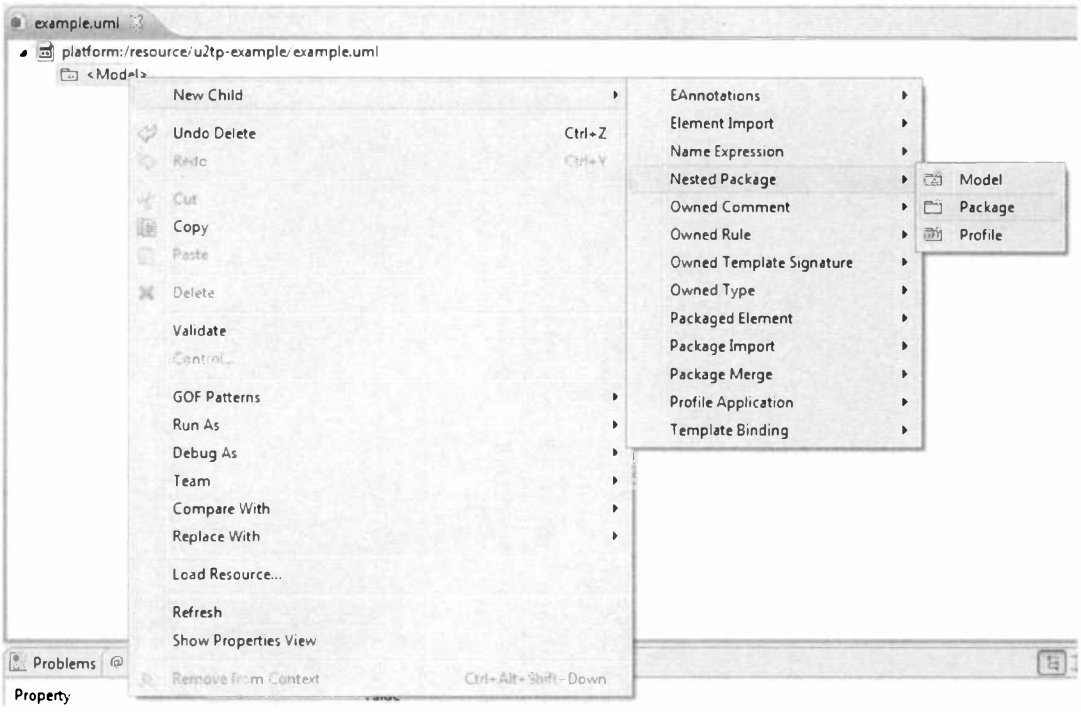


Figura 26 – Creando paquetes en un diagrama UML

Repetimos el proceso para crear los demás paquetes correspondientes al sistema de foros, para luego agregar las clases. Aunque para estas entidades debemos seleccionarlas desde la opción ***New Child / Packaged Element / Class***. Al completar la lista de entidades necesarias, agregar los atributos (***New Child / Owned Attribute / Attribute***) y métodos (***New Child / Owned Operation / Operation***). También completamos el paquete ***forumstest*** que contendrá las clases de pruebas con las entidades y métodos que vamos a verificar como muestra la figura 27.

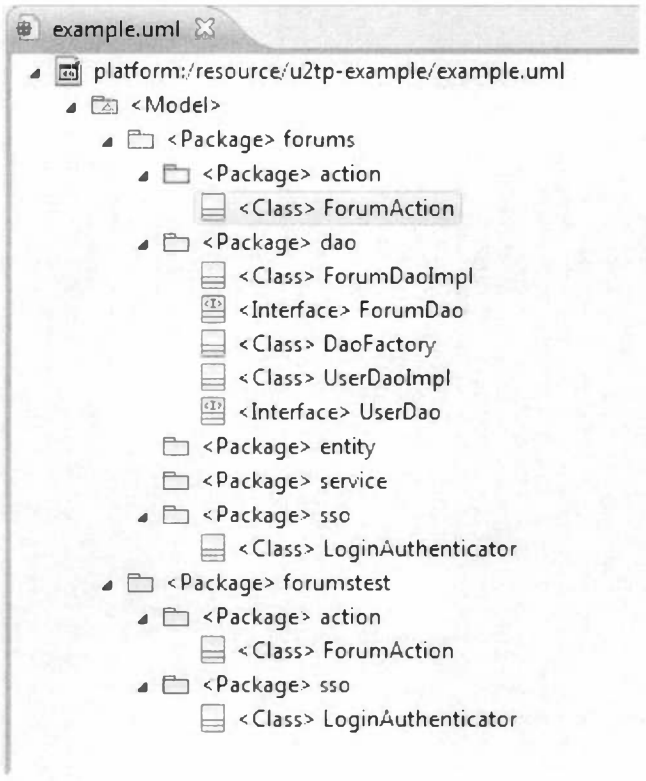


Figura 27 – Diagrama de clases inicial

A partir de nuestro diagrama de clases con sus atributos y métodos, opcionalmente podemos modelar el comportamiento de los casos de pruebas a través de diagramas de secuencias. Esto permite agregar comportamiento al código de prueba generado en forma automática. Para continuar con nuestro ejemplo, seleccionamos los siguientes métodos del paquete de pruebas ***forumstest***:

- Consultar y mostrar un foro determinado.

  Este caso de prueba se denomina show() y se encuentra en la clase ForumAction del paquete ***action***.

- Consultar todos los foros.

  El caso de prueba showAll() se encuentra en la clase y paquete mencionado anteriormente.

- Validar la identidad del usuario cuando éste se autentica en el sistema.

El último de los casos de prueba se denomina validateLogin() y se encuentra en la clase LoginAuthenticator del paquete ***sso***.

Los diagramas de secuencias, representados por el objeto ***<Interaction>***, están organizados bajo el objeto ***<Collaboration>***. Así podemos empezar a agregar los demás objetos que definen la secuencia y los mensajes que se intercambian entre las distintas instancias de las clases. Los principales objetos agregados que podemos mencionar son los conectores, representados por el objeto ***Connector***, las líneas de vida (***Lifeline***) y los mensajes (***Message***) entre otros. La figura 28 muestra parte de la definición de los diagramas.

Figura 28 – Diagrama de secuencia del ejemplo

2. Cargando y utilizando el Perfil de Pruebas UML

Antes de comenzar a utilizar el perfil de pruebas debemos cargarlo e inicializarlo en nuestro modelo. Una vez cargado el perfil podemos agregar los estereotipos de pruebas definidos sobre los objetos del diseño facilitando así su traducción a través de un mecanismo automático.

Cargando el perfil a través de ***UML Editor / Load Resource...*** como muestra la figura 29 agrega el perfil dentro del archivo example.uml.

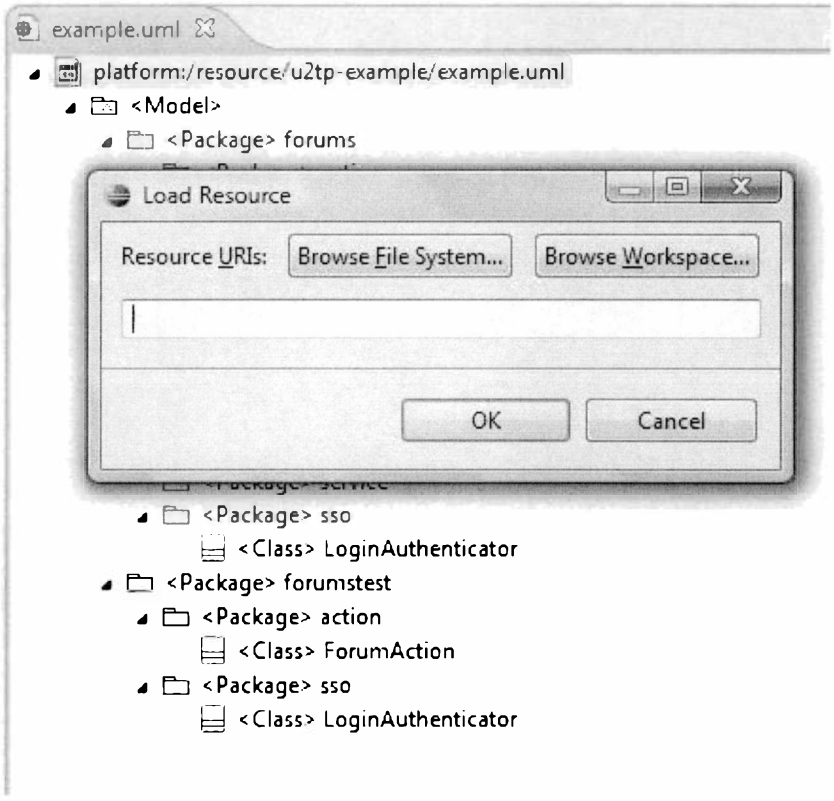


Figura 29 –Cargando un perfil de pruebas UML

Una vez terminada esta acción, hay que indicar que dicho perfil puede estar disponible para aplicarlo sobre el modelo ***Model***. Para eso, seleccionamos el objeto ***Model*** de nuestro caso de ejemplo y elegimos el perfil a través de ***UML Editor / Package / Apply Profile...*** como muestra la figura 30.

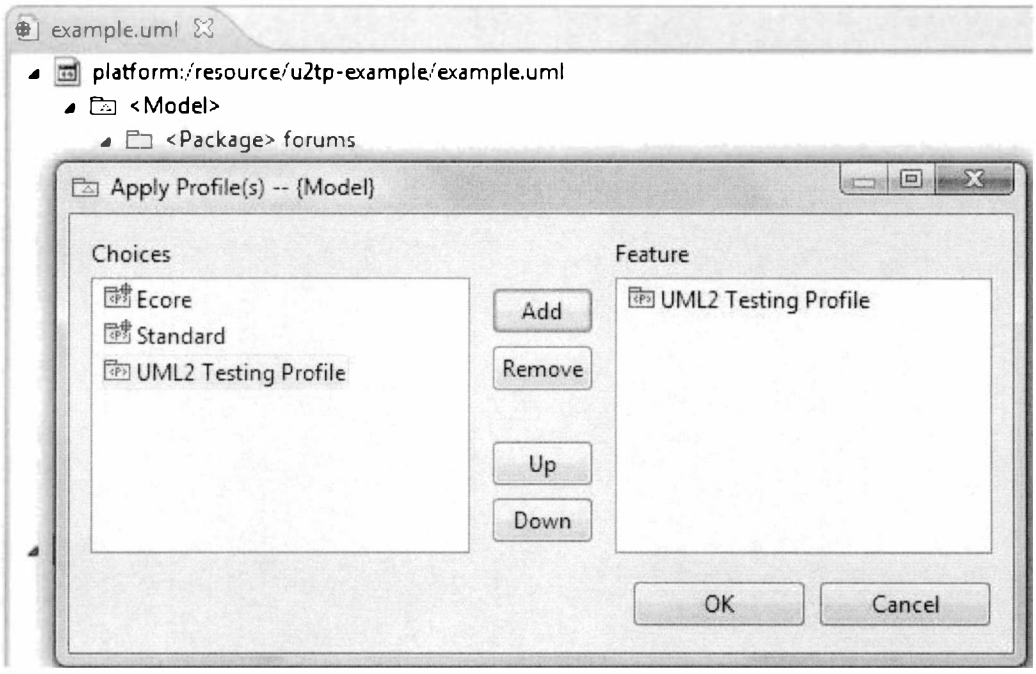


Figura 30 – Seleccionando el perfil de pruebas UML

Ahora estamos en condiciones de comenzar a aplicar los estereotipos del perfil de Pruebas UML sobre los distintos elementos del modelo de acuerdo al tipo de objeto para el que pueda aplicarse.

La figura 31 muestra cómo aplicamos desde el menú **UML Editor / Element / Apply Stereotype** el estereotipo ***SUT*** sobre una clase del modelo.

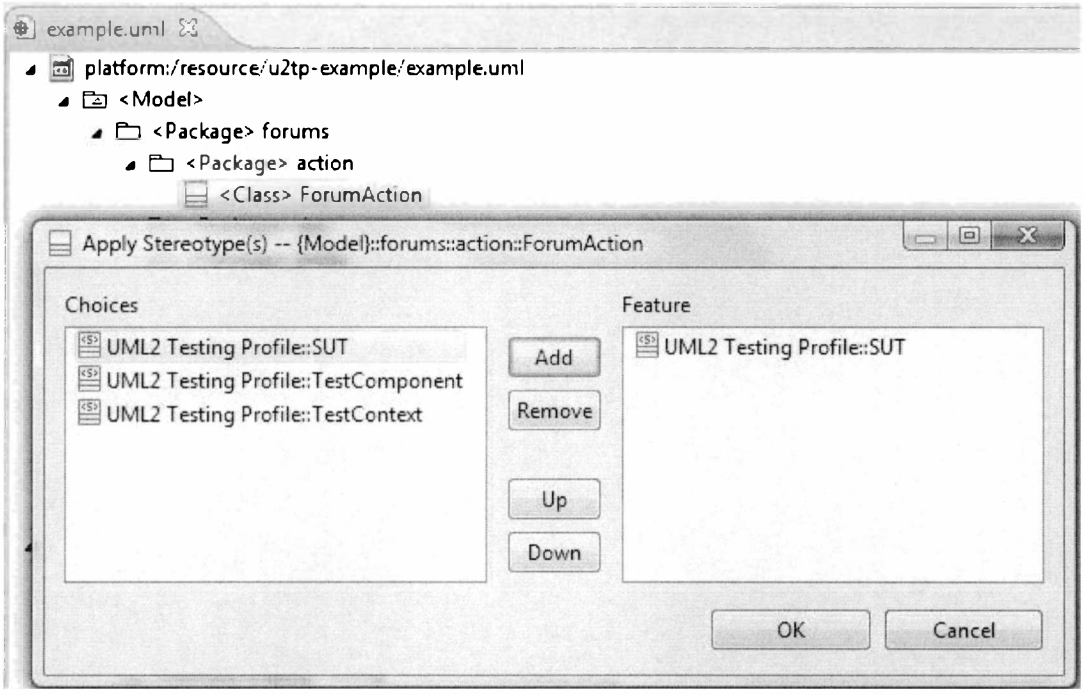


Figura 31 – Aplicando el estereotipo SUT

Lo mismo podemos hacer para los demás objetos, indicando el tipo de estereotipo que necesitamos aplicar, como pueden ser ***<<TestComponent>>*** para las clases o interfaces.

De igual manera que podemos utilizar los estereotipos ***<<TestContext>>***, ***<<TestCase>>*** y ***<<Verdict>>*** para especificar las clases u operaciones de pruebas para los dos últimos como muestra la figura 32.

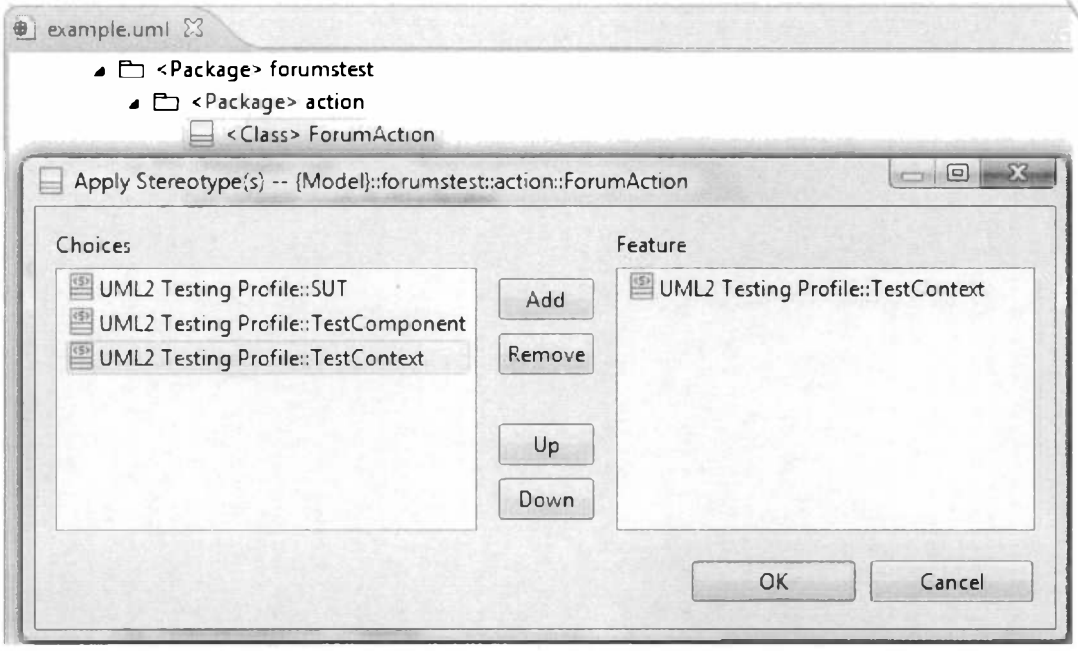


Figura 32 – Aplicando el estereotipo TestContext

Llegado a este punto, estamos en condiciones de utilizar el plug-in desarrollado y procesar el modelo UML2 diseñado.

3. Generando el código de pruebas

Para finalizar nuestro ejemplo, vamos a transformar el modelo de prueba a código. Para completar esta actividad, solo debemos seleccionar el modelo UML2 y listar el menú contextual de Eclipse presionando el botón derecho del mouse. Allí deberá aparecer listado la opción ***JUnit code generator*** dentro del menú ***u2tp.codegen*** como muestra figura 33.

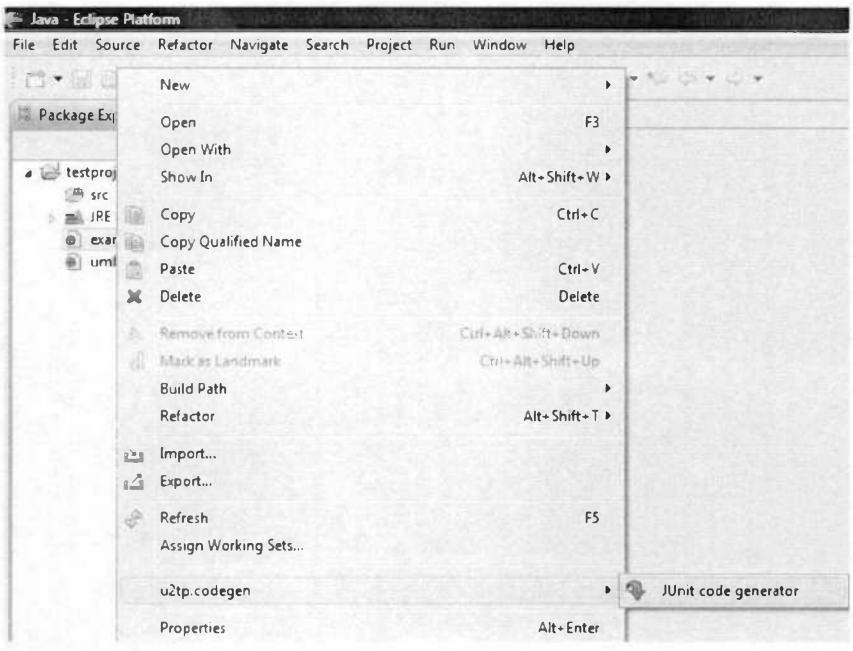


Figura 33 - Plug-in para transformar un modelo de pruebas a código JUnit

Como resultado de la ejecución del plug-in desarrollado obtenemos el código de pruebas JUnit y Easy Mock de manera automática y con soporte a los casos de pruebas con comportamiento de manera más completa.

4. Resumen

En este capítulo hemos desarrollado un caso de estudio desde el punto de vista del usuario del plug-in construido para nuestra propuesta de tesis. Definimos un modelo de pruebas completo y mostramos cómo aplicar el modelo independiente de la plataforma (PIM) en el entorno Eclipse, para luego transformarlo a código de prueba JUnit.

Como corolario, hemos demostrado la utilidad de nuestra herramienta para generar y transformar modelos de pruebas en forma automática. Su valor agregado también está dado a medida que el proceso de transformación se hace más complejo y/o repetitivo incrementando la productividad, eficiencia y confiabilidad del producto final en todas las etapas del ciclo de vida del desarrollo de software.

TRABAJOS RELACIONADOS

En este capítulo presentamos una descripción de algunos de los trabajos sobre desarrollo de software dirigidos por modelos de pruebas, junto a las propuestas de transformaciones que guardan alguna relación con nuestro trabajo de tesis.

Korat [Korat] es un framework de testing automático de programas Java. Dada una especificación formal de un método, Korat utiliza la precondición del método para generar automáticamente todos los casos de pruebas de tamaño pequeño a través del método de model checking. Korat ejecuta el método de cada caso de prueba y verifica los resultados de cada salida con los métodos de las post condiciones. Para generar el caso de prueba de un método, Korat construye un predicado Java, como puede ser un método que devuelva un valor booleano, desde los métodos de las precondiciones. En este artículo se ilustra el uso de Korat para probar distintas estructuras de datos, incluyendo algunas del framework de Colecciones de Java. Los programadores pueden utilizar JML para escribir los métodos de las pre y pos condiciones como así también de las invariantes. El prototipo Korat implementado utiliza Java Modeling Language (JML) para realizar las especificaciones y generar los oráculos de pruebas en forma automática desde los métodos de las post condiciones. El conjunto de herramientas de JML traduce las post condiciones JML en aserciones Java en tiempo de ejecución. Si la ejecución de un método no cumple dicha aserción, entonces arroja una excepción para indicar que la post condición no se cumple. Los oráculos de pruebas atrapan dichas excepciones y las reportan. Korat también utiliza *JML+JUnit* para combinar los oráculos de pruebas JML con JUnit lo que permite automatizar la ejecución y verificación de los resultados. De todos modos, *JML+JUnit* necesita que los programadores provean el conjunto de posibilidades para todos los parámetros de los métodos.

Dentro del contexto de MDT, Heckel y Lohmann proponen en [Towards Model-Driven Testing] utilizar patrones de diseño comunes para crear casos de pruebas en entornos de testing local o distribuido. Para modelar los diagramas UML utiliza reglas gráficas de reacciones para describir el estado de la transformación de datos de los modelos a casos de pruebas asociados con los estados iniciales y la secuencia de los métodos utilizados. Aunque esto permitiría generar el código de pruebas en forma automática a través de la simulación de los modelos, hasta el momento no cuenta con una herramienta de soporte, siendo un tema pendiente para las futuras investigaciones.

En el artículo [From-U2TP-models-to-executable-tests-with-TTCN-3] se presenta una propuesta que tiene como objetivo derivar oráculos de pruebas ejecutables desde diagramas U2TP en forma automática a código *Testing and Test Control Notation versión 3* (TTCN-3). Para eso, define las reglas de transformación concretas de mapeo entre los conceptos de los meta-modelos U2TP de origen a los conceptos de los meta-modelos TTCN-3 destino. El proceso de mapeo incluye cada elemento del meta-modelo de origen, por lo tanto mapea cada estereotipo, interface, tipo primitivo, propiedades, operaciones y parámetros a las meta-clases y asociaciones correspondiente del meta-modelo de destino. La definición de las reglas de

transformación está casi completa; aunque existen casos especiales en determinados diagramas de pruebas para revisar como son los de secuencia, actividad o interacción. Además, no todos los conceptos de U2TP están mapeados, especialmente aquellos como *Timezone* o *Scheduler*. La herramienta creada es un plug-in de Eclipse U2TP basado en un proyecto UML2.0; y las reglas de transformación están desarrolladas en Java. Las transformaciones generan objetos dentro de una instancia de meta-modelo TTCN-3 lo que permite la compilación y ejecución de las pruebas diseñadas previamente en U2TP. Los resultados de la transformación generan el esqueleto del código de pruebas en TTCN-3, por lo tanto requiere el esfuerzo adicional para que un programador complete las definiciones de las pruebas.

[MDT-Dai] introduce una metodología sobre cómo aplicar los conceptos de U2TP a un modelo de diseño UML en forma efectiva para obtener un modelo de pruebas. En esta metodología, las clases y objetos son agrupados en conjunto para definir los componentes de pruebas o SUT. Para realizar las transformaciones provee a los diseñadores de las pruebas de unos mecanismos denominados *directivas de pruebas*, y de un meta-modelo llamado *Test Directive Meta-Model*. Para crear una instancia del meta-modelo U2TP, las reglas de transformaciones se aplican tanto sobre el meta-modelo de UML como así también sobre el meta-modelo *Test Directive Meta-Model*. Los tres modelos están basados en MOF. La transformación del modelo UML al modelo U2TP se especifica por un conjunto de reglas definidas en los meta-modelos [Model-Transformation] de acuerdo a la especificación Query/View/Transformation (QVT) definidas por [CBOP-DSTC-IBM]. El lenguaje de transformación introducido es orientado a aspectos, declarativo y basado en patrones. Las *reglas de transformación* se utilizan para describir la correspondencia entre los patrones de los elementos en el modelo de origen y los elementos a ser creados en el modelo destino. Los *patrones* son definiciones reusables, cuando se utilizan en la regla de origen, un patrón se consulta; cuando se utiliza en el destino, funciona como una *plantilla* para los elementos del modelo. El *seguimiento de relaciones* asocia los elementos del modelo de origen con aquellos del modelo destino. Finalmente, el autor concluye que las reglas de transformación no están completas. Además, las reglas especificadas no se probaron a través de una herramienta que automatice la transformación de los modelos UML a modelos de pruebas utilizando U2TP.

En [MDT-Dai extension] la definición del proceso de derivación de pruebas por modelos sigue la propuesta de [MDT-Dai]. A partir de los modelos de diseño en UML, se propone realizar transformaciones a modelos de prueba basados en U2TP utilizando QVT para la generación automática de los casos de pruebas. Para lograrlo, se define una extensión del metamodelo de UML de forma que se puedan anotar los diagramas de secuencia con información que, luego, pueda ser utilizada para generar el oráculo de pruebas. Esta información es anotada en OCL como pre y pos condiciones en el diagrama. Los diagramas de secuencia extendidos con pre y pos condiciones son transformados posteriormente en modelos de prueba que son instancias de U2TP. El diagrama de secuencia se anota de forma que incluya información sobre el resultado esperado y el estado inicial del mismo. Por lo tanto, esta investigación se centra principalmente en la generación automática de pruebas a partir de los modelos de diseño del sistema (diagrama de secuencia y diagrama de clases, que corresponden al PIM) sean transformados mediante QVT en un modelo de pruebas, que luego puede ser refinado según la plataforma final, también mediante transformaciones QVT (a un modelo JUnit, por ejemplo). Para llevar a cabo la propuesta, resta la realización de las transformaciones QVT definidas en el trabajo que permita llevar los resultados de esta investigación al ámbito de las pruebas en líneas de producto de software.

1. Resumen

Al repasar los trabajos presentados en este capítulo, vemos que en la ingeniería de software existen mayores esfuerzos y diversas investigaciones dedicados al proceso de generación de código de pruebas en forma automática derivado de los modelos.

En contraste con la propuesta de esta tesis, [Korat] es una herramienta que no utiliza los estándares de UML y sus extensiones para diseñar modelos pruebas (U2TP). Si bien [Towards Model-Driven Testing] utiliza UML como lenguaje de modelación, la propuesta de reglas gráficas de transformación no tiene una herramienta de soporte que demuestre la factibilidad de su implementación, además de no seguir los estándares especificados por U2TP para la creación de modelos de pruebas. En [From-U2TP-models-to-executable-tests-with-TTCN-3] encontramos la utilización de las notaciones UML2.0 y U2TP estándares para diseñar los modelos de pruebas. Además, proporciona la implementación a través de un plug-in de Eclipse de la herramienta de generación del esqueleto del código de pruebas en forma automática en código TTCN-3 a partir de reglas de mapeo entre los conceptos de los metamodelos de ambas especificaciones. Finalmente, nuestra propuesta sigue la definición del proceso de los trabajos [MDT-Dai] y [MDT-Dai-extension] este último derivado del primero. Aunque [MDT-Dai] introduce una metodología sobre cómo aplicar los conceptos de U2TP a un modelo de diseño UML para obtener un modelo de pruebas, las reglas de transformación no están completas ni se probaron a través de una herramienta que automatice el proceso. La motivación de la investigación de [MDT-Dai-extension] se centra sobre las transformaciones de modelos de pruebas basados en U2TP ya si a código de prueba ejecutable en algún lenguaje, para lo cual define una extension del metamodelo UML en OCL, que luego utiliza para lograr las transformaciones. Este trabajo no logra aun probar dichas definiciones sobre un producto de software que automatice el proceso.

En contraste con las investigaciones estudiadas, nuestra propuesta define una extensión al metamodelo U2TP para diseñar los casos de pruebas aplicables sobre diagramas de clases y secuencia. Además, proporciona las definiciones formales completas para transformar dichas anotaciones en código de prueba JUnit ejecutable, brindando la posibilidad no sólo de generar el esqueleto del código prueba, sino también agregar cierto comportamiento especificado por el usuario en los diagramas de secuencia utilizando el framework EasyMock. Por lo tanto, a través de la implementación de una herramienta basada en un plug-in de Eclipse, demostramos la factibilidad de la propuesta desarrollada en el trabajo de tesis. Hasta el momento y de acuerdo al libro [Model-Driven-Testing-U2TP], no existe ningún componente de software que colabore en la generación de código JUnit en forma automática basada en modelos de pruebas diseñados con U2TP, lo que sería considerado de gran utilidad y aporte al estudio de MDT una propuesta en dicho sentido.

CONCLUSIONES FINALES Y TRABAJOS FUTUROS

Los modelos de pruebas requieren técnicas de validación y verificación para transformar modelos UML en forma automática a código que detecte errores o fallas en el producto final. El testing es una técnica de prueba de caja negra que permite automatizar el proceso de generación de casos de prueba a código. Esta es una técnica simple y práctica que no requiere experiencia en métodos formales para crear las especificaciones de los casos de pruebas. Las pruebas unitarias se pueden ejecutar en forma automática y permiten evaluar formalmente la corrección del producto final con respecto a sus modelos, brindando un marco de documentación del código además de asegurar que los nuevos cambios introducidos no generen errores.

En este trabajo hemos desarrollado una herramienta que permite transformar en forma automática modelos de pruebas estructurales (diagramas de clases) y de comportamientos (diagramas de secuencia) a código de prueba soportado por el framework de testing unitario JUnit. El mayor inconveniente en la traducción de los modelos de pruebas son los conceptos dinámicos que por lo general no se pueden automatizar en su totalidad y que en algunos casos exceden a la definición del framework JUnit. Con el objetivo de enriquecer las transformaciones de los diagramas de comportamiento, hemos utilizado la extensión del framework EasyMock para transformar los componentes dinámicos que permitan completar el código resultante y así simular el comportamiento de los objetos complejos.

Hemos creado también los estereotipos necesarios para modelar los dominios de pruebas utilizando el Perfil de Pruebas UML. Estos brindan un marco más amigable en la creación de casos de pruebas a través del uso del lenguaje de modelado estándar UML. Brindamos además las reglas de transformación de los modelos de pruebas que evalúan la calidad del software con respecto a sus modelos a través del lenguaje formal de transformaciones de modelo a texto MOFScript.

En base a los resultados obtenidos nos planteamos los siguientes trabajos futuros:

- Extender la herramienta para soportar la transformación automática de diagramas estructurales y de comportamiento adicionales a los de clases y secuencia hasta ahora soportados.
- Permitir la transformación de modelos a otros frameworks de testing unitario.
- Agregar mayor precisión en la traducción de métodos de comportamiento.
- Extender la herramienta y el lenguaje de modelado de pruebas para permitir la definición de pre y pos condiciones sobre los casos de pruebas y su correspondiente traducción a código ejecutable. Brindar también la posibilidad de especificar la secuencia de ejecución de los casos de pruebas.

84

GLOSARIO

Abstract Window Toolkit (AWT)

Es un kit de herramientas de gráficos, interfaz de usuario, y sistema de ventanas independiente de la plataforma original de Java. AWT es ahora parte de las Java Foundation Classes (JFC) - la API estándar para suministrar una interfaz gráfica de usuario (GUI) para un programa Java.

anotación Java

Es una forma de añadir metadatos al código fuente Java que están disponibles para la aplicación en tiempo de ejecución. Muchas veces se usa como una alternativa a la tecnología XML. Las Anotaciones Java pueden añadirse a los elementos de programa tales como clases, métodos, campos, parámetros, variables locales, y paquetes.

Ant

Es una herramienta multi-plataforma hecha en Java usada en programación para la realización de tareas mecánicas y repetitivas, normalmente durante la fase de compilación y construcción. Tiene la ventaja de no depender de las órdenes de Intérprete de comandos de cada sistema operativo, sino que se basa en archivos de configuración XML y clases Java para la realización de las distintas tareas.

Application Programming Interface (API)

Una Interfaz de Programación de Aplicaciones es el conjunto de funciones y procedimientos (o métodos, si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

Arquitectura Dirigida por Modelos (en inglés Model Driven Architecture, MDA)

Una aproximación a la especificación del sistema que separa la especificación funcional de la especificación de implementación sobre una plataforma tecnológica.

bytecode

El bytecode es un código intermedio más abstracto que el código máquina. Habitualmente es tratado como un archivo binario que contiene un programa ejecutable similar a un módulo objeto, que es un archivo binario producido por el compilador cuyo contenido es el código objeto o código máquina.

clase

Las clases son declaraciones o abstracciones que describen un conjunto de objetos que comparten las mismas características, limitaciones y semántica.

classpath

En el lenguaje de programación Java se entiende por Classpath una opción admitida en la línea de órdenes o mediante variable de entorno que indica a la Máquina Virtual de Java dónde buscar paquetes y clases definidas por el usuario a la hora de ejecutar programas.

Common Warehouse Metamodel (CWM)

Es un conjunto de interfaces estándar utilizado para el intercambio de metadata entre herramientas, plataformas y repositorios en ambientes warehouse heterogéneos y distribuidos.

diagrama de clase

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.

diagrama de secuencia

Un diagrama de secuencia muestra la interacción de los mensajes que se intercambian, junto con las ocurrencias de los eventos correspondientes sobre la línea de vida del objeto.

estereotipo

Una clase que define como una metaclase (o estereotipo) se puede extender, y permite el uso de la plataforma o el dominio de una terminología o notación además de las utilizadas por la metaclase extendida. Algunos estereotipos están predefinidos en UML, otros pueden ser definidos por el usuario. Los estereotipos son uno de los mecanismos de extensibilidad de UML.

eXtreme programming

La programación extrema es un proceso ágil de desarrollo de software que se adapta a los cambios de requisitos en cualquier punto de la vida del proyecto al considerar que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos.

framework

Un framework es una estructura de soporte definida, mediante la cual otro proyecto de software puede ser organizado y desarrollado. Provee una estructura y una metodología de trabajo la cual extiende o utiliza las aplicaciones del dominio.

Lenguaje de Marcas Extensible (en inglés Extensible Markup Language, XML)

Es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C) que permite definir la gramática de lenguajes específicos.

Lenguaje Unificado de Modelado (en inglés Unified Modeling Language, UML)

Un lenguaje estándar (OMG) que sirve para especificar la estructura y el comportamiento de los sistemas. La norma define una sintaxis abstracta y una sintaxis gráfica concreta.

mapeo

Especificación de un mecanismo para la transformación de los elementos de un modelo conforme a un metamodelo en elementos de otro modelo que se ajusta a otro (posiblemente el mismo) metamodelo.

Máquina Virtual Java (en inglés Java Virtual Machine, JVM)

Es un programa nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (el Java bytecode), el cual es generado por el compilador del lenguaje Java.

metadatos

Representa los modelos de datos. Por ejemplo, un modelo de UML, un esquema de base de datos relacional expresado utilizando metamodelo.

metaclase

Una clase de la cual sus instancias son clases. Las metaclases son típicamente utilizadas para construir metamodelos.

meta-metamodelo

Es modelo que define el lenguaje para expresar el metamodelo. La relación entre el meta-metamodelo y el metamodelo es análoga a la relación entre un metamodelo y un modelo.

metamodelo

Un metamodelo es un modelo que define el lenguaje para expresar un modelo.

Meta Object Facility (MOF)

Es un estándar de OMG, estrechamente relacionado con UML, que permite la gestión de metadatos y la definición del lenguaje.

modelo

Una especificación formal de la función, estructura y/o comportamiento de una aplicación o sistema.

Modelo Específico de la Plataforma (en inglés Platform Specific Model, PSM)

Un modelo de un subsistema que incluye información acerca de la tecnología específica que se utiliza sobre una plataforma en particular, y por lo tanto posiblemente contiene elementos que son específicos de la plataforma.

Modelo Independiente de plataforma (en inglés Platform Independent Model, PIM)

Un modelo de un subsistema que no contiene información específica de la plataforma, o la tecnología que se utiliza.

Object Management Group (OMG)

Es un consorcio dedicado al establecimiento de estándares de tecnologías orientadas a objetos. Actualmente también provee los estándares de modelado.

perfil

Un paquete estereotipado que contiene los elementos del modelo que se han adaptado para un dominio específico o propósito utilizando los mecanismos de extensión, tales como los estereotipos, las definiciones y restricciones. Un perfil puede también especificar librerías de modelos de la que depende y el subconjunto del metamodelo que extiende.

perfil UML

Un conjunto normalizado de extensiones y restricciones que adapta UML a determinado uso particular.

plataforma

Una plataforma es un conjunto de subsistemas/tecnologías que proporcionan un conjunto coherente de funcionalidades a través de las interfaces y patrones de uso especificados que cualquier subsistema que depende de la plataforma puede utilizar sin preocuparse por los detalles de cómo la funcionalidad proporcionada por la plataforma fue implementada.

Swing

Es una biblioteca gráfica para Java que incluye widgets para interfaz gráfica de usuario tales como cajas de texto, botones, desplegables y tablas.

widget

Es una pequeña aplicación o programa, usualmente presentado en archivos pequeños que son ejecutados por un motor de *widgets* o Widget Engine; facilitan el acceso a funciones frecuentemente usadas además de proveer de información visual.

World Wide Web Consortium (W3C)

El World Wide Web Consortium es un consorcio internacional que produce recomendaciones para la World Wide Web. Está dirigida por Tim Berners-Lee, el creador original de URL (Uniform Resource Locator, Localizador Uniforme de Recursos), HTTP (HyperText Transfer Protocol, Protocolo de Transferencia de HiperTexto) y HTML (Lenguaje de Marcado de HiperTexto) que son las principales tecnologías sobre las que se basa la Web.

XMI

Es un estándar de la OMG que mapea MOF a XML. Define como deben emplearse las etiquetas XML que son utilizadas para representar modelos MOF serializados en XML.

BIBLIOGRAFIA

[AGEDIS] http://www.agedis.de/.

[AGEDIS 01] El-Far, I., Whittaker, J.: Model-based software testing. In Marciniak, J., Encyclopedia on Software Engineering. Wiley, 2001.

[AGEDIS 04] Hartman, A., Nagin, K.: The AGEDIS tools for model based testing. In: UML Satellite Activities, 2004, 277–280.

[Ant] Apache Ant, http://ant.apache.org/index.html.

[Antlr] http://www.antlr.org/.

[Art Testing] Myers, G., The Art of Software Testing, Wiley John and Sons, 2004.

[Booch 04] Booch, G. et al. "An MDA Manifesto". In Frankel, D. and Parodi J. (eds) The MDA Journal: Model Driven Architecture Straight from the Masters, 2004.

[Bei 90] B. Beizer: Software Testing Techniques, Van Nostrand Reinhold, 1990.

[Bei 95] Beizer, B., Black-Box Testing: Techniques for functional testing of software and systems. John Wiley & Sons, Ltd., 1995.

[CBOP-DSTC-IBM] CBOP/DSTC/IBM: MOF Query/Views/Transformations, 2nd Revised Submission (ad/04-01-06). OMG. 2004.

[CWM] Common Warehouse Metamodel (CWM™), http://www.omg.org/technology/documents/formal/cwm.htm.

[DDH 72] Dahl, O., Dijkstra, E., Hoare C., "Structured Programming." Academic Press, New York, 1972.

[EasyMock] http://www.easymock.org.

[Eclipse] The Eclipse Project. Home Page. Copyright IBM Corp, 2000. http://www.eclipse.org.

[EMF] Eclipse Modeling Framework EMF. http://www.eclipse.org/modeling/emf/.

[ESC 2] ESCJava2, http://kind.ucd.ie/products/opensource/ESCJava2/.

[From-U2TP-models-to-executable-tests-with-TTCN-3] Zander J., Dai Z., Schieferdecker I., Din G., From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing, 2005.

[GROOVE] http://groove.sourceforge.net/groove-index.html.

[Intro U2TP] Lidia, F., Vallecillo A., Una Introducción a los Perfiles UML, Novática, 2004.

[Java] http://java.sun.com.

[JML 2] JML, http://www.cs.ucf.edu/~leavens/JML/.

[JPF 2] Java PathFinder, http://javapathfinder.sourceforge.net/.

[JUnit] JUnit testing framework, http://www.junit.org/.

[Korat] Boyapati, C., Khurshid, S., Marinov, D., Korat: Automated Testing Based on Java Predicates, MIT Laboratory for Computer Science. 200 Technology Square Cambridge, MA 02139 USA, 2002.

[MDAG] MDA Guide Version 1.0.1, http://www.omg.org/docs/omg/03-06-01.pdf.

[MDT] Utting, M., Legeard, B., Practical model-based testing, a tools approach. Morgan Kaufmann Publishers, 2007.

[Model-Driven-Testing-U2TP] Baker P., Dai Z., Grabowski J., Haugen O., Schieferdecker I., Williams C., Model-Driven Testing - Using the UML Testing Profile, Springer, 2008.

[MDT-Dai] Zhen Ru Dai, Model-Driven Testing with UML 2.0, Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany. Descargado de http://www.cs.kent.ac.uk/projects/kmf/mdaworkshop/submissions/Dai.pdf en marzo 2009.

[MDT-Dai extension] Lamancha, B., Mateo, P., Garcia-Rodriguez, I., Usaola, M., Propuesta para pruebas dirigidas por modelos usando el perfil de pruebas de UML 2.0, Universidad de la República de Uruguay, Uruguay / Universidad de Castilla-La Mancha, España, 2008. Descargado de http://www.sistedes.es/TJISBD/Vol-2/No-4/articles/pris-08-perez-perfilUML.pdf en marzo 2009.

[Model-Transformation] Duddy, K., Gerber, A., Lawley, M., Raymond, K., Steel, J.: Model Transformation: A declarative, reusable patterns approach. (In: 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)) pp. 174–185

[MOF] Meta Object Facility (MOF) 2.0 Core Specification. OMG, http://www.omg.org/docs.

[Oldevik 06] Oldevik, Jon. MOFScript User Guide, Version 0.6 (MOFScript v 1.1.11), 2006. Descargado de http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf en enero 2009.

[OMG] Object Management Group (OMG). http://www.omg.org/.

[PGP 08] Pons, C., Giandini R., Pérez G., Desarrollo de Software Dirigido por Modelos, Conceptos teóricos y su aplicación práctica, Facultad de Informática, Universidad Nacional de La Plata, 2008.

[PG 08] Pons, C., Garcia, D., A Lightweight Approach for the Semantic Validation of Model Refinements. Electronic Notes in Theoretical Computer Science (ENTCS). ELSEVIER. ISSN 1571-0661, 2008. Descargado de http://www.lifia.info.unlp.edu.ar/papers/2008/Pons2008.pdf en enero 2009.

[PS] Pruebas de software. Consultado de http://es.wikipedia.org/wiki/Pruebas_de_software en enero 2009.

[QVT] Query/View/Transformation (QVT) Specification. Final Adopted Specification ptc/07-07-07. OMG, 2007.

[Towards Model-Driven Testing] Heckel, R., Lohmann, M., Towards Model-Driven Testing, University of Paderborn, Paderborn, Germany, 2003. Descargado de http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/TACos03-Heckel-Lohmann.pdf en enero 2009.

[U2TP 04] Consortium: UML 2.0 Testing Profile, Final Adopted Specification at OMG (ptc/04-04-02), 2004.

[UL 07] Utting, M., Legeard, B., Practical model-based testing, a tools approach. Morgan Kaufmann Publishers, 2006.

[UML 05] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification, formal/2005-07-04. Descargado de http://www.omg.org en enero 2009.

[UML IL] UML 2.0. The Unified Modeling Language Infrastructure version 2.0, OMG Final Adopted Specification. March 2005. Descargado de http://www.omg.org en enero 2009.

[XMI] XML Metadata Interchange Specification, http://www.omg.org/technology/documents/formal/xmi.htm.

[XML] XML Value Type Specification, http://www.omg.org/technology/documents/formal/xmlvalue.htm.

[XP] eXtreme Programming, http://www.extremeprogramming.org/.