Automatización del proceso de generación automática de modelos de servicios en SoaML desde modelos de Procesos de Negocio en BPMN 2.0 con QVT

Informe de Proyecto de Grado

Miguel Merlino
Diego Bortot

Tutor: Dra. Andrea Delgado

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo – Uruguay

Abril de 2015

Resumen

El Desarrollo Dirigido por Modelos (Model-Driven Development, MDD) se basa en la generación y utilización de modelos como artefactos de primer orden durante todo el ciclo de desarrollo de software, y en fomentar y facilitar su reutilización.

La Arquitectura Dirigida por Modelos (Model-Driven Architecture, MDA) es el marco de trabajo adoptado por el Object Management Group (OMG) con el objetivo de implementar las ideas de MDD. Está compuesto por un conjunto de estándares, entre los cuales se encuentra el *Query/View/Transformation* (QVT) que define los lenguajes para consulta y trasformación de modelos.

Para el desarrollo y la adopción de este paradigma es necesario contar, entre otras cosas, con herramientas que den soporte a la metodología, y que automaticen y faciliten las tareas asociadas con el proceso, fundamentalmente la automatización de las transformaciones. La plataforma Eclipse y el Eclipse *Modeling Framework* (EMF) proveen muchas de estas herramientas, y un ambiente extensible que permite por medio de plugins agregar funcionalidades.

El presente proyecto describe la implementación de un plugin de Eclipse que facilita la utilización de motores de transformación QVT *Relations* (QVTr) abstrayendo al usuario de las particularidades de configuración y ejecución de cada motor. Ofrece una interfaz intuitiva y a la vez flexible, siendo adecuada tanto para usuarios asiduos como esporádicos.

Si bien el plugin fue desarrollado con el objetivo de soportar la transformación de modelos de procesos de negocio en *Business Process Model and Notation* (BPMN 2.0) a modelos de servicio en *Service Oriented Architecture modeling Language* (SoaML), puede ser utilizado para transformar cualquier modelo que pueda serializarse a *XML Metada Interchange* (XMI). El formato del resultado de la trasformación será XMI, por lo que pueden producirse modelos de cualquier tipo que pueda ser de-serializado de XMI.

También se ofrece la posibilidad de extender el plugin para usarlo con nuevos motores (incluso motores no QVTr), agregando implementaciones específicas a las interfaces de extensión proporcionadas. El plug-in desarrollado fue validado mediante casos de estudio con modelos reales integrando dos motores QVT distintos: MediniQVT y Modelmorf, lo que permitió demostrar las capacidades de extensión y soporte del mismo.

Como trabajo futuro se plantean tres posibles mejoras al plugin: instalación/detección de motores de transformación, ejecución de transformaciones en tiempo real, y agregado de transformación de modelos de salidas. Se analizan y plantean algunas consideraciones para llevarlas a cabo.

Palabras Clave: MDD, MDE, MDA, BPMN, SoaML, Eclipse, QVT, Medini, ModelMorf.

Tabla de contenidos

1. Introducción

El presente documento es parte del proyecto de grado para la carrera de Ingeniería en Computación de la Udelar, propuesto por el grupo COAL (Componentes, Objetos, Arquitecturas, Lenguajes) en la línea de investigación “Ingeniería y desarrollo dirigido por modelos”.

1.1 Contexto y motivación

Para el desarrollo y la adopción del paradigma de MDD y la aplicación de los estándares de MDA [1] es necesario contar, entre otras cosas, con herramientas que den soporte a la metodología y los estándares, y que automaticen y faciliten las tareas asociadas con el proceso. La plataforma Eclipse y el Eclipse *Modeling Framework* (EMF) proveen muchas de estas herramientas, y un ambiente extensible que permite por medio de plugins agregar funcionalidades. Algunas empresas u organizaciones han realizado implementaciones de los estándares MDA, en particular de motores de transformación para lenguajes QVTr. A su vez, algunos de estos se implementaron como plugins de Eclipse. Sin embargo, cada uno tiene sus particularidades a la hora de configurar y ejecutar las transformaciones.

Por otro lado, la implementación de procesos de negocio (PN) se realiza frecuentemente en base a servicios y Web Services. La OMG dispone del estándar BPMN 2.0 para el modelado de PN´s, y de SoaML (perfil de UML) para el modelado de servicios. Ambos conforman con el estándar de la OMG *Meta Object Facility* (MOF) para la definición de metamodelos, y por lo tanto pueden ser relacionados por transformaciones QVT. Automatizar la transformación entre estos modelos es de suma importancia para facilitar la vinculación entre el espacio del problema y el espacio de la solución.

1.2 Objetivos

El objetivo general de este proyecto es desarrollar un plug-in de Eclipse que soporte el proceso de generación automática de modelos de servicios en SoaML desde modelos de Procesos de Negocio en BPMN 2.0, integrando diversas transformaciones QVTr ya definidas, incluyendo la evaluación de motores QVTr existentes adecuados al entorno Eclipse.

Los objetivos particulares definidos para el proyecto son los siguientes:

1) facilitar la aplicación de transformaciones entre modelos BPMN 2.0 y modelos SoaML soportando el proceso definido para la definición y ejecución de las mismas

2) evaluar motores QVTr existentes para posibilitar el uso de cualquier motor de transformación QVTr, abstrayendo al usuario de las particularidades de su funcionamiento y configuración.

3) desarrollar un plug-in de Eclipse para la automatización de la generación automática de modelos utilizando distintos motores QVTr

4) validar el funcionamiento del plug-in desarrollado mediante su utilización en un caso de estudio para la transformación de modelos BPMN 2.0 a SoaML con al menos dos motores QVTr

1.3 Cumplimiento de objetivos y aportes del proyecto

El plugin desarrollado cumple con los objetivos propuestos de facilitar la aplicación de

transformaciones, y la utilización para ello de diferentes motores QVTr.

Así mismo se logró brindar una interfaz consistente, intuitiva y flexible para realizar estas operaciones en el ambiente Eclipse.

Si bien el plugin fue desarrollado con el objetivo de transformar modelos BPMN a SoaML, puede ser utilizado para transformar cualquier modelo de origen que pueda serializarse a XMI a cualquier modelo de destino en formato XMI. También es factible extender el plugin para ejecutar transformaciones en motores no QVTr.

1.4 Desarrollo del proyecto

El proyecto fue desarrollado entre los meses de abril de 2014 y abril de 2015.El cronograma planteado originalmente para el proyecto era el siguiente:

Abril, Mayo, Junio:

- Estudio del estado del arte en desarrollo dirigido por modelos, así como modelado de procesos de negocio y servicios, y estándares existentes para modelado y transformaciones entre modelos
    - estudio de la teoría de metamodelado, perfiles UML, estándar XMI para intercambio de modelos
    - estándar BPMN 2.0 para modelado de procesos de negocio y SoaML para modelado de servicios
    - estandares MDA para desarrollo dirigido por modelo, tipos y niveles de modelos, y QVT para transformaciones
- Estudio del entorno Eclipse y desarrollo de plug-ins, motores QVT relacional para Eclipse y elementos de la definición de transformaciones QVT existente

Julio, Octubre:

- Definición e Implementación del plug-in de Eclipse para la automatización de la generación automática de modelos de servicios en SoaML desde modelos de Procesos de Negocio en BPMN 2.0 con QVT
- Requerimientos (Casos de Uso), definición y priorización
- Caso de estudio de aplicación del soporte a la generación automática desde modelos BPMN 2.0 a modelos SoaML con el plug-in Eclipse desarrollado, que podrá ser de laboratorio (ejemplo del estándar de BPMN 2.0) o real de alguna organización afín

Noviembre, Diciembre:

- Realización del informe final del proyecto de grado, correcciones y defensa

El cronograma fue respetado en la primera etapa, hasta Julio. Luego de esto surgen sucesivas demoras que imposibilitaron finalizar el informe para diciembre de 2014. Algunas de las causas de estas demoras fueron: problemas para la pruebas de transformaciones en distintos motores debidas a diferencias de sintaxis; escasa documentación de estas diferencias, y en general del funcionamiento de los motores; escasa documentación de algunas funcionalidades de Eclipse que se pretendía usar para el proyecto (como las instalación programática de nuevo software); problemas en las

instalación de motores, por incompatibilidad de versiones entre los motores y sus dependencias; aparición de nueva literatura que fue revisada e incorporada al informe.

En Marzo de 2015 se retomaron las actividades y se realizaron las correcciones al informe, correcciones y nuevas pruebas al plugin y nuevos ejemplos de funcionamiento.

Para el desarrollo se utilizó el repositorio SVN Assembla[1] para el código del plugin, y una carpeta compartida en Google Drive para los documentos y material.

1.5 Estructura del documento

El presente documento se organiza de la siguiente manera: En el Capítulo 2 se introduce el marco teórico y las herramientas que dan base al desarrollo. En el Capítulo 3 se establecen los requisitos para el plugin. En el Capítulo 4 se describe la solución propuesta, se analizan las técnicas utilizadas, y las consideraciones durante el desarrollo. En el Capítulo 5 se describe el proceso de verificación. El Capítulos 6 presenta los casos de estudio normales y de extensión respectivamente. Finalmente en el Capítulo 7 se presentan las conclusiones y el trabajo futuro.

1 https://www.assembla.com

2. Marco teórico

En las siguientes secciones se presentan los conceptos, definiciones y herramientas relevantes para el desarrollo del proyecto.

2.1 Model-Driven Development / Engineering (MDD/MDE)

Podemos ver un modelo como una representación de un sistema o un aspecto de este, usualmente simplificada, que no necesita ser completa, cuyo propósito puede ser facilitar la comprensión y análisis del sistema que modela, facilitar la comunicación entre actores, estimar costos, validar decisiones de diseño, etc. Un sistema se puede representar por un conjunto de modelos que capturan cada uno un aspecto específico de este.([3],[4])

Un metamodelo es una clase especial de modelo que define la estructura, la semántica, y las restricciones que deben cumplir los modelos que se deriven de éste [25]. Cada lenguaje de modelado tiene un metamodelo que define su sintaxis abstracta, la semántica de sus artefactos, y como se relacionan entre sí.

El modelado en el proceso de desarrollo de software es realizado en general como documentación del sistema que difícilmente es mantenida luego de implementado y liberado el mismo. El Desarrollo Dirigido por Modelos (Model Driven Development, MDD) basa el desarrollo de software en modelos, utilizando como artefactos de primer orden metamodelos, modelos y lenguajes que permiten transformaciones entre éstos. Estas transformaciones convierten sucesivamente un modelo en otro modelo del mismo sistema refinando el nivel de abstracción hasta llegar al código asociado. “MDD es simplemente la noción de que se puede construir un modelo de un sistema, que luego se puede transformar en el sistema mismo.” [3]

Aunque muchas veces se usan MDD y MDE indistintamente, MDE se puede ver como un concepto más amplio, que incluye a MDD: “MDD se enfoca en la generación de implementaciones a partir de modelos. En contraste, MDE incluye otros usos de modelos precisos para soportar el proceso de desarrollo, como ingeniería inversa dirigida por modelos y evolución dirigida por modelos” [6]. En [4] en la sección “*Some open research issues in MDE*”, se hace referencia otras incumbencias de MDE, como por ejemplo: el estudio de la relación entre los modelos y los sistemas que estos representan; la teoría de modelado; las relaciones entre meta-modelos (por ejemplo la extensión) y entre sus elementos; la identificación y definición de las posibles operaciones sobre modelos y meta-modelos, por ejemplo, combinación, weaving, diferencia, métrica, alineación entre meta-modelos, etc.; el manejo de los problemas derivados de la evolución y versionado de modelos y meta-modelos; el estudio de las relaciones semánticas entre meta-modelos, y las posibilidades de transponer conocimiento generado en un área de dominio a otra, e incluso generar nuevo conocimiento (esta idea esta presente en las ontologías, que se asemejan fuertemente con los meta-modelos).

Algunas de las motivaciones u objetivos de MDD/E son: generar técnicas y herramientas que permitan el modelado tanto en el espacio del problema como de la solución, y la articulación entre ellos definiendo relaciones precisas y transformaciones; facilitar la reutilización del conocimiento generado, permitiendo la migración a diferentes plataformas o dominios; mejorar la productividad automatizando tareas de transformación y generación de artefactos ejecutables. Según [5] “Las tecnologías MDE ofrecen un enfoque prometedor para remediar la inhabilidad de los lenguajes de tercera generación para mitigar la complejidad de las plataformas y expresar eficazmente conceptos del dominio”

Se pueden identificar 3 aspectos principales de MDE: principios, estándares y herramientas. En la literatura se citan algunos ejemplos de iniciativas que apuntan a adoptar, desarrollar, o implementar alguna combinación de estos aspectos: Model-Centric Software Development (MCSD, Lookheed Martin)[5], Software Factories (Microsoft)[27], Model Driven Architecture (MDA, marco de estándares de OMG)[1], Generic Modeling Environment (GME, Institute for Software Integrated Systems)[28], Eclipse Modeling Framework[29], MINERVA (Universidad de Castilla-La Macha) [17], Model-based systems engineering (MBSE)[30], MOD Reseach Group[31] y una extensa lista de herramientas que pueden ser encontradas en Internet con diversas funcionalidades de modelado, generación, importación/exportación, transformación, etc.

En [6] los autores analizan los resultados de un estudio sobre la adopción de MDE en la industria. A pesar de que existen casos de fracaso, señalan que la adopción de MDE es generalizada, y su aplicación es muy variada, yendo desde esfuerzos a nivel industria para definir modelos precisos para cierto dominio de aplicación, hasta un uso muy restringido para la generación del código para una única familia de aplicaciones en una compañía.

Algunas de las observaciones que realizan son:

- los factores organizacionales, sociales y psicológicos son tan importantes para la adopción como los técnicos.
- son más exitosos los casos en que la adopción se da de abajo hacia arriba, es decir, que los desarrolladores empiezan a experimentar y usar conceptos y herramientas MDE cuando lo creen conveniente o útil. Los casos de imposición desde la gerencia tienden a fracasar.
- las organizaciones usan MDE como y cuando es apropiado, y en combinación con otros métodos de maneras muy flexibles, en lugar de seguir un enfoque metodológico estricto o pesado.
- la adopción es más exitosa en empresas que apuntan a un dominio en particular, que en empresas que desarrollan software más general.
- la generación de código no es la motivación mayor para la adopción. Las ventajas percibidas apuntan a la mejora en la documentación, y construcción incremental de definiciones arquitectónicas, que los desarrolladores van construyendo al identificar fragmentos similares de código que usan habitualmente.
- el éxito requiere una motivación comercial más allá de la promesa de MDE de poder hacer las cosas más rápido y a menor costo. Usualmente esto no convence a las empresas de tomar el riesgo de adopción de la metodología. Las empresas que adoptan MDE lo hacen porque les permite desarrollos que no podrían realizar de otra forma.
- las técnicas y herramientas más modernas no hacen un buen trabajo en el soporte a las actividades del desarrollo de software. No hay consenso en cuanto a las herramientas y lenguajes usados por las empresas. UML se utiliza en forma parcial o informal, o directamente no se utiliza. Solo dos empresas de las entrevistadas dicen usar MDA.

2.2 Model-Driven Architecture (MDA)

Model-Driven Architecture (MDA) es el marco de trabajo adoptado en 2001 por la Object Management Group (OMG) que puede ser definida como “la realización de los principios de MDE

alrededor de un conjunto de estándares de la OMG" [4]. En el 2003 se publicó la "MDA Guide 1.0" que establece el marco conceptual que luego es plasmado en los estándares y especificaciones propuestas por la OMG. La versión actual es la 2.0 de junio de 2014 [2].

El objetivo principal de MDA es el de valorizar los modelos y el modelado para facilitar el manejo de la complejidad e interdependencia de sistemas complejos. Para esto los estándares de MDA definen la *estructura*, la *semántica*, y la *notación* de los modelos. Los modelos que siguen estas definiciones son llamados "modelos MDA" y puede entonces ser utilizados para producir documentación, para relevar y especificar sistemas, para generar implementaciones y ejecutables, pueden ser intercambiados entre distintas herramientas, transformados, consultados y presentados de diferentes maneras. También se definen estándares para el intercambio a través de serialización a XML.

MDA permite el uso de diferentes lenguajes de modelado adecuados al dominio, y la representación de un sistema a diversos niveles de abstracción y puntos de vista, así como la vinculación entre todos ellos.

Los conceptos más importantes de MDA son los diferentes tipos de modelos y sus niveles de abstracción, y las transformaciones entre ellos.

Los modelos son separados principalmente en 2 tipos:

- **Plataform independent Model (PIM)** Es un modelo que representa al sistema (o una parte de este) independiente de una plataforma particular. Es decir, se puede transformar a cualquier plataforma tecnológica específica, de un nivel de abstracción menor.
- **Plataform specific Model (PSM)** El PSM representa al sistema de una manera más específica con respecto a la plataforma de implementación.

El concepto de plataforma puede existir en varios niveles de abstracción. Por ejemplo, se pueden ver los Web Services como una plataforma de implementación para un modelo de procesos de negocios, y a su vez, los Web Services pueden ser implementados en las plataformas Java o .Net. De la misma manera, el lenguaje C puede ser considerado un modelo independiente de la plataforma de implementación, que puede ser un procesador Intel o uno ARM. Por esto, los conceptos de PIM/PSM pueden verse en todos los niveles de abstracción de un sistema. Históricamente la MDA se refiere a los modelos de negocios puros como Computation Independent Model (CIM).

Las transformaciones son definidas para pasar de un nivel de abstracción (PIM) a otro más bajo (PSM), o bien, para obtener una nueva representación con un nivel de abstracción similar al del modelo de origen. De esta forma un modelo puede ser reutilizado para producir documentación para los distintos *stakeholders*, o PSMs para varias plataformas.

La OMG propone y adopta varios estándares que están estrechamente relacionados con la implementación de los conceptos de MDA, por ejemplo, MOF, UML, BPMN, QVT, XMI. En las secciones siguientes introduciremos algunos de ellos.

2.2.1 Meta Object Facility (MOF)

Un Meta-modelo es una clase especial de modelo que define la estructura, la semántica, y las restricciones que deben cumplir los modelos que se deriven de éste.

Cada lenguaje de modelado (UML, BPMN, etc) tiene su meta-modelo. Para poder relacionar estos meta-modelos, la OMG adopto la especificación MOF [7] . MOF es un lenguaje simple para definir meta-modelos. Todos los meta-modelos que conformen con MOF, y sus modelos derivados, pueden ser procesados en el ecosistema de MDA. Es decir, MOF es el meta-lenguaje unificador de todos los meta-lenguajes y lenguajes de modelado que quieran integrar la MDA. Al ser una especificación abierta, cualquier organización, no solo la OMG, puede definir lenguajes o herramientas que conformen con MOF, y de esta forma permitir su integración.

La OMG define una arquitectura de cuatro capas, como se ve en la Figura 2.1. En la capa M0 se ubica el sistema a representar, las instancias de la realidad modelada. En la capa M1 se ubica el modelo de ese sistema, por ejemplo, un diagrama de clases de UML. En la capa M2 se encuentran los meta-modelos de UML o BPMN, entre otros. Y en la capa M3 se encuentra MOF, lo que brinda una alto grado de libertad y flexibilidad para definir nuevos lenguajes, sin perder interoperatividad.

MOF esta basado en un subconjunto de UML. Ambos comparten el mismo meta-modelo, al que se agregan restricciones *Object Constraint Lengauge* (OCL) para definir el subconjunto de elementos relevantes para MOF.

El estándar establece 2 niveles de conformidad con MOF:

- Essential MOF (EMOF): es un subconjunto de MOF que se corresponde con las estructuras encontradas en los lenguajes orientados a objetos. Su objetivo principal es permitir definir meta-modelos simples que luego pueden ser extendidos hacia CMOF, de esta forma facilitando la integración de herramientas con el estándar.
- Complete MOF (CMOF): es la especificación más completa de MOF, y es la utilizada para definir, por ejemplo, el meta-modelo de UML.

La versión actual de MOF es la 2.4.2 de abril del 2014, adoptada como estándar internacional ISO/IEC 19508.

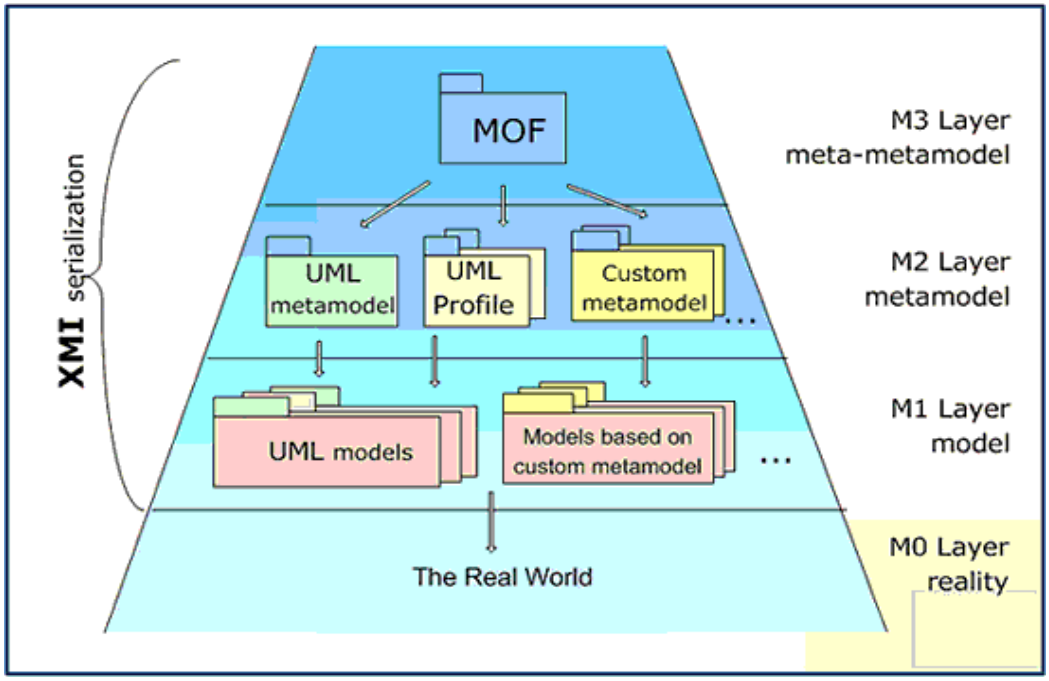


Figura 2.1: Arquitectura MDA

2.2.2 Query/View/Transformation (QVT)

Un pilar fundamental de MDD/E y MDA son las transformaciones. Estas permiten la navegabilidad entre los diferentes modelos que representan un sistema, tanto verticalmente (de la especificación abstracta al código ejecutable) como horizontalmente (diferentes aspectos o vistas de un sistema, con un nivel de abstracción similar).

La OMG adopta el estándar QVT [8] para definir consultas, vistas y transformaciones entre modelos. En este se proponen la arquitectura, los lenguajes, las mapeos operacionales y el núcleo del lenguaje de la especificación. QVT también se define como lenguaje/modelo que conforma con MOF como se ve en la Figura 2.2. Allí se ven los modelos de origen y destino (Ma y Mb), el modelo de la transformación (Mt), y los meta-modelos con los que conforma cada uno (MMa, MMb y QVT respectivamente) que a su vez conforman con MOF.

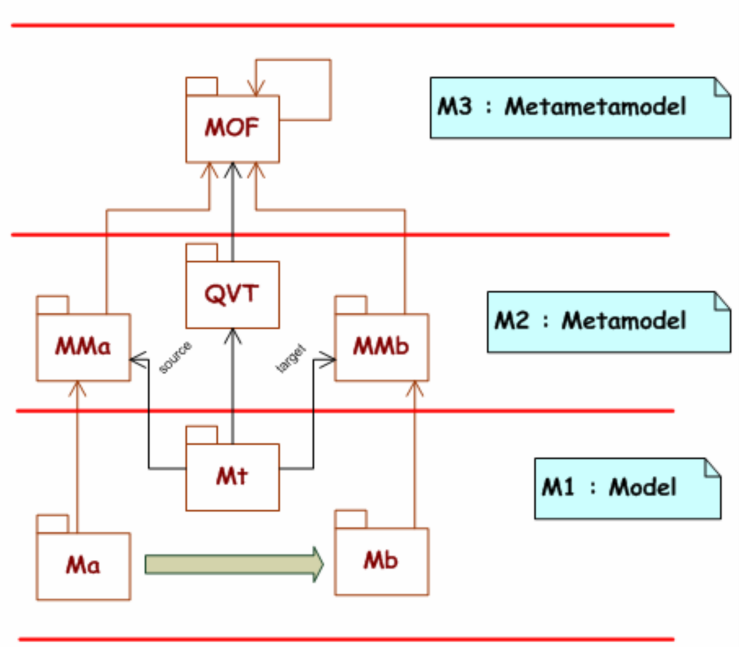


Figura 2.2: QVT en arquitectura MDA

QVT esta formado por tres lenguajes de transformación relacionados: 2 lenguajes declarativos (Relations y Core) y un lenguaje imperativo (Operational Mappings). Esto le brinda a QVT flexibilidad a la hora de expresar las transformaciones, que pueden aprovechar las facilidades que brinda uno u otro paradigma de acuerdo a las necesidades. Los lenguajes operan sobre modelos que conformen con el meta-modelo MOF, y se organizan en una arquitectura de 2 niveles que se puede ver en la Figura 2.3.

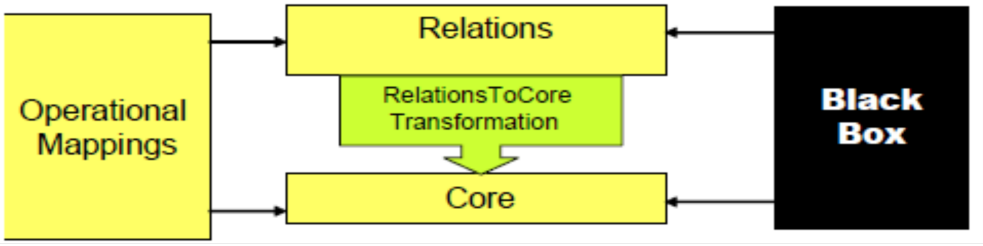


Figura 2.3: Arquitectura QVT

Además de estos 3 lenguajes estándar, la especificación provee un mecanismo no estándar llamado Black-Box que permite escribir transformaciones en cualquier lenguaje de programación que pueda ser vinculado a MOF, directa o indirectamente a través de otro lenguaje. Esto abre la posibilidad de aplicar algoritmos complejos a las transformaciones, o utilizar bibliotecas especificas de dominio (científico, matemático, de negocios, etc) para calcular propiedades de los modelos u otras operaciones. También permite ocultar los detalles de una transformación. Esto le da a la arquitectura una enorme flexibilidad, pero conlleva el riesgo de dar al plug-in que lo implemente acceso a referencias de objetos en los modelos, para ejecutar operaciones arbitrarias, que no son implícitamente trazables.

QVT actualmente se encuentra en la versión 1.2 de febrero del 2014.

2.2.3 XML Metada Interchange (XMI)

Otro aspecto fundamental de la concepción MDA, y tal vez decisivo para su adopción, es la posibilidad de intercambiar modelos libremente entre herramientas de distintos proveedores. La OMG propone para esto XMI.

XMI es un estándar para serializar modelos y meta-modelos que conformen con MOF a XML. Esto facilita el almacenamiento, transmisión e intercambio de modelos. El estándar define los siguientes aspectos:

- la representación de objetos mediante elementos y atributos de XML
- un mecanismo estándar para vincular objetos dentro del mismo archivo XML o en otros archivos
- la validación de documentos XMI usando esquemas XML
- Identificación de objetos, lo que permite que se referencien unos a otros mediante sus identificadores (IDs o UUIDs[2])

La adopción del estándar es variada, y no todas las herramientas que lo soportan se ajustan fielmente a las especificaciones, o las implementan completamente. Además, las diferentes versiones del estándar tienen diferencias que usualmente se manejan ofreciendo al usuario que seleccione que versión de XMI va a importar o exportar. También existe una correspondencia entre el lenguaje usado y la versión XMI que lo serializa, por ejemplo, a UML 1.3 le corresponde XMI 1.0 o 1.1, a UML 1.4 los XMI 1.2 o 2.0, a UML 2.0 el XMI 2.1, etc, lo cual agrega más complejidad a la implementación de la serialización.

La versión actual de XMI es la 2.4.2 de abril de 2014, también adoptada como estándar internacional ISO/IEC 19509.

2.3 Perfiles UML

Los perfiles UML son una herramienta para extender el lenguaje UML, los cuales permiten construir modelos UML para dominios particulares, en los cuales se aplican estereotipos y valores adicionales a elementos, atributos, asociaciones,etc.

Constituyen un mecanismo para extender la sintaxis y semántica UML para expresar conceptos

2 UUID: Universal Unique Identifier

particulares a un determinado dominio, que el core del lenguaje UML no es capaz de expresar de forma trivial. Esto permite adaptar los elementos de un meta-modelo UML a las necesidades de cierto dominio, como ser una plataforma o dominio de aplicación.

Los perfiles UML logran evitar tener que crear un nuevo lenguaje a partir del MOF (Meta Object Facility) si se quieren agregar nuevos elementos y restricciones al lenguaje UML sin alterar su estructura base.

Por más información, consultar A.2 de anexo.

2.4 Estándares de modelado

2.4.1 BPMN

BPMN (Business Process Model and Notation) es un estándar de modelado desarrollado por la OMG [10]. El principal objetivo de BPMN es proveer una notación que sea fácilmente comprensible para los usuarios de los procesos de negocio, entre ellos analistas que diagraman los principales flujos del negocio, hasta los desarrolladores responsables de llevar a cabo la implementación de esos procesos empleando determinadas tecnologías afines y aquellos responsables de monitorear y gestionar dichos procesos. De esta forma, BPMN constituye un estándar que permite estrechar la brecha entre el diseño de los procesos y su implementación.

2.4.2 SoaML

SoaML (Service Oriented Architecture modeling Language) es un proyecto open source de la OMG para diseñar y modelar servicios en una arquitectura de SOA . Define perfil UML desde un meta-modelo afín para dicho propósito [13].

Por mas información, consultar A.1 y A.3 de anexo.

2.5 Herramientas

2.5.1 Plataforma Eclipse

La plataforma Eclipse permite levantar un entorno integrado de desarrollo (IDE) que no está dedicado a un lenguaje de programación específico. Abarca los aspectos funcionales básicos de cualquier IDE y permite desarrollar aplicaciones a partir de los componentes base de la misma. Esta diseñada para ser fácilmente extensible por terceras partes. En el núcleo de la plataforma esta el SDK de eclipse, y se pueden construir o extender innumerables componentes desde el SDK.

En la figura 2.8 muestra los componentes y las APIs de mayor relevancia de la plataforma.

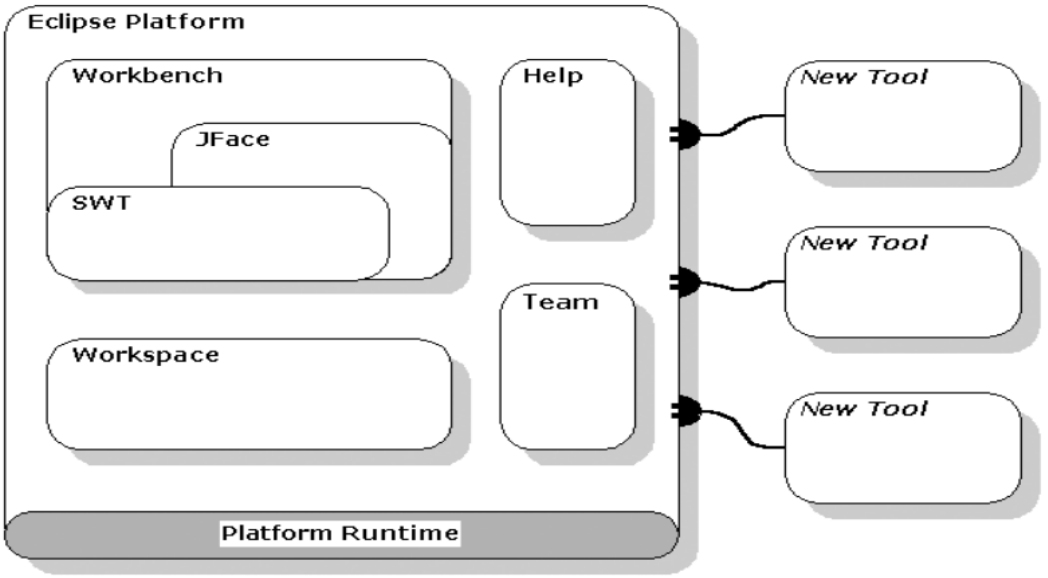


Figura 2.4: Plataforma Eclipse

La principal función de la plataforma Eclipse es proveer mecanismos y reglas de integración a diversos proveedores de software, pudiendo lograr la integración de gran variedad de herramientas de desarrollo disímiles. Desarrollar una herramienta o aplicación sobre la plataforma Eclipse garantiza su eventual integración con otras herramientas y/o aplicaciones de la plataforma.

La estructura de la plataforma comprende un abanico amplio de funcionalidades, entre las más genéricas:

- Construcción de diferentes herramientas de desarrollo de aplicaciones
- Soporte para gran cantidad de herramientas de diverso tipo (desarrollo, modelado, etc.).
- Manipulación y uso de diversos tipos de datos, lenguajes y archivos, tales como HTML, Java, C, JSP (Java Server Pages), EJB (Enterprise Java Beans), XML, GIF, JPEG, texto plano.
- Soporte de entorno de desarrollo de aplicaciones con o sin interfaz gráfica de usuario.
- Portabilidad en la mayoría de los sistemas operativos, incluyendo Windows, Linux, Mac OS, Solaris.
- Uso de Java como lenguaje base de programación. La plataforma puede utilizarse como un Java IDE agregando componentes de Java (como el JDT). Sin embargo, su uso no está limitado sólo al lenguaje Java. Por ejemplo, se puede utilizar como IDE para desarrollo en C/C++ o COBOL importando los plug-ins necesarios que brinden el soporte necesario.
- Facilidad para integración de herramientas atravesando distintos lenguajes, tipos de datos y de contenido

En general, el SDK de eclipse está conformado por los diferentes capas. La capa que provee

herramientas necesarias para construir plugins es el PDE (Plugin Development Environment) [26]. Dichas herramientas permiten crear, desarrollar, testear, debuggear, compilar y deployar plugins de Eclipse. Es también un entorno dentro de Eclipse para desarrollo de componentes en la plataforma Java en general, no solo para el desarrollo de plugins.

Por más detalles, consultarse a la sección B del anexo.

2.5.2 Motores QVT Relacionales

Los motores de transformación son aplicaciones que pueden transformar un modelo de entrada en otro de salida aplicándole un script de transformación. En esta sección se hace una reseña de los motores que pueden ejecutar transformaciones escritas en el lenguaje QVT Relations (QVTr).

Medini QVT [18]

Se distribuye como plug-in para Eclipse o como RCP de Eclipse. Provee un motor de ejecución de transformaciones QVTr que se distribuye con licencia *Eclipse Public Licence* (EPL). También ofrece un editor y un debuger con licencia gratuita para uso no comercial. Este motor fue utilizado para la validación del plugin desarrollado.

La ultima versión disponible es la 1.7 para Eclipse 3.6.

ModelMorf [19] **/QVTR2** [20]

ModelMorf es un motor propietario de la empresa Tata Consultancy Service que ofrece una licencia gratuita para ejecutar la aplicación por linea de comando. En el archivo *readme.html* que se instala con la aplicación se detallan las características no implementadas del estándar QVT, y otras funcionalidades particulares del producto.

Este motor fue utilizado como motor alternativo para la validación del plugin desarrollado.

La última versión es de 2009. Cabe aclarar que el sitio de ModelMorf no esta en linea desde finales del 2014.

QVTR2 es un plug-in de Eclipse que utiliza como motor de transformación a ModelMorf.

QVTd (QVT Declarative) [21].

QVTd es un proyecto de Eclipse enmarcado en el EMF (Eclipse Modeling Framework). Es una implementación parcial de los lenguajes QVTc (Core) y QVTr. El plugin provee modelos, parsers y editores para QVTc/r. La versión 1.0 proyectada para Eclipse Mars será la primera que ejecute QVTr.

La última versión es la 0.11 para Eclipse Luna.

Echo [22]

El proyecto Echo es una herramienta de reparación y transformación de modelos basada en Alloy

(lenguaje de modelado estructurado basado en lógica de primer orden). Se instala como Plugin de Eclipse 4.3. Ejecuta transformaciones QVTr con anotaciones OCL traduciéndolas a Alloy, valiéndose de los componentes de Eclipse QVTd y OCL para el parseo.

Opera en tiempo real, es decir, informa de inconsistencias encontradas durante la edición del modelo. La consistencia se verifica entre el modelo y su metamodelo y/o entre 2 modelos relacionados por una transformación QVTr.

La última versión es la 0.3.1 de abril de 2014.

3. Requisitos del proyecto

En las secciones que siguen se presentan los casos de uso que modelan los requisitos funcionales para el plugin a desarrollar, y sus requisitos no funcionales.

3.1 Casos de uso

Los casos de uso identificados en los requerimientos funcionales para el plug-in a desarrollar son los siguientes:

- Seleccionar motor
- Registrar metamodelo
- Alta de configuración
- Modificación de configuración
- Baja de configuración
- Ejecutar transformación
- Generar xmi

La figura 3.1 muestra el Diagrama de los Casos de Uso detectados.

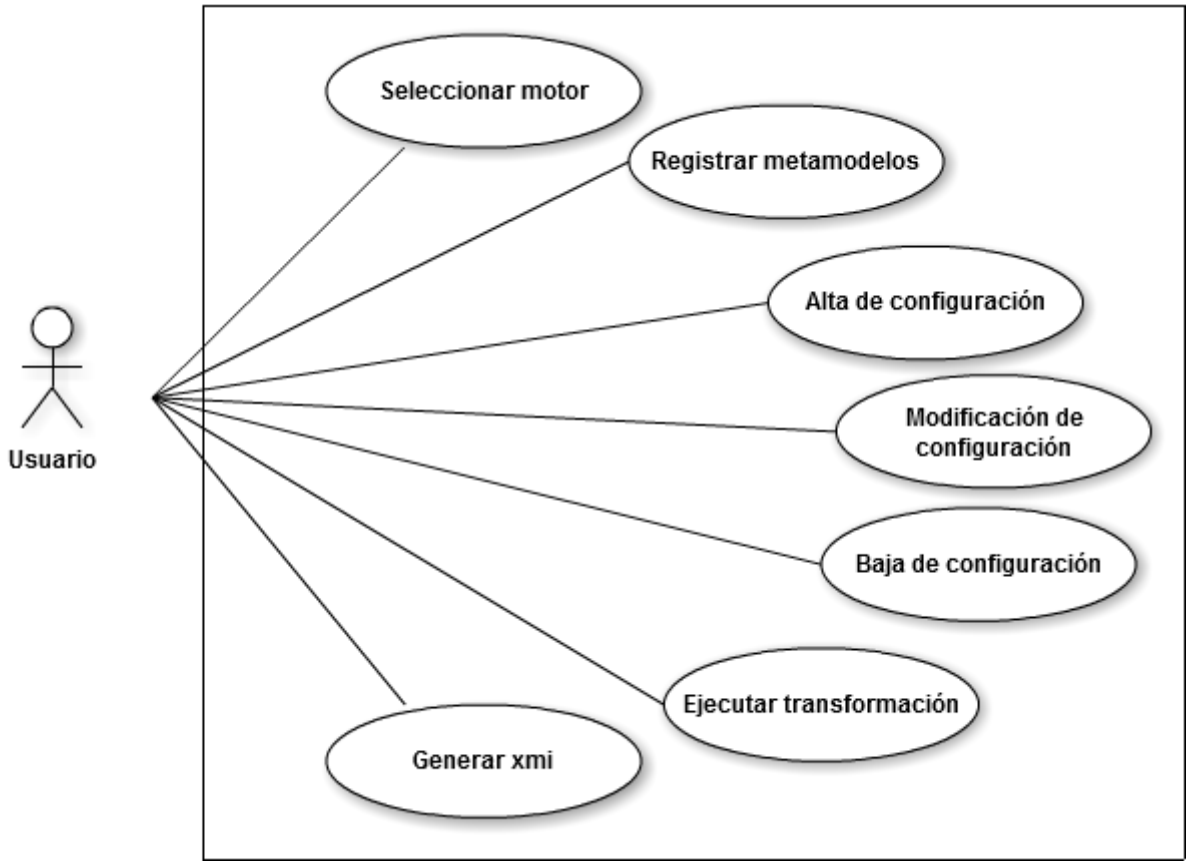


Figura 3.1: Diagrama de casos de uso

A continuación se brinda la descripción general de cada caso de uso.

3.1.1 Seleccionar motor.

El sistema le presenta al usuario la lista de motores instalados disponibles para que el usuario elija con cual va a definir la configuración.

Una vez elegido el sistema recuerda la selección para los siguientes pasos.

3.1.2 Registrar metamodelos.

El usuario registra un metamodelo en el motor seleccionado. El metamodelo debe ser de una extensión reconocida por el motor seleccionado en el caso de uso “Seleccionar Motor”.

El usuario debe proporcionar la ruta al metamodelo, ya sea absoluta o relativa al workspace de Eclipse.

Una vez seleccionados, el sistema actualiza la lista de metamodelos asociados al motor de transformación correspondiente.

3.1.3 Alta de configuración.

El usuario da de alta una nueva configuración en el motor seleccionado en el caso de uso “Seleccionar Motor”. Deben existir metamodelos asociados al motor registrados en el caso de uso “Registrar metamodelos”.

El usuario ingresa los datos de la configuración: nombre, ruta a los modelos de origen y destino, ruta al script de transformación, y opcionalmente ruta al directorio de trazas.

Al finalizar el sistema valida los datos ingresados y de ser correctos registra la nueva configuración asociada al motor correspondiente.

3.1.4 Modificación de configuración.

El usuario modifica los parámetros de una configuración existente. Para esto el sistema lista las configuraciones existentes asociadas al motor seleccionado en el caso de uso “Seleccionar motor” y el usuario selecciona la configuración a modificar por su nombre. El sistema muestra los datos de la configuración, que el usuario podrá modificar.

Al finalizar el sistema valida los datos ingresados y de ser correctos guarda los cambios en la configuración

3.1.5 Baja de configuración.

El usuario elimina una configuración existente en el motor seleccionado. Para esto el sistema lista las configuraciones existentes asociadas al motor seleccionado en el caso de uso“Seleccionar motor”. El usuario selecciona la configuración a eliminar por su nombre y ejecuta el comando de eliminación. El sistema elimina la configuración y la asociación con el motor correspondiente.

3.1.6 Ejecutar transformación.

El usuario ejecuta una transformación. Para esto el sistema lista las configuraciones existentes asociadas al motor seleccionado en en caso de uso “Seleccionar motor”. El usuario selecciona la configuración a ejecutar por su nombre y presiona el comando de ejecución.

El sistema ejecuta la configuración seleccionada en el motor correspondiente, e informa el resultado de la transformación.

3.1.7 Generar xmi

El usuario selecciona un modelo BPMN2.0 (u otro formato serializable a XMI) y la transformación XSLT correspondiente y el sistema genera un archivo en formato de intercambio .xmi equivalente.

3.2 Requisitos no funcionales

A continuación se detallan los requisitos no funcionales relevados.

3.2.1 Extensibilidad

Se plantea la necesidad de que el plugin sea extensible a otros motores de transformación no contemplados, o que pudieran desarrollarse en el futuro. Es aceptable que la extensión tenga que ser realizada por un usuario con conocimientos de programación, y que requiera implementar clases o interfaces, modificar archivos de configuración, o recompilar componentes.

3.2.2 Diseño de la interfaz

Se plantean los siguientes requisitos para la interfaz gráfica:

- debe estar en idioma inglés
- debe mostrarle intuitivamente al usuario que pasos tiene que realizar, que requisitos tiene que cumplir para llevar a cabo la tarea.
- Debe permitir acceso directo a cierta etapa del proceso si es que las etapas anteriores están cumplidas, agilizando así la reutilización.
- Debe nuclear en un solo lugar (por ejemplo, un menú) el acceso a todas las operaciones o etapas del proceso.

4. Solución propuesta

Para la solución se plantea realizar un plugin con estructura de wizard, que guíe al usuario por los pasos que tiene que realizar. A su vez, permitirá al usuario invocar un paso particular del flujo si es que los pasos anteriores ya fueron completados con anterioridad.

Se realizó evitando generar una dependencia con motores de ejecución de transformaciones, de tal forma que el plugin pueda cargarse en el ambiente aun si no existe ningún motor instalado.

Se provee una interfaz de "motor abstracto" para poder extender el plugin a nuevos motores no existentes en la actualidad.

4.1 Análisis de la solución

En la Figura 4.1 se muestra el modelo de dominio resultante del análisis.

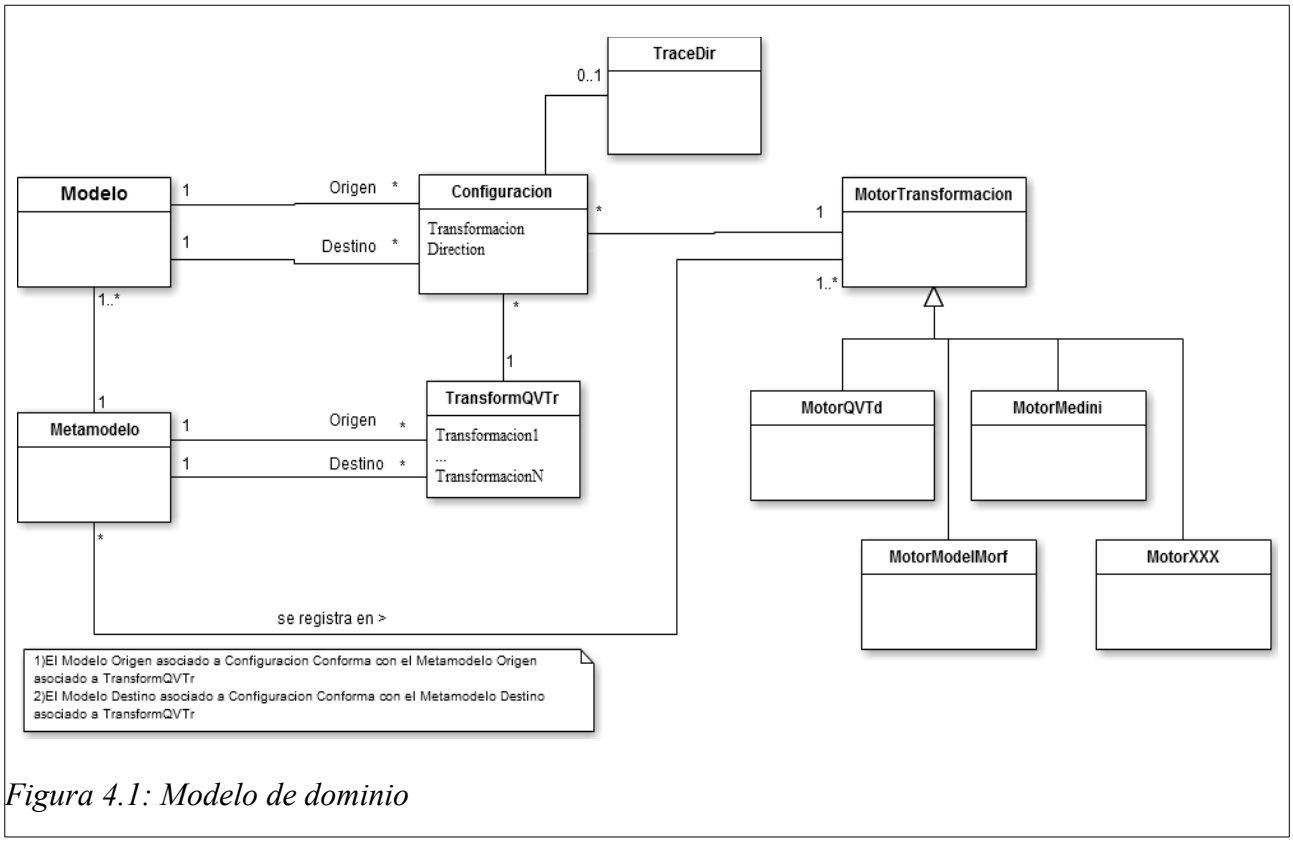


Figura 4.1: Modelo de dominio

- Configuración. Una configuración esta conformada por un modelo origen y destino, una transformación QVTr, posiblemente un directorio de transformación (TraceDir) y un motor asociado que utiliza la configuración para ejecutar una transformación.
- Transformación QVTr. La transformación en sí se ejecuta en el motor, por lo tanto debe contener referencias a los metamodelos origen (que utiliza el modelo origen) y destino (que utiliza el modelo destino).

- Modelo. Tiene asociado un metamodelo que contiene definiciones de los elementos y puede participar en 1 o más configuraciones.
- Metamodelo. Puede estar incluido en una o más transformaciones y se registra en el motor de transformación.
- Motor de transformación. Representa un motor abstracto. Contiene ciertas primitivas y operaciones elementales que debe de realizar cualquier implementación de un motor qvt. Los motores qvt conocidos son el Medini y ModelMorf.

4.2 Arquitectura general

En la figura 4.2 se muestra la integración del plugin en Eclipse. Se integra instalando el plugin dentro de la plataforma. Para eso hay que agregar físicamente el plugin en el directorio de instalación de Eclipse y al reiniciarse queda deployado en la plataforma.

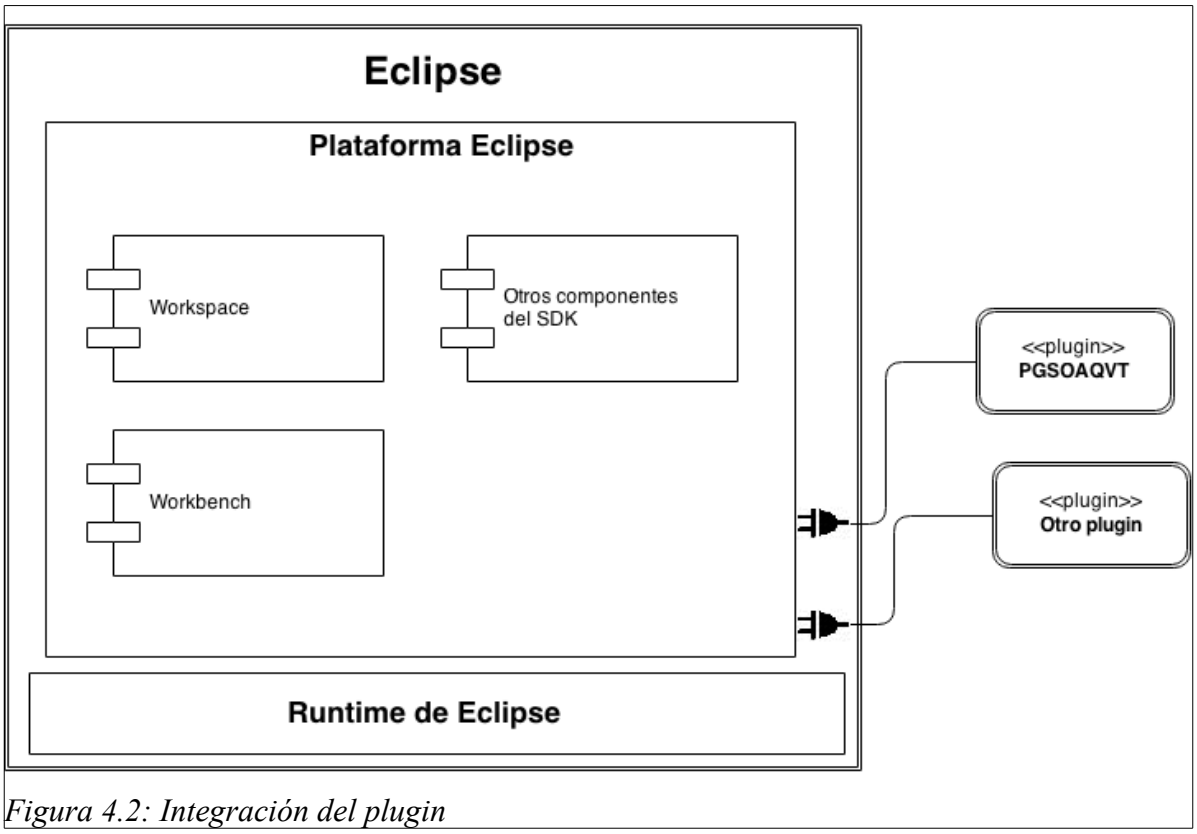


Figura 4.2: Integración del plugin

En el diagrama de la figura 4.3 se muestran los subsistemas de Eclipse que utiliza el plugin.

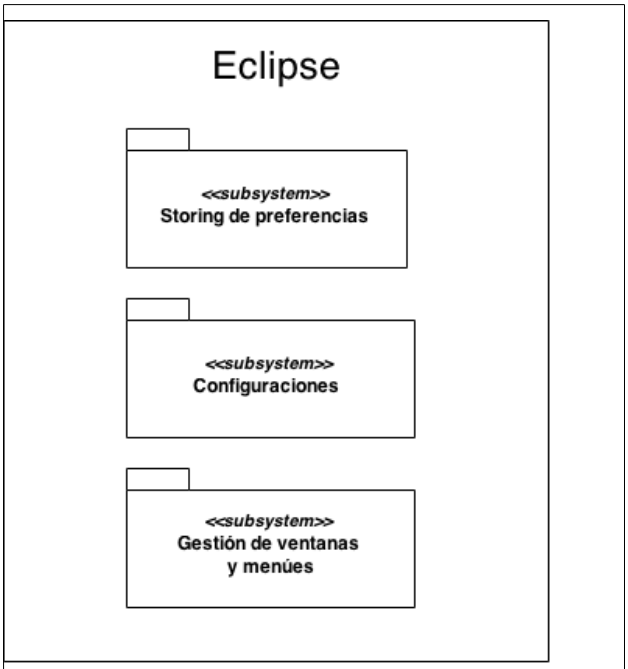


Figura 4.3: Subsistemas utilizados por el plugin

Los subsistemas utilizados por el plugin son: *storing* de preferencias, configuraciones y gestión de ventanas y menúes. En el diseño del plugin se mostrará qué función cumplen y como se utilizan los componentes de estos subsistemas en el plugin.

4.3 Diseño del plugin

Subsistemas

En la figura 4.4 se muestra el Diagrama de subsistemas de la solución, y luego se describen los subsistemas identificados y sus funcionalidades.

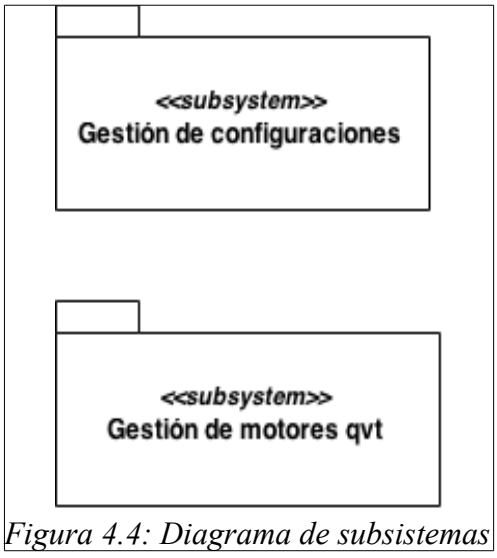


Figura 4.4: Diagrama de subsistemas

Se pueden identificar dos grandes conjuntos de funcionalidades del plugin: gestión de configuraciones y gestión de motores qvt. La gestión de configuraciones agrupa todas las operaciones que afectan directa o indirectamente a las configuraciones qvt. Alta, modificación, baja, ejecución de configuraciones están entre las operaciones más importantes que implementa este módulo. La gestión de motores qvt reúne las operaciones y componentes encargados de realizar la instalación y registro de motores qvt en la plataforma, registro de los metamodelos en los motores y uso de las APIs de los motores para ejecutar las transformaciones.

Componentes

A continuación, se muestran dos diagramas de componentes. El de la figura 4.5 es un diagrama de componentes de alto nivel mientras que el de la figura 4.6 ilustra componentes de mayor granularidad partiendo del anterior. Además, el último incluye uso de componentes externos del plugin relacionados al SDK de Eclipse.

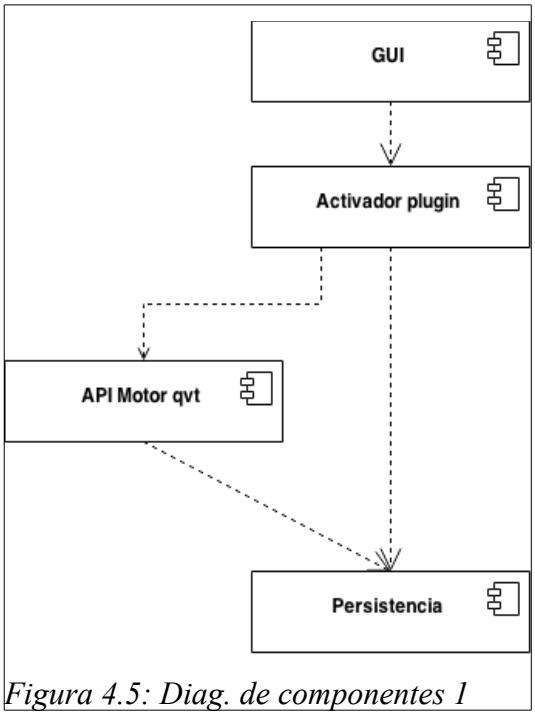


Figura 4.5: Diag. de componentes 1

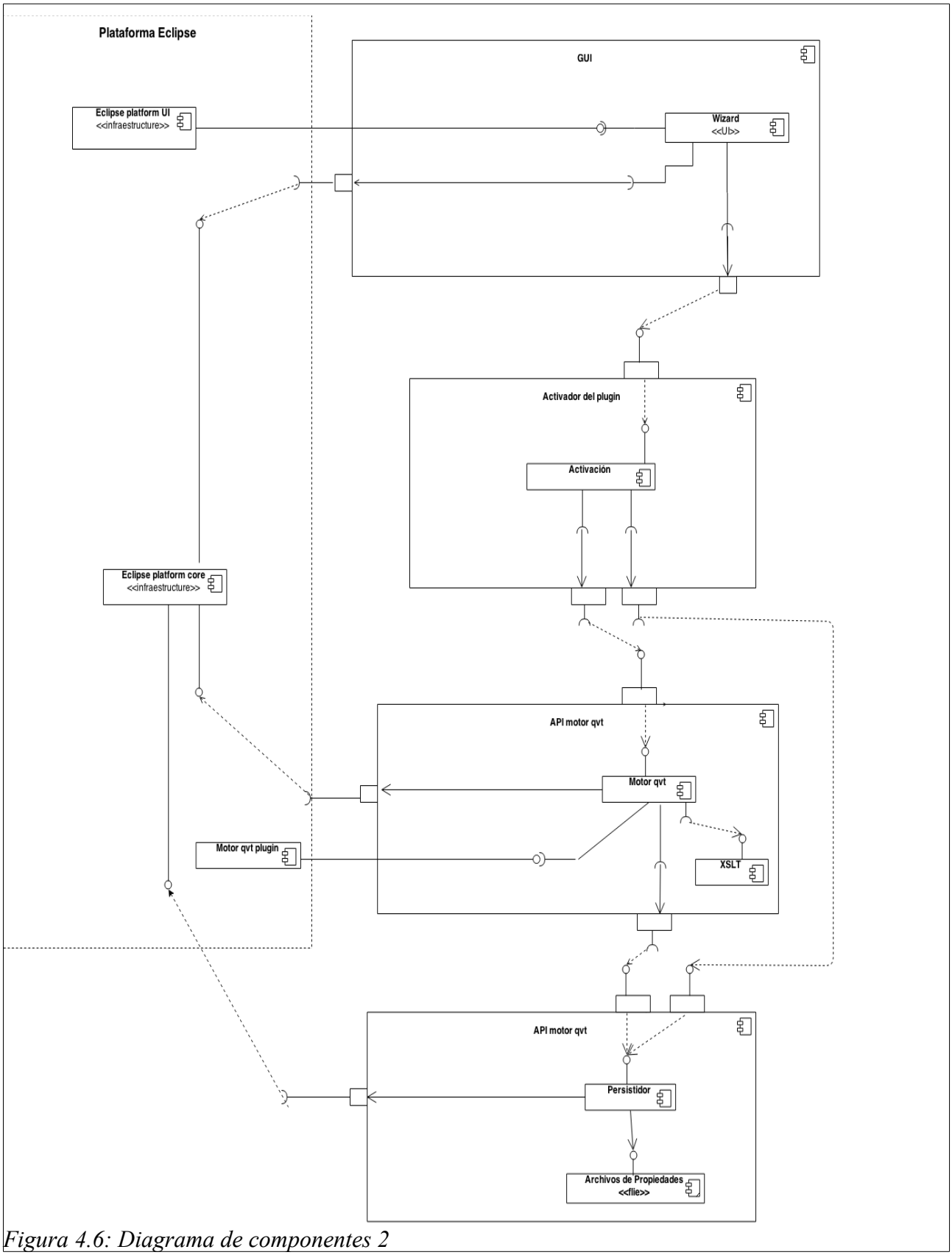


Figura 4.6: Diagrama de componentes 2

Se identificaron cuatro grandes componentes (figura 4.5). Se describen en líneas generales:

- GUI. Aglomera todos los componentes relacionados a la interfaz de usuario.
- Activador plugin. Comprende todos los componentes clave para llevar a cabo la activación adecuada del plugin en la plataforma Eclipse.
- API Motor qvt. Comprende los componentes asociados a los motores qvt utilizados en la implementación.
- Persistencia. Maneja todos lo relacionado al almacenamiento/recuperación de datos del plugin.

GUI utiliza interfaces del Activador plugin a través de las cuales accede tanto a las APIs de los motores como para el acceso a datos. GUI solo se comunica con el Activador. Tanto el Activador del plugin como las APIs de motores interactúan con el módulo de persistencia.

En el segundo diagrama (figura 4.6) se muestran los componentes del plugin y su relación con componentes de Eclipse.

El componente GUI encapsula todos los elementos que forman parte de la interfaz gráfica (Wizards, ventanas del Wizard, menú, etc..) y requiere de componentes de la UI de la plataforma Eclipse para la creación de las interfaces gráficas. Además, utiliza algunos componentes del core de la plataforma para devolver por consola el log de las transformaciones.

El componente Activador plugin se encuentra el componente de Activación que aglomera las clases que tienen como propósito tanto de levantar el plugin como realizar acciones y definir eventos al momento de activación.

El componente de las APIs de motores comprende el componente de implementaciones de motores qvt asociados al plugin desarrollado y sus respectivos componentes externos instalados en la plataforma que implementan los motores en sí. Se acceden a las APIs que estos proveen para invocar las transformaciones. Además la implementación del motor abstracto puede crear configuraciones ejecutables desde el SDK de Eclipse, por lo cual deberá comunicarse con interfaces del core de la plataforma para obtener parámetros de las configuraciones.

Por último, el componente de Persistencia engloba aquellas clases que realizan acceso a datos. Como se verá en el capítulo 4, podemos persistir en el plugin del motor o en un archivo de propiedades de pgsoaqvt. En el primer caso, utilizamos los componentes el core de Eclipse que proveen acceso al store de preferencias. En el segundo caso, realizamos la persistencia en archivos de propiedades.

Clases

Finalmente, la solución cuenta con el siguiente Diagrama de clases de diseño de la figura 4.7.

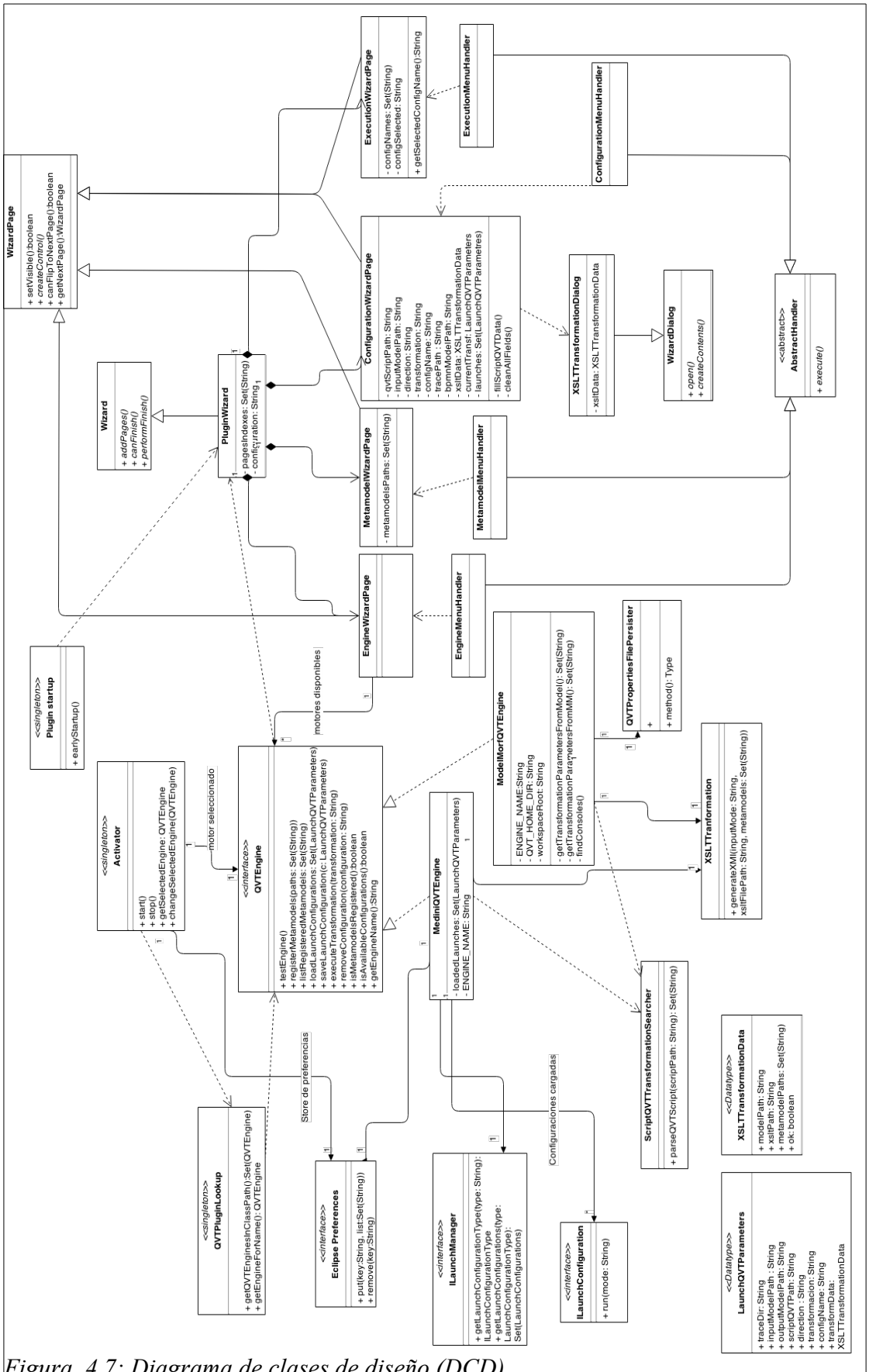


Figura 4.7: Diagrama de clases de diseño (DCD)

En el DCD cabe realizar algunas observaciones:

- La clase PluginWizard se componete de las clases *WizardPage, las cuales están relacionadas con la GUI. Cada clase *Handler tiene dependencia sobre una clase WizardPage que se encarga de instanciarla. El uso de las wizard pages y los handlers se esclarecerá en el análisis de la interfaz en la sección 4.4 y en la implementación en la sección 4.5.1.
- La interfaz QVTEngine contiene las operaciones necesarias de un motor abstracto para el plugin desarrollado. Se verá con mayor detalle la semántica de las operaciones en la sección 4.5.4.
- La clase ModelMorf es una implementación de QVTEngine que se realiza durante el caso de estudio de extensibilidad, detallado en el capítulo 6.
- El MediniQVT tiene asociada la interfaz IeclipsePreferences y el ModelMorf tiene asociada la clase QVTPropertiesFilePersister. Ambas están relacionadas con los modos de persistencia que se detallarán en la sección 4.5.5.
- Se utiliza el Datatype LaunchQVTParameters para almacenar datos de las transformaciones. Su uso se detallará en la sección 4.5.5 de implementación de la extensión.

4.4 Diseño de la interfaz

4.4.1 Flujo de uso

En la figura 4.5 se muestra un diagrama de actividad UML con el flujo de uso de la interfaz del plugin.

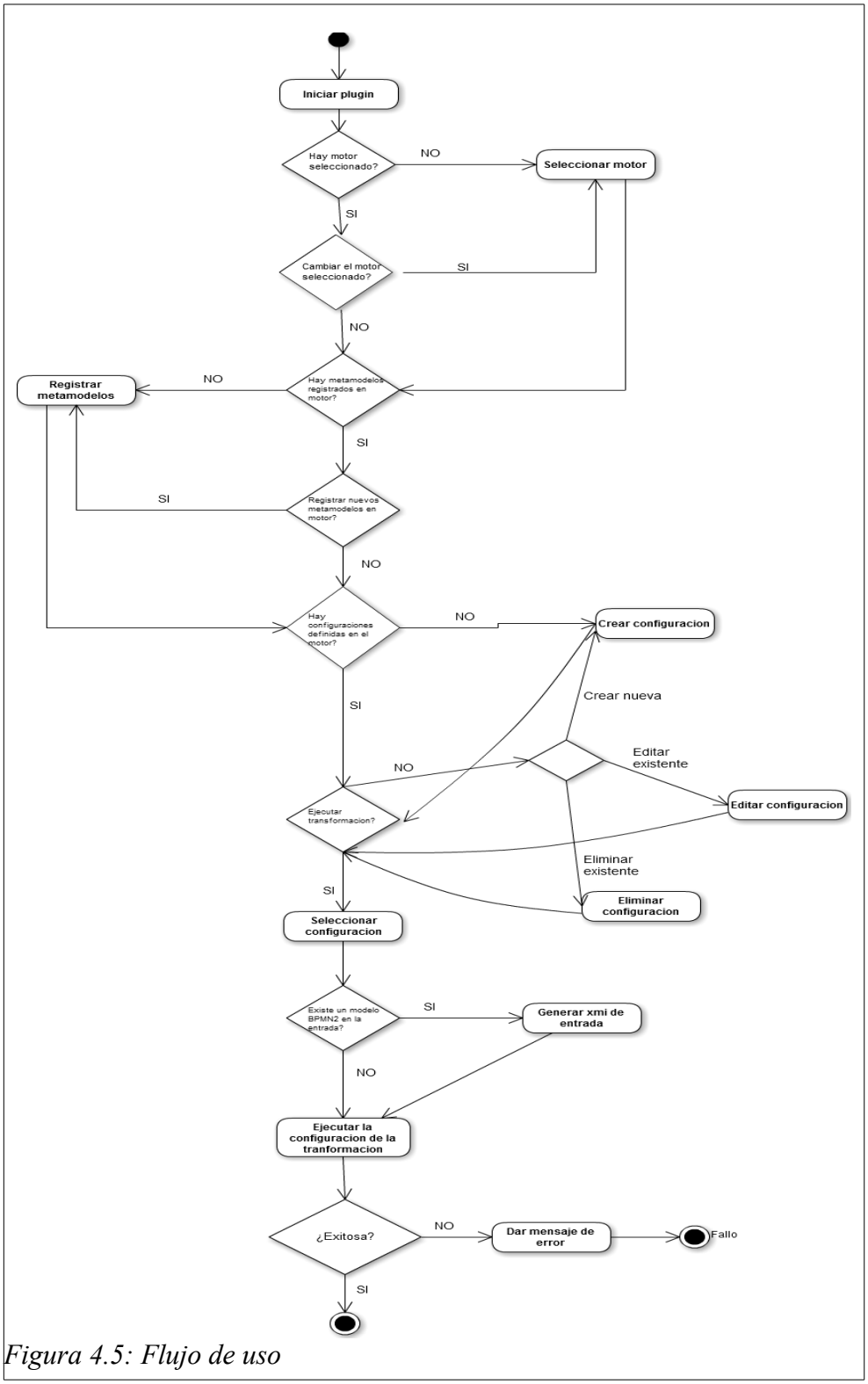


Figura 4.5: Flujo de uso

El plugin arranca en la actividad “Iniciar plugin”. El sistema se fija si hay un motor previamente seleccionado por el usuario y guardado en el sistema. Si no existe referencia a ningún motor en la memoria del plugin, se procede a la actividad “Seleccionar motor”, que le pide al usuario que seleccione uno de los motores qvt disponibles instalados. En caso de que ya exista un motor

seleccionado pero el usuario desee modificarlo, se continua en la misma actividad bajo la misma lógica. Si ya hay un motor seleccionado y además el usuario quiere seguir trabajando con este, se saltea la etapa de selección del motor.

El sistema sigue el flujo de los metamodelos. Si no hay metamodelos registrados en el motor, entonces el plugin retoma la actividad “Registrar Metamodelos” que obliga al usuario a registrar metamodelos en el motor. Se invoca esta actividad de igual manera si el usuario desea agregar/quitar/cambiar metamodelos registrados. En caso de que ya haya metamodelos en el motor y el usuario no desee realizar la operación, se saltea esta actividad y el plugin entra en el flujo de las configuraciones.

Con respecto a las configuraciones, el sistema admite un flujo más dinámico. Si no hay configuraciones de transformaciones del motor seleccionado, se obliga al usuario a crear una nueva configuración en “Crear configuración”. En caso de que ya existan configuraciones guardadas en el motor, el usuario puede decidir entre ejecutar una configuración, o realizar ABM de configuraciones. Por eso existe una decisión con tres caminos de salida: “Crear nueva” que lleva a la acción de “Crear configuración”, “Editar existente” que lleva a la acción de “Editar Configuración” o “Eliminar existente” que desemboca en la acción “Eliminar configuración”. De cada una de estas tres se vuelve a la decisión de ejecutar transformación, por lo cual potencialmente el usuario podría hacer un loop de ABM de configuraciones. Si ya existían configuraciones en el motor y el usuario deseaba directamente ejecutar alguna, se saltean las actividades anteriores. La condición “Ejecutar transformación” lleva a la acción “Seleccionar transformación” que pide al usuario seleccionar una transformación a ejecutar en el motor. Si la configuración tenía definida un modelo de entrada con una transformación xslt, previamente a la ejecución se genera el modelo de entrada al motor en la actividad “Generar xmi”.

En la fase final del flujo, se ejecuta la transformación en el motor en la actividad “Ejecutar transformación”. Si esta es exitosa, fin del flujo. Si hubo algún error en el proceso, se despliega al usuario la razón del error y fin del flujo.

Se puede ver que varias de las acciones en el diagrama se mapean y tienen correlación con los casos de uso detallados en el capítulo 3, de la siguiente forma:

- “Seleccionar motor” al CU Seleccionar motor.
- “Registrar metamodelos” al CU Registrar metamodelos
- “Crear configuración” al CU Alta de Configuración.
- “Editar configuración” al CU Modificar configuración.
- “Eliminar configuración” al CU Baja de Configuración.
- “Generar xmi de entrada” al CU Generar XMI

4.4.2 Descripción

La interfaz cuenta con un menú principal con cuatro opciones y un wizard. Cada opción del menú principal representa un paso en la secuencia del flujo en la definición de la transformación. Las opciones del menú son: "Select engine", "Register metamodels", "Configurations" y "Execute Transformation". En la figura 4.6 se muestra la UI claramente.

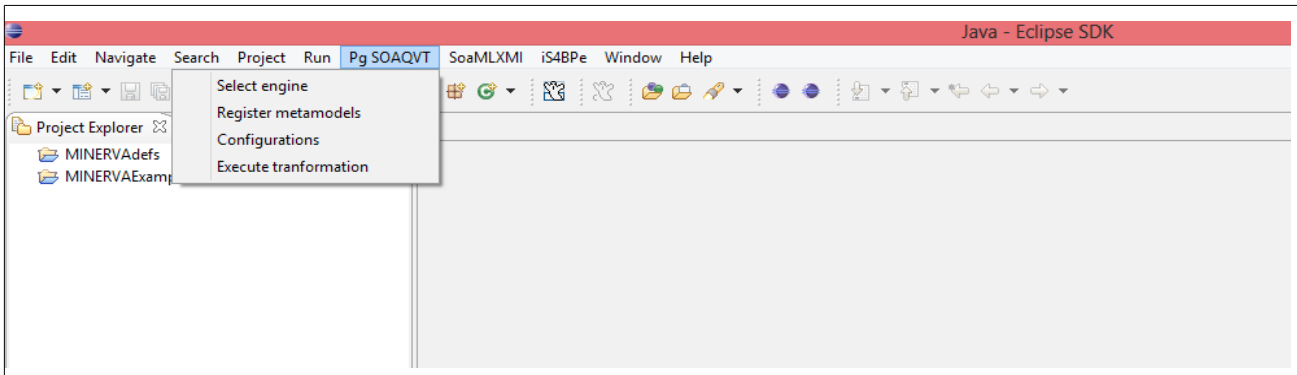


Figura 4.6: Menú del plugin

Al hacer click en una de las opciones, se abre el paso correspondiente al flujo. A su vez, al abrir el menú principal, aparecen como habilitadas las opciones correspondientes a los pasos que son capaces de realizarse. Por ejemplo, si no se ha seleccionado un motor aún, solo aparece habilitada la opción "Select engine", porque se debe seleccionar primero un motor antes de avanzar; las otras opciones aparecen deshabilitadas, tal como se muestra en la figura 4.7.

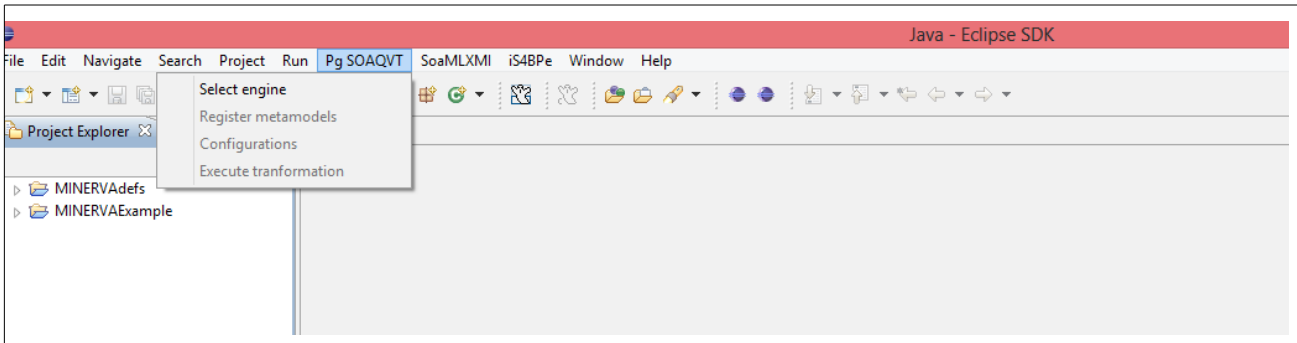


Figura 4.7

Si, por ejemplo, ya se ha seleccionado un motor pero no se ha registrado ningún metamodelo, aparecen habilitadas las opciones "Select engine", porque se podría cambiar el motor seleccionado en este estado, y "Register Metamodels", para agregar metamodelos al motor actual, como se muestra en la figura 4.8.

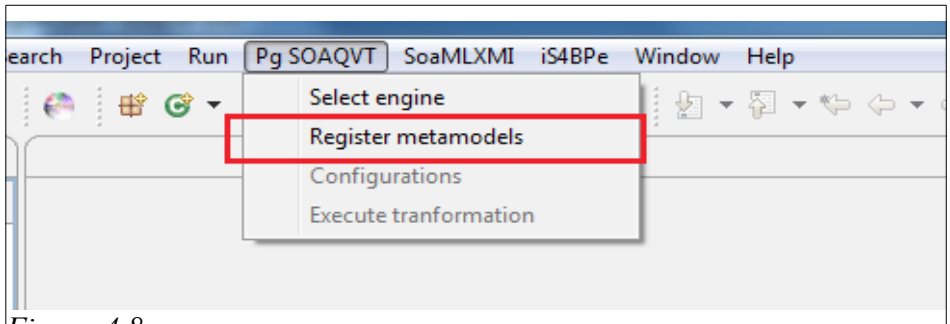


Figura 4.8

Como se visualiza en la figura 4.9, vemos la correspondencia que hay entre cada opción del menú y la secuencia lógica de pasos para ejecutar la transformación acorde al flujo de uso. Los pasos se organizan en un wizard y cada ventana del wizard se corresponde con una sola opción del menú.

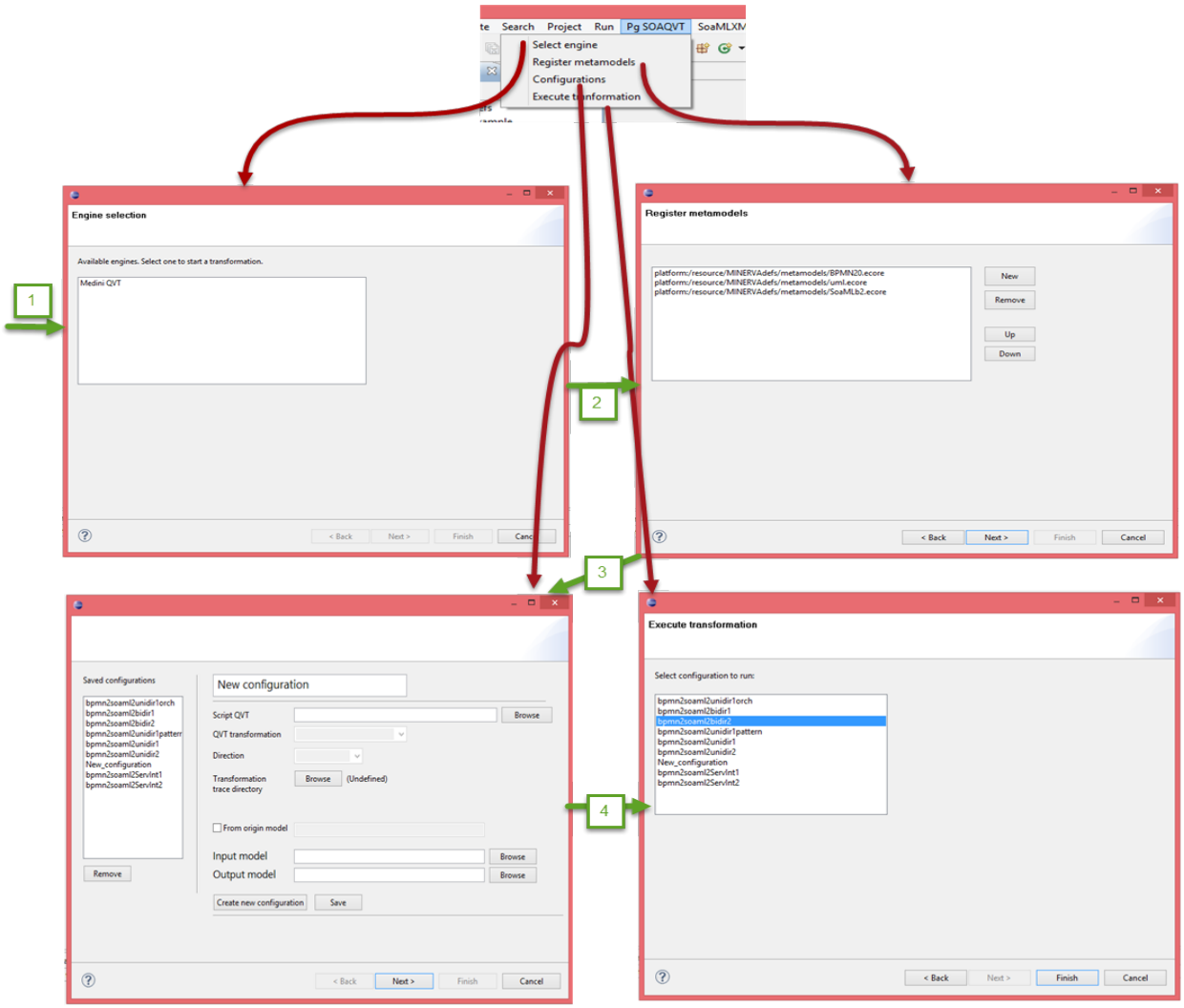


Figura 4.9: Correspondencia de las ventanas y el menú

Por ejemplo, al dar click en la opción del menú "Register Metamodels", esta abrirá la segunda ventana de la secuencia.

Se hará una descripción de cada una de las ventanas en detalle y sus componentes.

- Ventana 1. Lista los motores instalados en una lista. Habilita al siguiente paso habiendo seleccionado un motor de dicha lista. En la figura 4.10, se ilustra el primer paso de la transformación de acuerdo al flujo de uso.

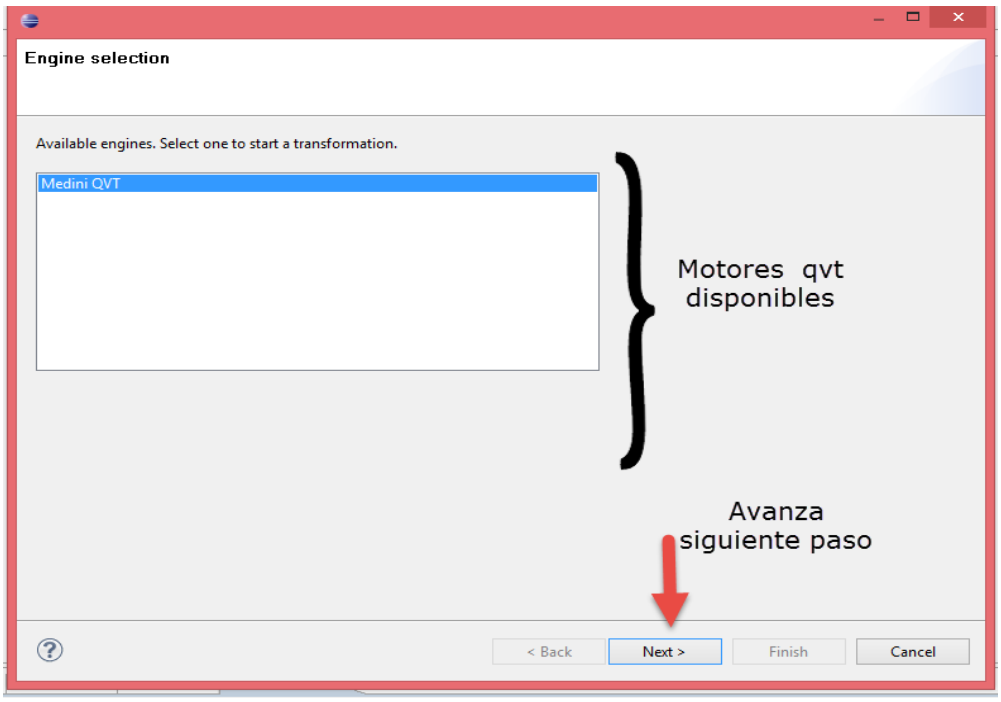


Figura 4.10: Primer página

- Ventana 2: En este paso se abre la ventana para registrar metamodelos. Se listan los metamodelos registrados en la ventana y aparecen algunos botones, tal como se muestra en la figura 4.11.

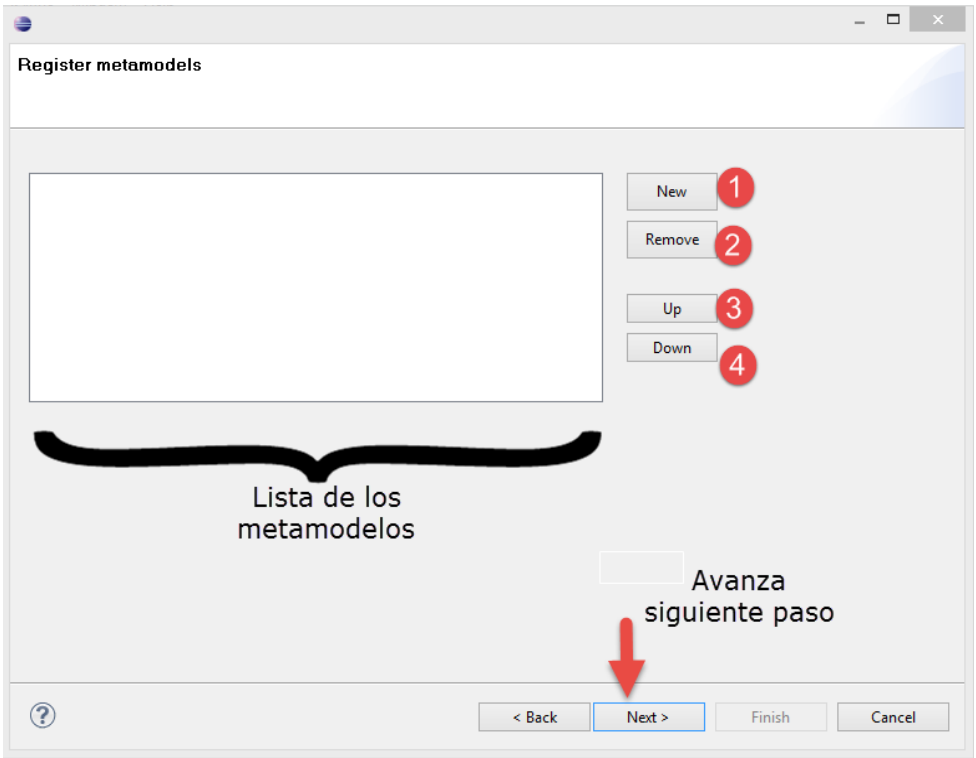


Figura 4.11: Segunda página

Se despliega una lista con las rutas de los metamodelos registrados. Aparecen varios botones. Un botón de New (1) que abre un file chooser (del file system o relativo al workspace) que permite seleccionar el archivo que contiene un metamodelo y agregarlo a la lista. Debajo un botón de Remove (2) que elimina un metamodelo de la lista que se encuentre seleccionado. Si no hay ninguno, la acción del botón no tiene efecto. Más abajo se presentan dos botones, Up (3) y Down (4), que define el orden de precedencia en que deben aplicarse los metamodelos. Al seleccionar un metamodelo y presionar en Up, el metamodelo se muestra un renglón más arriba. Análogamente al botón de Down, el metamodelo va a aparecer un renglón más abajo. Si hay al menos un metamodelo en la lista, se habilita el botón de Next.

- Ventana 3. Se avanza a la pagina de configuraciones. La figura 4.12 muestra la ventana. Aquí hay varios elementos gráficos.

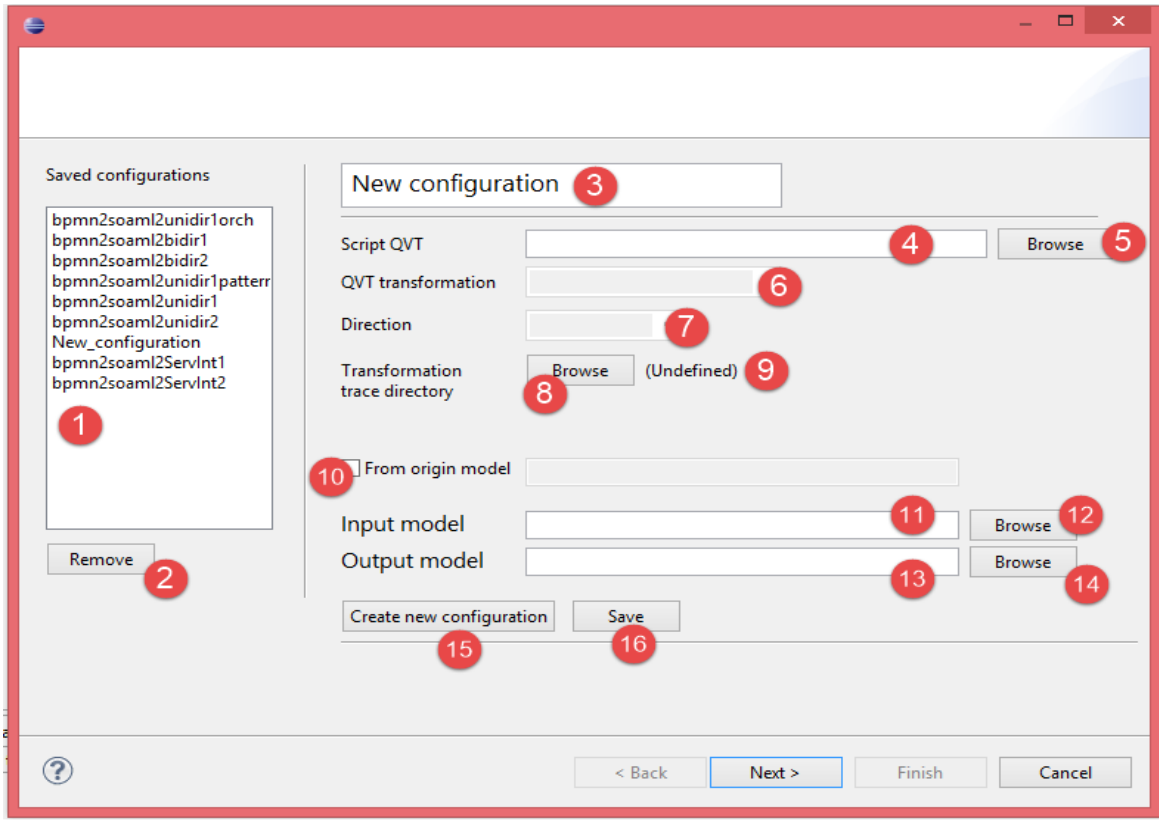


Figura 4.12: Tercer página

A la izquierda de la página, aparece una lista con los nombres de las configuraciones definidas (1) y un botón Remove (2) que elimina una configuración de la lista si hay alguna seleccionada. En la parte derecha, tenemos un panel con configuración en blanco. En la parte superior hay un campo de texto con el nombre de una nueva configuración (3) y un campo de texto con la ruta al script qvt (4). Dicha ruta se puede elegir desde un file chooser que surge al presionar el botón Browse (5) relativo al workspace y que automáticamente se carga en (4). Después se tienen un combo box deshabilitado para la selección de una transformación del script (6), un combo box deshabilitado para la selección de la dirección (7) y un botón "Browse" (8) que abre un file chooser relativo al workspace para actualizar la ruta del directorio de la traza de la transformación. Dicha ruta se mostrara en un label en (9). Después aparece un check box "From input model" (10) que al hacer

checked abre un popup con campos para agregar un modelo y una transformación. En (11) hay un campo de texto que guarda la ruta al modelo de entrada de la transformación, y que se puede buscar desde un file chooser (relativo al workspace o absoluto) que se abre al presionar el botón Browse en (12). Análogamente, se guarda la ruta del modelo de salida en (13) y (14). Finalmente, debajo del panel de la derecha tenemos dos botones más: “Create new configuration”, que al presionarlo hace limpiar todos los campos de la configuración actual y fija (10) en unchecked, y “Save” que guarda la nueva configuración en el motor y agrega el nombre puesto en (3) a la lista en (1).

Al seleccionar una de las configuraciones de la lista (1), se cargan los datos de la misma en la derecha para su edición. Y al dar luego “Save” se guarda la instancia de la misma con los nuevos datos.

Al hacer check en (10), se abre un popup para el ingreso de los datos de la pre-transformación opcional. La interfaz que aparece se muestra en la figura 4.13.

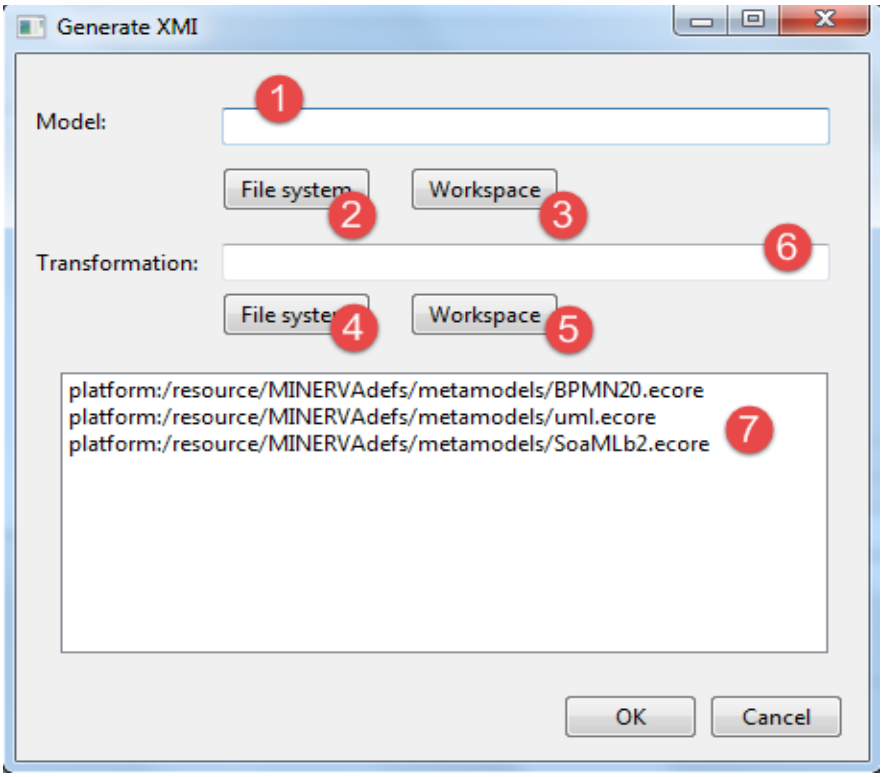


Figura 4.13: Diálogo para generar modelo de entrada con xslt

Aparecen dos campos de texto (1) y (6) para almacenar las rutas al modelo de entrada y a la transformación xslt. Los botones File System (2) y (4) permiten respectivamente examinar en el file system para seleccionar los archivos correspondientes; los botones Workspace (3) y (5) permiten respectivamente examinar el file system relativo al workspace del entorno. Es necesario indicar para el modelo de la pre-transformación, a qué metamodelos registrados debe apuntar, por lo tanto se listan en (7) todos los metamodelos registrados. El usuario debe al menos seleccionar uno de la lista. Finalmente, se validan los campos al presionar en OK.

- Ventana 4. La ultima ventana muestra una lista con las transformaciones. Al seleccionar una

de la lista, se puede finalizar el wizard ya que se deshabilita el botón de fin. La figura 4.14 muestra la ventana.

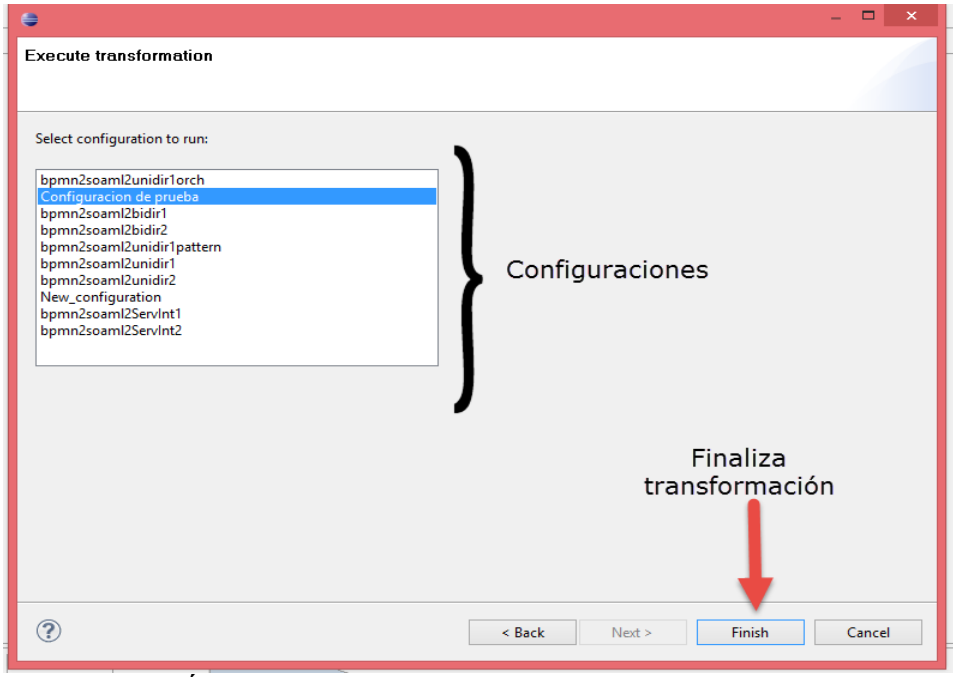


Figura 4.14: Última ventana

En cualquiera de las ventanas se puede volver al paso previo y hacer modificaciones.

4.5 Implementación

4.5.1 Interfaces gráficas

Como se mencionó previamente en la sección 4.4, la interfaz cuenta con un menú principal con varias opciones. También cuenta con un wizard compuesto por una secuencia de ventanas.

Menú

El menú se implementó básicamente mediante extensiones de los plugins nativos de eclipse org.eclipse.ui.menus, org.eclipse.ui.commands y org.eclipse.ui.handlers. La definición de las extensiones se realiza en el archivo plugin.xml que se adjunta en el anexo D.

El menú principal se implementó a partir de una extensión del plugin de eclipse org.eclipse.ui.menus. La extensión requiere de algunos parámetros, como ser un identificador y un nombre. Utilizamos un objeto menuContribution que es el elemento de extensión de org.eclipse.ui.menus que permite agregar un nuevo menú a la barra de menús del IDE; para este caso el que se va a usar para nuestro plugin. Dentro de la menuContribution, agregamos un menú, donde especificamos una etiqueta (para nuestro plugin, "Pg SOAQVT") e identificador, y debajo del menú las opciones del menú con sus respectivos nombres y en este orden: Select Engine, Register Metamodels, Configurations, Execute Tranformation.

Para cada una de las opciones del menú se definió como un command (del plugin nativo org .eclipse.ui.commands). A cada command se le asoció un manejador, el cual define qué acción realizar cuando se hace click en determinada opción. Dichas acciones están definidas como extensiones del plugin org.eclipse.ui.handlers, que tienen como elemento de extensión un handler que hace referencia a una clase que implementa la clase abstracta AbstractHandler, como se puede apreciar en el diagrama de clases de diseño de la sección 4.3 en las clases *Handler, y se implementa la acción particular sobrescribiendo el método execute. Las clases que actúan como handlers se ubican en el package gui.menus.handlers.

En síntesis, el menú es una menuContribution extensión de org.eclipse.ui.menus. Cada opción del menú hace referencia a un comando y para cada comando existe un handler que apunta a ese comando y a una clase que implementa la acción del comando, es decir, de la acción de la opción.

Por otro lado, el paso máximo de inicio que puede hacer el usuario según el flujo de uso se muestra habilitando/deshabilitando las opciones del menú. Para esto, se realizó una extensión del plugin nativo org.eclipse.ui.startup. Dicha extensión referencia una clase que sobrescribe el método earlyStartup de la interfaz Istartup que se ejecuta inmediatamente cuando el plugin es levantado en la plataforma. En dicha clase básicamente se obtiene la instancia del menú de nuestro plugin y se le asocia un listener al evento que se captura cuando el usuario abre el menú. En dicho listener, se activan o desactivan las opciones según el estado del plugin. Si no hay un motor seleccionado, se habilita la opción "Select Engine" y se deshabilitan el resto; si hay un motor pero no hay metamodelos registrados en el mismo, se habilitan sólo "Select Engine" y "Register Metamodels"; si hay motor, hay metamodelos pero no hay configuraciones en el motor definidas, se activa hasta "Configurations"; y por último, si hay configuraciones definidas, se habilitan todas las opciones.

Wizard y ventanas

La secuencia de ventanas se origina en un wizard. Cada ventana es una página del wizard. El wizard se crea en cada uno de los handlers de las opciones del menú. Cuando se hace click en una opción, se crea el wizard y se avanza a la página del paso correspondiente.

En el package gui.wizards.service se encuentra la clase PluginWizard que construye el Wizard y se encuentran las clases que conforman las páginas del wizard, extendiendo la clase WizardPage, en el siguiente orden:

- EngineWizardPage: genera la primer página del wizard. Se muestra a partir de la selección en la opción del menú "Select engine". Muestra los motores registrados en el plugin y realiza la selección del motor.
- MetamodelWizardPage: Se puede avanzar a esta página seleccionando en la opción del menú "Register Metamodels". Muestra los metamodelos registrados en el motor y agrega nuevos.
- ConfigurationsWizardPage: Se puede avanzar a esta página seleccionando en la opción del menú "Configurations". Provee la interfaz para ABM de configuraciones de transformación.
- ExecutionWizardPage: genera la página final. Se puede avanzar a esta página seleccionando en la opción del menú "Execute Tranformation". Muestra las configuraciones creadas y se selecciona una a ejecutar.

La clase PluginWizard únicamente crea y agrega las wizard pages mencionadas al wizard, define como acción para el botón Finish del wizard correr la transformación y valida el estado del plugin previamente a ejecutar la acción final.

Cada clase de página del wizard almacena en memoria los datos de los componentes. Heredan de la clase WizardPage de SWT y sobrescriben en general las siguientes operaciones:

- `createControl`: se ejecuta cuando se crea la página. Aquí se inicializan los componentes gráficos de la página y se los carga con los datos iniciales. Además, se asigna el comportamiento de dichos componentes al generarse eventos sobre ellos. Por ejemplo, al hacer click en un botón y se abra un file chooser para la selección de un archivo del file system. Otro ejemplo es la ventana de configuraciones donde al seleccionar una configuración existente de la lista produce que los campos de la página se carguen con los datos de la misma.
- Constructor. Recibe datos de las páginas precedentes y los almacena en memoria.
- `CanFlipToNextPage`. Devuelve un valor booleano que indica si pasar a la siguiente página.
- `GetNextPage`. Avanza el wizard a la página subsiguiente. Puede también actualizar el estado de esta página previo a realizar el avance.
- `SetVisible`. Se encarga de hacer visible la página actual. En algunas páginas también puede actualizar algún componete gráfico en la acción del usuario de hacer back y forward en el wizard luego que las páginas ya fueron instanciadas.

Los componentes gráficos principales empleados en las páginas fueron botones(Button), campos de texto(TextField), checkboxes (Checkbox), selectores de archivos (FileChooser) y combo boxes, todos componentes de la libreria SWT.

4.5.2 Bibliotecas utilizadas

Extensible Component Scanner (ExtCoS)

Es un componente java que provee una API para realizar búsquedas en el proyecto de clases, interfaces y enumerados. También permite hacer búsquedas consultando por extensión, herencia o uso de determinadas anotaciones. Permite realizar consultas tipo SQL dentro de un paquete determinado del proyecto filtrando por nombre y tipo de la entidad, y retorna los resultados encontrados en una lista de tipo Class<?>.

Se utilizó la versión 0.4b de la librería.

Utilitarios

Se encuentran en los paquetes *.utils del proyecto del plugin como clases java (ver si es necesario el DCD de la sección 4.3).

- ScriptQVTTransformationSearcher. Se encarga de realizar lecturas del script qvt que se ingresa en el paso 3. Se implementó un método que permite obtener mediante expresiones

regulares las firmas de las transformaciones de un script qvt y devuelve los nombres de las transformaciones y sus parámetros. Estos datos se presentan al usuario en el paso 3 cuando elige el script qvt del workspace.

- XSLTTransformation. Se encarga de realizar la transformación xslt. Se utiliza básicamente una operación que recibe la ruta a un modelo, la ruta a un archivo de transformación en formato xslt y crea un nuevo archivo con el resultado de la transformación. Se utiliza en la ejecución de la transformación.
- SWTResourceManager. Provee primitivas para acceder a más alto nivel a fuentes y colores de la herramienta SWT.

Windows builder

Para el diseño de las interfaces, se utilizó el plugin de Eclipse WindowsBuilder. Es una herramienta que permite desarrollar interfaces nativas para el entorno. Dentro de Windows Biulder, se empleó específicamente la librería SWT Designer, la cual emplea un editor visual para crear interfaces gráficas.

El plugin de Windows Builder recrea un árbol de sintaxis para parsear el código fuente y devolver la salida en la previsualización, al estilo de Swing. Muestra una paleta con los componentes gráficos de SWT (Botones, listas, campos de texto, checkboxes, separadores, labels, etc..) . Estos se arrastran al lugar deseado para agregarlos a la ventana (página del wizard) sobre la que se esta trabajando y se genera el código fuente de forma automática. Con el editor gráfico se pueden ir cambiando de posición los elementos sobre la ventana e irles agregando/modificando propiedades. También los cambios en el código se reflejan en el editor al guardar el fuente.

De esta manera, se agilizó enormemente el proceso de diseño de las interfaces ya que no hubo que relevantar el plugin cada vez que se hizo algún cambio a nivel de interfaz.

Junit4

Provee un framework para realizar tests unitarios. Se utilizó para realizar los tests, como se verá en el capítulo 5, con el fin de depurar todas las funcionalidades. Los tests realizados fueron orientados a test de plugins, corriendo las pruebas en una instancia de Eclipse con el plugin instalado.

4.5.3 Reflection

El lookup de motores se realiza en la clase QVTPluginLookup. Contiene las operaciones que utilizan reflection para búsquedas de implementaciones de motores qvt. Cuenta con dos operaciones básicas que se invocan durante el paso 1 para la selección del motor.

QVTPluginLookup tiene la operación `getQVTEnginesInClassPath` que obtiene los motores registrados en el plugin y retorna una lista con los motores qvt que implementan la interfaz del motor abstracto que están en el plugin. Se realiza una consulta con la libreria Extension Component Scanner para obtener todas las clases del package engines que cumplan con la condición de ser implementaciones del motor abstracto y retorna una lista con estas clases.

Algo que puede ocurrir es que una implementación del motor abstracto no cuente con los jars

necesarios para invocar a sus operaciones, sea porque no estén referenciadas o no hayan sido descargadas o instaladas. Cuando suceda esto, la clase del motor puede no compilar y ese motor no va a poder ser utilizado para transformaciones en nuestro plugin . Es por eso que iterando sobre la lista resultado de `getQVTEnginesInClassPath`, se intenta obtener una instancia de cada clase, de la forma:

```
for (Class<?> cls : classesList){

                        ....

      try {

            QVTEngine engine = (QVTEngine)cls.newInstance();

            engine.testEngine();

            availEngines.add(engine);

      } catch (Throwable e) {

            ...

      }

}

return availEngines;
```

ClassesList es el resultado de la consulta devuelta por la ExtCos. Como vemos, se obtiene una instancia de la clase y se invoca el método testEngine que es un método auxiliar definido en el motor abstracto para chequear que la clase pueda ser utilizada para llamar al motor que representa. En caso de un error, por ejemplo de compilación, en la clase, el método arroja una instancia de tipo Throwable, que implica que la clase no está disponible para hacer correr transformaciones y no se la agrega a la lista resultado.

La clase QVTPluginLookup además cuenta con otra operación que recibe como parámetro el nombre de un motor qvt y devuelve la instancia del motor abstracto con dicho nombre, utilizando análogamente la librería ExtCoS.

4.5.4 Extensión

La extensión del plugin se basa en agregar una nueva implementación del motor abstracto. Dicha implementación es una clase que debe localizarse en el package "engines" e implementar la interfaz QVTEngine. La figura 4.15 muestra las primitivas que debe de proveer cualquier motor qvt para funcionar en nuestro plugin, definidas en QVTEngine.

```
package engines;

import java.util.Collection;

public interface QVTEngine {
    public void testEngine() throws Throwable;

    public String registerMetamodels(Collection<String> filePaths) throws Throwable;

    public void executeTransformation(String configurationName) throws Throwable;

    public Collection<String> listRegisteredMetamodels()  throws Throwable;

    public Map<String,LaunchQVTParameters> loadLaunchConfigurations()  throws Throwable;

    public void saveLaunchConfiguration(LaunchQVTParameters parameters) throws Throwable;

    public void removeLaunchConfiguration(String configName) throws Throwable;

    public boolean isMetamodelsRegistered() throws Throwable;

    public boolean isAvailableConfigurations() throws Throwable;

    public String getEngineName();

}
```

Figura 4.15: Interfaz del motor abstracto

La semántica de las operaciones es la siguiente:

- `TestEngine`. Realiza chequeos a nivel del motor qvt a fin de comprobar que se pueda utilizar desde nuestro plugin. Si testEngine libera una excepción, este motor no se lista en la lista de motores instalados en el paso de la selección de motor, porque se entiende que no se cumplen todas las condiciones para que pueda utilizarse.
- `RegisterMetamodels`. Registra las rutas relativas al workspace de los metamodelos pasadas por parámetro en el motor.
- `ExecuteTransformation`. Ejecuta la transformación de la configuración pasada por parámetro en el motor.
- `ListRegisteredMetamodels`. Devuelve las rutas a los metamodelos registrados en el motor.
- `LoadLaunchConfigurations`. Devuelve un mapa con las configuraciones guardadas en el motor. Dicho mapa se conforma por clave el nombre de la configuración y por valor un objeto con los parámetros: script qvt, modelos de entrada y salida, nombre y dirección de la transformación y directorio de la traza. También puede incluir un modelo origen con una transformación xslt asociada y que según vimos da como resultado el modelo de entrada de la transformación.
- `SaveLaunchConfiguration`. Almacena en el motor los parámetros de la configuración asociada.
- `RemoveLaunchConfiguration`. Elimina del motor la configuración con el nombre dado.
- `IsRegisteredMetamodels`. Devuelve un valor booleano que indica si hay algún metamodelo registrado en el motor.

- `IsAvailableConfigurations`. Devuelve un valor booleano que indica si hay alguna configuracion guardada en el motor.
- `GetEngineName`. Devuelve el valor correspondiente al nombre del motor.

Luego de hacer esta implementación, es necesario actualizar las dependencias del plugin, como se explica en la sección 4.5.6.

Hecho lo anterior, quedaría registrado el nuevo motor en nuestro plugin para ser utilizado.

4.5.5 Persistencia

El plugin utiliza dos mecanismos de persistencia:

- En el plugin del motor qvt. Se persisten los datos de las configuraciones, de los metamodelos registrados e información adicional en el store de preferencias del plugin que lleva a cabo la transformación. A nivel implementación, se utiliza la API de preferencias de Eclipse para realizar operaciones de almacenamiento y carga de preferencias, pasando como parámetro el ID del plugin del motor qvt competente. Específicamente, se utilizó la interfaz IEclipsePreferences del paquete org.eclipse.core.runtime.preferences.

Cuando se manda ejecutar la transformación en el motor, el plugin motor es responsable de hacer los chequeos correspondientes obteniendo los datos persistidos en su store de preferencias.

- En almacenamiento propio de PG SOAQVT. Si el plugin del motor qvt no utiliza preferencias para persistir datos, el plugin realiza un almacenamiento del estado actual del flujo de la transformación en un archivo de propiedades de java. En dicho archivo se almacenan las referencias a metamodelos y la definición de las configuraciones para un motor específico. Se utiliza la estructura de parejas {clave:valor} para acceder a los datos.

La clave se compone de diversas formas dependiendo del objeto a representar.

  - Metamodelos:

    <Nombre_Motor>.Metamodel.<número_order>=<Ruta al metamodelo>

    Ej: MediniQVT.Metamodel.0=/workspace/metamodels/bpmn.ecore

  - Confiuraciones:

    <Nombre_Motor>.Configuration.<nombre_configuración>.<parámetro>=<valor>

    <parámetro>,<valor>) se encuentra en

    {("sciptqvt",<ruta relativa al script qvt>),

    ("inputModel",<ruta al modelo de entrada>),

    ("outputModel",<ruta al modelo de salida>),

    ("tranformation",<transformación del script qvt>)

("direction",<dirección de la tranformación>),

("traceDir",<directorio de traza>),

(“originModel”,<Modelo original de entrada>),

(“xsltPath”,<ruta a la transformación xslt>)}

Ejs:

MediniQVT.Configuration.nuevaConfig.scriptqvt=/workspace/qvt/scripts/script.qvt

MediniQVT.Configuration.nuevaConfig.transformation=bpmn2tosoaml2

MediniQVT.Configuration.nuevaConfig.inputModel=C:\models\bpmnmodel.xmi

originModel y xsltPath sólo son necesarias si queremos una transformación xslt previa que devuelva el modelo de entrada de la transformación qvt.

Al momento de ejecutar la transformación, el plugin es responsable de obtener los parámetros de la transformación del archivo de propiedades y pasárselos al motor correspondiente. Esto se hace en la clase engines.persitence.QVTPropertiesFilePersister, la cual realiza el almacenamiento y la carga de datos en y desde el archivo de propiedades respecto a metamodelos y configuraciones. Cuenta con los siguientes métodos:

- `SaveMetamodels`. Crea el archivo de propiedades si no existe. Se obtienen las propiedades actuales y se cachean en memoria. Se guardan las rutas de los metamodelos en disco según el formato especificado y se actualiza el cache de propiedades.
- `SaveConfiguration`. Crea el archivo de propiedades si no existe. Se obtienen las propiedades actuales y se cachean en memoria. Se guardan los datos de la configuración en disco según el formato especificado y se actualiza el cache de propiedades.
- `GetAllConfigurationParams`. Se obtienen los datos de las configuraciones cacheadas en memoria. Si no hay propiedades actuales, se inicializa el archivo de propiedad en disco.
- `GetMetamodels`. Obtiene las rutas de los metamodelos de las propiedades en cache. Si no hay propiedades actuales, se inicializa el archivo de propiedad en disco.
- `GetConfiguration`. Hace un llamado a `GetAllConfigurationParams` y obtiene la configuración con el nombre dado.

A nivel de nuestro plugin, se utiliza un store de preferencias propio para guardar el nombre del último motor seleccionado por el usuario.

4.5.6 Dependencias

Las dependencias del plugin se editan abriendo el proyecto y el archivo plugin.xml en al pestaña *dependencies*.

En el plugin desarrollado hubo que definir varias dependencias. Están las dependencias sobre plugins requeridos por este (required-plugins) y las dependencias de paquetes utilizados (imported-packages) sin especificar el plugin de origen.

Tiene como plugins requeridos algunos del core de la plataforma Eclipse. Entre ellos están:

- org.eclipse.ui.: Provee interfaces java para interacción y extensión de la UI de la plataforma Eclipse. Dentro de este paquete se utilizó la clase PlatformUI que provee acceso a los resources del workbench. Se necesita de este plugin para la creación del menú, los ítems del menú y los wizards que conforman la UI del plugin.
- org.eclipse.core.runtime. Contiene primitivas para realizar operaciones sobre el runtime de la plataforma. La definición del activador de un plugin requiere extender la clase abstracta AbstractUIPlugin que se encuentra en este paquete. Además contiene la API de acceso al store de preferencias de los plugins, necesaria para persistir valores.
- org.eclipse.ui.console. Este plugin provee acceso a un conjunto de interfaces que facilitan la creación y el despliegue de consolas en la vista de consolas. Permitió activar y poner foco en la vista de la consola estándar para mostrar el log completo de la ejecución de la transformación qvt seleccionada.
- org.eclipse.debug.core: Expone clases e interfaces que soportan facilidades comunes entre diversas modos de depuración. Permitió el acceso a los tipos de configuración y configuraciones respectivas almacenadas en la plataforma. Se empleó este plugin para realizar el ABM de configuraciones de cualquier tipo y ejecución de determinada configuración.
- org.junit v4. Plugin que contiene la librería Junit4, descripta en la sección 4.5.2.

Estos plugins requeridos son mandatorios para que levante pgsoaqvt. Si estas dependencias no se encuentran o no se pueden resolver, el plugin no levanta.

Luego se especifican las dependencias necesarias por el motor MediniQVT. Se agregaron los paquetes utilizados a la lista de “Imported Packages” y las marcamos como opcionales, ya que tal como se menciona en la sección 4.5.7, estos pueden no haberse cargado en el entorno y por lo tanto hacer fallar el plugin al levantar. Los paquetes importados fueron:

- de.ikv.medini.debug.core. Se empleo para obtener el valor de constantes para acceder a los parametros de las configuraciones de MediniQVT.
- de.ikv.medini.qvt.ui. Provee constantes generales del motor MediniQVT, como ser su PLUGIN_ID.
- de.ikv.medini.qvt.ui.preferences. Provee constantes para el acceso al store de preferencias del motor MediniQVT.

La figura 4.16 muestra las dependencias de nuestro plugin en el archivo plugin.xml con la presentación visual.

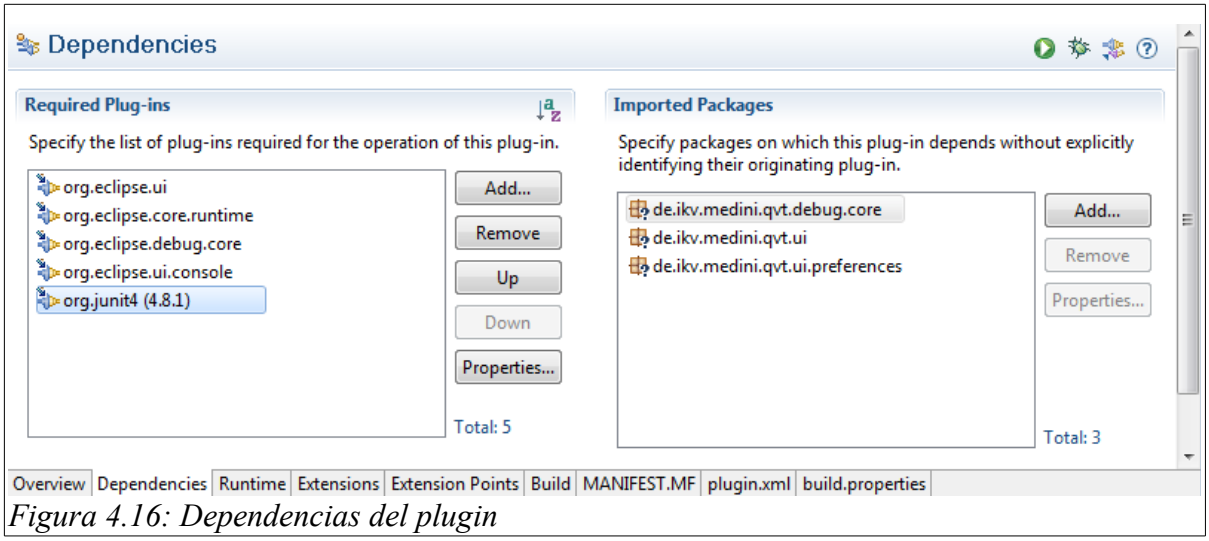


Figura 4.16: Dependencias del plugin

4.5.7 Alternativas y problemas encontrados

En esta sección se explicarán algunos inconvenientes encontrados durante el desarrollo del plugin, y cuáles fueron las decisiones tomadas para solventarlos.

Instalación del motor

En una primera etapa, se quiso lograr la instalación del motor mediante una conexión a Internet. Se pensó en tener guardada en un archivo de configuración la url para descargar el compresible, pudiendo lograr descargar el MediniQVT al PC, descomprimir el contenido y mover los jars del motor a la carpeta plugins del Eclipse. Al hacer esto e instalando nuestro plugin en un Eclipse sin el MediniQVT, se vio que el resultado no era exitoso. La razón era que Medini necesita de otras librerías para poder funcionar, como por ejemplo el EMF, y no se estaba resolviendo el árbol de dependencias. Esto podría pasar potencialmente con cualquier motor qvt a instalar. En consecuencia, quedó descartada esta idea.

En una segunda etapa, se pensó en invocar al Install New Software del Eclipse programáticamente. Se tendría la url al update site guardado en un archivo de configuración y cuando el usuario eligiera el motor a instalar, automáticamente se le abriría el cuadro de dialogo de actualizaciones. Esta idea parecía atractiva, ya que el manager de Eclipse para instalar plugins resuelve el árbol de dependencias y descarga todos los jars que sean necesarios. Solo sería necesario hacer un Restart del entorno para eventualmente cargar los plugins ya instalados. Dada la escasa y poco clara documentación en la web, además del uso de una API de Eclipse que pareció tener cierta complejidad para utilizar el Update Manager, se estimó que llevaría una curva de aprendizaje importante en el desarrollo para lograr una funcionalidad no imprescindible, por lo que se descartó también esta idea.

Por lo que finalmente se optó fue dejar que el usuario instale manualmente el plugin de un motor

qvt llamando al Install New Software. Para eso tendría que ir a buscar en la web el update site del mismo y realizar él la instalación. El plugin ya no instalaría motores sino que los registraría mediante implementaciones de una interfaz de motor abstracto, dando métodos a las operaciones del mismo.

Detección y disponibilidad de los motores instalados

Otro inconveniente que surgió en el lapso del desarrollo fue la detección de los motores instalados y ver si están disponibles para ser utilizados. Cuando instalamos el plugin en un Eclipse que no tenia el MediniQVT instalado, el mismo no levantaba. Esto era porque la clase del plugin para Medini no compilaba ya que no encontraba los jars propios en la plataforma. Dado que a nuestro plugin podría usar N motores, seria necesario usar algún mecanismo para detectar las implementaciones residentes en el plugin. Se decidió usar reflection para obtener las clases que fueran herencia del motor abstracto en el classpath y preguntar si se puede obtener una instancia de forma dinámica de esa clase para manejar excepciones de posibles errores, por ejemplo de compilación por falta de librerías o de la inexistencia de algún directorio o archivo clave. Un motor que supera estos controles se acepta como disponible para ser utilizado en el plugin.

Manejo de dependencias

Se presentaron problemas al cargar nuestro plugin en una plataforma donde no estuviera el motor Medini instalado. Por este motivo, hubo que cambiar las dependencias de nuestro plugin de los motores haciéndolas opcionales, para que el mismo pueda levantar sin necesidad de que las librerías de las dependencias que estuvieran presentes.

Transformaciones xslt

Al realizar la transformación xslt de un modelo para convertirlo a un modelo origen bpmn de la configuración, el modelo xmi no era interpretado correctamente por Medini. En el archivo de salida de la xslt no aparecían las referencias a los metamodelos correspondientes al modelo origen, por lo que hubo que editar programáticamente el resultado de la xslt para incluir dichas referencias. Es por esta razón que se solicita al usuario agregar la referencia al metamodelo en el paso 3 del flujo cuando se especifica un modelo inicial. Con esto, se agrega la ruta relativa al metamodelo en el atributo xsi:schemaLocation del encabezado del xmi.

La xslt utilizada y provista por la OMG no deja un xmi de origen válido para MediniQVT, por lo que la ejecución de la transformación desde un bpmn no resultó exitosa y genera un modelo de salida vacío. Para conseguir que la xslt devolviera un modelo origen valido, hubo que editar la xslt de forma de adecuarla a los tags y elementos de la sintaxis requerida por el motor.

Restricciones de uso de ModelMorf

ModelMorf no resuelve referencias a archivos externos. Se tuvo que utilizar, como se verá en el caso de estudio con ModelMorf en la sección 6.1.2, una versión de la transformación “recortada” para poder utilizar correctamente el motor.

5. Verificación del plugin.

Para la verificación del plugin utilizamos como herramienta la librería Junit4 de Java.

Cada test prueba la lógica del plugin. Se contemplaron casos de prueba donde el plugin debe devolver algún resultado de error y otros donde debe tener un resultado de éxito. Se utilizó para las pruebas la implementación del motor abstracto de MediniQVT pero eventualmente se podrían realizar análogas pruebas para más motores. Además, sólo se testean flujos de ejecución con determinadas entradas y salidas de nuestro plugin y no sobre los resultados de la ejecución de transformaciones de Medini.

5.1 Definición de los casos de prueba

Los casos de prueba contemplados agruparon por casos de uso:

- Selección de motor
  - selección de motor inválido
  - detección y listado correcto de los motores instalados.
- Registrar metamodelos
  - Registrar metamodelo con ruta inválida.
  - Registrar metamodelos con rutas válidas y listado de los metamodelos registrados.
- ABM de configuraciones
  - Crear configuración con nombre inválido.
  - Crear con ruta a script qvt inválida
  - Crear con script qvt con errores de sintaxis.
  - Crear sin definir transformación o dirección de la transformación.
  - Crear con ruta a modelo de entrada inválida o al modelo de salida inválida.
  - Crear con ruta a directorio de traza inválida.
  - Crear configuración válida y listado de configuraciones guardadas.
  - Crear configuración válida con nombre duplicado.
  - Crear otra configuración válida con éxito y modificar el nombre con el de una ya existente.
  - Eliminar configuración inexistente.
  - Eliminar configuración existente y listado de configuraciones guardadas.
- Ejecutar transformación.
  - A partir de configuración inexistente.
  - A partir de configuración existente sin registrar metamodelos.

    - De configuración existente con metamodelos registrados correspondientes, chequeando la existencia del archivo de modelo de salida generado.
- Transformación xslt
    - Con archivo xslt con errores o inexistente.
    - Con modelo de entrada con errores o inexistente.
    - Con modelo de salida no sobreescribible (bloqueado)

Se creó para cada CU una clase de test con su propio conjunto de tests. Para correr los tests, se crearon configuraciones de Junit de "Junit Plug-in test", provistas por PDE. Este tipo de configuraciones permiten correr una clase o un conjunto de clases con métodos de prueba anotados con @Test levantando una nueva instancia del Eclipse definiendo diversos parámetros: directorio de workspace, argumentos de la VM, qué plugins de la plataforma van a ser cargados, etc. Todas las configuraciones corrieron sobre el mismo workspace, cuyo directorio llamémosle <workspace_test_path>. El mismo se crea previamente a ejecutarse los tests de una clase. Esto se hace con el fin de mantener el estado del workpace entre diferentes conjunto de tests y no hacer que el resultado el test de una funcionalidad se vea condicionada a la anterior.

Los clases de los tests de la verificación fueron ubicados en el package pgsoaqvt.test. Cada clase extiende de una clase abstracta AbstractTest que previamente a ejecutarse los tests de la clase se encarga de copiar toda la estructura de directorios a ser utilizados para los casos de prueba ubicada en un directorio fuente <sources_path> a el <workspace_test_path> y luego que en la instancia de Eclipse levantada para las pruebas se importen como proyectos en el workspace <workspace_test_path>. El <sources_path> está hardcodeado en la clase AbstractTest. También permite obtener una instancia de la implementación de los motores utilizados. Se corrió cada clase con una configuración de Junit particular.

Se realizaron seis clases básicas:

- `EngineSelection`. Implementa los casos de prueba de selección de motor. Se utilizó con la extensión con ModelMorf. Se chequea que los motores instalados en la plataforma y detectados por reflection sean exactamente ModelMorf y MediniQVT. También se busca instanciar un motor con nombre no existente.
- `EngineSelectionWithExclusions`. Contiene una lógica similar a la de la clase EngineSelection, utilizando la extensión de ModelMorf. La diferencia es que corre sobre una configuración donde a la instancia de Eclipse Application se le excluyen los plugins de MediniQVT. Por ello, se chequea que en la lista de motores detectados por el plugin no esté conformada sólo por el motor ModelMorf.
- `MetamodelsTest`. Implementa los casos de prueba de registrar metamodelos.
- `ConfigurationsTest`. Implementa los casos de prueba de crear/modificar/eliminar configuraciones. Para el caso de probar un script qvt con errores de sintaxis se toma en cuenta la salida de la clase utilitaria ScriptQVTTranformationsSearcher que según ya vimos tiene la responsabilidad de parsear las firmas de las transformaciones.
- `ExecutionTest`. Implementa los casos de prueba de ejecución de transformaciones.
- `XSLTTest`. Implementa los casos de prueba de las transformaciones xslt.

El <sources_path> utilizado para las pruebas tiene como contenido los siguientes directorios:

- BPMN2SOAML: contiene scripts de transformaciones qvt bien y mal formadas y archivos de metamodelos para definir transformaciones bpmn2->soaml. Contiene además modelos de bpmn en formato bpmn y xmi y una transformación xslt de bpmn2 a xmi.
- UML2RDBMS: contiene scripts de tranformación qvt, archivos de metamodelo y modelos de entrada para realizar transformaciones uml->rdbms.

5.2 Resultados obtenidos

Los resultados fueron variando acorde a las diferentes versiones del plugin. En el proceso de desarrollo del plugin, se identifican cinco versiones. Cada una se distingue en funcionalidades implementadas y casos de uso abarcados. Las versiones que se tuvieron son:

- 1.0. El objetivo de esta primer versión fue comprender el uso y manejo de las configuraciones de Eclipse y cómo se definen en el MediniQVT desde su API. Sólo se implementó el alta y modificación de configuraciones en el MediniQVT, junto con las validaciones necesarias para tener configuraciones correctas (sin nombre vacío, con modelo de entrada y salida, etc.). No están presentes la etapa de selección del motor ni la de ejecución en el motor de la transformación asociada a la configuración.
- 1.1. A esta versión se le agrega la baja de configuraciones en el MediniQVT. Se termina de ensamblar el uso de las configuraciones de Eclipse con el motor Medini. Además, se incorporan el paso previo de registro de metamodelos en el Medini y el de la ejecución de la transformación. Esta versión tuvo como hito la ejecución exitosa de una transformación desde el plugin desarrollado, registrando metamodelos y definiendo una configuración adecuada.
- 1.2. Esta versión incorpora la posibilidad de realizar una transformación previa a la ejecución del motor. Partiendo del modelo original, en formato bpmn2, se pudo llegar a un modelo de salida, transformando el modelo original en modelo de entrada de la configuración mediante una XSLT y luego ejecutando en el Medini. La transformación XSLT queda asociada a la configuración.
- 1.3. Esta versión incorpora el paso de selección del motor. Si bien sólo se cuenta con el motor Medini hasta esta etapa, se implementa la detección de los motores instalados mediante Reflection y el listado del primer paso.
- 1.4. Se incrementa la versión 1.3 con la implementación de la extensión para el motor ModelMorf. Se realiza el registro de metamodelos y alta, modificación y baja de configuraciones asociadas al motor ModelMorf. Esta versión tuvo como hito la capacidad tangible del plugin de poder ser extendido para otros motores QVT. Definiendo una configuración, se logra la ejecución exitosa de la misma en el ModelMof desde el plugin desarrollado.
- 1.5. Es la versión actual del plugin. Se permite generar el modelo de entrada mediante una transformación XSLT opcionalmente, definiendo los parámetros de la misma en una configuración de ModelMorf, al igual que se hizo para Medini en la versión 1.2.

Cabe mencionar que las versiones mayores incluyen depuración de bugs que aparecieron en

funcionalidades de versiones anteriores.

En la tabla siguiente se muestra para cada versión del plugin, cuáles casos de prueba fueron exitosos y cuáles no. La 'X' indica que fue exitoso.

Caso de prueba\Versión		1.0	1.1	1.2	1.3	1.4	1.5
Selección de motor inválido					X	X	X
Detección y listado de motores instalados					X	X	X
MediniQVT	Registrar metamodelo con ruta inválida		X	X	X	X	X
	Registro exitoso de metamodelos		X	X	X	X	X
	Listado de metamodelos registrados		X	X	X	X	X
	Crear config. con nombre inválido	X	X	X	X	X	X
	Crear config. con ruta a script qvt inválida	X	X	X	X	X	X
	Crear config. sin definir dirección o transformación	X	X	X	X	X	X
	Crear config con ruta modelo de entrada/salida inválida	X	X	X	X	X	X
	Crear config. Con directorio de traza inválido	X	X	X	X	X	X
	Crear config. Válida con nombre duplicado.	X	X	X	X	X	X
	Eliminar configuración existente		X	X	X	X	X
	Ejecutar config. A partir de config. inexistente		X	X	X	X	X
	Ejecutar config. A partir de config existente sin registrar metamodelos		X	X	X	X	X
	Ejecutar config. Con confi. Válida y metamodelos registrados		X	X	X	X	X
	Transformación xslt con errores			X	X	X	X
	Transformación xslt con modelo de entrada con errores o inexistente			X	X	X	X

	Transformación xslt con modelo de salida bloqueado			X	X	X	X
ModelMorf	Registrar metamodelo con ruta inválida					X	X
	Registro exitoso de metamodelos					X	X
	Listado de metamodelos registrados					X	X
	Crear config. con nombre inválido					X	X
	Crear config. con ruta a script qvt inválida					X	X
	Crear config. sin definir dirección o transformación					X	X
	Crear config con ruta modelo de entrada/salida inválida					X	X
	Crear config. Con directorio de traza inválido					X	X
	Crear config. Válida con nombre duplicado.					X	X
	Eliminar configuración existente					X	X
	Ejecutar config. A partir de config. inexistente					X	X
	Ejecutar config. A partir de config existente sin registrar metamodelos					X	X
	Ejecutar config. Con confi. Válida y metamodelos registrados					X	X
	Transformación xslt con errores						X
	Transformación xslt con modelo de entrada con errores o inexistente						X
	Transformación xslt con modelo de salida bloqueado						X

6. Caso de estudio

El caso de estudio se divide en dos secciones: la transformación de un modelo bpmn2 a SoaML con el motor MediniQVT y con el motor ModelMorf, y una transformación de un modelo uml a rdbms con el motor ModelMorf para mostrar el requisito de la extensibilidad del plugin.

6.1 Caso de estudio con transformaciones dadas

Se mostrará como con el plugin desarrollado se puede definir y transformar un modelo en bpmn2 convertido a formato de intercambio xmi a un modelo en SoaML expresado también en xmi. El modelo bpmn2 sobre el cual se realizará la transformación se muestra en la figura 6.1.

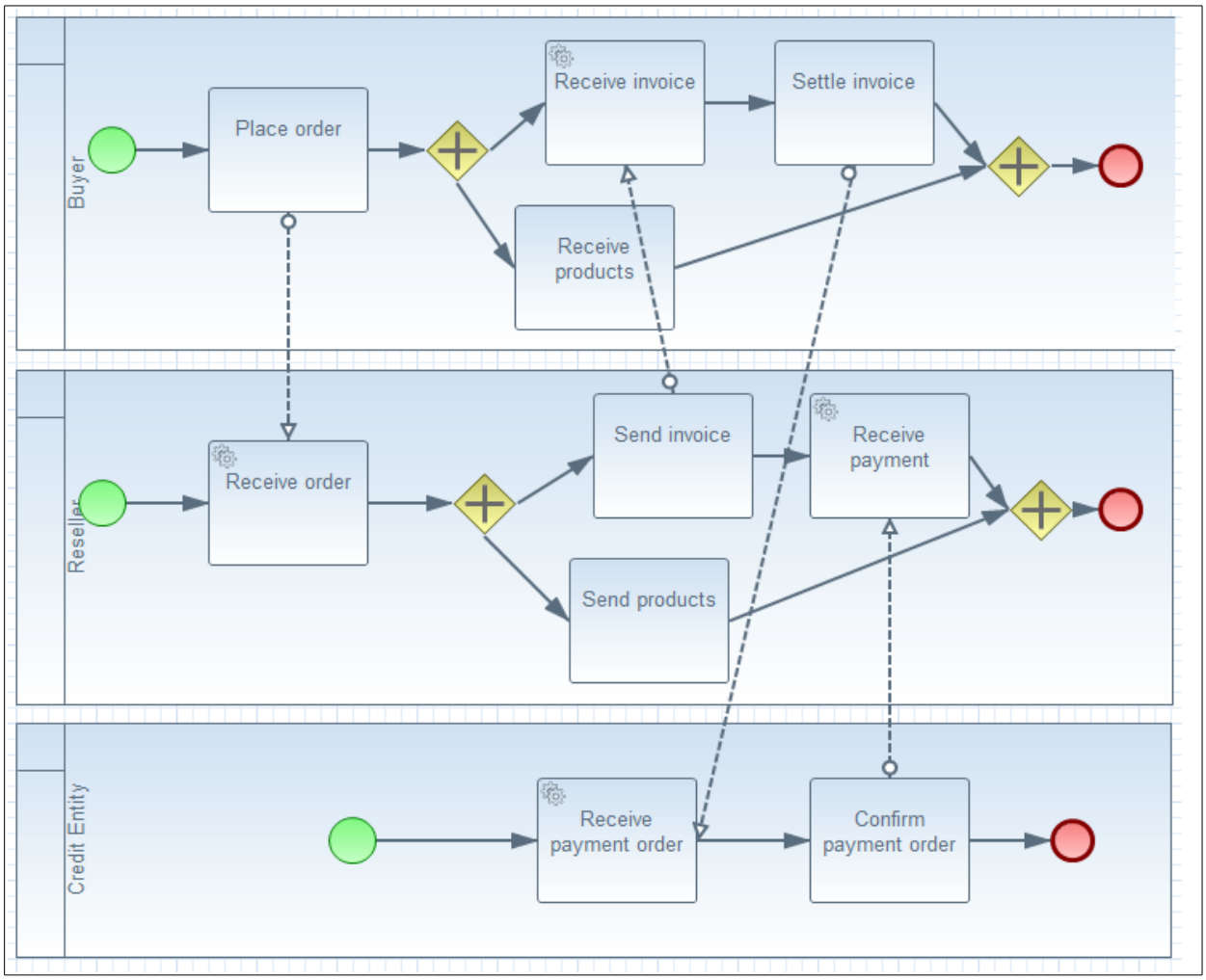


Figura 6.1: Modelo bpmn del caso de estudio

En el proceso se muestran tres participantes (pools): Credit Entity (entidad crediticia), Reseller (revendedor) y Buyer (comprador). El comprador comienza realizando una orden de compra que la se envía al revendedor. El revendedor puede en ese momento enviar los productos de la orden de compra al comprador y finalizar la venta o emitir una factura de compra (invoice). En el segundo

caso, cuando el comprador liquida la factura puede realizar el pago a crédito. La entidad crediticia recibe la orden de pago. Luego de confirmada, el revendedor recibe el pago correspondiente a la factura y finaliza la venta.

En la transformación con ambos motores, el primer paso es abrir una instancia el entorno Eclipse con el plugin pgsoaqvt instalado. En la barra del menú debe aparecer el menú Pg SOAQVT que indica que el plugin está disponible.

6.1.1 MediniQVT

Se va necesitar un workspace con todos los recursos necesarios: metamodelos, scripts de transformaciones, modelo de entrada y directorio de transformaciones. La figura 6.2 rebela la estructura del workpsace.

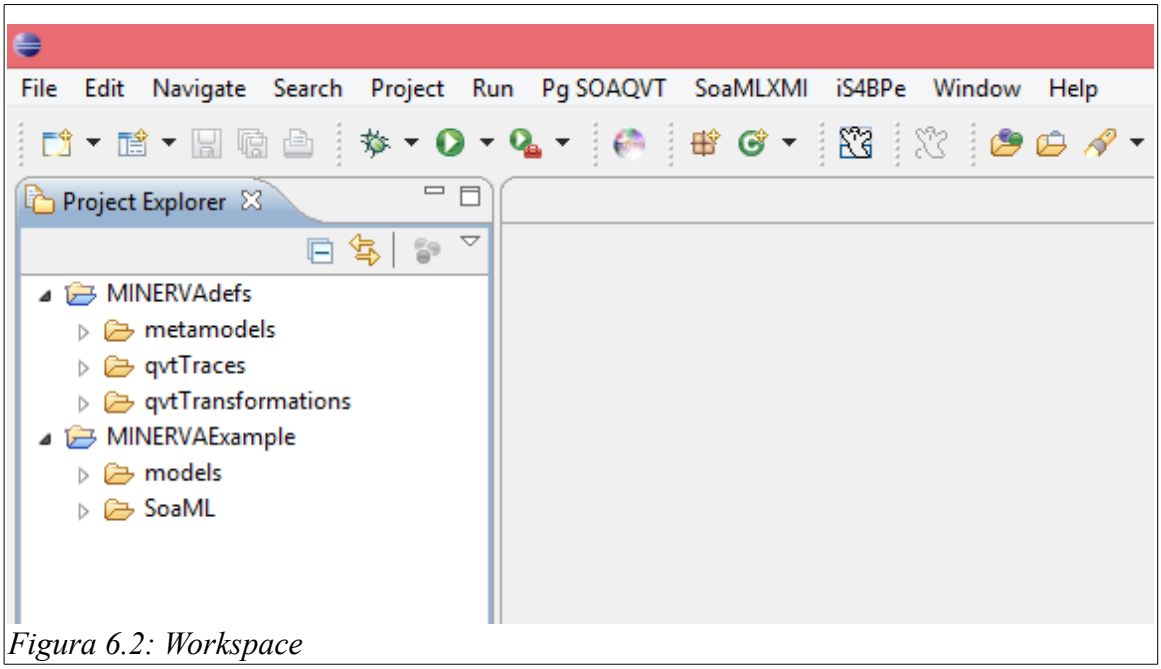


Figura 6.2: Workspace

Se detalla a continuación la estructura de directorios de un workspace de ejemplo:

- En MINERVAdefs están todos los archivos que son imprescindibles para poder definir una transformación. En el subdirectorio "metamodels" se agregaron los metamodelos necesarios, el subdirectorio "qvtTraces" es en el que se van a persistir la traza de las transformaciones y el subdirectorio "qvtTransformations" contiene la lista de scripts de transformaciones qvt para el motor
- En MINERVAExample, se tienen los modelos de entrada en formato xmi y es donde se van guardar los modelos de salida resultantes de las transformaciones.

Se comienza por el primer paso de la transformación. Para eso seleccionamos la opción del menú “Select engine” para en primera instancia seleccionar el motor de transformación. En la figura 6.3

se selecciona de la lista el motor MediniQVT y se avanza al siguiente paso presionando Next.

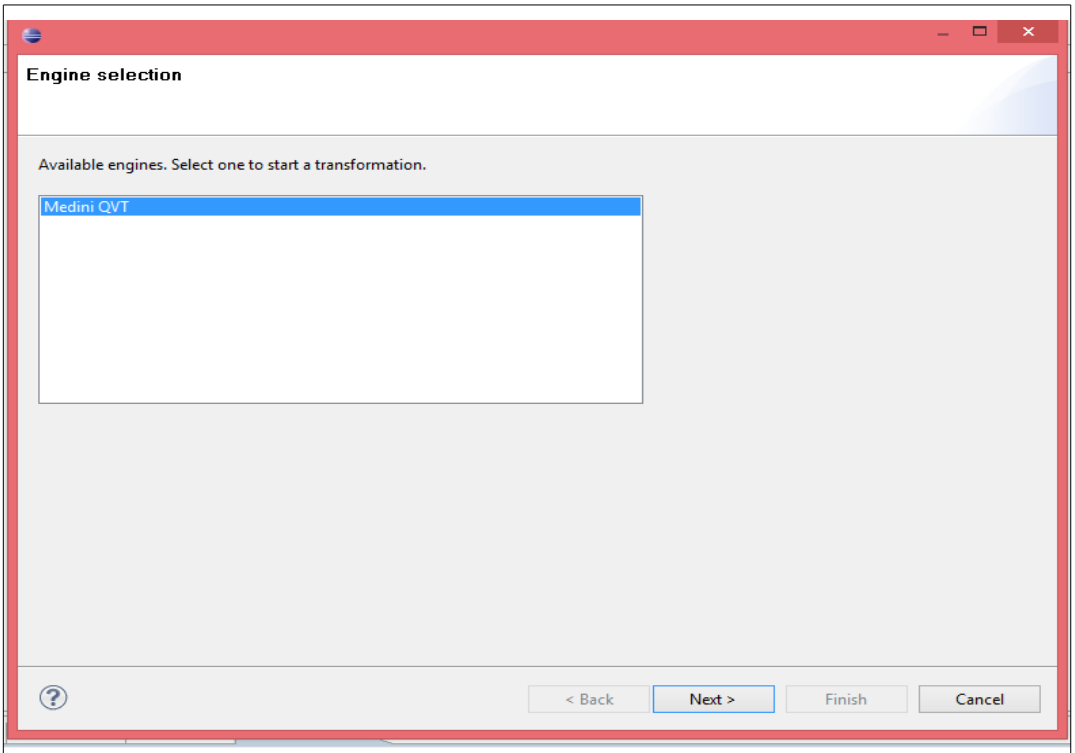


Figura 6.3

El siguiente paso que se presenta es para registrar metamodelos. Suponiendo que no hay ningún metamodelo registrado, nos aparece una lista vacía, tal como se ilustra en la figura 6.4. No se permite avanzar al siguiente paso hasta no tener registrado al menos un metamodelo.

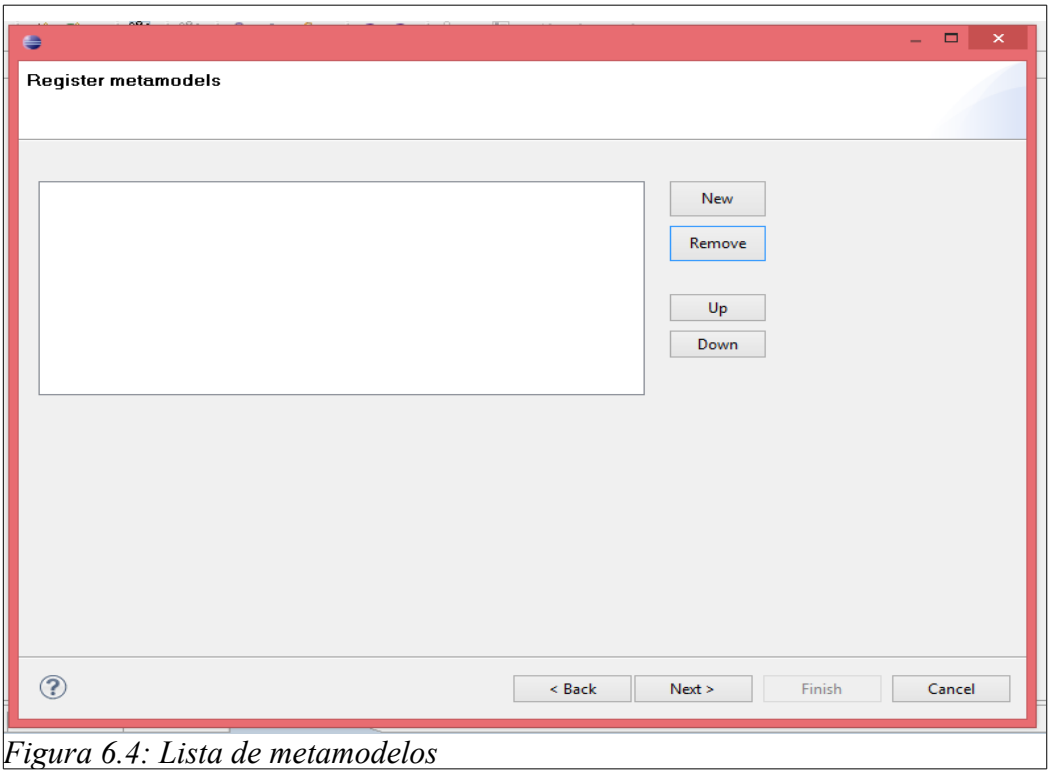


Figura 6.4: Lista de metamodelos

Se agregan los metamodelos correspondientes a la transformación a definir. Se hace click en el botón New y se añaden las rutas relativas al workspace a los metamodelos Bpmn20.ecore, Soaml.ecore y uml.ecore (que dan las definiciones de los modelos utilizados). La figura 6.5 muestra un explorador con los metamodelos en MINERVADefs/metamodels . La figura 6.6 muestra los metamodelos agregados con sus respectivas rutas a la lista de la ventana del paso 2.

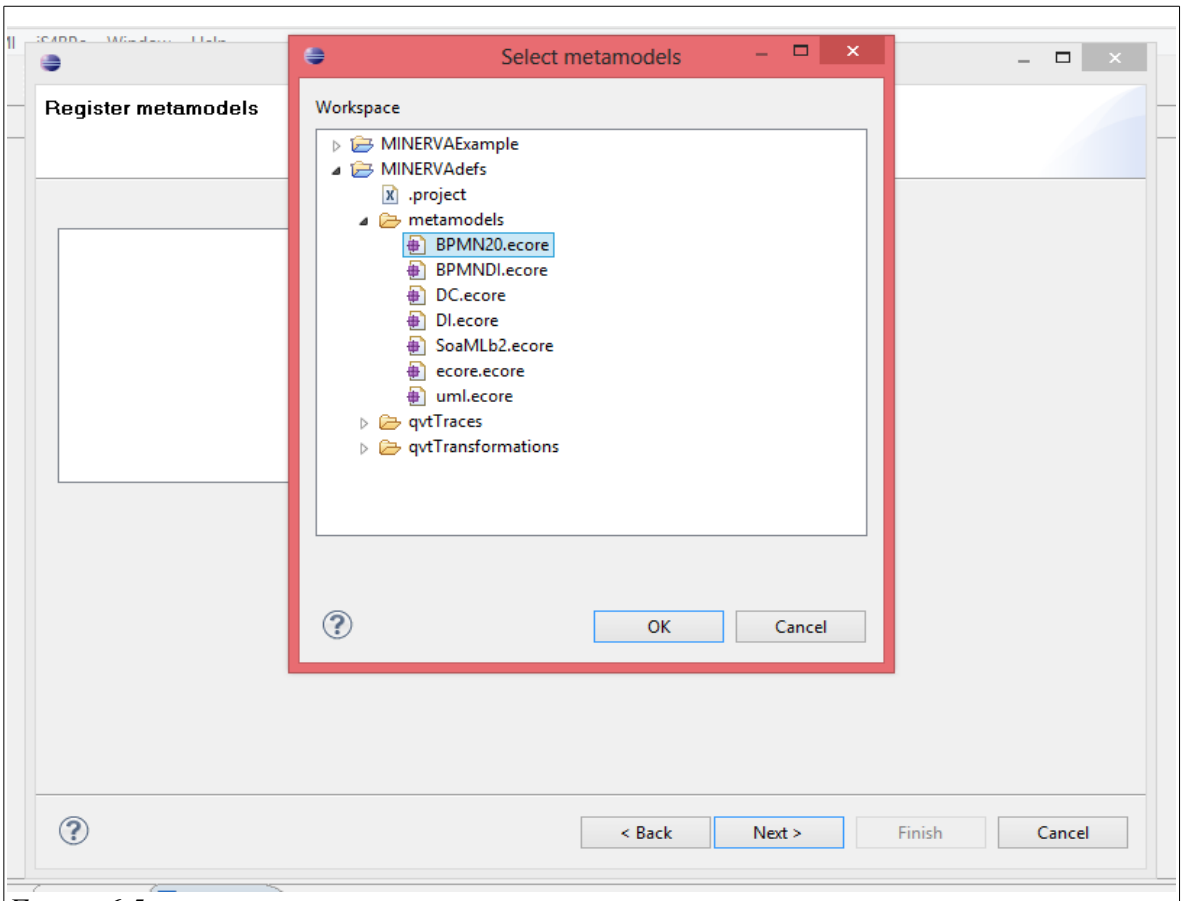


Figura 6.5

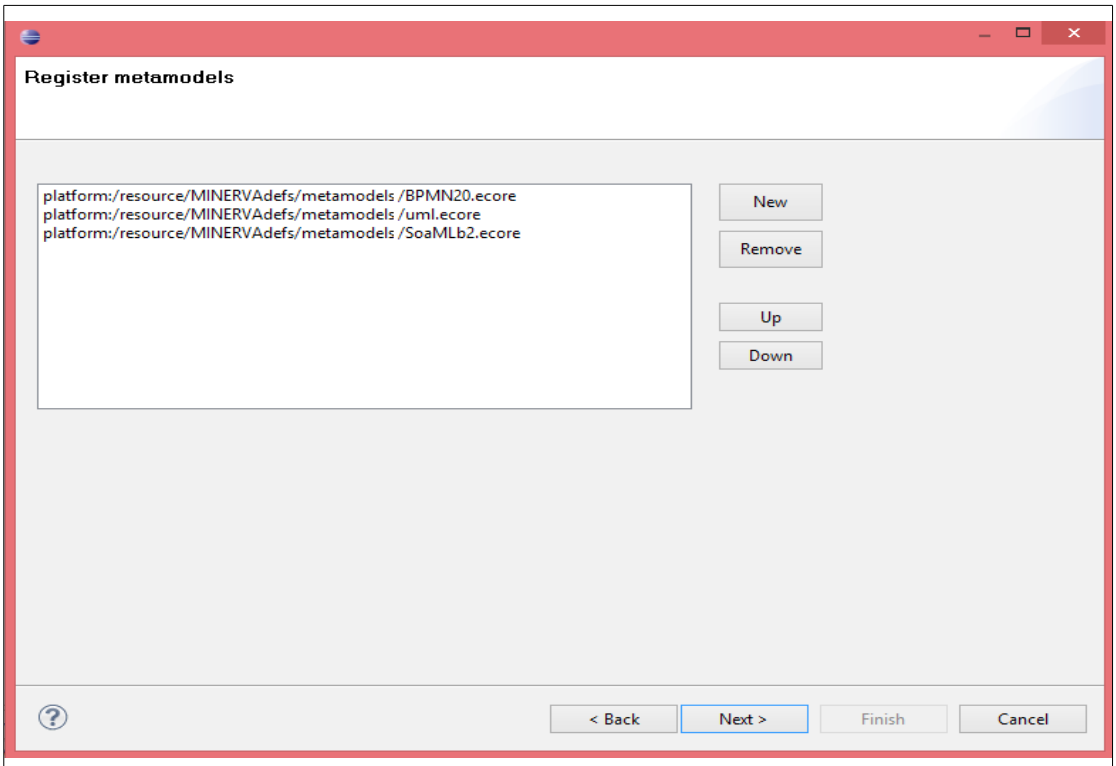


Figura 6.6

En el paso subsiguiente, se creará una nueva configuración para la transformación deseada. Damos Next y nos encontramos con la ventana que permite dar de alta las transformaciones. Por defecto, tenemos los datos vacíos de una posible nueva configuración, como en la figura 6.7.

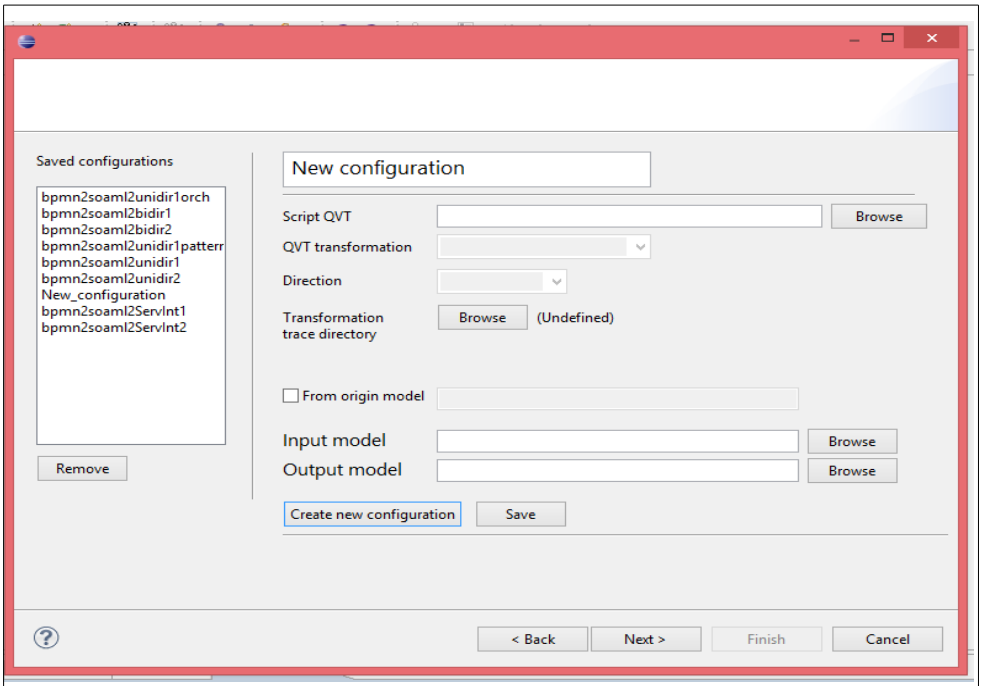


Figura 6.7: Configuración vacía

En primera instancia se selecciona del workspace la ruta al script de la transformación. Del subdirectorio qvtTransformations elegir el script bmpn2soaml2unidir2 que define una transformación qvt de bpmn a soaml. El script seleccionado aparece en la figura 6.8.

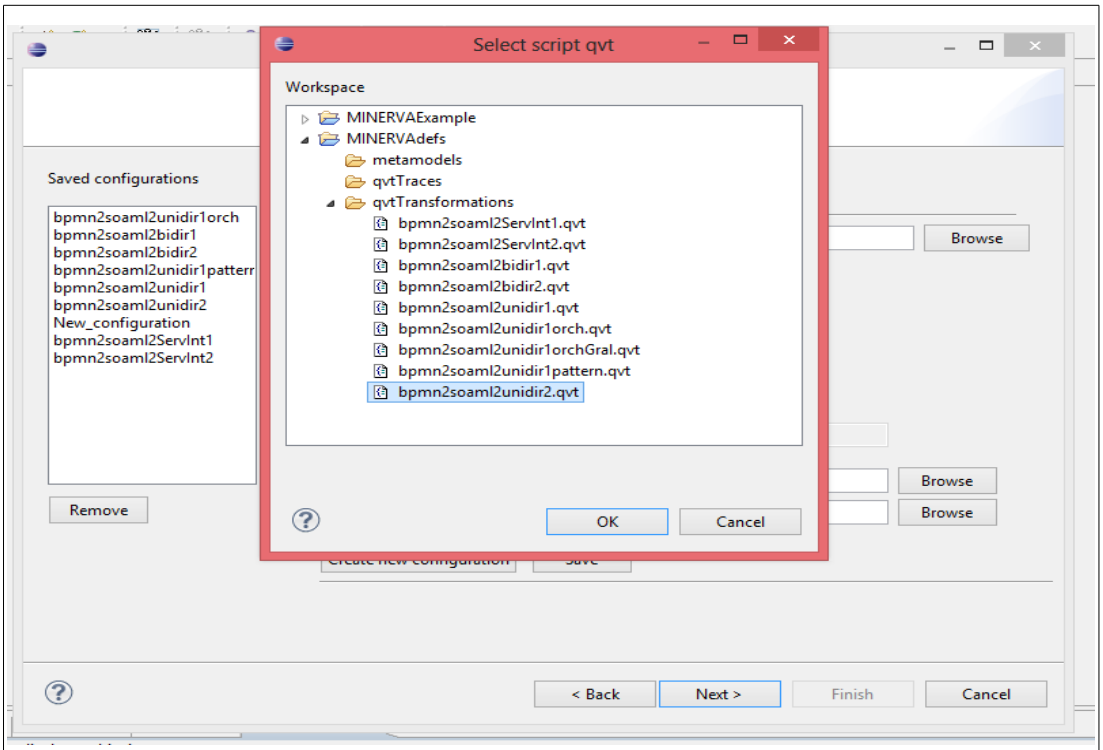


Figura 6.8

Se especifica la transformación (única) del script de nombre bpmn2soaml2 y la dirección hacia soaml. Luego agregar un directorio de transformación en MINERVAdefs/qvtTraces haciendo click en el botón "Browse" junto a la etiqueta "Trace directory". La figura 6.9 muestra el directorio a seleccionar.

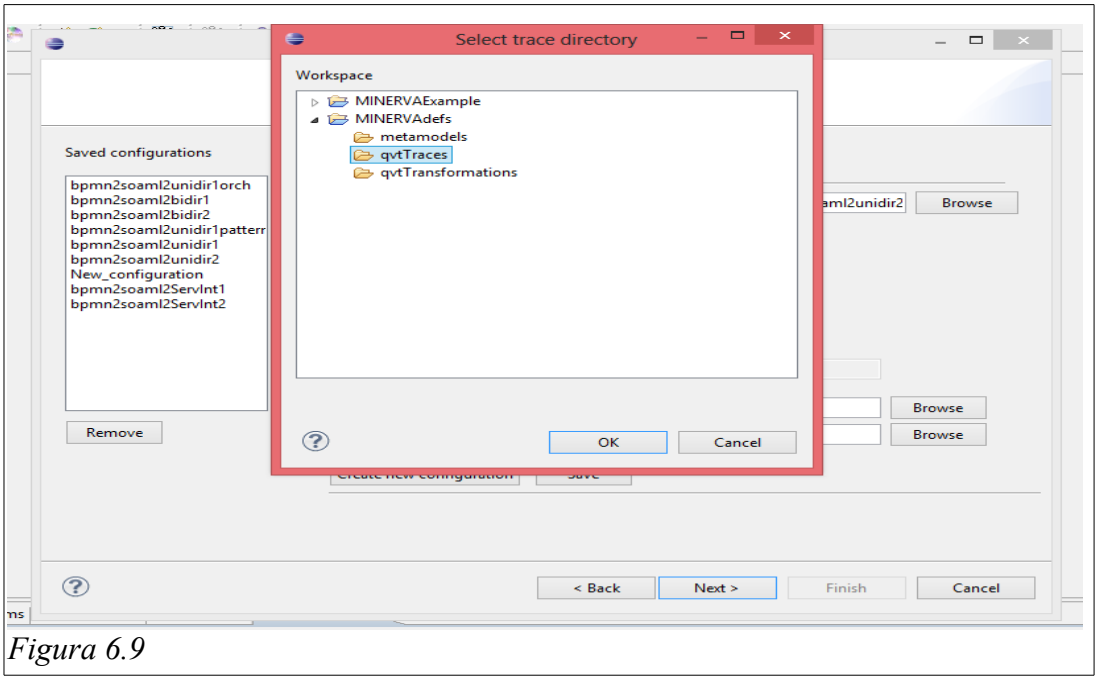


Figura 6.9

Por último, se definen cuáles van a hacer los modelos de entrada y salida. Como modelo de entrada, seleccionamos el archivo (del file system), el ResellerBP.xmi que está bajo MINERVAExample/models tal como se muestra en la figura 6.10.

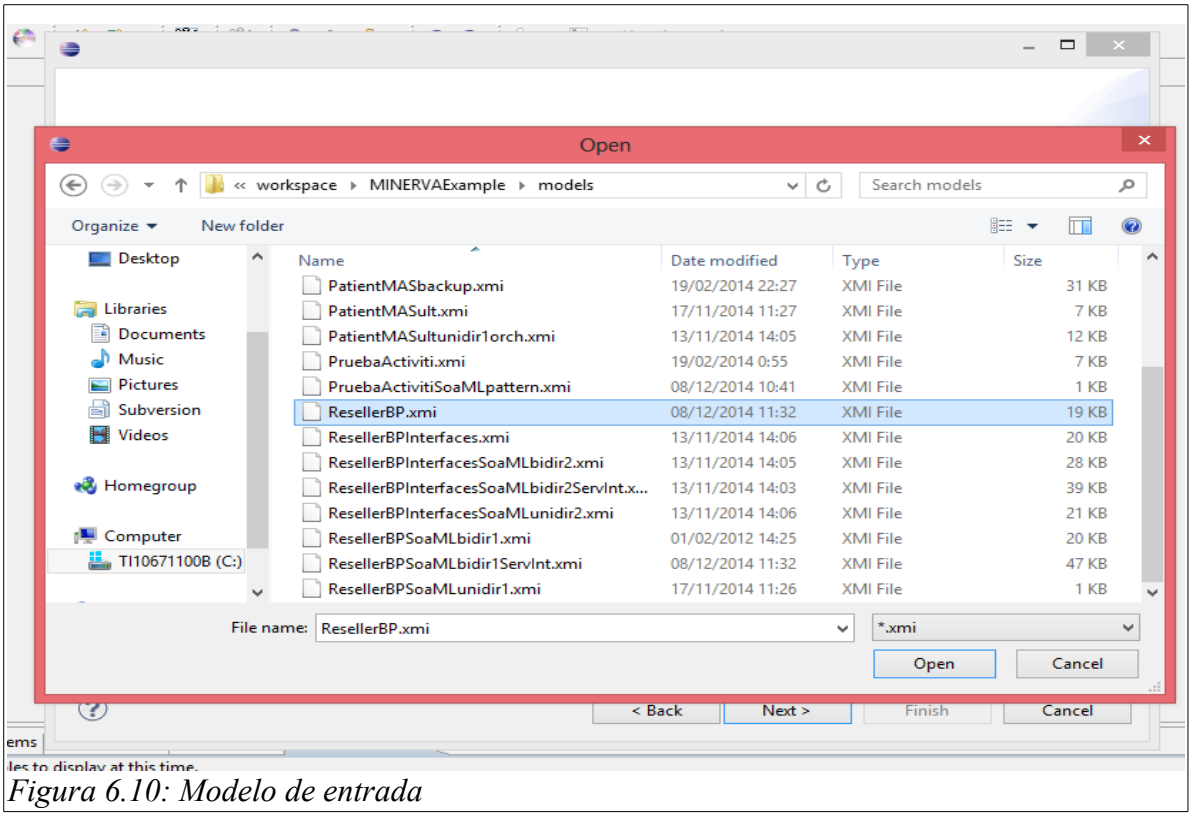


Figura 6.10: Modelo de entrada

Y se define una ruta al modelo de salida como C:\Workspace\resultado.xmi. Se le da el nombre de la nueva configuración como "Configuración de prueba". La figura 6.11 muestra la nueva configuración con todos sus datos.

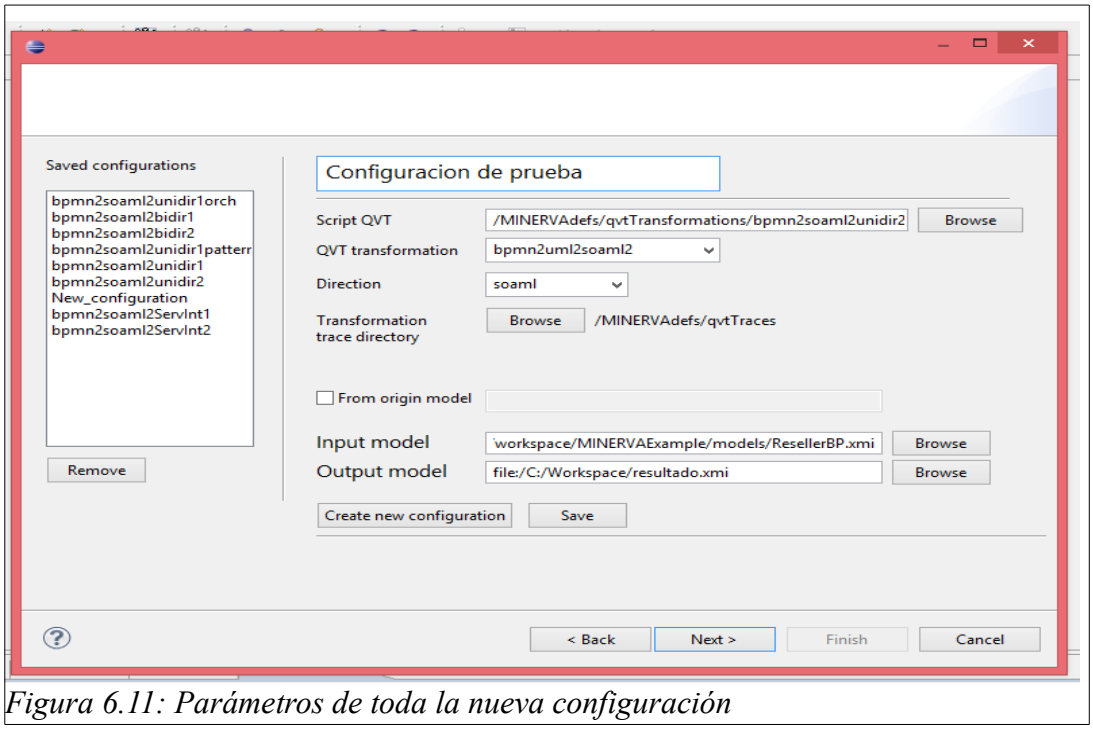


Figura 6.11: Parámetros de toda la nueva configuración

Se procede presionando en "Save" para guardar la nueva configuración en el motor. Como todos los parámetros fueron correctos, se salva la misma con éxito. Luego haciendo click en Next se avanza al siguiente paso. En este, que se muestra en la figura 6.12, se listan los nombres de todas las configuraciones definidas, entre ellas la recientemente creada.

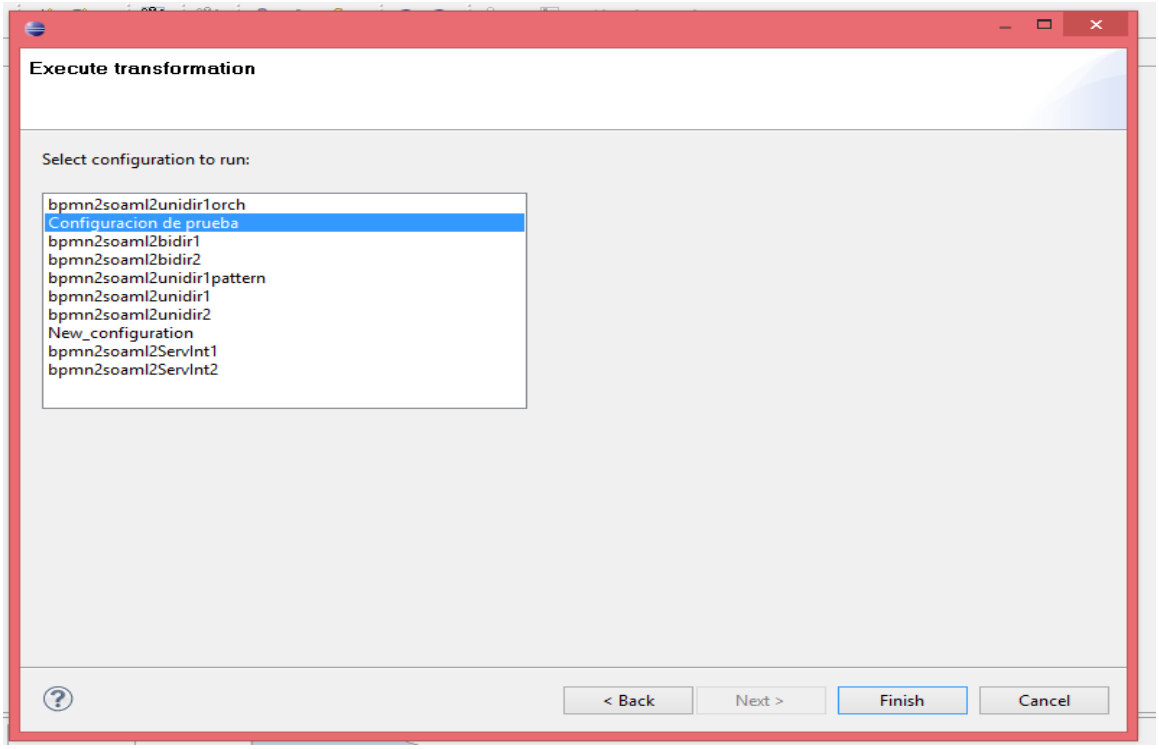


Figura 6.12: Selección de la nueva configuración

Se elige la misma y luego click en Finish. Se ejecuta la transformación de la configuración en el motor y se muestra en la figura 6.13 el log de la transformación en la vista de la consola.

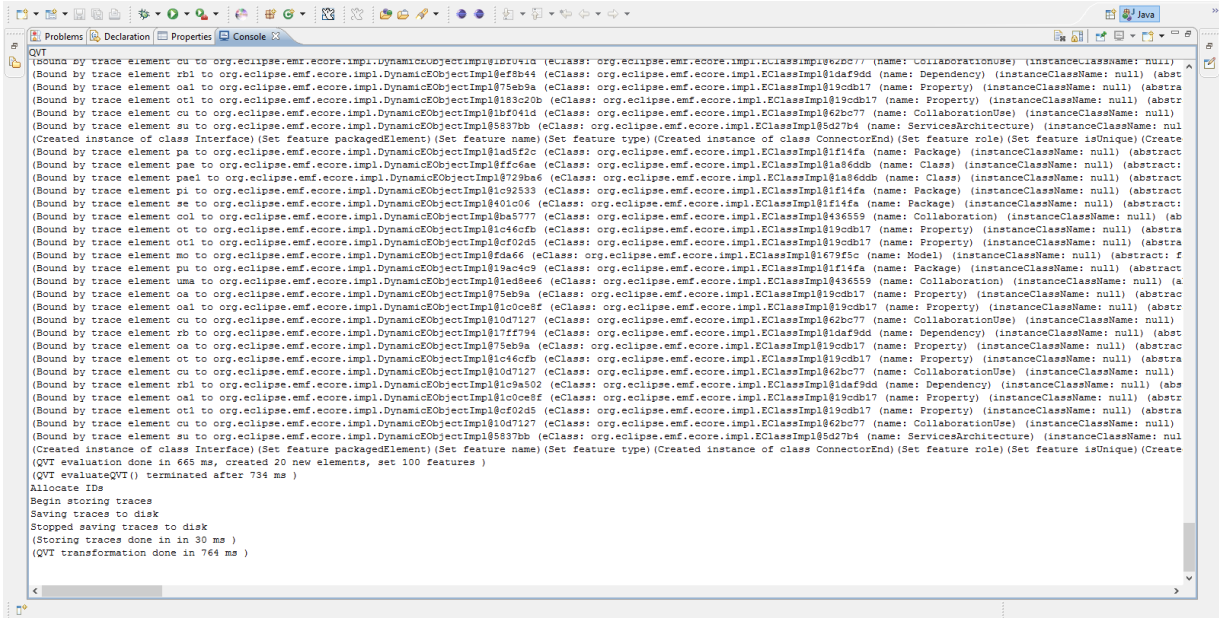


Figura 6.13: Log de la transformación

Al finalizar la ejecución, se despliega un mensaje de éxito en pantalla, tal como ocurre en este caso y se muestra en la figura 6.14.

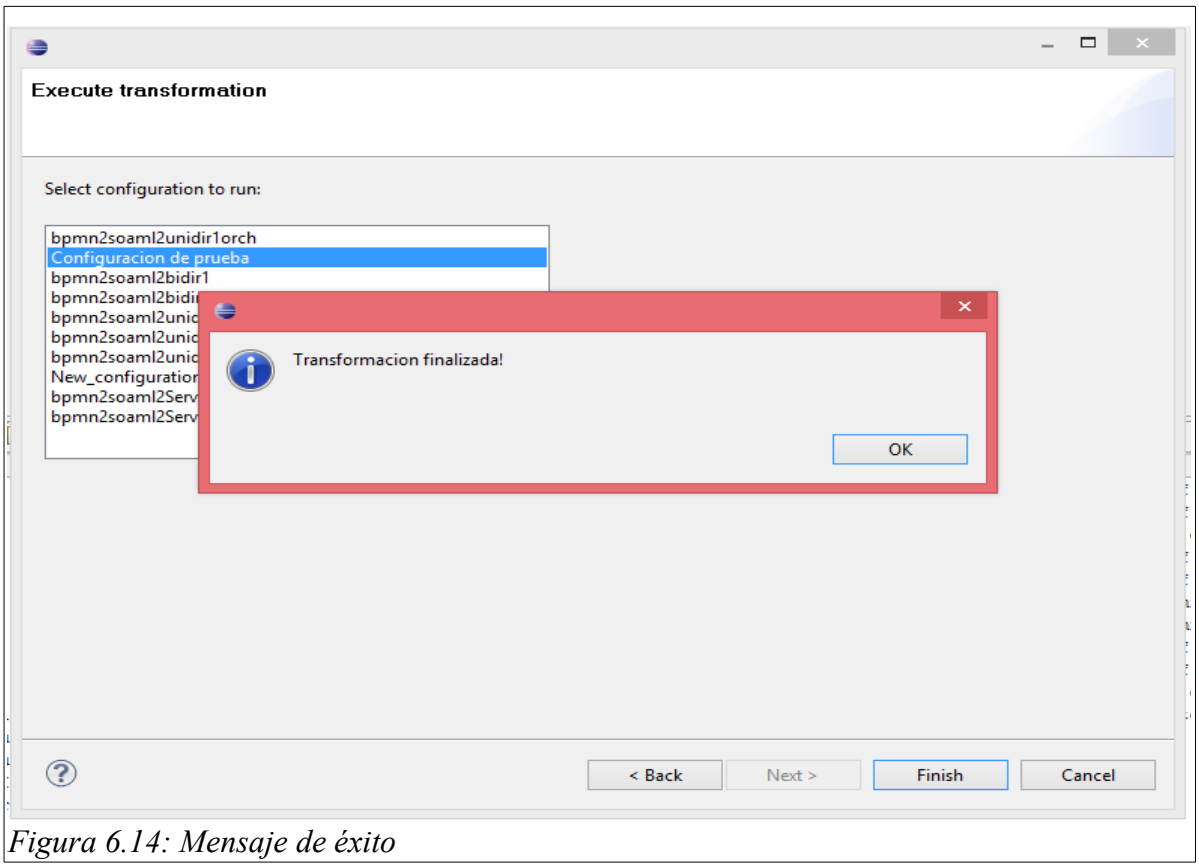


Figura 6.14: Mensaje de éxito

Si abrimos el archivo C:\Workspace\resultado.xmi desde un editor de texto, se verá el modelo de salida generado con MedinQVT con elementos de SoaML.

6.1.2 ModelMorf

Luego de utilizar Medini como motor, se quiso ejecutar la misma transformación en otro motor. El motor elegido fue el ModelMorf que es el utilizado en el caso de extensibilidad que se presenta en la sección 6.2 (donde se describirán los detalles particulares del motor y su utilización). El objetivo en esta instancia es comprobar si es factible ejecutar exitosamente una configuración definida en MediniQVT en ModelMorf.

Se encontraron algunos problemas de compatibilidad para reutilizar la misma configuración.

En primer lugar, como se definió en la transformación de Medini de la sección anterior, se utilizaron metamodelos en formato ecore (BPMN20.ecore, uml.ecore y SoaMLb2.ecore) que es la implementación de emof para Eclipse. ModelMorf solo acepta los metamodelos en formato xml definidos mediante la especificación emof (Essential MOF) o cmof (Complete MOF). Por otra parte, el motor ModelMorf no resuelve las referencias externas entre archivos de metamodelo y solo permite recibir hasta dos metamodelos por línea de comando, cada uno correspondiente a los parámetros de la transformación qvt (en la especificación del motor descargado de [16] en el apartado "Features with implementation restrictions" lo menciona explícitamente: “Only two non-

primitive domains are supported"). Ergo, el metamodelo SoaML de Medini no puede ser usado de esta forma en ModelMorf debido a que incluye una referencia a las definiciones uml en uml.ecore. Lo mismo ocurre con el BPMN20. ecore que utiliza elementos de los metamodelos BPMNDI.ecore, DI.ecore y DC.ecore que son referenciados. Debido a esta limitante del motor, se debe que utilizar versiones reducidas de elementos en los metamodelos. La solución para este problema siguió los siguientes pasos:

1. Convertir de formato ecore a formato emof y renombrando con la extensión xml de los metamodelos BPMN20.ecore, SoaML.ecore, BPMNDI.ecore, DI.ecore y DC.ecore. Esto se hizo posible utilizando las Ecore Tools de Eclipse [32].
2. En el metamodelo BPMN20.xml de bpmn suprimir las definiciones que produzcan error de dependencias.
3. En el metamodelo SoaMLb2.xml de soaml sustituir las definiciones de los elementos que referencian al namespace del metamodelo uml.ecore por las equivalentes en el namespace de emof http://schema.omg.org/spec/mof/2.0/emof.xmi.
4. En el modelo de entrada bpmn cambiar la localización del esquema con los metamodelos necesarios: BPMN20.xml y BPMNDI.xml.

En segundo lugar, hay algunas diferencias en la sintaxis y las definiciones en el script qvt exigida por Medini y por ModelMorf. Ellas son:

- los parámetros de la firma de la transformación deben estar separados por "," en Medini y por ";" en ModelMorf.
- no admite dos enforce domain en una top relation.
- debe recibir exactamente dos parámetros.
- solo admite dos dominios: uno enforce y otro checkonly.

Para correr la transformación, hubo que modificar la qvt utilizada (bpmn2soaml2bidir.qvt) para dejarla sintácticamente correcta según las restricciones de ModelMorf.

Se va a utilizar un proyecto diferente del utilizado para Medini para ejecutar la transformación en ModelMorf. El proyecto utilizado tiene la estructura de la figura 6.15.Los archivos del proyecto

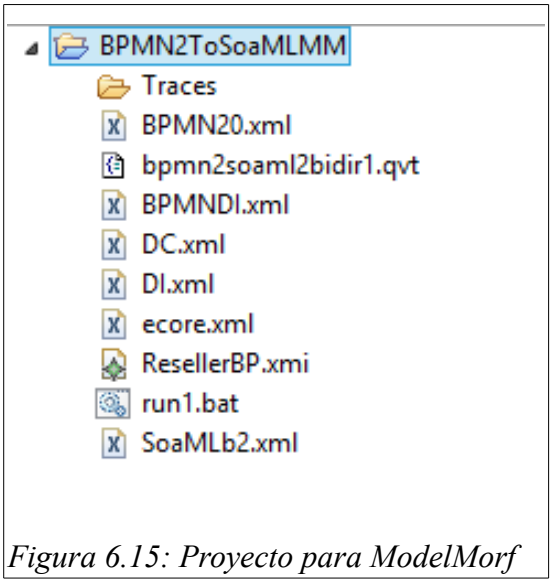


Figura 6.15: Proyecto para ModelMorf

son:

- BPMN20.xml y SoaMLb2.xml. Metamodelos para los modelos de entrada y salida respectivamente parámetros de la configuración.
- Bpmn2soaml2bidir1.qvt. Transformación qvt utilizada en el caso con Medini pero ajustada para ModelMorf.
- ResellerBP.xmi. Modelo de entrada.
- BPMNDI.xml, DC.xml, DI.xml, ecore.xml. Metamodelos apuntados por el modelo de entrada.
- Traces. Directorio para almacenar la traza de la transformación.

Se comienza por el primer paso de la transformación. Para eso seleccionamos la opción del menú "Select engine" para en primera instancia seleccionar el motor de transformación. En la figura 6.16 se selecciona de la lista el motor ModelMorf y se avanza al siguiente paso presionando Next.

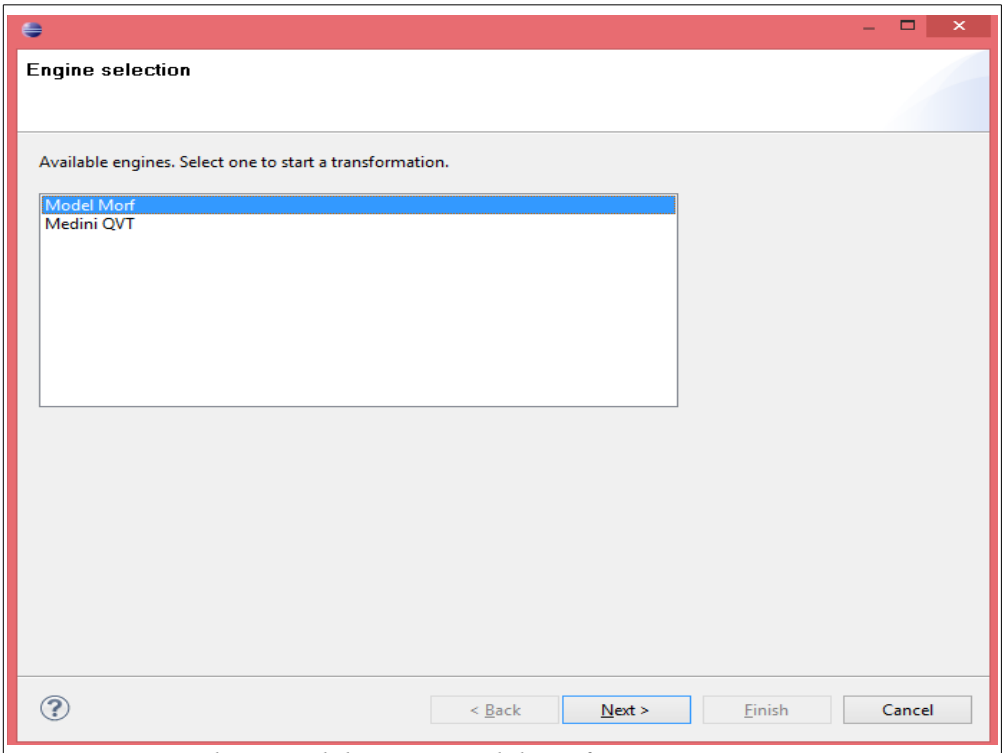


Figura 6.16: Selección del motor ModelMorf

El siguiente paso que se presenta es para registrar metamodelos. Suponiendo que no hay ningún metamodelo registrado, nos aparece una lista vacía. No se permite avanzar al siguiente paso hasta no tener registrado al menos un metamodelo. Se agregan los metamodelos correspondientes a la transformación a definir. Se hace click en el botón New y se añaden las rutas relativas al workspace a los metamodelos Bpmn20.xml y SoaMLb2.xml (que dan las definiciones de los modelos utilizados). La figura 6.17 muestra un explorador con los metamodelos xml a agregar. La figura 6.18 muestra los metamodelos agregados con sus respectivas rutas a la lista de la ventana del paso 2.

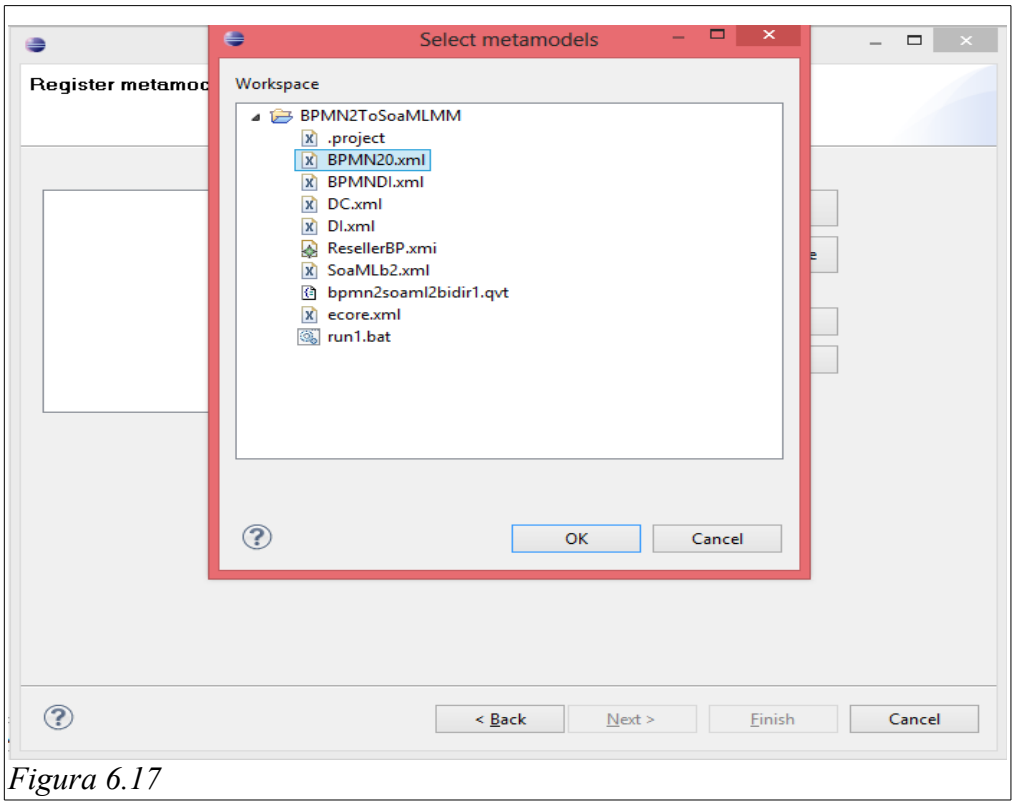


Figura 6.17

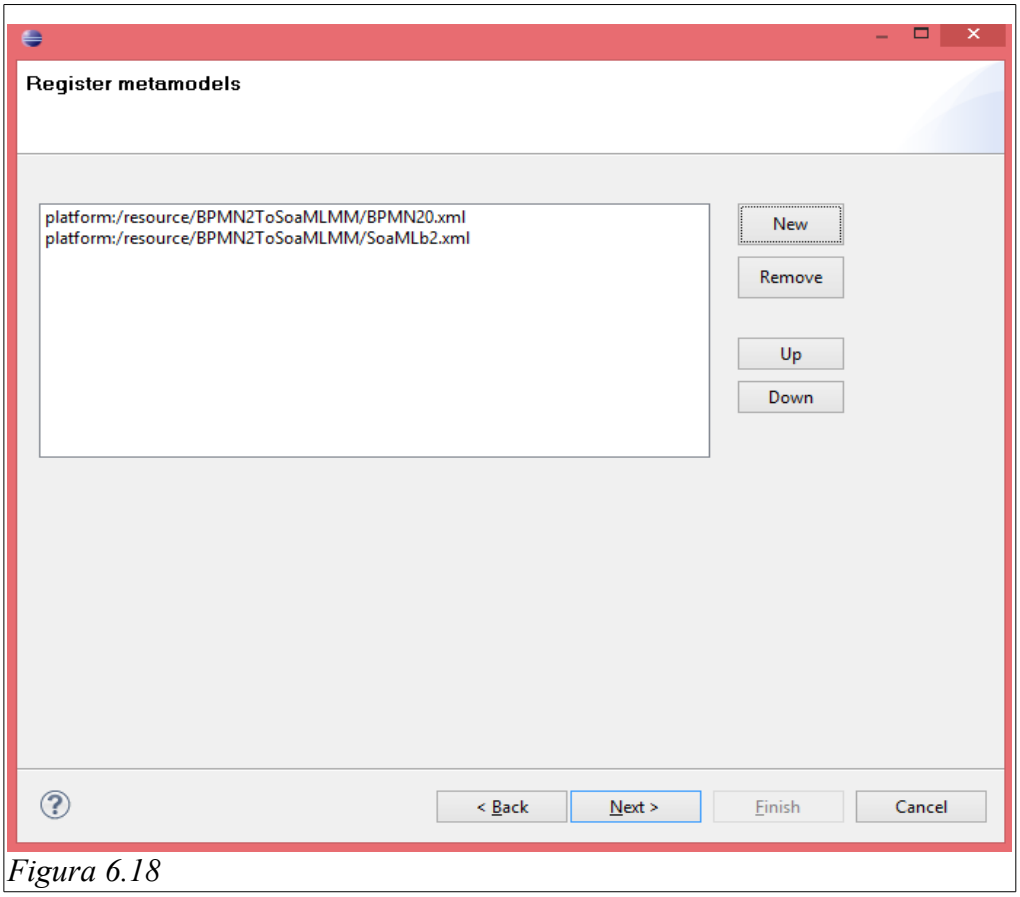


Figura 6.18

En el paso subsiguiente, se creará una nueva configuración para la transformación deseada. Damos Next y nos encontramos con la ventana que permite dar de alta las transformaciones. Se crea una configuración de nombre "bpmn2soaml2" con datos de la nueva configuración. El modelo de entrada es ResellerBP.xmi y el de salida tiene el nombre"ResellerBPSoaMLService.xmi". La transformación es bpmn2soaml2bidir1.qvt. La figura 6.19 muestra la configuración creada.

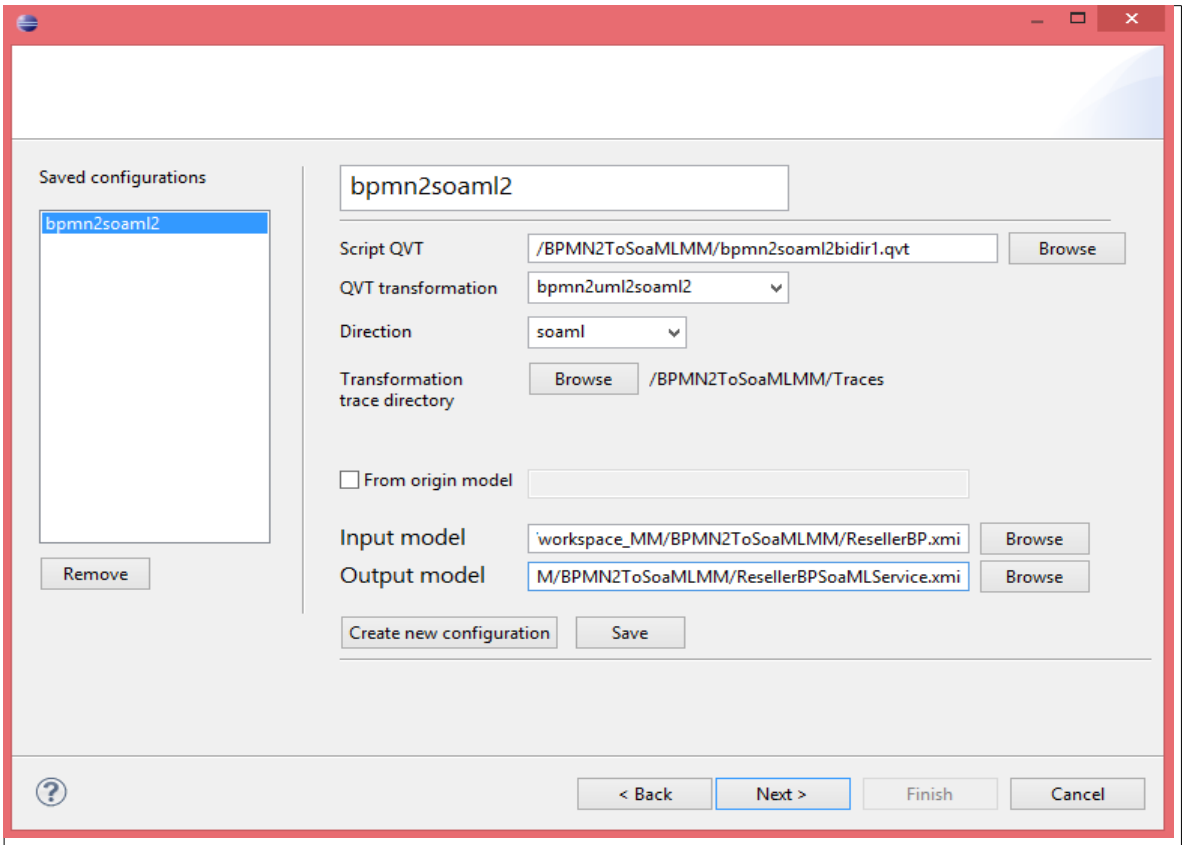


Figura 6.19: Configuración ModelMorf

Luego haciendo click en Next se avanza al siguiente paso. En este, que se muestra en la figura 6.20, se listan los nombres de todas las configuraciones definidas, entre ellas la recientemente creada.

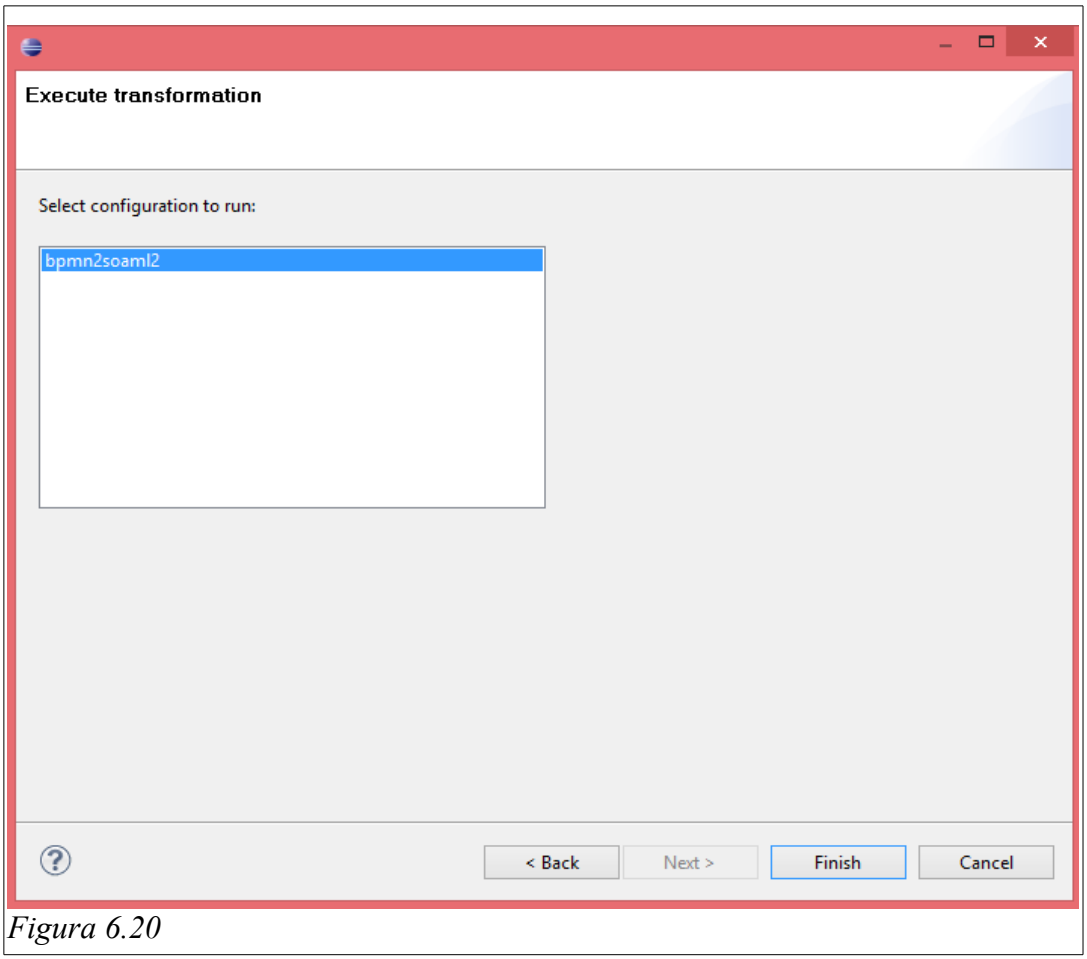


Figura 6.20

Se elige la misma y luego click en Finish. Se ejecuta la transformación de la configuración en el motor y se muestra en la figura 6.21 el log de la transformación en la vista de la consola.

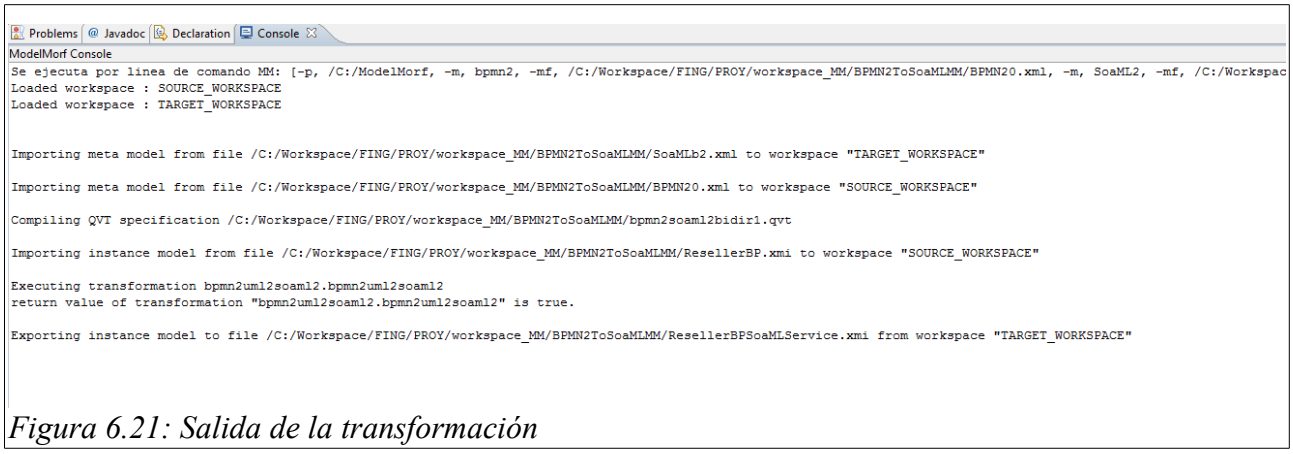


Figura 6.21: Salida de la transformación

Al finalizar la ejecución, se despliega un mensaje de éxito en pantalla, tal como ocurre en este caso y se muestra en la figura 6.22.

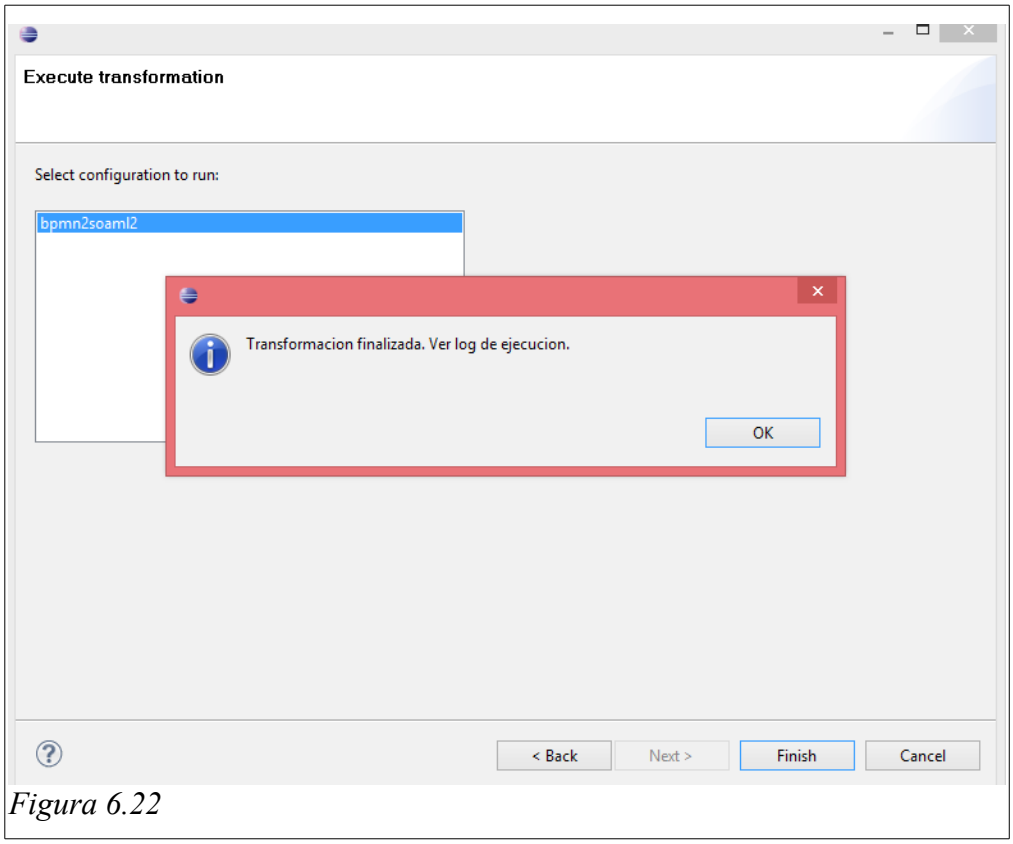


Figura 6.22

Si abrimos el archivo ResellerBPSoaMLService.xmi desde un editor de texto, se verá el modelo de salida generado con ModelMorf con elementos de SoaML.

6.1.3 Resultados

El modelo de salida se puede exportar a SoaML con algún plugin de Eclipse como [17]. Del empaquetado xmi /xml se obtienen los siguientes diagramas SoaML:diagrama de participantes (figura 6.23), diagrama de arquitectura de servicios (figura 6.24), diagrama de mensajes (figura 6.25), diagrama de contratos de servicio (figura 6.26) y el diagrama de interfaces (figura 6.27).

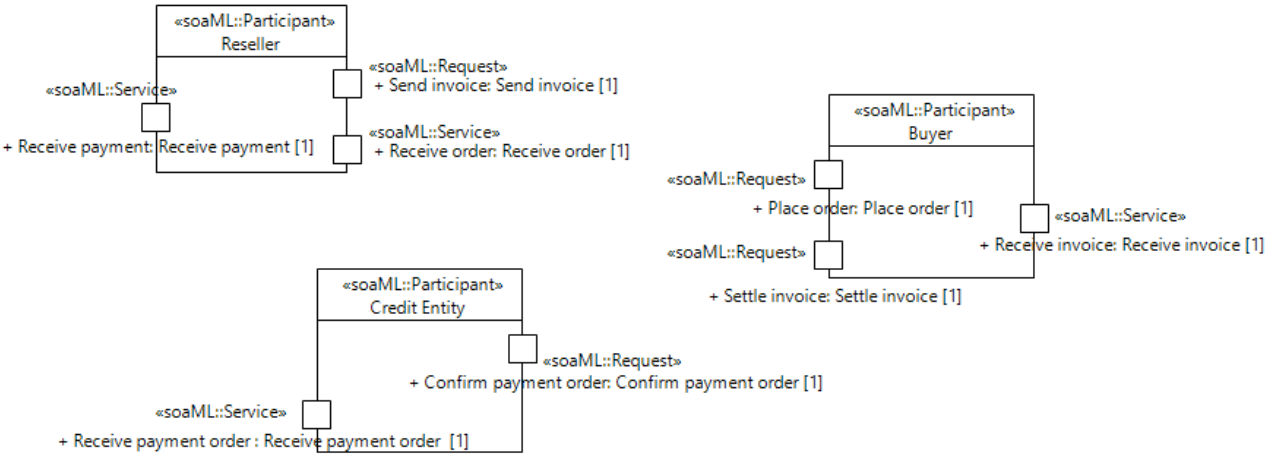


Figura 6.23: Diagrama de Participantes SoaML

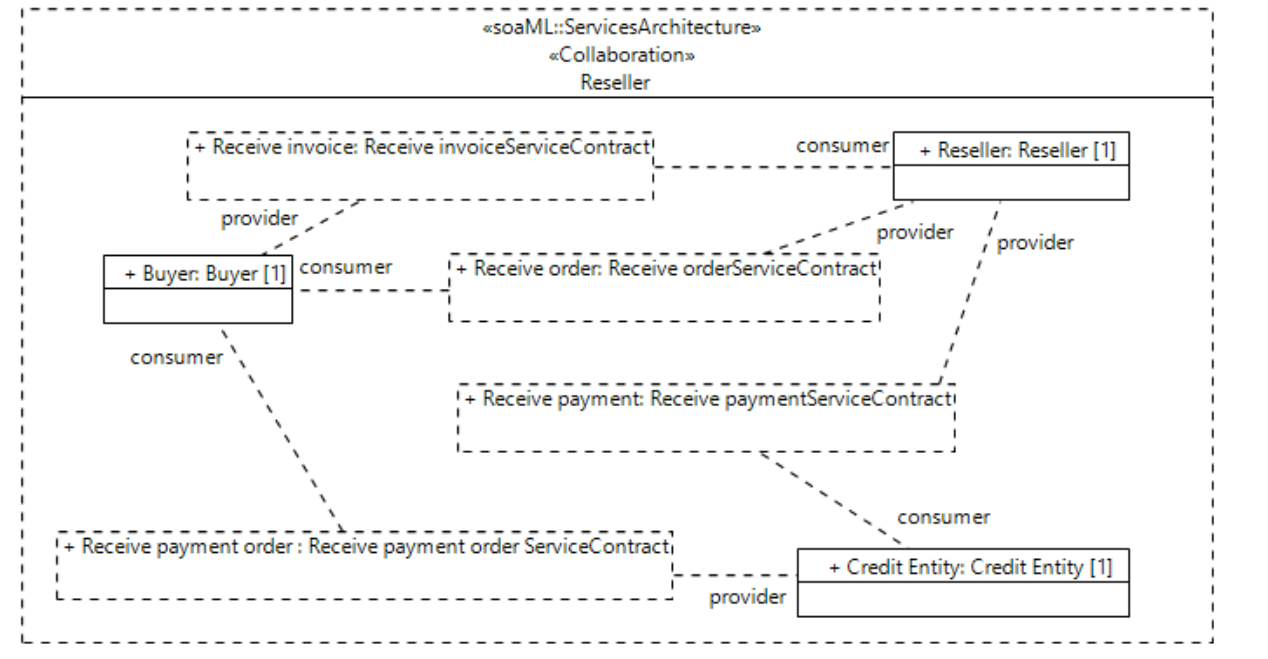


Figura 6.24: Diagrama de Arquitectura SoaML

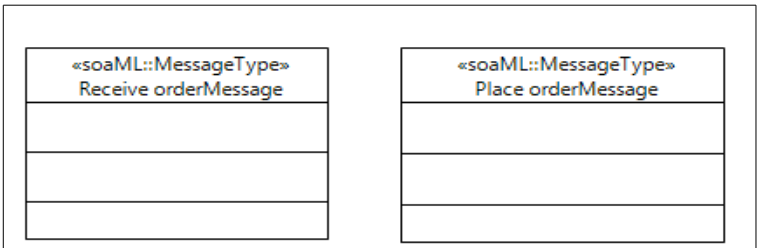


Figura 6.25: Diagrama de Mensajes SoaML

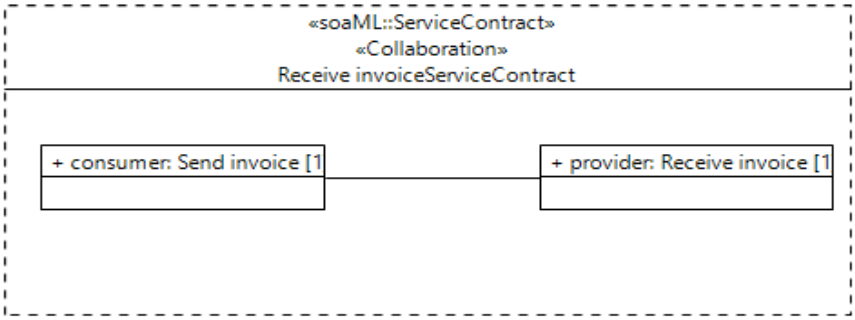


Figura 6.26: Diagrama de Contrato de Servicio para envío/recepción de facturas SoaML

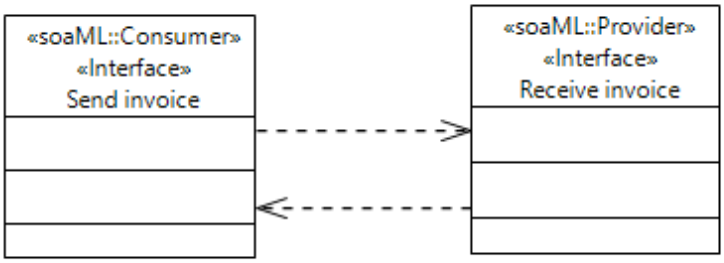


Figura 6.27: Diagrama de Interfaces SoaML para envío/recepción de facturas

En el modelo SoaML de salida se aprecian tres participantes (como en el bpmn de entrada)tal como se muestra en el diagrama de participantes de la figura 6.23: Reseller (revendedor), Buyer (comprador) y Credit Entity (Entidad crediticia). El vendedor expone los servicios de recibir las órdenes y los pagos, y consume un servicio que le permite enviar las facturas. El comprador define una interfaz de servicio para recibir la factura, y consume servicios para iniciar una orden y liquidar una factura. Por último, la entidad crediticia expone una interfaz para recibir los pagos, y requiere de servicios para dar la confirmación de los mismos.

En el diagrama de arquitectura de la figura 6.24 se ilustran claramente las interacciones de los participantes. Se pueden ver diversos contratos de servicio definido, entre ellos:

ReceiveInvoiceServiceContract provisto por el comprador para recibir las facturas y consumido por el revendedor que las emite; el contrato ReceiveOrderServiceContract provisto por el revendedor para recibir las órdenes del comprador y consumido por éste para tal propósito; ReceivePaymentServiceContract lo provee la entidad crediticia y lo consume el revendedor; y por último, en el contrato ReceivePaymentOrder es provista por la entidad de crédito y quien consume las operaciones es el comprador para realizar los pagos a crédito.

6.2 Caso de estudio de extensibilidad

Como se vio en el capítulo 3, uno de los requisitos no funcionales del plugin a desarrollar es lograr que sea extensible para cualquier motor qvt. En este capítulo, mostraremos paso por paso un ejemplo de extensibilidad.

Para el caso de extensibilidad, se utilizó el motor qvt relacional ModelMorf. Los fuentes del motor se pueden descargar en [16]. El archivo descargado es un .zip que contiene un ejecutable para realizar la instalación. Debemos descomprimirlo y ejecutar el archivo install.bat. Éste produce un directorio por defecto en C:\ModelMorf con todos los archivos fuente del motor y comprende parte de la siguiente estructura de archivos:

- qvt.jar. Es el empaquetado principal del motor ya que almacena todas las clases que constituyen la implementación del mismo.
- ExtJar. Es un directorio que contiene otros archivos .jar. Se encuentran allí todas las dependencias externas necesarias de qvt.jar para funcionar apropiadamente.
- Config. Es el directorio de configuración del motor. Allí se destacan, particularmente, el qvt-config.xml que contiene parámetros de la configuración inicial del motor y el config.xml que define rutas a archivos de salida que surgen como resultado de las transformaciones.
- Workspace. Directorio por defecto de trazas de las transformaciones (definidos en el config.xml).

Se debe inicialmente definir los parámetros de configuración inicial de ModelMorf. Llámesele <MM_path> al directorio de instalación de ModelMorf. Se debe abre el archivo <MM_path>/config/qvt-config y para agregar el path al JAVA_HOME en la entrada "JAVA_HOME". El path no debe contener espacios y debe apuntar a una JVM de 32 bits, ya que el qvt.jar no corre en 64 bits. Por otro lado, se debe abrir el archivo <MM_path>/config/config.xml y modificar las rutas en los atributos modelPath con rutas válidas, por ejemplo las que vienen por defecto son C:/ModelMorf/workspace/*.out reemplazarlas por <MM_path>/workspace/*.out. Como último paso, se debería obtener una licencia válida para utilizar el ModelMorf. El archivo de la licencia, con extensión .dat, debe descargarse y moverse a <MM_path>/config.

El ModelMorf se instala en el sistema operativo a partir de una estructura de directorios, no existe como plugin instalable en la plataforma Eclipse. Por lo tanto, a diferencia de MediniQVT que sí dispone de los plugins para Eclipse, las referencias del plugin al motor ModelMorf se definen de diferente forma.

Para empezar, debemos abrir el proyecto pgsoaqvt del plugin. En primer lugar, se debe agregar el ModelMorf como librerias externas al plugin, es decir, incluir los jars de ModelMorf. Para eso, hay que crear un directorio ModelMorf bajo la carpeta lib del proyecto en la cual copiar el archivo qvt.jar y la carpeta extJar de <MM_path>. La estructura de directorios debe quedar como se muestra en la figura 6.28.

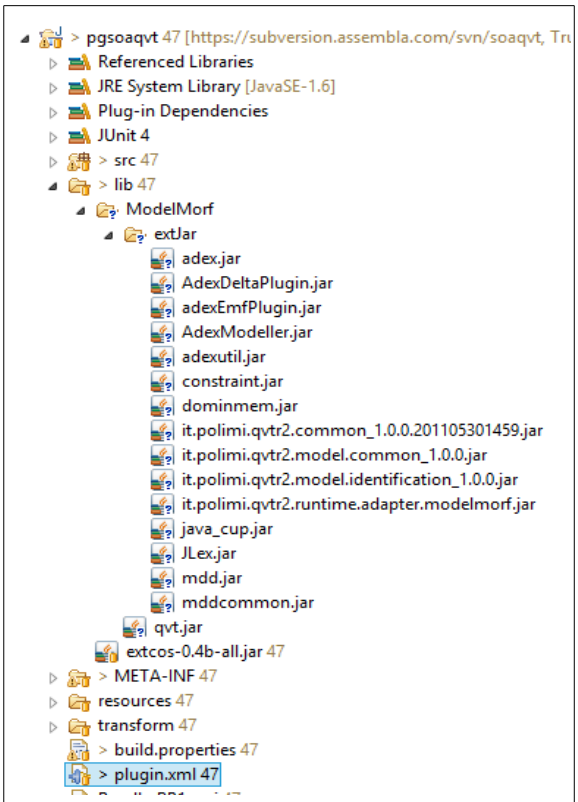


Figura 6.28: Librerías del ModelMorf

Luego hay que agregar las dependencias en el proyecto del plugin a los jars recien incluidos. Para eso, hay que abrir el archivo plugin.xml, ir a la pestaña "runtime" y presionar en "Add.." en la sección Classpath. Allí se deben setear las referencias a los jars bajo lib/ModelMorf como se muestra en la figura 6.29.

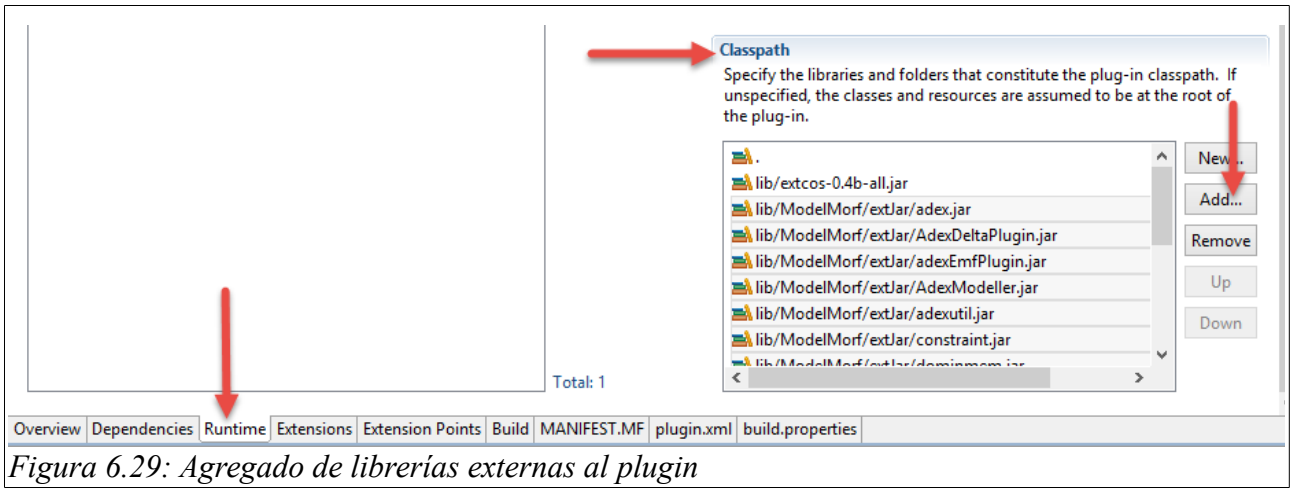


Figura 6.29: Agregado de librerías externas al plugin

La característica principal de ModelMorf es que las transformaciones se ejecutan desde línea de comando. Es así que la API que provee sigue el mismo procedimiento. Simplemente se invoca un método de una clase del qvt.jar que recibe como parámetros los argumentos que se pasarían si se fuera a correr el motor por consola. Al recibir los argumentos de la transformación por línea de comando (metamodelos, modelos de entrada y salida, script qvt, etc.), no maneja estado y por ende no cuenta con un mecanismo de persistencia propio, por lo cual vamos a necesitar implementar la

persistencia desde pgsoaqvt para ir manteniendo consistencia en el flujo de uso. Para ver más detalle de los modos de persistencia, revisar la sección 4.5. Esto es necesario tenerlo en cuenta al momento de realizar la implementación y a lo cual se detallará posteriormente. La sintaxis de comando de ModelMorf es la siguiente:

```
modelmorf  ( -m <alias_metamodelo> -mf <ruta_metamodelo> )+ -c <ruta_script_qvt>

( -u <variable_modelo>  -f <ruta_modelo> )+ -t <nombre_trasnformación> -d
<dirección_transformación> -q <modo_ejecucion> [ -tox <ruta_directorio_traza> ].
```

<modo_ejecucion> puede tomar los valores checkout o enforce. El primero solo valida que la transformación sea exitosa pero no construye el modelo de salida. La opción enforce sí lo hace, por lo tanto es la que se usará para la transformación. La sintaxis se dilucidará en la implementación del motor abstracto para ModelMorf.

El paso siguiente es proceder a implementar la extensión, siguiendo los lineamientos de la sección 4.5.4. En primer lugar, se deben hacer cambios en las dependencias del plugin correspondientes al motor. Hay que agregar los plugins y paquetes necesarios de los cuales se hace uso, tal como se explica la sección 4.5.6). En la figura 6.30 se presenta el archivo plugin.xml y los elementos agregados.

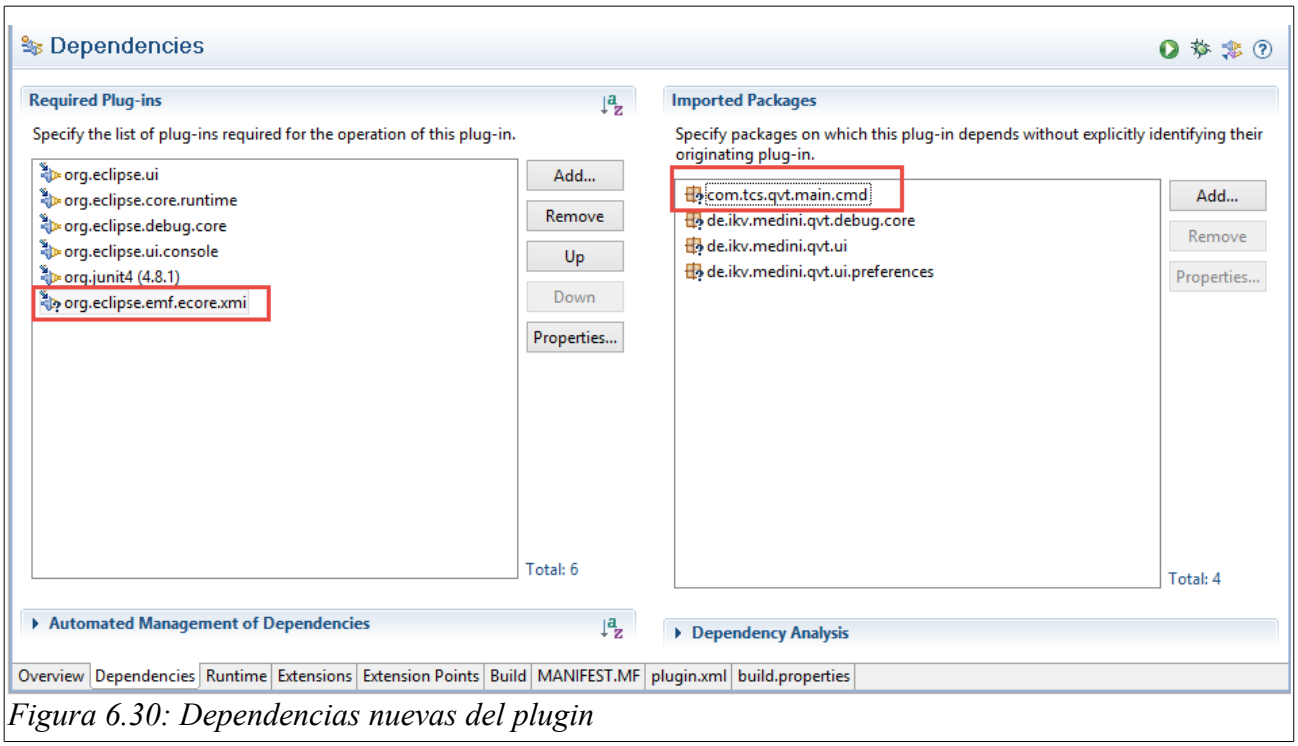


Figura 6.30: Dependencias nuevas del plugin

Es necesario agregar como dependencias:

- org.eclipse.emf.ecore.xmi.: Plugin añadido a la lista de plugins requeridos. Es el plugin de EMF de Eclipse para almacenar y operar sobre los contenidos de los modelos. Marcado como opcional.
- Com.tcs.qvt.main.cmd.: Package añadido a la lista de paquetes importados. Como después se verá, se utiliza para invocar la transformación en el ModelMorf desde la extensión. Marcado como opcional.

Actualizadas las referencias y las dependencias del proyecto, procedemos a crear los nuevos fuentes. Vamos al package "engines" y creamos una clase "ModelMorfQVTEngine" que implemente la interfaz de motor abstracto "QVTEngine". En la figuras 6.31 y 6.32 se muestran estos cambios.

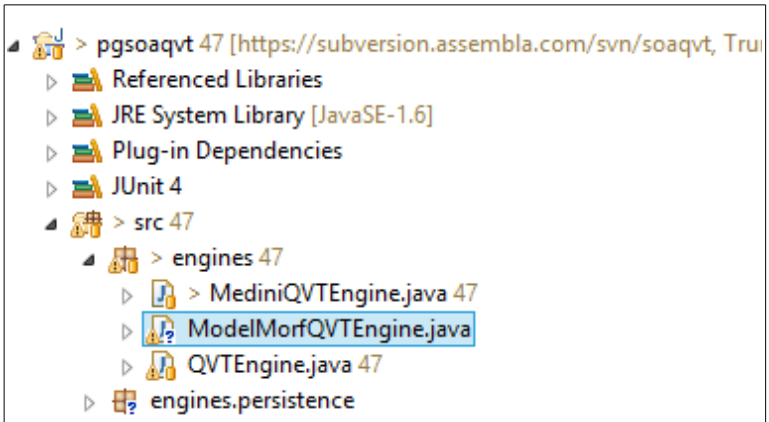


Figura 6.31

```
public class ModelMorfQVTEngine implements QVTEngine{

    private QVTPropertiesFilePersister persister=new QVTPropertiesFilePersister();
    private String workspaceRoot;
    private static final String ENGINE_NAME="Model Morf";
    private static final String QVT_HOME_DIR="/C:/ModelMorf";
```

Figura 6.32

En la clase ModelMorfQVTEngine se definieron algunas propiedades, tal como se muestra también en la figura 6.32:

- `persister`. Es una instancia de QVTPropertiesFilePersister para persistir datos de las transformaciones en un archivo de propiedades propietario de pgsoaqvt para el motor debido a que, como se dijo antes, ModelMorf no cuenta con persistencia propia.
- `engine_name`. Nombre del motor qvt.
- `qvt_home_dir`. Ruta absoluta por defecto al directorio de instalación de ModelMorf.
- `workspaceRoot`. Almacena en memoria la ruta absoluta al directorio del workspace de la instancia de Eclipse.

ModelMorfQVTEngine

Contiene la implementación del motor abstracto para ModelMorf. Para más información de la interfaz del motor abstracto, referirse a la sección 4.5.4. Se describirán someramente los métodos implementados:

- `registerMetamodels`. Se persisten las rutas de los metamodelos en el archivo de propiedades para ModelMorf. La sintaxis de almacenamiento se detalla en la sección 4.5.5.
- `listRegisteredMetamodels`. Se obtienen las rutas relativas de los metamodelos del archivo de propiedades de ModelMorf.
- `isMetamodelsRegistered`. Evalúa que la condición de que la cantidad de metamodelos guardados en el archivo de propiedades sea positiva.
- `saveLaunchConfiguration`. Se persisten los parámetros de la configuración (script qvt, rutas a modelos de entrada y salida, ruta a directorio de traza, nombre de la transformación y dirección) en el archivo de propiedades de ModelMorf. La sintaxis de almacenamiento se detalla en la sección 4.5.5.
- `IsAvailableConfigurations`. Evalúa la condición de que la cantidad de configuraciones guardadas en el archivo de propiedades sea positiva.
- `LoadLaunchConfigurations`. Obtiene todos los datos de las configuraciones del archivo de propiedades y los datos particulares de esa configuración.
- `getEngineName`. Retorna el nombre del motor o el valor de la variable estática ENGINE_NAME.
- `RemoveLaunchConfiguration`. Elimina la configuración con el nombre recibido como parámetro del archivo de propiedades. Para eso, obtiene todas las configuraciones y elimina aquellas que contengan el nombre en la clave. Se persisten todas las configuraciones resultantes de dicho filtro.
- `TestEngine`. Verifica que el motor este disponible para ser utilizado, instanciando la clase com.tcs.qvt.main y la existencia del directorio de instalación del motor. La ruta al directorio se pasa por argumento al levantar Eclipse. Se comprueba que exista y se extrae de los argumentos, a fin de chequear su existencia. En caso contrario (el directorio pasado es inexistente o no se pasa por argumento), se utiliza el directorio por defecto QVT_HOME_DIR. En caso de la no existencia de dicho directorio, QVT_HOME_DIR o argumento, se devuelve una excepción.
- `ExecuteTransformation`. Implementa la funcionalidad de invocar la transformación qvt en el motor ModelMorf. Esto lo realiza en los siguientes pasos:

  1. Se obtienen los parámetros de la configuración con el nombre pasado por parámetro del archivo de propiedades.
  2. Se define una línea de comando inicial vacía. Se le concatena (-p <MM_path>) para especificar el directorio donde reside el motor.
  3. Se obtienen los bindings de la transformación seleccionada del script qvt de la configuración con los metamodelos. Para eso, se obtienen las nombres tipo de la firma de la transformación y luego se busca en todos los metamodelos registrados el correspondiente a ese nombre, leyendo y evaluando la igualdad con el valor del atributo "name" del primer elemento del xml del metamodelo. A modo de ejemplo la firma de la figura 6.33.

```
transformation UmlToRdbms(uml:umlMM; rdbms:rdbmsMM)
```

Figura 6.33

Los nombres tipo son umlMM y rdbmsMM. Si estuviera el metamodelo registrado umlMM.xml con el primer elemento de la forma:

```
<emof:Package xmi:version="2.0"  xmlns:xmi="http://www.omg.org/XMI" ...
              name="umlMM" uri="umlMM">
      ....
</emof:Package>
```

Entonces se asocia el alias "umlMM" al metamodelo umlMM.xml. Este paso se utiliza para construir las parejas (-m <metamodel_alias> -mf <metamodel_file>)+ de la línea de comando a ejecutarse.

4. Se obtienen los bindings de la transformación seleccionada del script qvt de la configuración con los modelos. Para eso, se obtienen las variables de la transformación de la firma de la transformación y luego se asocian con el modelo de entrada y de salida según la dirección de la transformación de la configuración.

   Partiendo del ejemplo de la firma de la figura 7.6, si tenemos un modelo de entrada y uno de salida de la configuración, ejemplos uml.xml y rdbms.xml respectivamente, y la dirección hacia "rdbms", entonces asociamos los aliases "uml" al modelo de entrada uml.xml y "rdbms" al modelo de salida rdbms.xml.

   Este paso se utiliza para construir las parejas (-u <model_alias> -f <model_file>)+ de la línea de comando a ejecutarse.

5. Se agrega el resto de los argumentos (-c <ruta_script_qvt>) [-tox <ruta_directorio_traza>] (-t <transformación>) (-d <dirección>).
6. Se pasa la línea de comando al método main de la clase `com.tcs.qvt.main.cmd.Main` que se agregó en la sección de paquetes importados en las dependencias del plugin. La siguiente línea finalmente lleva a cabo la transformación:

```
com.tcs.qvt.main.cmd.Main.main ( arguments );
```

Todas las rutas de los pasos 1 al 6 deben ser absolutas. Las que son relativas al workspace se pueden convertir en absolutas haciendo con un append del valor de la propiedad "workspaceRoot" que se definió en ModelMorfQVTEngine.

Culmina de esta manera la implementación del motor abstracto para ModelMorf. Para reemplazar el puntero al directorio por defecto de ModelMorf c:\ModelMorf en el plugin, se debe ejecutar eclipse con los siguientes argumentos:

```
>eclipse.exe [otros_argumentos..] -qvtEngine ModelMorf -path <MM_path>
```

Ejemplo de extensibilidad.

Se mostrará un ejemplo de uso de ModelMorf en el plugin donde se desea convertir un modelo uml a un modelo rdbms equivalente. El modelo uml sobre el cual se realizará la transformación se muestra en la figura 6.34.

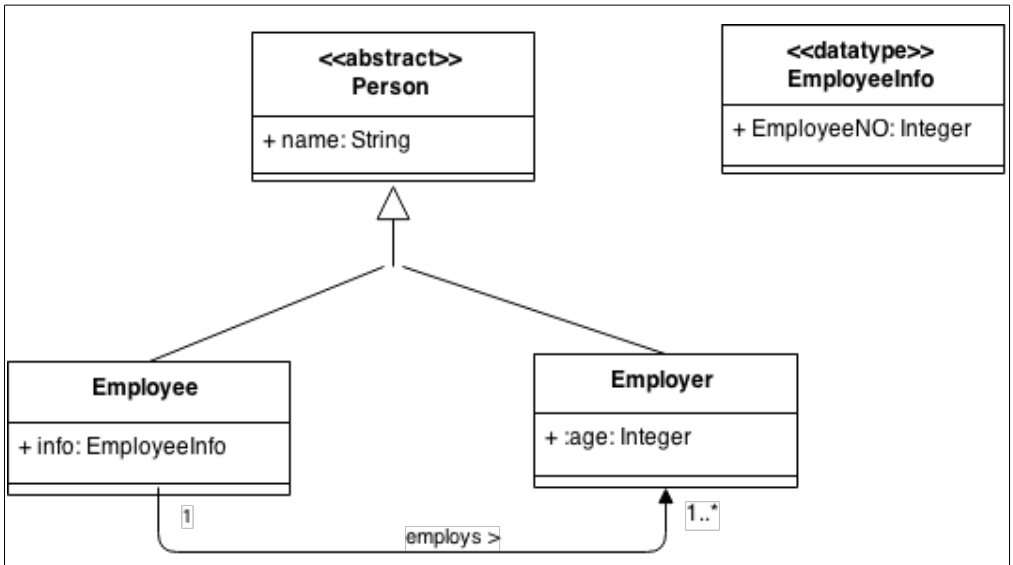


Figura 6.34: Modelo UML de entrada

En dicho modelo, se ven una clase Employee (empleado) que tiene número de empleado y una clase Employer (empleador) con edad. El empleador emplea a un empleado con la asociación Employs. Ambos heredan de una clase Person (persona) abstracta que guarda el nombre del empleador y

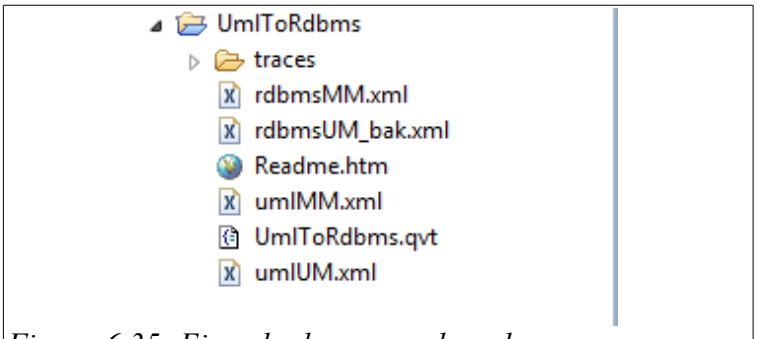


Figura 6.35: Ejemplo de uso en el workspace

empleado.

Para efectuar la transformación, el primer paso es abrir una instancia el entorno Eclipse con el plugin pgsoaqvt instalado e importar del workspace el ejemplo que se muestra en la figura 6.35.

Contiene el modelo de entrada en umlUM.xml. Luego yendo al menú pg SOAQVT se selecciona la opción Select Engine. Allí seleccionar ModelMorf tal como se muestra en la figura 6.36.

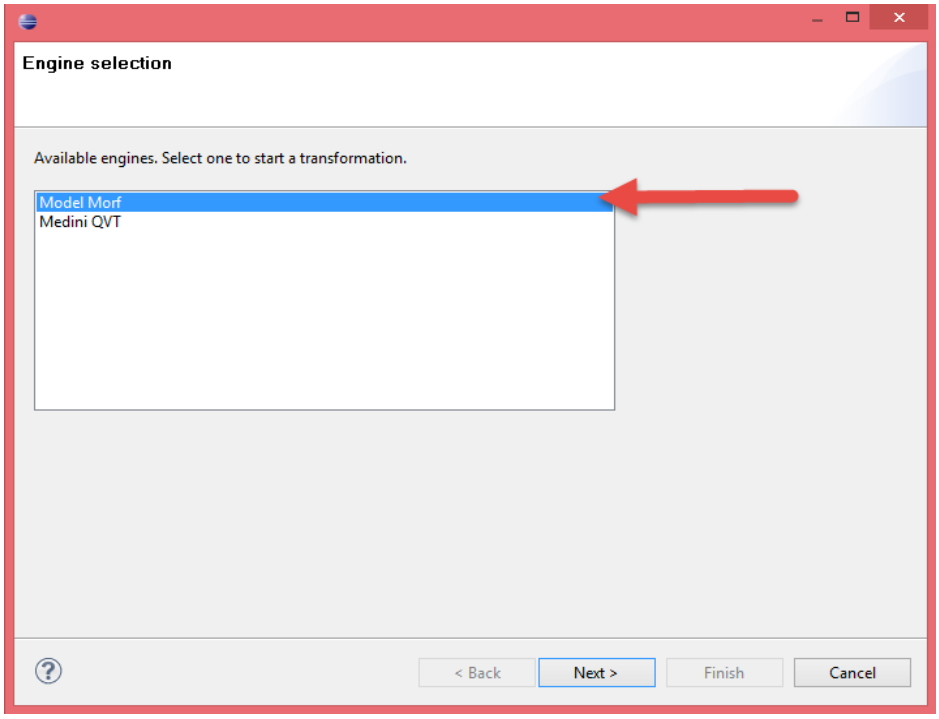


Figura 6.36: Selección del motor ModelMorf

Se pasa a la ventana de registrar metamodelos. Allí se agregan los metamodelos umlMM.xml y rdbmsMM.xml correspondientes a los modelos de entrada y salida, como se muestra en la figura 6.37.

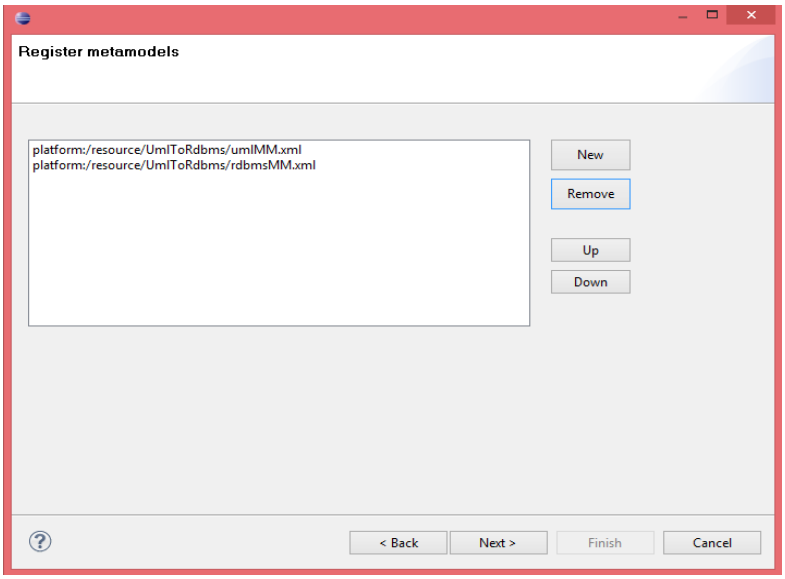


Figura 6.37: Metamodelos al motor

Paso siguiente, representado en la figura 6.38, se crea una nueva configuración con:

- Script qvt: /UmlToRdbms/UmlToRdbms.qvt
- Transformacion: umlToRdbms
- Dirección: rdbms
- Directorio de traza: /UmlToRdbms/traces
- Modelo entrada: umlUM.xml
- Modelo salida: salida.xml

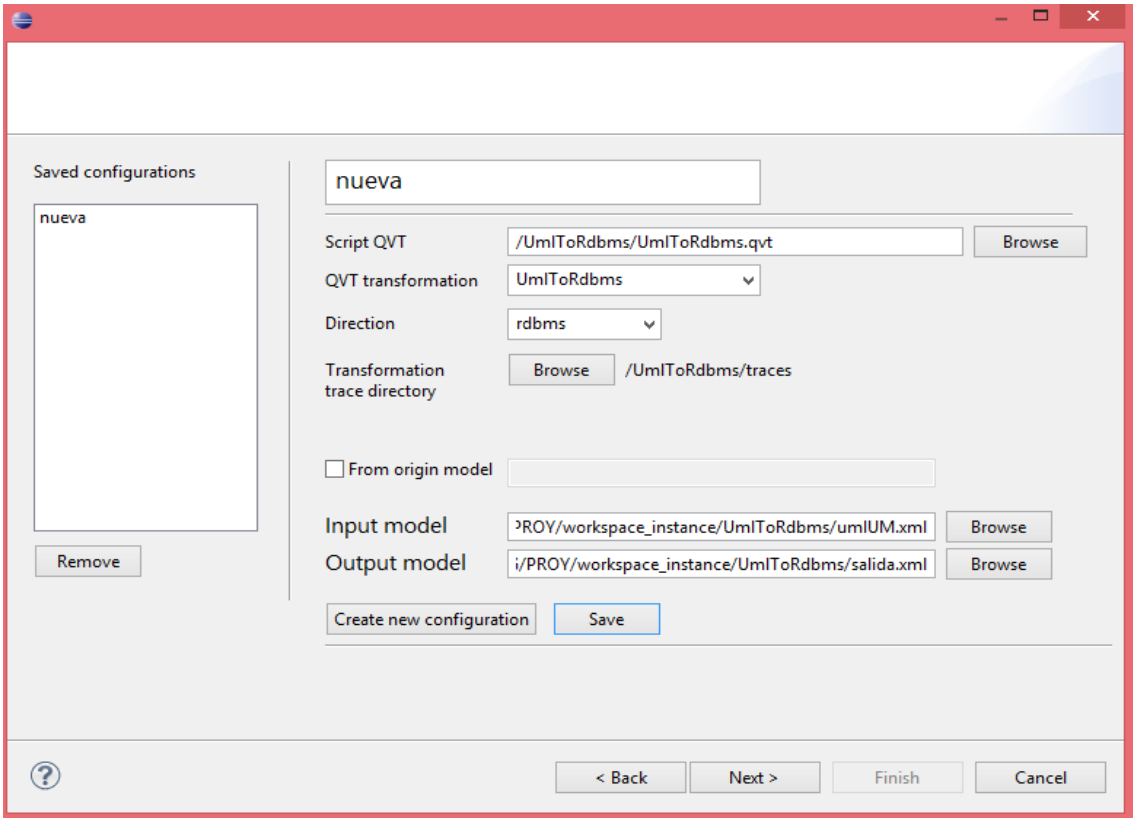


Figura 6.38: Parámetros de la nueva configuración

Finalmente, se elige la configuración creada a ejecutar. La figura 6.39 muestra la ventana de selección.

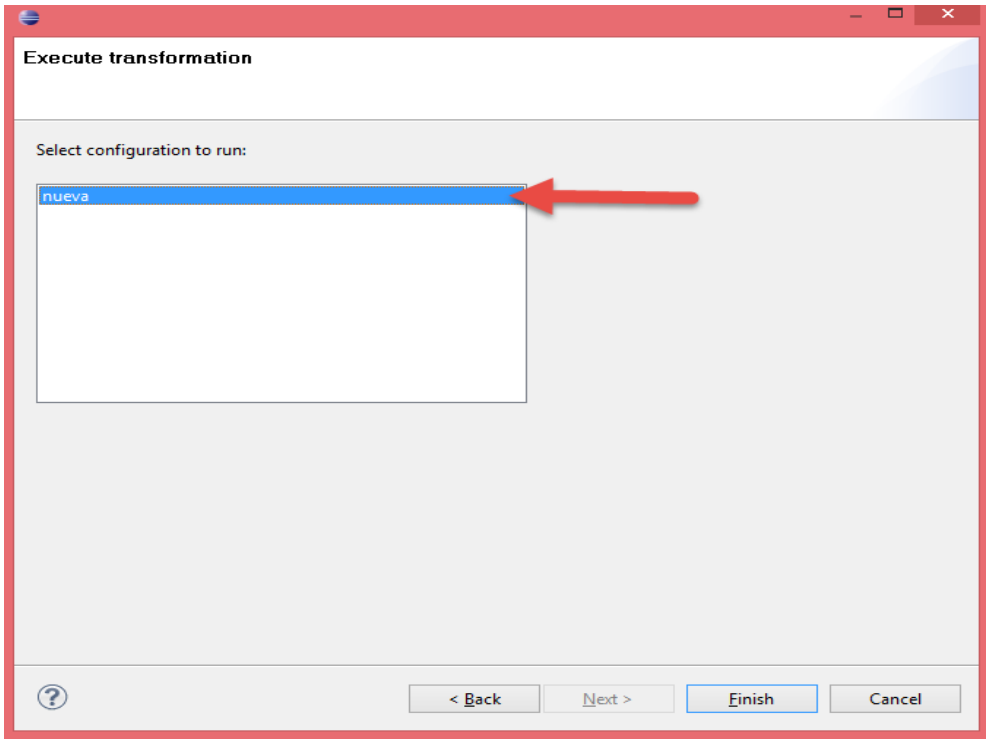


Figura 6.39

Al correr la transformación, puede solicitar agregar la licencia de ModelMorf, la cual se encuentra en QVT_HOME_DIR/config. En la figura 6.40 se muestra el archivo de salida generado en el workspace.

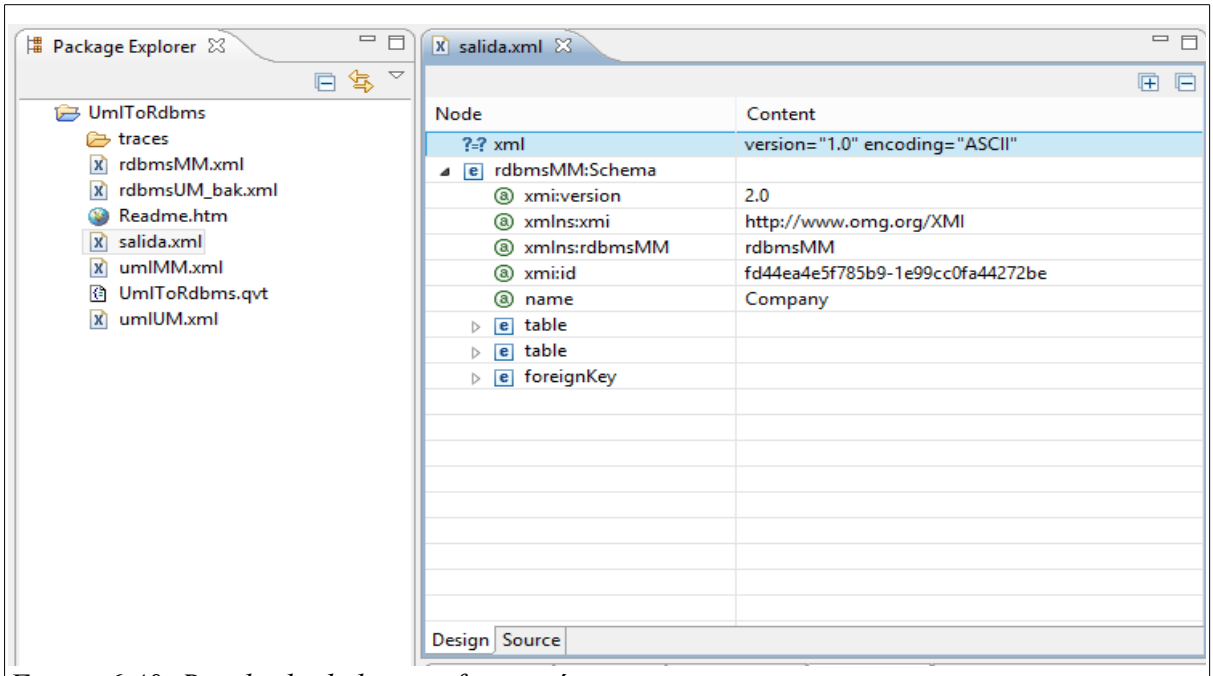


Figura 6.40: Resultado de la transformación

En el modelo de la figura 6.41 se representa el resultado de la transformación.

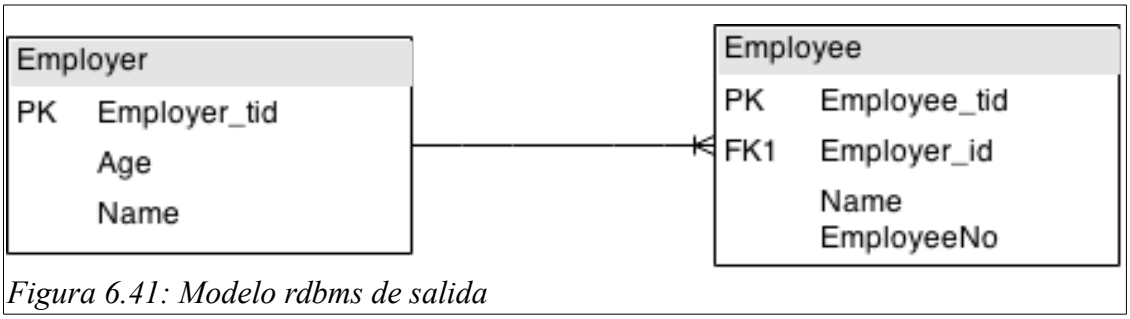


Figura 6.41: Modelo rdbms de salida

Contienen dos tablas: Employer (empleadores) y Employee (empleados). El empleado tiene una clave foránea "Employer_id" a la tabla Empleador y como un empleador puede emplear a muchos empleados, la relación es 1:N.

7. Conclusiones, trabajo futuro

Las nociones de MDD/E aun no han sido adoptadas plenamente por la industria. El uso de modelos como documentación o como asistencia en el análisis y diseño es generalizado, pero en forma parcial o informal. El objetivo de pasar de un representación del sistema al sistema mismo (que se empezó a perseguir con las herramientas CASE de los años 80´s) no ha sido alcanzado, excepto por algunos sectores de la industria con dominios muy acotados y/o críticos.

Algunos de los factores que contribuyen a esta situación son la falta de madurez en los estándares, y en los casos de implementación, el apego relativo o incompleto a estos (por ejemplo, los diferentes "dialectos" QVTr para los motores de transformación). Además, como se menciono, hay factores organizacionales y psicológicos que tienen que ser estudiados y encarados en conjunto con otras disciplinas y ramas de las ciencias cognitivas.

En cuanto al plugin desarrollado, se logró abstraer al usuario de las particularidades de configuración y ejecución de los motores de transformación QVTr, permitiendo además elegir el motor a utilizar para las ejecuciones. El intercambio de motores para una configuración, debido a la incompatibilidad de los motores QVT en cuanto a la sintaxis de los *scripts*, no es aplicable al momento, por lo que se optó por vincular las configuraciones a su motor de ejecución. Pero cuando los motores logren una mayor adhesión al estándar y puedan resolver estas diferencias el plugin podrá ser modificado para ejecutar las transformaciones del mismo lenguaje libremente. También se logró una interfaz concisa e intuitiva.

El plugin es fácilmente extensible para su uso con otros motores de transformación, como también para transformaciones escritas en otros lenguajes, tanto QVT como no QVT.

Entre las mejoras posibles al plugin se encuentran la instalación automática de motores invocando programáticamente a la API de instalación de software de Eclipse. Aún de este manera, el usuario necesitaría saber cuales son los plugins que corresponden a motores de transformación ya que no existe una denominación estándar para estos, y que pueda ser consumida programáticamente. Esta limitación aplica también a la detección de motores ya instalados en la plataforma. Por otro lado, seguiría siendo necesaria la implementación explícita de la interfaz "QVTEngine" para el nuevo plugin encontrado, ya que las API de los motores no son estándar, ni se requiere que estén hechas para acoplarse a un *extension point* del tipo Eclipse que pueda proveer el plugin desarrollado.

Otra mejora posible es brindar la capacidad de ejecutar una transformación (configuración) en tiempo real, es decir, durante la edición o al guardar los cambios en el modelo de origen. Para esto, se deberán tener las siguientes consideraciones:

- la configuración a ejecutar debería ser seleccionada explícitamente por el usuario, ya que podrían existir mas de una configuración con el mismo modelo de origen, y con distintos destinos, scripts de transformación y/o motores. Ejecutar automáticamente todas las transformaciones posibles podría sobrecargar al sistema. Para mas flexibilidad podría permitirse al usuario elegir mas de una transformación, balanceando entre cobertura y performance.
- Si se están utilizando plugins de visualización de diagramas, el plugin de destino tendría que tener la capacidad de responder al evento de cambio en el archivo XMI que se genera en la transformación para que los cambios sean visualizados automáticamente sin necesidad de recargar el archivo. Si ademas el plugin destino requiere transformar XMI a otro formato para poder visualizar el diagrama, se deberá automatizar esa transformación también.

Otra funcionalidad que podría agregarse al plugin desarrollado es la de aplicar una transformación XSLT al modelo XMI de salida. De esta forma se podría lograr una transformación completa entre dos formatos pasando por XMI en una sola operación.

Referencias y bibliografía

[1] http://www.omg.org/mda/ [Consultado: Marzo 2015]

[2] GUIDE, M. D. A. version 1.0, version 2.0, document OMG: http://www.*OMG.org/mda*.

[3] MELLOR, Stephen J.; CLARK, Tony; FUTAGAMI, Takao. Model-driven development: guest editors' introduction. *IEEE software*, 2003, vol. 20, no 5, p. 14-18.

[4] BÉZIVIN, Jean. On the unification power of models. *Software & Systems Modeling*, 2005, vol. 4, no 2, p. 171-188.

[5] SCHMIDT, Douglas C. Guest editor's introduction: Model-driven engineering. *Computer*, 2006, vol. 39, no 2, p. 0025-31.

[6] J. Hutchinson, M. Rouncefield, and J. Whittle, "The State of Practice in Model-Driven Engineering", IEEE Software May/June 2014, pp. 79-85.

[7] http://www.omg.org/spec/MOF/ [Consultado: Marzo 2015]

[8] http://www.omg.org/spec/QVT/ [Consultado: Marzo 2015]

[9] http://www.omg.org/spec/XMI/ [Consultado: Marzo 2015]

[10] http://www.omg.org/spec/BPMN/2.0/PDF/ [Consultado: Marzo 2015]

[11] http://www.omg.org/spec/UML/2.5/Beta2/ [Consultado: Marzo 2015]

[12] http://www.uml-diagrams.org/profile-diagrams.html [Consultado: Marzo 2015]

[13] http://www.omg.org/spec/SoaML/1.0.1/PDF/ [Consultado: Marzo 2015]

[14] http://sisas.modelbased.net/practice.business.service_modelling.base-sintef/guidances/guidelines/business_soaml_guidance_59D58B7.html [Consultado: Marzo 2015]

[15] http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf [Consultado: Marzo 2015]

[16] https://code.google.com/p/qvtr2/wiki/UsingModelMorf [Consultado: Marzo 2015]

[17] http://alarcos.esi.uclm.es/MINERVA/TOOLS/soamlPlugin.htm [Consultado: Agosto 2014]

[18] http://projects.ikv.de/qvt/wiki [Consultado: Marzo 2015]

[19] http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm[Consultado: Octubre 2014]

[20] https://code.google.com/p/qvtr2/ [Consultado: Marzo 2015]

[21] https://projects.eclipse.org/projects/modeling.mmt.qvtd [Consultado: Marzo 2015]

[22] https://github.com/haslab/echo [Consultado: Marzo 2015]

[23] http://www.uml-diagrams.org/uml-25-diagrams.html [Consultado: Abril 2015]

[24] https://www.visual-paradigm.com/tutorials/databasedesign.jsp [Consultado: Abril 2015]

[25] MELLOR, Stephen J. (ed.). *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.

[26] https://eclipse.org/pde/ [Consultado: Abril 2015]

[27] J.Greenfield et al., *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*, John Wiley & Sons, 2004.

[28] Generic Modeling Environment: http://www.isis.vanderbilt.edu/projects/gme/ [Consultado: Abril 2015]

[29] Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf [Consultado: Abril 2015]

[30] Model-based systems engineering: http://mbse.gfse.de/ [Consultado: Abril 2015]

[31] MOD Reseach Group: http://modelbased.net/ [Consultado: Abril 2015]

[32] Ecore Tools: http://eclipse.org/ecoretools/overview.html [Consultado: Abril 2015]

Apéndices

A. Estándares de modelado y entorno Eclipse

1. BPMN

La especificación y semántica de los elementos de modelado de BPMN (artefactos, eventos, asociaciones, etc.) se definen en el metamodelo de BPMN. El metamodelo de BPMN es un diagrama de clases UML que contiene las definiciones y atributos de los elementos. Teniendo en cuenta otorga la capacidad del metamodelado, BPMN puede ser extendido o ampliado a nuevas versiones mediante cambios en su metamodelo siguiendo cumpliendo reglas y restricciones de la especificación. Inclusive, puede ser interrelacionado con herramientas de modelado con otros elementos de otro lenguaje de modelado OMG y realizar transformaciones a partir de los respectivos metamodelos. Realizan el mapeo de objetos semánticamente equivalentes de un metamodelo de entrada al metamodelo de salida.

La figura A.1 muestra el metamodelo de BPMN con los principales elementos de modelado del lenguaje: proceso, coreografía, colaboración, subproceso, actividad , grupos, actividad, asociación, gateway.

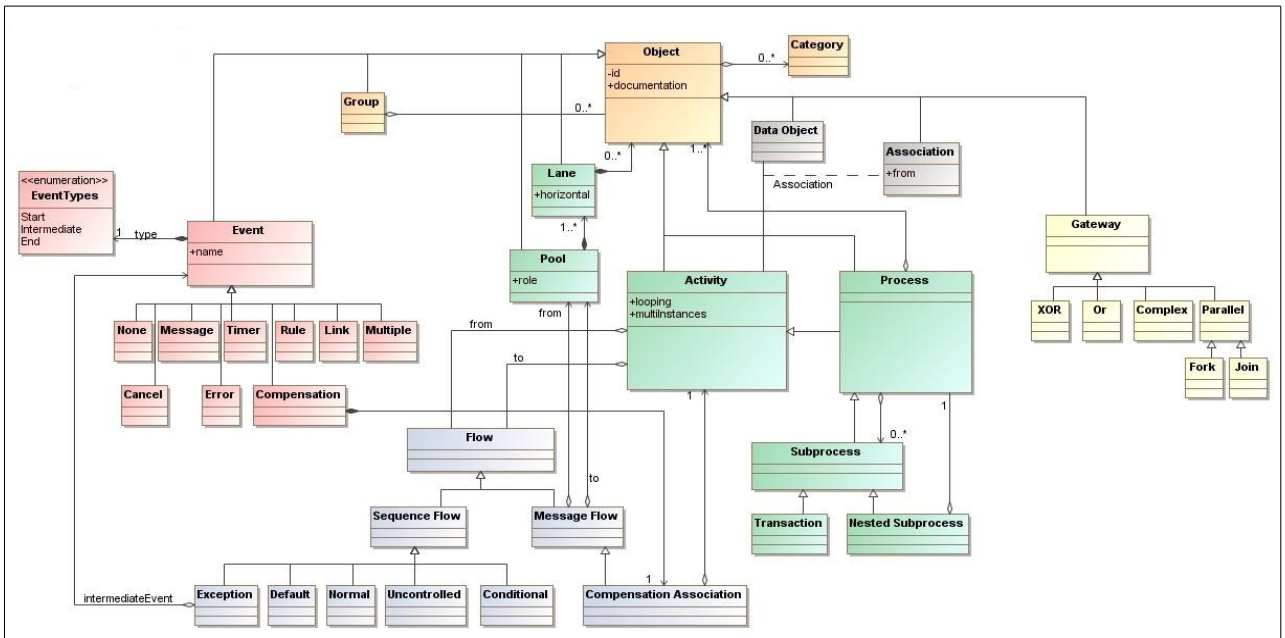


Figura A.1: Metamodelo de BPMN

Los elementos gráficos de BPMN se agrupan en cuatro categorías principales:

Objetos de flujo

- Eventos. Un evento está representado por un círculo y refiere a algo que "ocurre" durante el proceso de negocio. Afectan un flujo del proceso que en general tiene un disparador (trigger) o un resultado (result). Hay tres tipos de eventos: Start, Intermediate y End. Los Start events

deben aparecer al principio del proceso, los Intermediate Events durante el proceso y los End events al final de un proceso. Los eventos pueden tener o no asociados disparadores de diverso tipo: tiempo (timer-event), basados en la recepción de un mensaje (message-based-event), generados por un error (error-event), condicionales (conditional- event), basados en señales (signal-events) o en excepciones (exception-event), etc..

- Actividades. Una actividad está representada por un rectángulo con puntas redondeadas. Representa un trabajo o actividad que es llevada a cabo durante el proceso. Existen 2 tipos de actividades: Task activities y Sub-process activities. Una task es utilizada cuando la actividad no puede ser dividida a mayor nivel de detalle, contrariamente a las sub-process que son actividades compuestas.
- Gateways. Un gateway es representado con una forma de rombo. Es utilizado para controlar la convergencia/divergencia de un flujo de secuencia. Determina decisiones dentro de un proceso, generando paralelismo, unión o entrelazado de flujos.

Objectos de asociación

- Flujo de secuencia. Representado con una flecha de estilo sólido, es utilizado para mostrar el orden en que un conjunto de actividades será ejecutado.
- Flujo de mensaje. Representado con una flecha con guiones, es utilizado para mostrar el flujo de mensajes entre dos proceso participantes. Dos pools separados en el diagrama representan dos participantes.
- Asociación. Representada por una flecha de estilo punteado, es utilizada para asociar datos, texto y otros artefactos en un flujo. Se usa para mostrar las entradas y salidas de las actividades.

Swimlanes

Es un mecanismo para organizar actividades en componentes visuales separados. Hay de 2 tipos:

- Pool. Representa un participante correspondiente a un proceso. Es un contenedor que agrupa actividades relacionadas y las separa de otro Pool.
- Lane. Es una subpartición dentro de un Pool que se extiende a lo largo del mismo, vertical u horizontalmente. Usados para organizar y categorizar actividades.

Artefactos

- Objetos de datos. Mecanismo para mostrar cómo ciertos datos son requeridos por actividades. Conectados por asociaciones a las actividades.
- Grupos. Representado por un rectángulo de línea punteada, se utiliza con propósitos para análisis y/o documentación. No afecta el flujo de secuencia.
- Anotación. Mecanismo para el diseñador para mostrar información en texto adicional que aporte al lector del diagrama BPMN.

Ejemplos:

1) Parte de un proceso que aprueba solicitudes que llegan por mail a un servidor web para acceder a ciertos datos en el módulo de administración (figura A.2).

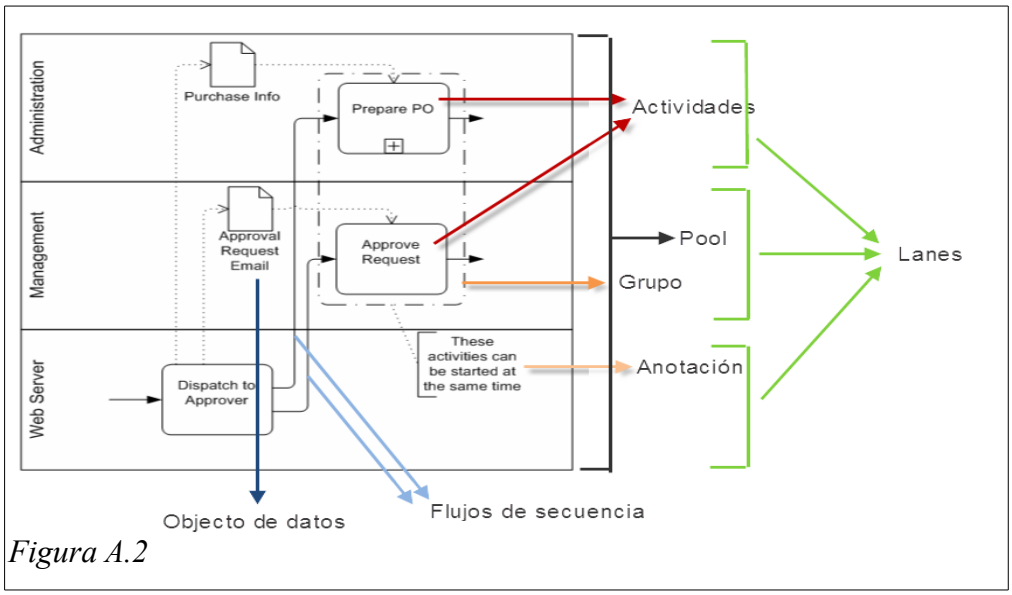


Figura A.2

2) Proceso que refleja las interacciones en la atención médico-paciente (figura A.3).

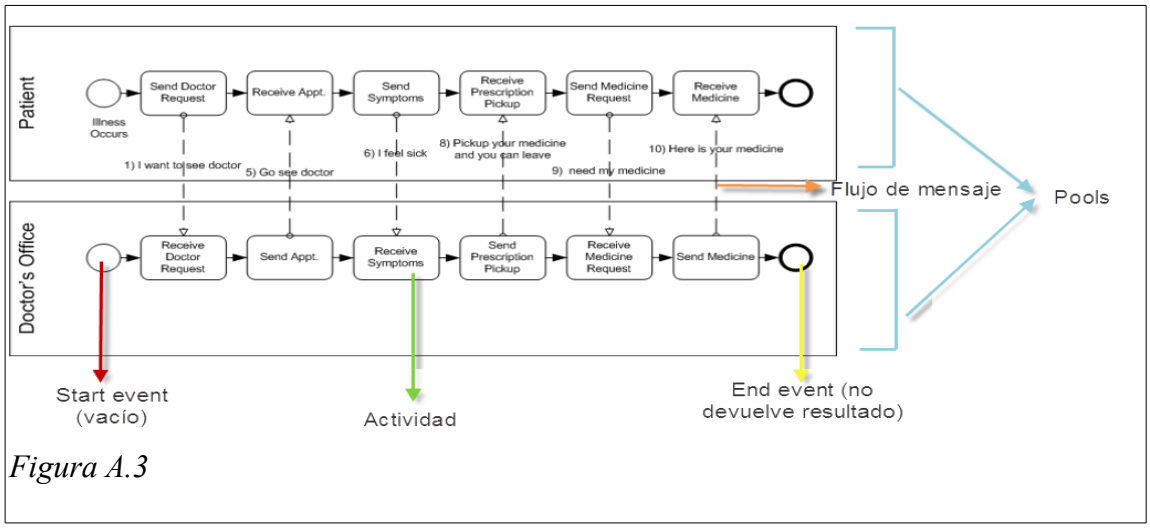
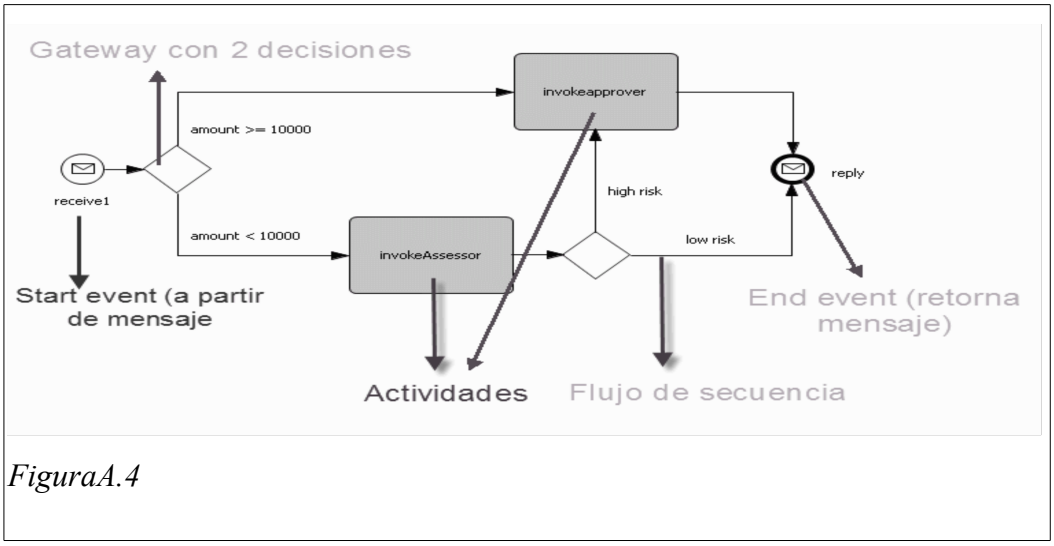


Figura A.3

3)Gateway con 2 decisiones (figura A.4)



FiguraA.4

2. Perfiles UML

Utilidad

Existen varias razones por las cuales un diseñador quiera poder extender y adaptar un meta-modelo existente, ya sea de UML o desde otro perfil:

- Disponer de una metodología y vocabulario propio de un dominio de aplicación o de una plataforma de implementación concreta (por ej., manejar dentro del modelo del sistema terminología propia de EJB).
- Definir una sintaxis para construcciones que no cuentan con una notación propia.
- Definir una nueva notación para símbolos ya existentes, más acorde con el dominio de la aplicación objetivo.
- Añadir cierta semántica que no aparece determinada de forma precisa o que no existe en el meta-modelo
- Añadir restricciones al meta-modelo
- Añadir información que puede ser útil a la hora de transformar el modelo a otros modelos, o directamente a código.

Se puede utilizar el lenguaje OCL (Object Constrained Language) de UML para definir la extensión del perfil. Un nuevo perfil se define básicamente en un paquete UML estereotipado <<Profile>> que extiende al meta-modelo de UML o a otro perfil.

3. SoaML

El metamodelo de SoaML extiende el metamodelo de UML2 de modo de soportar modelado de servicios. Esta extensión busca abarcar diversos escenarios como la definición y descripción de un servicio, SOA y contratos de servicio. El metamodelo extiende UML2 en cinco elementos:

Participantes(Participants), Servicios(Services), Interfaces(Interfaces), Contratos de Servicios (Service Contracts) y datos de servicio (Service data). En la figura A.5 se ilustran los participantes e interfaces de servicio en el metamodelo de SoaML y las relaciones entre ellos.

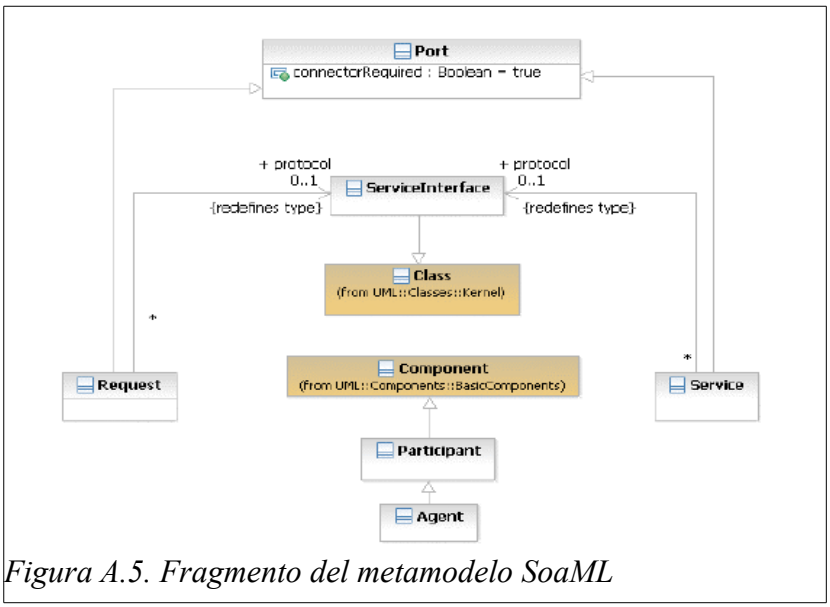


Figura A.5. Fragmento del metamodelo SoaML

SoaML en el contexto de SOA y MDA

La arquitectura orientada por servicios (SOA) permite describir organizaciones, personas y sistemas de forma de maximizar la escalabilidad e interoperabilidad. En el enfoque SOA tenemos actores que proveen servicios y otros que los consumen. Los servicios contienen cierta funcionalidad necesaria para culminar una tarea particular, mientras que los actores consumidores sólo conocen cuál es la funcionalidad expuesta ignorando de qué forma está implementada. Ellos ingresan entradas al servicio y éste les retorna las salidas resultantes que cumplen con sus expectativas de forma eficiente, a modo de caja negra. El uso de los servicios puede darse a cambio de ofrecer cierto valor, estableciendo una relación de intercambio con la comunidad (qué podría ser el público en general).

SOA, por lo tanto, es un paradigma arquitectónico para definir cómo las personas, organizaciones y sistemas proveen y utilizan servicios, alcanzando determinados resultados esperados. SoaML en este marco provee un mecanismo estándar de modelar y brindar una arquitectura a soluciones SOA utilizando UML.

SoaML no está ligado estrictamente a ningún tipo de tecnología particular, si bien se basa implícitamente en el uso de las tecnologías existentes para poder alcanzar determinada solución. SoaML puede ser utilizado dentro del paradigma MDA (Model-Driven Architecture) para mapear la arquitectura de los sistemas de negocio a las tecnologías necesarias que permitan la automatización de los procesos de negocio, manteniendo independientes entre sí el modelado de los procesos y la implementación en tecnologías particulares.

Algunos conceptos y notación básica

Los servicios son provistos por proveedores y consumidos por clientes, llamados participantes. Los

participantes ofrecen sus servicios a través de puertos utilizando el estereotipo <<Service>> y los pedidos a los puertos del servicio se anotan con el estereotipo <<Request>>. Un puerto de servicio tiene un tipo que describe cómo se utiliza; puede estar representado por una interfaz UML (para los servicios más simples) o mediante una ServiceInterface, una interfaz de servicio definida a partir de la notación de SoaML. En ambos casos el tipo del puerto especifica lo necesario para realizar la interacción entre proveedor-cliente del servicio.

Elementos básicos

Las entidades se denotan con el estereotipo <<Participant>>. Un puerto de un servicio provisto donde se expone una interfaz del servicio se denota con <<Service>> y el puerto del servicio del lado del consumidor se denota con <<Consumer>>. Un ejemplo básico de notación SoaML es la figura A.6.

Figura A.6

La figura muestra un participante Shipper que expone la interfaz "Shippment Status" en un puerto Service y otro participante Dealer que consume la interfaz "Shippment Status" en un puerto Request con el nombre Shippment Status.

Interfaz de servicio (ServiceInterface)

Como una interfaz UML simple, una ServiceInterface es una interfaz en SoaML que especifica un servicio y tipifica puertos. Una ServiceInterface, a diferencia de lo que puede ser la interfaz nativa de UML, puede especificar un servicio bidireccional basándose en un protocolo adecuado, donde tanto los proveedores como consumidores adquieren responsabilidades en cuanto a invocación o respuesta de operaciones, envío y recepción de mensajes o manejo de eventos. Algunos datos a tener en cuenta por una ServiceInterface son:

- Una interfaz simple UML de servicio puede ser realizada o utilizada por una ServiceInterface.
- Una ServiceInterface define roles en los que jugarán los participantes ligados al propio servicio.
- Una ServiceInterface define cómo son las interacciones válidas entre proveedor/consumidor. Especifica un protocolo de comunicación básico entre ambas entidades.
- Los puertos de una ServiceInterface especifican los requerimientos de los participantes en la provisión de un servicio y son entablados desde el punto de vista del proveedor.

La Figura A.7 muestra dos interfaces UML, OrderPlacer y OrderTaker. Mediante la

ServiceInterface “Place Order Service” que realiza las otras dos, se define a Order Placer con el rol consumer y a provider con el rol de proveedor.

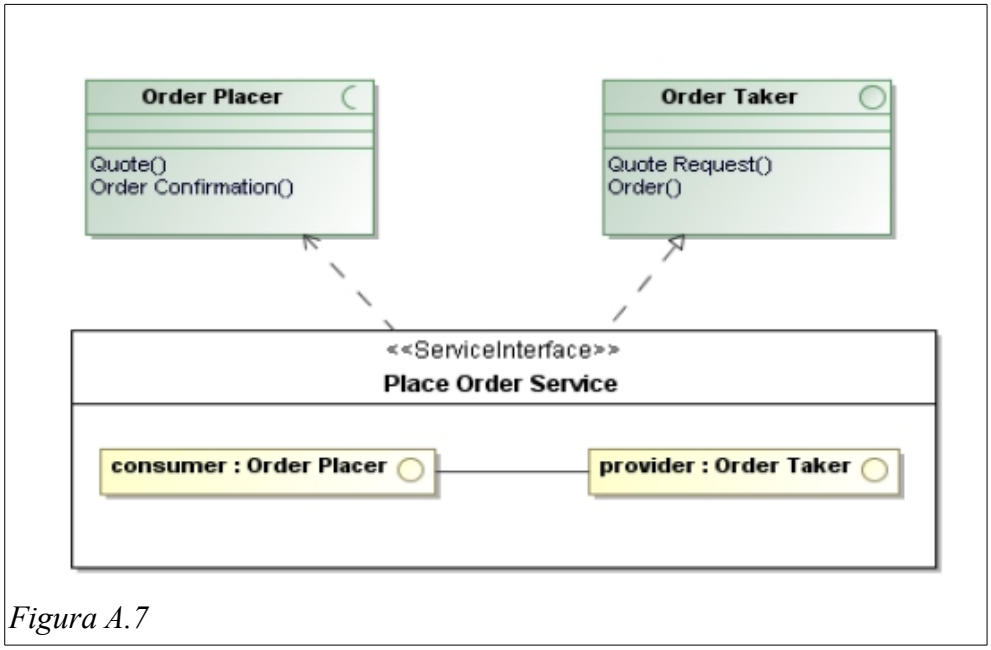


Figura A.7

Las interacciones obligatorias y opcionales entre los roles de una ServiceInterface se puede mostrar en un diagrama de coreografía, muy similar al diagrama nativo UML de actividad. Siguiendo el ejemplo anterior, el consumidor OrderPlacer realiza un pedido de Quote al OrderTaker y este le

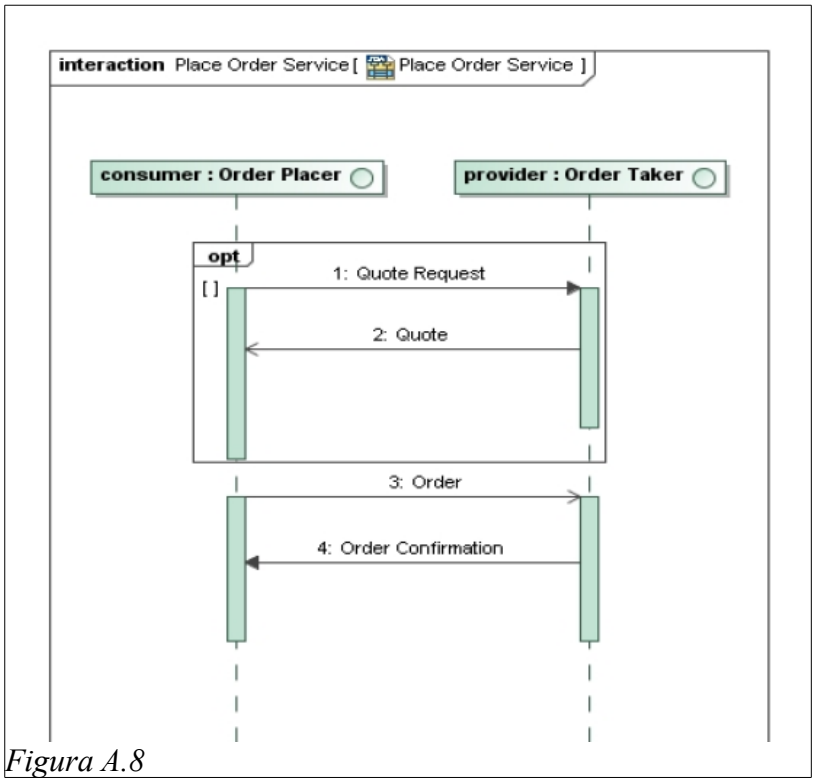


Figura A.8

provee la Quote. Luego el consumidor realiza un pedido de Order y es respondido con una confirmación (Figura A.8).

Arquitectura de servicios

Bajo el modelo SOA, se pueden representar las interacciones entre clientes-proveedores mediante un modelo arquitectónico de servicios, otro de los elementos brindados por SoaML. Un modelo de este tipo engloba a todos los servicios en determinado contexto e ilustra cómo los participantes se relacionan entre sí para consumar ciertos objetivos. En una arquitectura de servicios es importante determinar que roles van a desempeñar los participantes. Un rol define una función básica a realizar por una entidad en un contexto particular. Los participantes adquieren roles que les atribuyen ciertas cualidades dentro de la arquitectura. La figura A.9 ilustra este concepto.

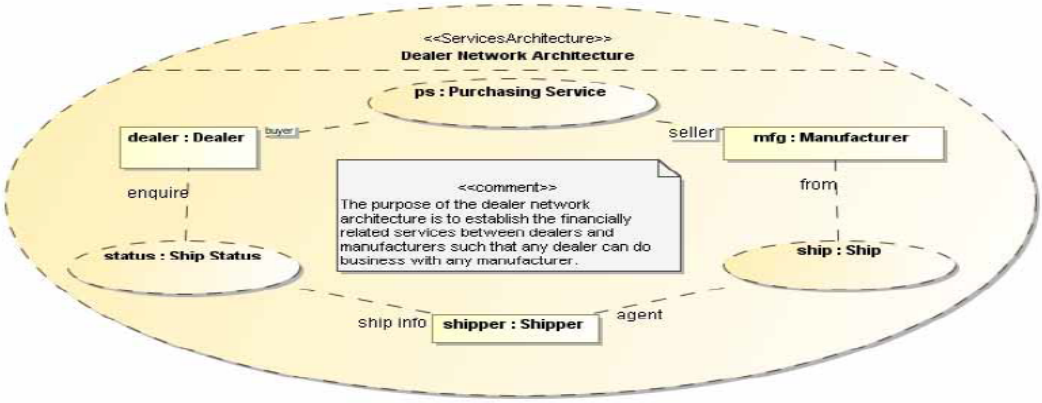


Figura A.9

En el modelo SoaML, el concepto está encapsulado en un diagrama de colaboración de UML estereotipado con <<ServiceArchitecture>>. Un participante puede tener asociada una arquitectura específica en la que las diferentes partes del mismo (ej.: departamentos dentro de una organización) trabajan en conjunto para proveer los servicios. Una arquitectura de servicios aparece en general realizando una interfaz de servicios UML de un participante.

La figura A.9 ilustra la arquitectura de servicios de un participante "Manufacturer" que consiste en un conjunto de componentes que interactuan a partir de determinados contratos de servicio (Service contracts) para realizar un servicio de compra-venta. En los componentes se indica el rol que ocupa (Fullfilment, Production) cada componente. Los roles internos dentro del participante se dibujan con bordes en líneas sólidas mientras que los externos con bordes en líneas punteadas.

Contrato de servicio (ServiceContract)

Figura A.10

Un contrato de servicio define los términos, condiciones, interfaces y coreografía que los participantes deben acordar para que el servicio se pueda ejecutar (figura A.10). Se puede ver como un binding entre proveedores y consumidores del servicio. A nivel del diagrama consiste en un diagrama de colaboración. Los roles definidos dentro de un contrato de servicio se establece mediante interfaces o interfaces de servicio (ServiceInterfaces) que son de ese rol.

Las interacciones de participantes con sus respectivos roles en un ServiceContract es representado en un diagrama de interacción o coreografía. El ServiceContract solo muestra en términos generales que roles y participantes se vinculan, y la coreografía desmenuza dicha relación en operaciones de mayor granularidad. La figura A.11 muestra la coreografía que describe las interacciones entre los componentes orderer y order processor.

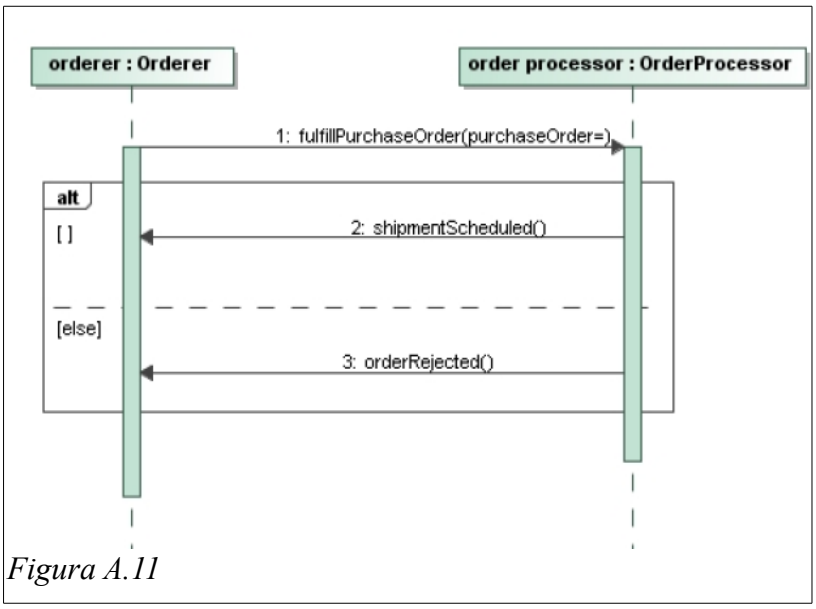


Figura A.11

4. UML

Generalidades

UML (Unified Modeling Language) es un lenguaje visual para especificación, construcción y documentación de artefactos de un sistema. Es un lenguaje de modelado de propósito general que puede ser utilizado para describir gran variedad de tipos de objetos y comportamientos, y puede ser aplicado a dominios heterogéneos (finanzas, salud, telecomunicaciones, etc.) y diferentes implementaciones (J2EE, .NET). La OMG adoptó la versión 1.1 en noviembre de 1997. Bajo la tutela de la OMG, UML ha resultado el lenguaje de modelado predominante en la industria de software.

El paradigma MDA de la OMG provee conceptualmente los elementos para definir la arquitectura de un sistema de software cuyo desarrollo está dirigido por modelos. UML, MOF y otras especificaciones juegan un rol importante en MDA ya que proveen lenguajes para la creación y transformación de modelos. La sintaxis abstracta de UML está especificada utilizando un metamodelo de UML que en sí es un modelo UML. El metamodelo de UML utiliza construcciones del subconjunto de UML que entran en la especificación de MOF 2. Por ejemplo, en el metamodelo raíz de UML la metaclase UML "Element" es una clase abstracta en el metamodelo UML que puede verse como una instancia de la metaclase Class con propiedad "isAbstract" en valor true. Otro ejemplo es la metaclase "Comment" que tiene un atributo "body", el cual puede verse como una instancia de la metaclase "Property" con nombre el valor del atributo "body" desde la perspectiva de MOF. Esto hace que UML sea definido sobre sí mismo, utilizando los mismos elementos del lenguaje, requiriendo de ciertas restricciones para asegurar que las construcciones sean bien formadas y sin entrar en definiciones recursivas. El metamodelo raíz de UML se muestra en la figura A.10.

Los modelos UML son almacenados en un repositorio de MOF 2 donde pueden ser alterados utilizando características y reglas de MOF e intercambiarse utilizando XMI.

Un modelo UML consisten en tres grandes categorías de elementos de modelado, cada uno de las cuales puede ser utilizado para representar información sobre las instancias de un sistema. Estas categorías son:

- Clasificadores. Un clasificador describe un conjunto de objetos con características estructurales y semánticas comunes. Un objeto es una instancia del sistema con estado y asociaciones con otros objetos. Los clasificadores UML predefinidos son: actor, asociación, clase, componente, datatype, nodo, subsistema, caso de uso.
- Eventos. Un evento describe un conjunto de posibles ocurrencias. Una ocurrencia es algo que sucede en cierto instante de tiempo y que trae consecuencias para el sistema. Un evento no necesariamente produce una transición de un estado del sistema a otro.
- Comportamientos. Un comportamiento describe un conjunto de posibles ejecuciones. Una ejecución es una secuencia de acciones realizadas en cierto período de tiempo que pueden generar o responder a ocurrencias de eventos y por ende cambiar el estado de los objetos. Algunos diagramas que forman parte de esta categoría: diagramas de actividad, diagramas de casos de uso, diagramas de estado, diagramas de secuencia y de comunicación, etc.

Metamodelo raíz de UML

En la figura A.12 se ilustra el metamodelo raíz de UML especifica los conceptos de modelado base de UML. La mayoría de las metaclases definidas en son abstractas, ya que comprenden los conceptos raíces de la jerarquía de los elementos de modelado UML.

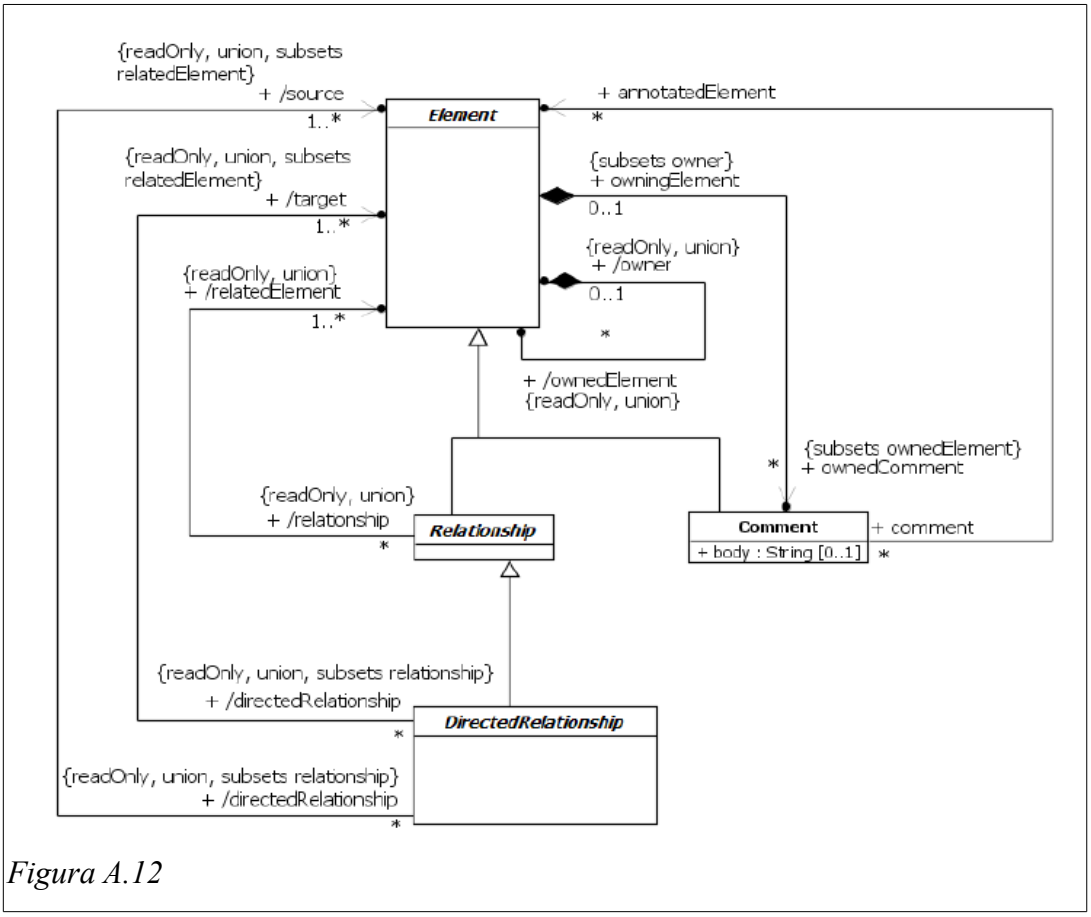


Figura A.12

Semántica del metamodelo raíz:

- Un **elemento** (Element) es un constituyente de un modelo. Los descendientes de un elemento proveen semántica apropiada para el concepto el cual representan. Cada elemento puede ser propietario y ser propiedad de otros elementos, formando una estructura de árbol. Los elementos hijos heredan las propiedades de los ancestros. Cuando un elemento se quita del modelo, todos sus elementos propios también se eliminan del modelo.
- Cada tipo de elemento puede contener **comentarios** (Comment). Los comentarios de un elemento no agregan semántica pero sí información útil para el lector del modelo.
- Una **relación** (relationship) es un elemento que especifica una asociación entre elementos. Los descendientes de relationship adhieren su propia semántica al concepto que representan. Una especialización de relación es relación directa (directed relationship) que representa el concepto de asociación desde un conjunto de elementos origen a un conjunto de elementos destino.

Diagramas y elementos principales de modelado

Diagramas de casos de uso. Describen las relaciones y dependencias entre un grupo de casos de uso y los actores participantes en el proceso. Se utilizan para facilitar la comunicación con los futuros usuarios del sistema y permiten determinar las características que tendrá el sistema. Elementos de un diagrama de caso de uso:

- Caso de uso. Descriptor de las interacciones típicas entre los usuarios de un sistema y el propio sistema. Especifica qué requisitos de funcionamiento debe de tener un sistema para llevar adecuadamente la interacción con los usuarios. Cada caso de uso está relacionado como mínimo con un actor y produce un resultado relevante para el usuario.
- Actor. Es una entidad externa que interactua con el sistema participando en un caso de uso. Los actores pueden ser humanos, otros sistemas u otros eventos externos.

Diagramas de clases. Muestran las diferentes clases, junto con sus métodos y atributos, que componen un sistema y cómo se relacionan unas con otras. Son diagramas "estáticos" ya que solo representan las relaciones entre las clases pero no los métodos que se invocan entre ellas. Una clase define atributos y métodos de una serie de objetos. Todas las instancias de la clase tienen el mismo comportamiento y el mismo conjunto de atributos, con valores de atributos que pueden ser distintos entre los objetos. Gráficamente, están representadas por rectángulos con el nombre de la clase y pueden mostrar atributos y operaciones en otros dos compartimentos del rectángulo.

Diagramas de secuencia. Muestran el intercambio de mensajes en un momento dado. Posen especial énfasis en el orden y el momento en que se envían los mensajes a los objetos. Los objetos están representados por líneas intermitentes verticales, con el nombre del objeto en la parte más alta. El eje de tiempo es vertical, incrementándose hacia abajo, de forma que los mensajes son enviados de un objeto a otro en forma de flechas con los nombres de la operación y los parámetros.

Diagramas de comunicación o colaboración. Muestran las interacciones que ocurren entre objetos que participan en una situación determinada. Son similares a los diagramas de secuencia en cuanto a semántica, pero fijan interés en las relaciones entre los objetos y su topología. En estos diagramas los mensajes enviados de un objeto a otro se representan mediante flechas, mostrando el nombre del mensaje, los parámetros y el número de secuencia del mensaje. Se emplean para representar un flujo de programa y un proceso en la lógica de negocio en un cierto período de tiempo.

Diagramas de estado. Muestran los diferentes estados de un objeto durante su ciclo de vida y los eventos que provocan cambios en el estado de un objeto. Los diferentes estados que puede adquirir un objeto se representan en una máquina de estado y cuyas transiciones se etiquetan con los posibles eventos que pueden afectar su estado actual.

Diagramas de actividad. Los diagramas de actividad describen la secuencia de actividades de un sistema. Son una forma particular de diagramas de estado, que únicamente contiene actividades. Estos diagramas están asociados a una clase, operación o caso de uso. Soportan tanto actividades secuenciales como paralelas. Una actividad se puede ver como un único paso de un proceso.

Otros diagramas UML comunes son: diagramas de componentes, diagramas de implementación y diagramas de entidad-relación.

5. Modelo relacional

El modelo relacional es un modelo de base de datos que representa el diseño de una base de datos

relacional. En el modelo relacional de una base, todos los datos se representan en términos de tuplas agrupadas en relaciones. Un modelo relacional ayuda a realizar consultas sobre la base, permite diseñar estructuras de datos para almacenar diversos tipos de datos y ejecutar procesos para realizar transformaciones de los datos, todo en base a que permite conocer el esquema relacional de la base.

Las bases relacionales almacenan datos en tablas. Las relaciones se definen entre tablas mediante referencias cruzadas. La forma en que se almacenan los datos permite a los usuarios comprender de manera sencilla la estructura y el contenido de los datos. Los elementos de modelado básicos del modelo relacional son:

- Tablas. Una base relacional consiste en una colección de tablas (entidades) de las cuales se quiere buscar la información. Una tabla consiste de columnas, que constituyen las propiedades de la tabla, y de filas con los registros almacenados y recuperar.
- Columna. Una columna puede referir a un campo o conjunto de campos en una tabla. Describe una propiedad de interés que se quiere almacenar.
- Relación. Es la asociación entre dos tablas. Asocia datos en las tablas de diversas formas dependiendo de la cardinalidad: 1:1, 1:N o N:N. Las relaciones se expresan en términos de claves foráneas en las tablas.

En la figura A.13, se presenta el metamodelo del modelo relacional.

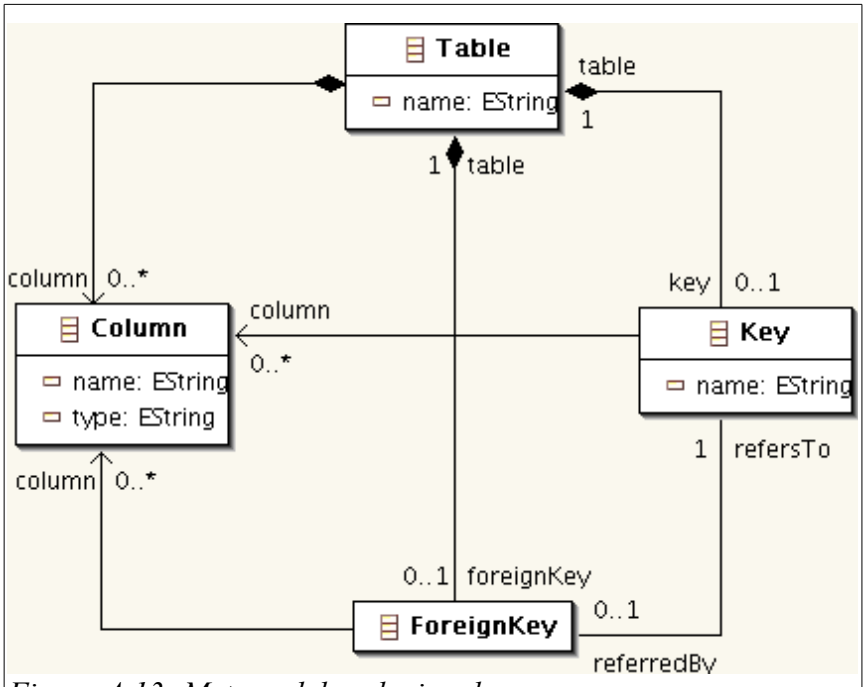


Figura A.13: Metamodelo relacional

B. Eclipse y desarrollo de plugins

En general, el SDK de eclipse está conformado por los diferentes capas:

- RCP: provee el conjunto mínimo de plugins necesario para desarrollar cualquier aplicación para Eclipse.
- JDT: alberga todo un Java IDE y es una plataforma en sí misma.
- IDE: instrumento de la plataforma para el desarrollo de múltiples aplicaciones.
- PDE: provee las herramientas necesarias para construir plugins y aplicaciones RCP.

La figura A.14 muestra las capas que conforman la plataforma Eclipse. La figura A.15 ilustra cómo cada capa alberga su propio conjunto de plugins que implementan la funcionalidad de cada capa.

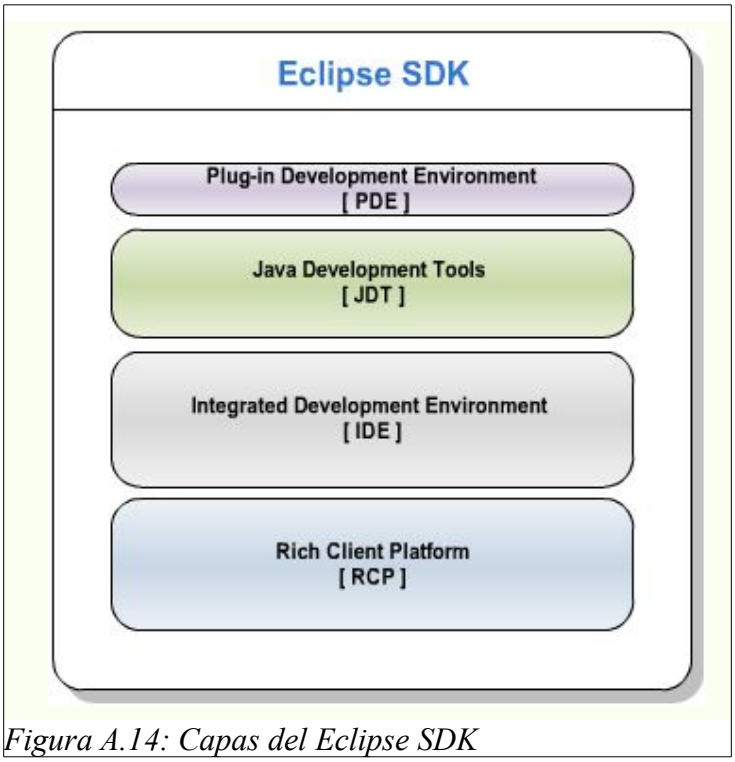


Figura A.14: Capas del Eclipse SDK

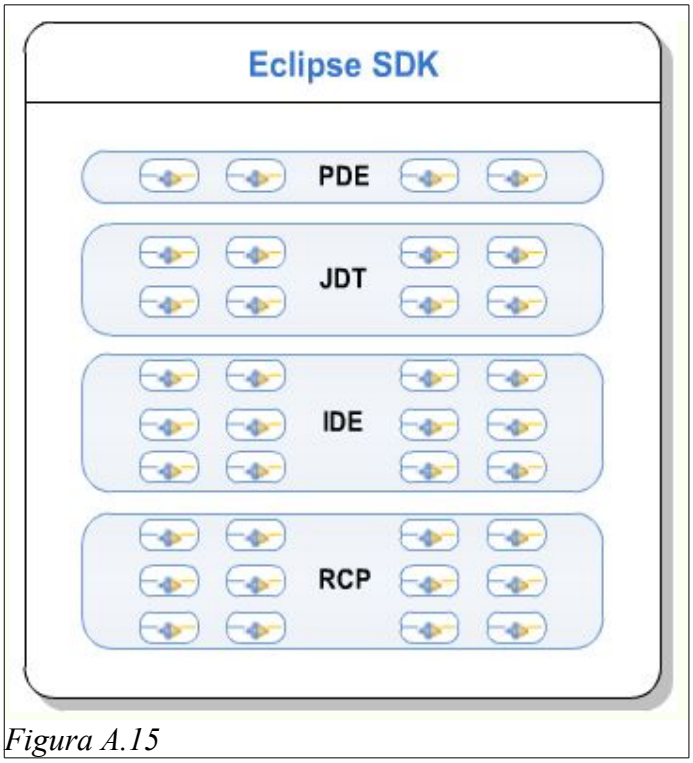


Figura A.15

Workspace

El workbench provee la UI con la personalidad de la plataforma Eclipse y los elementos con los que el usuario interactúa con las herramientas de la plataforma. El workbench está implementado utilizando SWT y Jface, pero no utiliza Java AWT ni Swing.

Una ventana en el workbench puede tener diferentes perspectivas, cada una de las cuales se puede hacer visible en cualquier momento. Una perspectiva es una colección de paneles. Cada perspectiva tiene sus propias vistas y editores ubicados de diferentes formas y posiciones, y pueden ser ocultadas cuando lo desee el usuario. Diversos tipos de vistas y editores pueden ser abiertas en

cualquier momento dentro de una perspectiva. El usuario puede rápidamente cambiar de perspectiva al momento de realizar otra tarea y configurarla de acuerdo a sus necesidades en función de adaptarse a determinada tarea. La plataforma provee perspectivas generales para navegabilidad entre recursos (manipular proyectos, editar archivos dentro de un proyecto, etc.), ayuda online y team support. Por otra parte, los plugins pueden aportar sus propias perspectivas.

La plataforma se encarga del manejo de la ventana principal del workbench y a las perspectivas. Los editores y las vistas son instanciados al momento de ser utilizados y eliminados al ser desactivados.

La figura A.16 ilustra los principales elementos de la UI del IDE: perspectivas, editores y vistas.

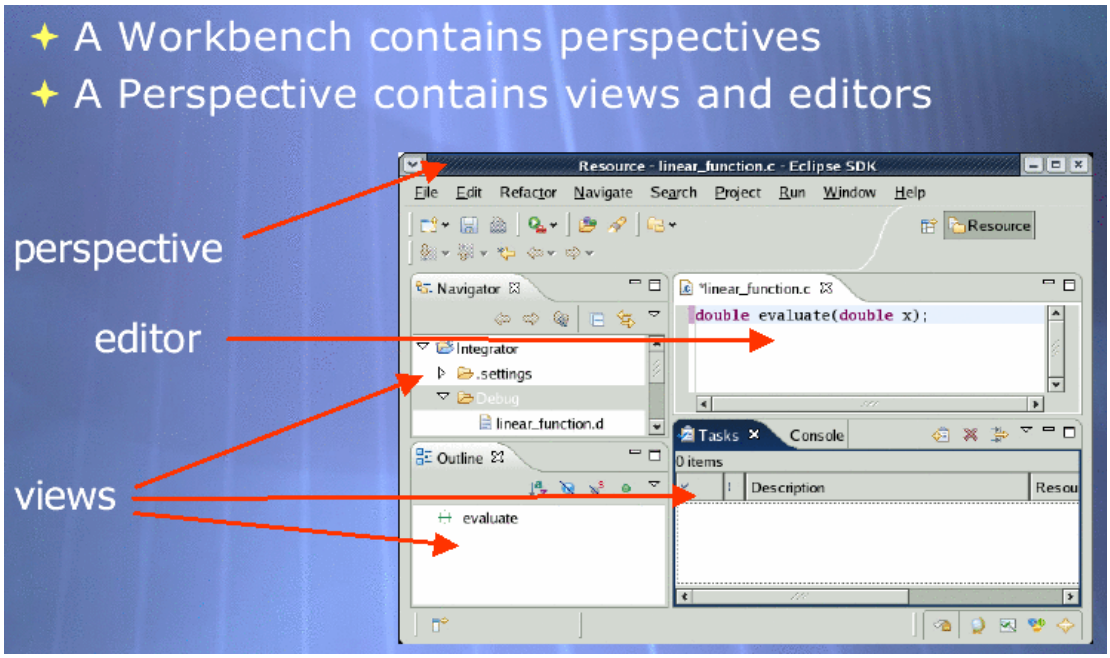


Figura A.16: Workbench y elementos

PDE: desarrollo de plugins

PDE provee herramientas para crear, desarrollar, testear, debuggear, compilar y deployar plugins de Eclipse. PDE es también un entorno dentro de Eclipse para desarrollo de componentes en la plataforma Java en general, no solo para el desarrollo de plugins. Está construido sobre la plataforma Eclipse y particularmente sobre JDT.

Componentes

El proyecto PDE está dividido en diversos componentes; los principales son: Build, UI y API tools.

Cada uno de estos componentes opera como si fuera un proyecto en sí mismo, con su propio con junto de funcionalidades, vistas y perspectivas. Ellos son:

- PDE Build. Su principal objetivo es facilitar el proceso de automatización de compilado de plugins. Esencialmente, produce scripts Ant, que provee mecanismos para compilado, ensamblado, testeo y ejecución de aplicaciones de Java, basados en información obtenida de archivos fuente del proyecto. Los scripts generados pueden utilizar otros proyectos referenciados localmente o en un repositorio.
- PDE UI. En general, modeladores y editores que facilitan el desarrollo de plugins. Algunas características que el componente provee al SDK:
  - Editores multipágina que manipulan todos los archivos de manifest del un plugin.
  - Herramientas de RCP: Editores y manuales que permiten definir, depurar y exportar productos a múltiples plataformas
  - Creación de nuevos plugins, fragmentos, características y hacer actualizaciones.
  - Importar plugins y features del file system.
  - Wizards que permiten compilar, empaquetar y exportar plugins.
  - Provee vistas que ayudan a los desarrolladores de plugins inspeccionar diversos aspectos durante el desarrollo.
  - Herramienta de conversión: wizards que permiten convertir un proyecto Java en un proyecto de naturaleza plugin.
  - Editores y herramientas para ayuda del desarrollador.
- PDE API Tools. Brinda a los desarrolladores una API de mantenimiento para reportes de defectos como incompatibilidades de binarios, números incorrectos de versiones de plugins, falta de tags apropiadas y uso de código que sobresale de la API entre plugins. Estas herramientas están integradas al SDK de Eclipse y se emplean para ayudar a automatizar el proceso de compilación. Algunas tareas de las cuales se responsabiliza el componente son:
  - Identificar compatibilidad de binarios entre versiones de un componente de software.
  - Actualizar números de versión de plugins basados en un esquema de versionado predefinido.
  - Proveer nueva documentación de tags y código.
  - Identificar ausencia de tipos en la API que son utilizados por fuera de la misma.
- PDE Incubator. Provee herramientas para testear nuevos procedimientos o funcionalidades que pueden aplicar al desarrollo general del espacio de plugins y eventualmente ser añadidas al SDK de Eclipse. Está dividido en áreas de trabajo (work areas) y componentes graduados (graduated components).
- PDE doc. Documentación de uso del PDE.

El desarrollo del PDE es promovido por la comunidad de forma abierta y transparente. El entorno admite diversas formas de reporte de bugs, contribuir mejoras, producir nuevos plugins o features,

participar en grupos de testing, etc..

JDT

El proyecto JDT contribuye con un conjunto de plugins que adhiere las características de un Java IDE en la plataforma Eclipse. Los plugins de JDT están categorizados en:

- JDT APT: agrega anotaciones de soporte a Java 5 o superior de Eclipse, procesadas por la herramienta apt de Sun.
- JDT Core: define la infraestructura central de JDT. Brinda un modelo de java para la navegabilidad entre los diversos recursos de un proyecto java. A su vez, proporciona facilidades para realizar búsquedas, asistencia y selección de código.
- JDT Debug: implementa la funcionalidad de depuración del Java IDE. Permite correr la VM de Java en modo ejecución o en modo debug, evaluación de expresiones en tiempo de depuración, inclusión dinámica de clases de la VM, etc..
- JDT UI: Realiza contribuciones al workbench de Eclipse: explorador de paquetes, vista de jerarquía de tipos, wizards para creación de elementos de java. Provee refactoreo para diversas operaciones: renombre y eliminación seguros de archivos, actualización de referencias, extracción de métodos.

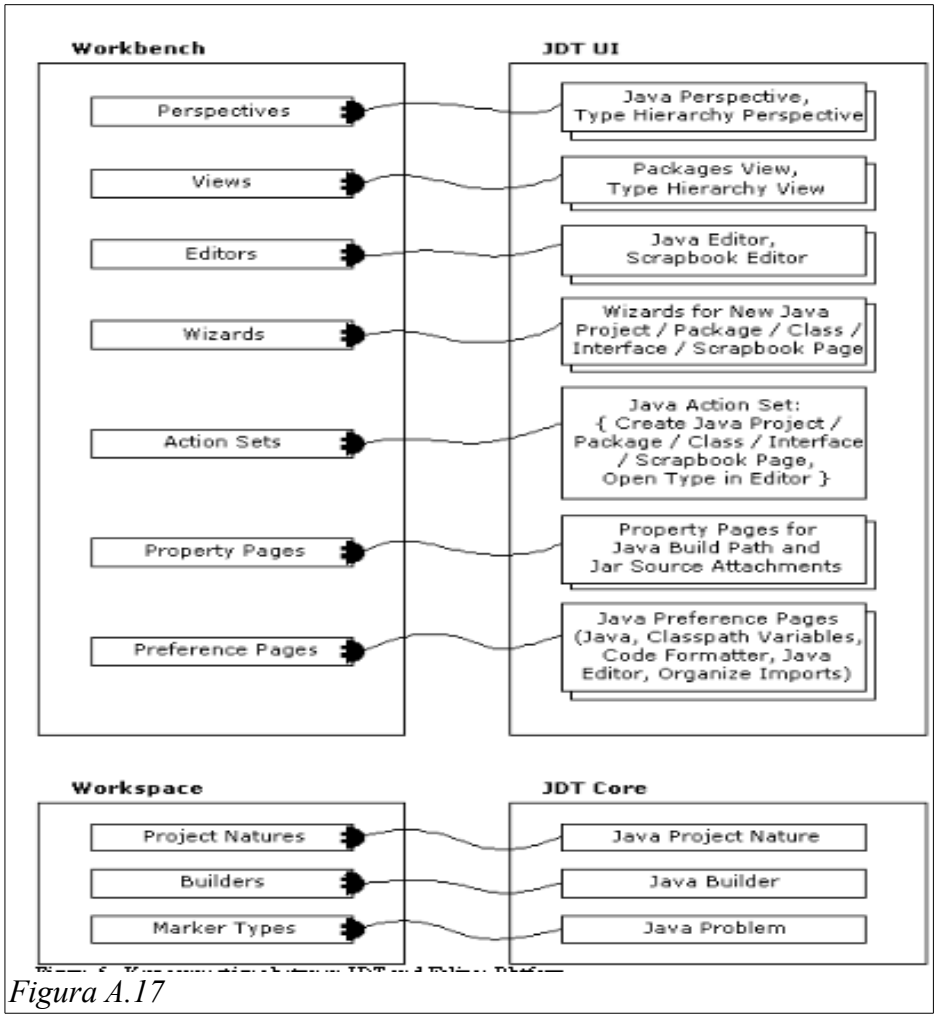


Figura A.17

La plataforma Eclipse define un conjunto de librerías y servicios que constituyen una infraestructura muy rica para poder construir herramientas y aplicaciones arbitrarias, en el cual destacamos el rol del RCP en la plataforma. RPC puede ser empleado para construir cualquier sistema arbitrario con diferentes propósitos y que no estrictamente esté ligado a la ingeniería de software. Permite crear aplicaciones de diverso tipo que trabajan en conjunción con servidores de aplicaciones, bases de datos y otros elementos de backend; y también contemplando las necesidades de los usuarios.

Una gran ventaja de la plataforma es que es extensible open source a través del desarrollo de componentes de plug-ins. Un plugin es un programa java que extiende la funcionalidad del entorno de alguna manera. Eclipse viene integrado con un conjunto estándar de plugins, incluyendo el JDT para trabajar sobre un IDE corriente de Java. Es open source ya que el código fuente de la plataforma está disponible para la comunidad de forma que los usuarios pueden libremente modificar y redistribuir sus propias piezas de software. La disponibilidad open source está limitada por el llamado copyleft: las licencias del software están protegidas por derechos de autor y se prohíbe su distribución exceptuando a los usuarios autorizados. El copyleft también requiere que cualquier software a ser redistribuido deba cumplir con un conjunto de reglas de distribución. Los desarrolladores de plugins para Eclipse pueden liberar o modificar cualquier código bajo las condiciones dispuestas en las cláusulas del CPL.

Instrucciones de desarrollo de un simple plugin

La forma más sencilla de crear un plugin es utilizando el PDE.

1. Descargar del sitio de descargas de http://www.eclipse.org la versión estándar de Eclipse. Descomprimir el .zip descargado.
2. Instalar el componte PDE. Ir al menu Help → Install New Software. Hacer click en el botón Add e insertar un nombre (por ejemplo, Eclipse PDE) y la URL del sitio de actualizaciones de PDE http://download.eclipse.org/eclipse/pde/visualization/updates luego presionar Ok. Hacer check en el nombre del componente a instalar "PDE Component" y seleccionar Next. Se instalará el componente. Cuando finaliza la instalación, deberá reiniciar el IDE para completar el proceso de instalación.
3. Crear un nuevo proyecto de plugin mediante File → New Proyect → Plug-in development → Plug-in project. Se creará a modo de ejemplo un proyecto de nombre com.example.hello. Ingresar el nombre correspondiente y dejar el resto de los campos tal como se muestra en la figura A.19.

La siguiente página del wizard muestra opciones para generación de una clase Java para activación. La clase activadora es necesaria si se quiere realizar alguna acción al momento de levantar o de bajar el plugin (por ejemplo, cargar ciertos recursos al ser activado y realizar un clean up al desactivarse). Recordar que los plugins de Eclipse se levantan a demanda del usuario. Creamos una clase java de activación con el nombre com.example.hello.Activator y damos el nombre "FING" al proveedor (Vendor) del plugin. Completar el resto de los campos tal como se muestra en la figura A.18.

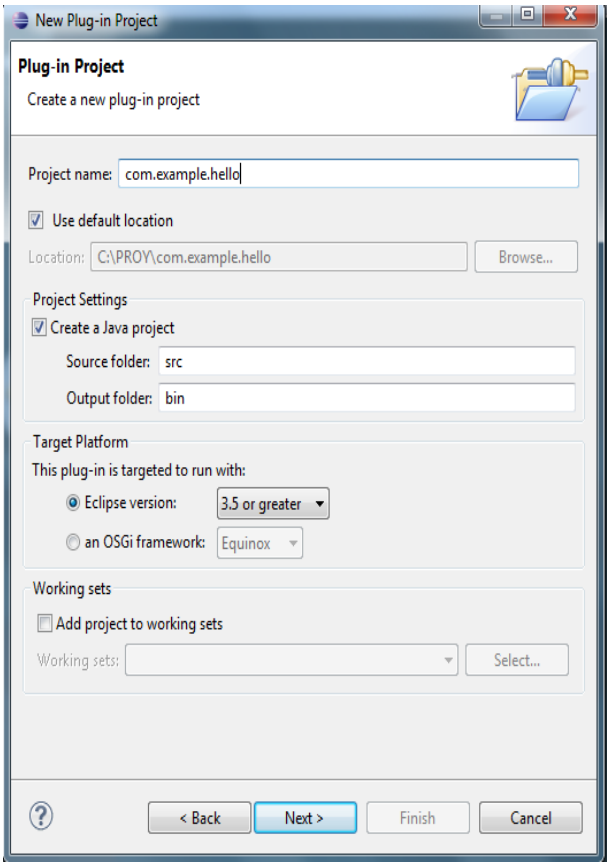


Figura A.19

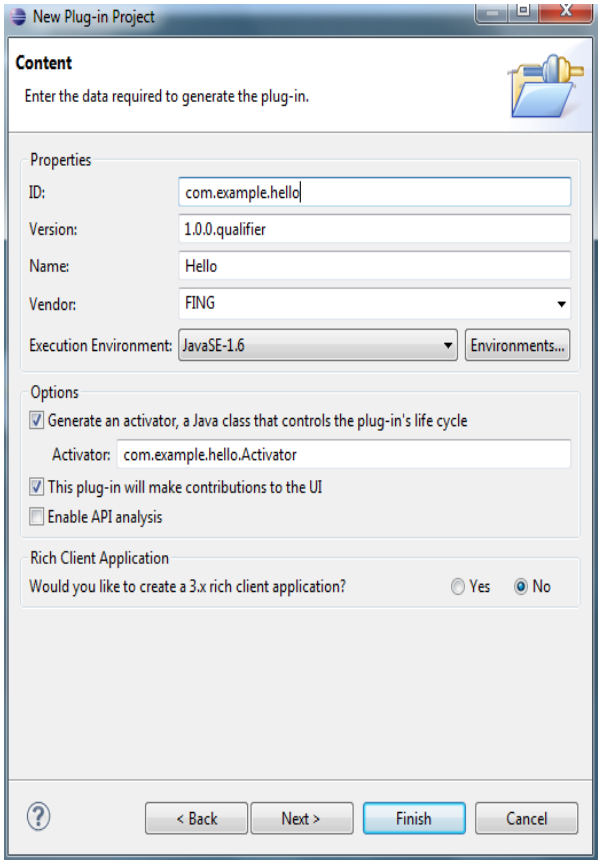


Figura A.18

Podemos crear un plugin de cero o podemos elegir arrancar uno a partir de un template. En este caso en el wizard siguiente vamos a crearlo utilizando el template "Plug-in with a view". El template del plugin solo genera una vista llamada SampleView dentro de una categoría de vista SampleCategory. La view contendrá tres elementos: "One", "Two" y "Three".

4. Damos siguiente en el Wizard.

Se creará el nuevo proyecto del plugin con el nombre dado en (3). Aparecen cuatro archivos en el proyecto:

- Activator.java. Esta es la principal clase del plugin. Cuando el plugin es activado, un método de inicio es llamado para inicializar el estado del plugin y cargar ciertos recursos.
- SampleView.java. Esta es la clase de la vista generada. Contribuye a la generación de la interfaz de usuario del plugin.
- Manifest.mf. Reside información de runtime del plugin
- Plugin.xml. Define información de extensión del plugin

Descripción del editor de manifest.

Haciendo doble click en el manifest.mf, se abre el editor del archivo correspondiente. Se muestran varias pestañas. Se describen a continuación algunas de ellas:

- Overview. Información general del plugin, la mayor parte provista durante los wizards de creación.
- Dependencies. Se muestra una lista de plugins de dependencia que son requeridos para que el plugin actual compile. En este caso,vemos que nuestro plugin tiene dependencias con los plugins eclipse.ui y eclipse.core.
- Runtime. Se declara todos los paquetes y jars que van a hacer expuestos para ser potencialmente utilizados por otros plugins en el futuro.
- Extensions. Hace sencilla la creación de extensiones para algún punto de extensión. Eclipse provee 213 puntos de extensión en los cuales los clientes pueden agregar funcionalidad. Básicamente, se selecciona un punto de extensión y se trabaja sobre este para construir las propias extensiones.

Testeo del plugin

Hay dos formas de testear un plugin. La primera, podemos construir un producto el cual crea un bundle del plugin, luego el bundle es volcado dentro de la carpeta del plugin en la instalación del eclipse. La segunda, correr el plugin desde la pestaña Overview del editor del manifest, haciendo click en la el link "Launch an Eclipse Application" (figura A.20). Presionando aquí, se abre una nueva del IDE con el plugin registrado. En la nueva instancia abierta, vamos a Window → Show View → Other → SampleCategory → SampleView. Finalmente, se despliega la vista del plugin.

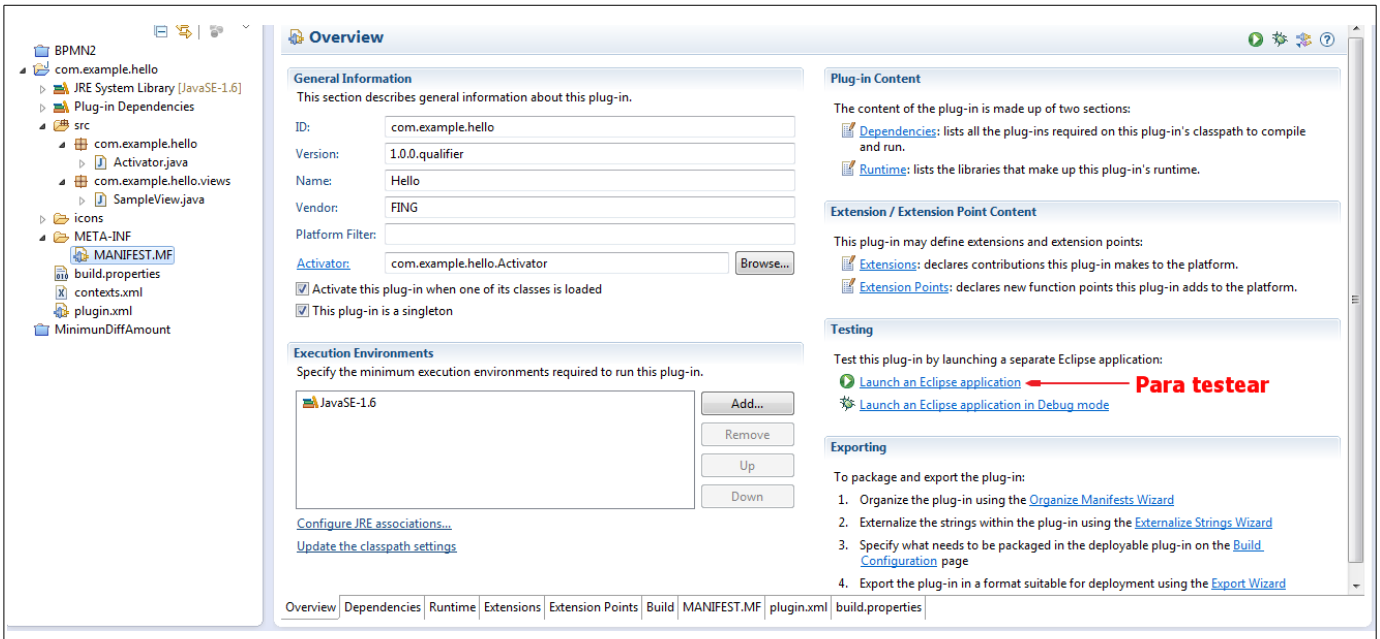


Figura A.20

C. Especificación de casos de uso

C.1 Seleccionar motor.

Actor principal:

Usuario

Descripción:

El usuario selecciona el motor en el cual va a ejecutar una transformación QVT.

Precondiciones:

1. Existe algún motor QVT registrado en el sistema.
2. El motor QVT a seleccionar debe estar registrado en el sistema. Esto implica que en el sistema estén los archivos con las librerías que implementan el motor y que exista una implementación de la API de transformación de la que hace uso el sistema para invocar las operaciones del motor.

Flujo de eventos

- Normal

Acción de Usuario	Respuesta del Sistema
1- El usuario selecciona la opción "Seleccionar motor"	
	2- El sistema muestra la lista con los nombres de los motores disponibles.
3- El usuario selecciona el motor en el cual va a trabajar.	
4-El usuario presiona "Ok".	
5-Fin CU	

Poscondiciones:

El sistema actualiza la referencia al motor seleccionado.

C.2 Registrar metamodelos.

Actor principal:

Usuario

Descripción:

El usuario registra un metamodelo en el motor seleccionado. El metamodelo debe ser de una extensión reconocida por el respectivo motor.

Precondiciones:

1. Hay un motor QVT seleccionado por el usuario.

Flujo de eventos

- Normal

Acción de Usuario	Respuesta del Sistema
1- El usuario selecciona la opción “Registrar metamodelo”	
	2- El sistema lista los metamodelos registrados del motor.
3- El usuario registra un metamodelo. Agrega la ruta al archivo del metamodelo a registrar. La ruta puede ser absoluta (en el file system) o relativa al workspace.	
	4-El sistema muestra la lista actualizada de metamodelos.
5-El usuario presiona “Ok”.	
6- Fin CU	

- Alternativo

5a)- El usuario desea registrar más metamodelos.

5A1) Vuelve a 3

Poscondiciones:

El sistema actualiza la lista los metamodelos registrados en el motor.

C.3 Alta de configuración.

Actor principal:

Usuario

Descripción:

El usuario da de alta una nueva configuración en el motor seleccionado.

Precondiciones:

1. Hay un motor QVT seleccionado por el usuario.
2. Existe en el workspace un script de transformación QVT con extensión aceptada por el motor.
3. Existe en el file system un archivo en formato .xmi de modelo de entrada de la transformación.

Flujo de eventos

- Normal

Acción de Usuario	Respuesta del Sistema
1- El usuario selecciona la opción "Alta de configuración"	
	2- El sistema despliega una ventana con los campos en blanco de los datos de la nueva configuración a ingresar.
3- El usuario ingresa los datos de la configuración. Ingresa un nombre de configuración y selecciona la ruta relativa al Workspace del archivo de script de transformación qvt y las rutas relativas al	

Workspace o absolutas en el file system a los archivos con los modelos de origen y destino en formato estándar xmi.	
4-El usuario presiona “Guardar”.	
	5-El sistema valida correctamente los campos ingresados y devuelve una respuesta acorde.
6- Fin CU	

- Alternativo

3a)- El usuario desea agregar un directorio donde guardar la traza de la transformación.

3A1) El usuario agrega la ruta relativa al workspace del directorio donde guardar la traza.

5a)- El sistema valida sin éxito los datos ingresados. Hay campos incorrectos.

5A1) La ruta al script de transformación QVT no es correcta.

5A11) El usuario ingresa correctamente la ruta correspondiente al script QVT.

5A2) Las rutas del modelo de entrada y/o de salida no son correctas.

5A21) El usuario ingresa correctamente las rutas a los archivos de los modelos que estén invalidas.

5A3) Falta el nombre de la configuración o ya existe una configuración con este nombre en el motor.

5A31) El usuario ingresa un nuevo nombre de configuración

Vuelve a 4)

Poscondiciones:

El sistema agrega la nueva configuración al motor seleccionado.

C.4 Modificación de configuración.

Actor principal:

Usuario

Descripción:

El usuario modifica los parámetros de una configuración existente.

Precondiciones:

1. Hay un motor QVT seleccionado por el usuario.
2. Existe en el workspace un script de transformación QVT con extensión aceptada por el motor.
3. Existe en el file system un archivo en formato xmi de modelo de entrada.
4. Existe una configuración en el motor seleccionado.

Flujo de eventos

- Normal

Acción de Usuario	Respuesta del Sistema
1- El usuario selecciona la opción "Modificación de configuración"	
	2- El sistema despliega una lista con los nombres de las configuraciones existentes en el motor.
3- El usuario selecciona una configuración.	
	4-El sistema despliega una ventana con los campos cargados de los datos de la configuración seleccionada.
5- El usuario actualiza los campos con los datos de la configuración.	
6-El usuario presiona "Guardar".	
	7-El sistema valida correctamente los campos ingresados y devuelve una respuesta acorde.
8- Fin CU	

- Alternativo

5a)- El usuario desea agregar un directorio donde guardar la traza de la transformación.

5A1) El usuario agrega la ruta relativa al workspace del directorio donde guardar la traza.

7a)- El sistema valida sin éxito los datos ingresados. Hay campos incorrectos.

7A1) La ruta al script de transformación QVT no es correcta.

7A11) El usuario ingresa correctamente la ruta correspondiente al script QVT.

7A2) Las rutas del modelo de entrada y/o de salida no son correctas.

7A21) El usuario ingresa correctamente las rutas a los archivos de los modelos que estén invalidas.

7A3) Falta el nombre de la configuración o ya existe una configuración con este nombre en el motor y no es la siendo modificada.

7A31) El usuario ingresa un nuevo nombre de configuración o el nombre actual.

Vuelve a 6)

Poscondiciones:

El sistema actualiza la configuración en el motor seleccionado.

C.5 Baja de configuración.

Actor principal:

Usuario

Descripción:

El usuario da de baja una configuración existente en el motor seleccionado.

Precondiciones:

1. Hay un motor QVT seleccionado por el usuario.
2. Existe una configuración en el motor seleccionado.

Flujo de eventos

- Normal

Acción de Usuario	Respuesta del Sistema

1- El usuario selecciona la opción “Baja de configuración”	
	2- El sistema despliega una lista con los nombres de las configuraciones existentes en el motor.
3- El usuario selecciona una configuración a eliminar.	
4-El usuario presiona “Ok”.	
	5-El sistema solicita confirmación.
6- El usuario confirma	
7- Fin CU	

- Alternativo

6A) El usuario no confirma la baja.

6A1) Vuelve a 3

Poscondiciones:

El sistema elimina la configuración elegida en el motor seleccionado.

C.6 Ejecutar transformación.

Actor principal:

Usuario

Descripción:

El usuario ejecuta la transformación asociada a una configuración determinada en el motor seleccionado.

Precondiciones:

1. Hay un motor QVT seleccionado por el usuario.
2. Existe una configuración en el motor seleccionado.

Flujo de eventos

- Normal

Acción de Usuario	Respuesta del Sistema
1- El usuario selecciona la opción “Ejecutar transformación”	
	2- El sistema despliega una lista con los nombres de las configuraciones existentes en el motor.
3- El usuario selecciona una configuración a ejecutar.	
4-El usuario presiona “Ejecutar”.	
	5-El sistema devuelve el resultado de la transformación
6- Fin CU	

Poscondiciones:

El sistema manda ejecutar en el motor seleccionado la transformación definida en la configuración. El motor chequea que los modelos de la transformación tengan asociados los respectivos metamodelos registrados. En caso de que no hayan sido registrados, el motor seleccionado devuelve un error y el sistema se lo muestra al usuario.

El motor carga el modelo de entrada y el script qvt y se ejecuta la transformación. Si la ejecución es exitosa, se genera el modelo de destino en formato xmi en la ruta especificada en la configuración y el sistema devuelve un mensaje de éxito. Si la configuración tiene definida un directorio de traza, el motor genera los archivos de la traza en el directorio.

C.7 Generar xmi

Actor principal:

Usuario

Descripción:

El usuario selecciona un modelo BPMN2.0 y genera un archivo en formato de intercambio xmi equivalente.

Precondiciones:

1. Existe una transformación XSLT de origen BPMN2.0 a destino xmi.
2. Existe un archivo en formato BPMN2.0 de entrada.

Flujo de eventos

- Normal

Acción de Usuario	Respuesta del Sistema
1- El usuario selecciona la opción "Transformar modelo de origen"	
2- El usuario ingresa las rutas (relativas al workspace o absolutas) al modelo BPMN2.0 y la transformación XSLT. También ingresa la ruta con el nombre del archivo destino xmi a crear.	
	3- El sistema ejecuta la transformación y devuelve un mensaje de resultado acorde.
4-Fin CU	

Poscondiciones:

El sistema crea en la ubicación correspondiente el archivo xmi que retorna la transformación xslt en caso de ser exitosa.

D. Plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
   <extension
         point="org.eclipse.ui.commands">
      <command
            id="pgsoaqvt.installengineqvt"
            name="installengine">
      </command>
      <command
            id="pgsoaqvt.registermetamodelqvt"
            name="registermetamodel">
      </command>
      <command
            id="pgsoaqvt.defineconfigurationqvt"
            name="defineconfiguration">
      </command>
      <command
            id="pgsoaqvt.executetransformationqvt"
            name="executetransformation">
      </command>
   </extension>
   <extension
         id="pgsoaqvt2014.menus"
         point="org.eclipse.ui.menus">
      <menuContribution
            locationURI="menu:org.eclipse.ui.main.menu?after=additions"
            allPopups="true">
         <menu
               id="plugin.menu.pgsoaqvtqvt"
               label="Pg SOAQVT"
               mnemonic="PSOA">
            <command
                  commandId="pgsoaqvt.installengineqvt"
                  id="pgsoaqvt.command.installengineqvt"
```

```
            label="Select engine"
            style="push">
         </command>
         <command
            commandId="pgsoaqvt.registermetamodelqvt"
            id="pgsoaqvt.command.metamodelsqvt"
            label="Register metamodels"
            style="push">
         </command>
         <command
            commandId="pgsoaqvt.defineconfigurationqvt"
            id="pgsoaqvt.command.configurationsqvt"
            label="Configurations"
            style="push">
         </command>
         <command
            commandId="pgsoaqvt.executetransformationqvt"
            id="pgsoaqvt.command.executeTransformationqvt"
            label="Execute tranformation"
            style="push">
         </command>
      </menu>
   </menuContribution>
</extension>
<extension
      point="org.eclipse.ui.handlers">
   <handler
         class="gui.menus.handlers.SelectEngineMenuHandler"
         commandId="pgsoaqvt.installengineqvt">
   </handler>
   <handler
         class="gui.menus.handlers.RegisterMetamodelsMenuHandler"
         commandId="pgsoaqvt.registermetamodelqvt">
   </handler>
   <handler
         class="gui.menus.handlers.DefineConfigMenuHandler"
```

```
            commandId="pgsoaqvt.defineconfigurationqvt">
      </handler>
      <handler
            class="gui.menus.handlers.LaunchConfigurationMenuHandler"
            commandId="pgsoaqvt.executetransformationqvt">
      </handler>
   </extension>
   <extension
         point="org.eclipse.ui.startup">
      <startup
            class="pgsoaqvt.PluginStartup">
      </startup>
   </extension>

</plugin>
```