ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Integración de Software para el Desarrollo de Aplicaciones Domóticas basada en Plug-in

AUTOR: **Francisco Javier Jiménez Castelo**
DIRECTOR: **Mª Francisca Rosique Contreras**

Abril, 2014

Autor	Francisco Javier Jiménez Castelo
E-mail del Autor	fjavierjimenezc@gmail.com
Director(es)	Mª Francisca Rosique Contreras
Título del PFC	Integración de Software para el Desarrollo de Aplicaciones Domóticas basada en Plug-in
Descriptores	Eclipse, Plug-in, Domotica.

Resúmen

Se ha desarrollado un Plug-in Eclipse para la herramienta Metadomo aprovechando la capacidad de Eclipse para ser extendido usando el mecanismo de extensión y puntos de extensión.
Con este Plug-in se pretende que el uso de metadomo para el usuario sea más fácil y para ello se ha realizado una interfaz amigable y sencilla, que proporciona la representación y seguimiento del flujo de trabajo típico de usuario en Metadomo, mejorando la operatividad y uso de esta herramienta. Además, la vista ha simplificado el número de operaciones que el usuario debe hacer para realizar las mismas tareas que en el entorno original.

Titulación	Ingeniero Técnico en Telecomunicaciones, esp. Telemática
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Abril, 2014

Índice

Índice de Figuras

Capítulo 1 Motivación

El entorno de Metadomo queremos que integre un conjunto de herramientas diseñadas para facilitar la creación y las transformaciones necesarias en un sistema domótico. Metadomo debe permite modelar gráficamente los componentes de manera intuitiva, generar automáticamente el código necesario de manera transparente para el usuario, y modernizar y reutilizar el software ya existente. Se desea que Metadomo proporcione un entorno totalmente unificado para facilitar su uso, aprendizaje y distribución.

El presente Proyecto se centra en la realización de un Plug-in Eclipse dirigido a integrar en un mismo IDE herramientas software existentes, concretamente, se aborda la herramienta Metadomo persiguiendo, por una parte, mejorar la usabilidad y operatividad del entorno para el desarrollo de aplicaciones domoticas, y por otra, facilitar su distribución y extensión. El objetivo principal de este Proyecto es la creación de un Plug-in Eclipse que integre la herramienta Metadomo, creando así un entorno de trabajo orientado a la domótica mediante el cual podamos realizar todas las operaciones que permite Metadomo, pero mejorando la usabilidad, operatividad y distribución de la herramienta. Para ello el Plugin desarrollado en este Proyecto se compone de los siguientes elementos: Una vista Eclipse que representa de forma gráfica el flujo de trabajo típico que un usuario ha de seguir en Metadomo y que permite realizar las operaciones de manera más intuitiva. Un botón en la barra de herramientas que habilita/deshabilita la vista en el entorno. Nueva opciones de menús para mejorar la accesibilidad de las funcionalidades de Metadomo. Una ayuda que expone el funcionamiento del Plug-in. Una perspectiva que engloba todo el entorno de trabajo del Plug-in.

Capítulo 2 El entorno de desarrollo Eclipse

En este capítulo daremos una introducción a la herramienta, reseñando históricamente sus orígenes, y describiremos la arquitectura de la plataforma Eclipse finalizando con una descripción de sus herramientas.

2.1 Introducción

Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar lo que el Proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados (del inglés IDE). Un IDE es un programa compuesto por un conjunto de herramientas útiles para un desarrollador de software. Como elementos básicos, un IDE cuenta con en un editor de código, un compilador/intérprete y un depurador. El IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse). Sin embargo, también se puede usar para otros tipos de aplicaciones cliente.

Gran parte de la programación de Eclipse fue realizada por IBM antes de que se creara el proyecto Eclipse como tal. El antecesor de Eclipse fue VisualAge y se construyó usando Smalltalk en un entorno de desarrollo llamado Envy. Con la aparición de Java en la década de los 90, IBM desarrolló una máquina virtual válida tanto para Smalltalk y Java. La rápida expansión de Java y sus ventajas con miras a una Internet en plena expansión obligaron a IBM a plantearse el abandono de esta máquina virtual dual y la construcción de una nueva plataforma basada en Java desde el principio. El producto final resultante fue Eclipse, que ya había costado unos 40 millones de dólares a IBM en el año 2001.

A finales de 2001 IBM, junto a Borland, crearon la fundación sin ánimo de lucro Eclipse, abriéndose así al mundo de código abierto. A este consorcio se han unido progresivamente importantes empresas del desarrollo de software a nivel mundial: Oracle, Rational Software, Red Hat, SuSe, HP, Serena, Ericsson, Novell, entre otras. Hay dos ausencias significativas: Microsoft y Sun Microsystems. Microsoft ha sido excluida por su posición de monopolio del mercado, y Sun Microsystem cuenta con su propio IDE y principal competencia de Eclipse: NetBeans. De hecho, el nombre de Eclipse fue elegido porque el objetivo era crear un IDE capaz de "eclipsar a Visual Studio" (Microsoft) así como "eclipsar el sol" (Sun Microsystem).

La última versión estable de Eclipse se encuentra disponible para los sistemas operativos Windows, Linux, Solaris, AIX, HP-UX y Mac OSX. Todas las versiones de Eclipse necesitan tener instalado en el sistema una máquina virtual Java (JVM), preferiblemente JRE (Java Runtime Environment) o JDK (Java Developer Kit) de Sun, que a principios de 2007 no son libres (aunque hay un anuncio por parte de Sun de que lo serán).

Eclipse fue liberado originalmente bajo la Common Public License, pero después fue relicenciado bajo la Eclipse Public License. La Free Software Foundation ha dicho que ambas licencias son licencias de software libre, pero son incompatibles con Licencia pública general de GNU (GNU GPL).

El término Eclipse además identifica a la comunidad de software libre para el desarrollo de la plataforma Eclipse. Este trabajo se divide en proyectos que tienen el objetivo de proporcionar una plataforma robusta, escalable y de calidad para el desarrollo de software con el IDE Eclipse. Este trabajo está coordinado por la Fundación Eclipse, que es una organización sin ánimo de lucro creada la promoción y evolución de la plataforma Eclipse dando soporte tanto a la comunidad como al ecosistema Eclipse.

2.2 Arquitectura de la plataforma Eclipse

Eclipse es un entorno flexible y extensible de desarrollo integrado (IDE). Podemos describir este IDE como:

- **Multi-plataforma:** Es compatible con Windows, Linux (motivo y GTK), Solaris, AIX, HPUX y MacOSX.

- **Multi-Lenguaje:** Eclipse está desarrollado utilizando el lenguaje Java, pero soporta la implementación de aplicaciones en Java, C/C++ y COBOL, adicionalmente, se está desarrollado soporte para lenguajes como Python, Perl, PHP, y otros. Los Plug-ins dirigidos a extender la funcionalidad de Eclipse deben estar escritos en Java.

- **Multi-función**: Además de ser utilizado para programación, Eclipse también soporta el modelado y análisis de software, creación Web, y muchas otras funciones.

Los bloques funcionales del IDE Eclipse se ilustran en la Figura 2.1. Cada bloque añadido a la estructura se construye sobre la base de los que hay por debajo de ella.
Es esta naturaleza modular de la plataforma Eclipse la que ha llevado a un crecimiento sin precedentes. Toda la plataforma es de código abierto y libre, por lo que otros proyectos de código abierto y productos comerciales se aprovechan de ello para añadir nuevas funcionalidades a Eclipse. En los siguientes sub-apartados, describimos los bloques funcionales que conforman el IDE.

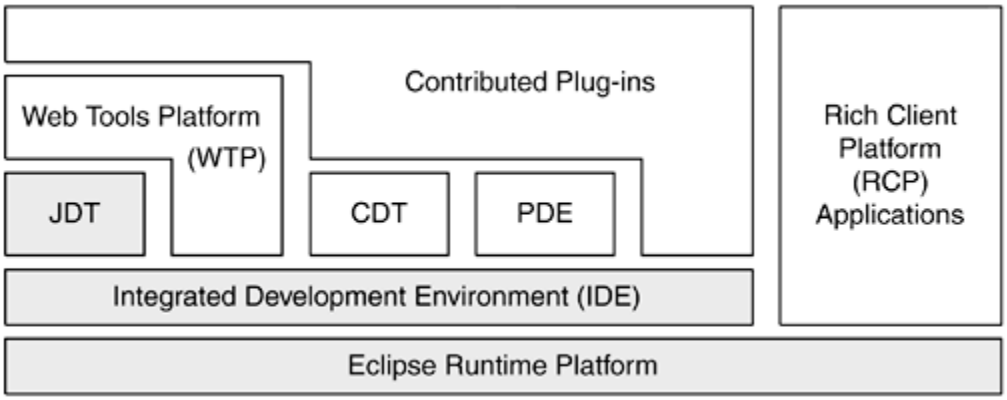

Figura 2.1: IDE Eclipse.

2.2.1 Eclipse Runtime Platform

La plataforma de ejecución proporciona los servicios de nivel más básicos, los principales se detallan a continuación.

- **Registro de Plug-ins.** Carga y maneja un registro de Plug-ins disponibles.
- **Recursos**. Gestión de los archivos y carpetas del sistema operativo, incluyendo la ubicación de los recursos enlazados, de forma independiente de la plataforma.
- **Componentes UI.** Los componentes de la interfaz gráfica de usuario de Eclipse están basados en las librerías SWT y JFace.
- **Actualizaciones**. Las aplicaciones de Eclipse han incorporado soporte para la instalación y la actualización de Plug-ins utilizando URLs que indican donde se localizan los respectivos recursos (pudiendo alojarse en servidores en Internet).
- **Ayuda.** Eclipse dispone de una serie de facilidades comunes compartidas por todos los Plug-ins.

2.2.2 Entorno de Desarrollo Integrado

El IDE Eclipse proporciona una experiencia de usuario común para actividades de muy diverso índole, multi-lenguaje y multi-plataforma. De forma que los Plug-ins Eclipse se construyen sobre los fundamentos de este IDE por lo que no necesitan "reinventar la rueda". Las características más significativas del IDE son resumidas a continuación. La Figura 2.2 muestra cada de los elementos que componen el IDE Eclipse.

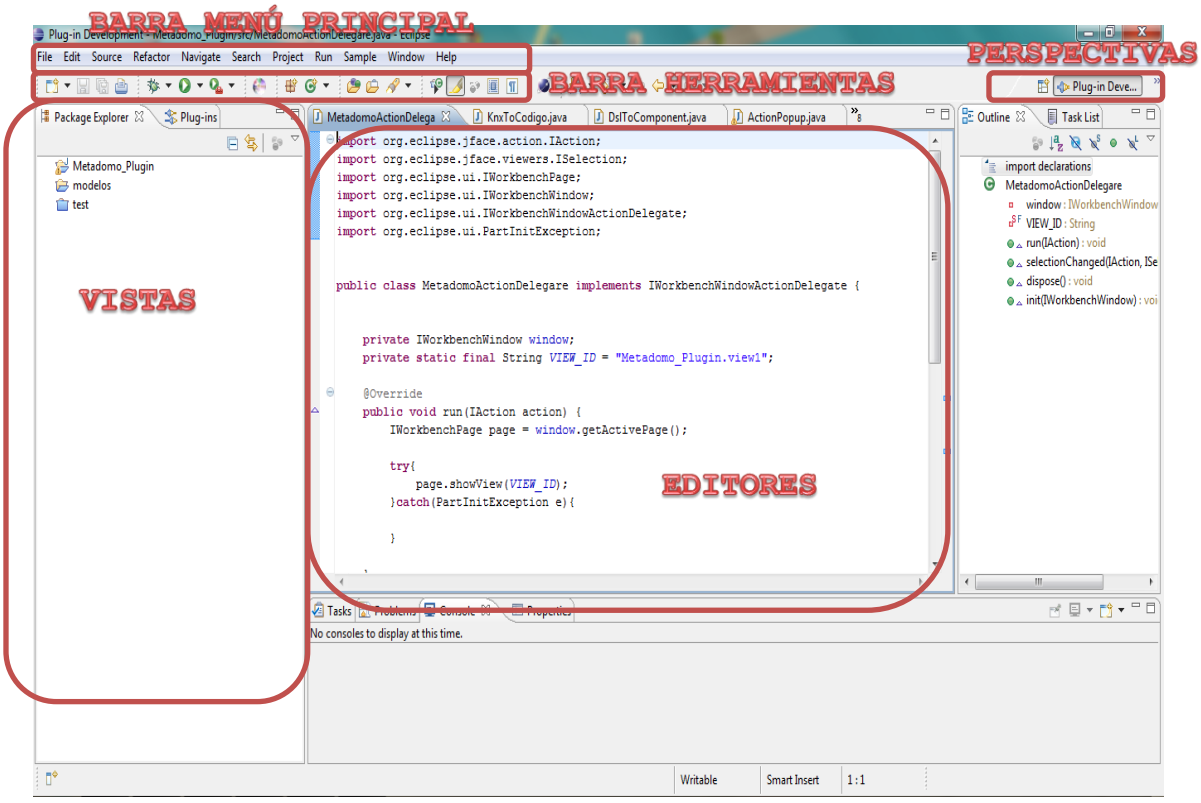

Figura 2.2: IDE Eclipse

- **Vistas.** Pueden tener múltiples y muy diferentes especificaciones unas de otras, tantas como el programador de éstas quiera proporcionales. En la Figura 2.2 vemos la vista Explorador de Paquetes en la cual podemos visualizar los Proyectos Eclipse existentes en el entorno de trabajo así como su contenido.

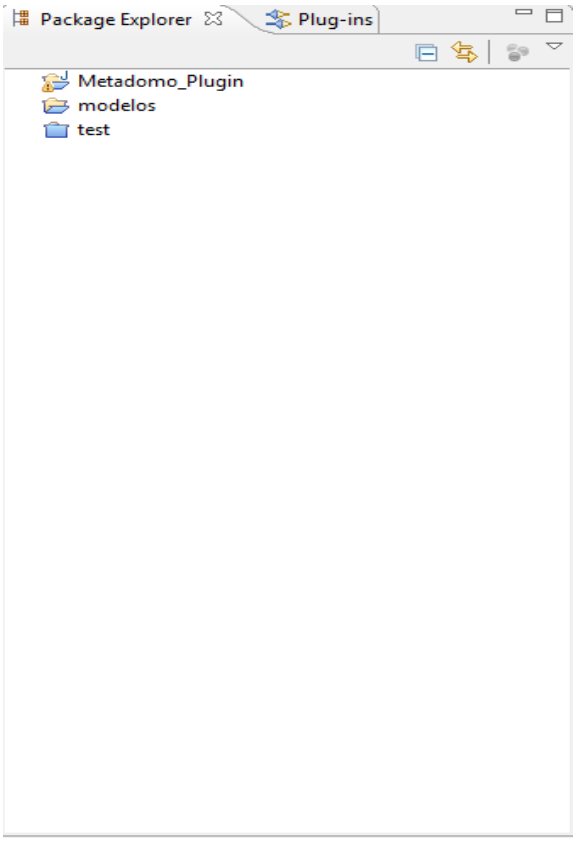

Figura 2.3: Vista del Explorador de paquetes

- **Perspectivas.** Engloba un conjunto de vistas que son usadas para un fin común, es decir, la perspectiva creada debe tener un nombre identificativo relacionado con la función desempeñada por el conjunto de vistas, botones, editores que esta engloba.

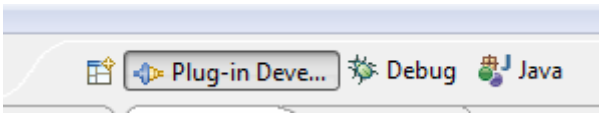

Figura 2.4: Perspectivas

- **Editores**. En la Figura 2.5 vemos ejemplo de editor, un editor es usado en Eclipse para abrir/visualizar un tipo de archivo siempre que Eclipse sea capaz de soportarlo, podemos visualizar archivos de texto, XML, etc.

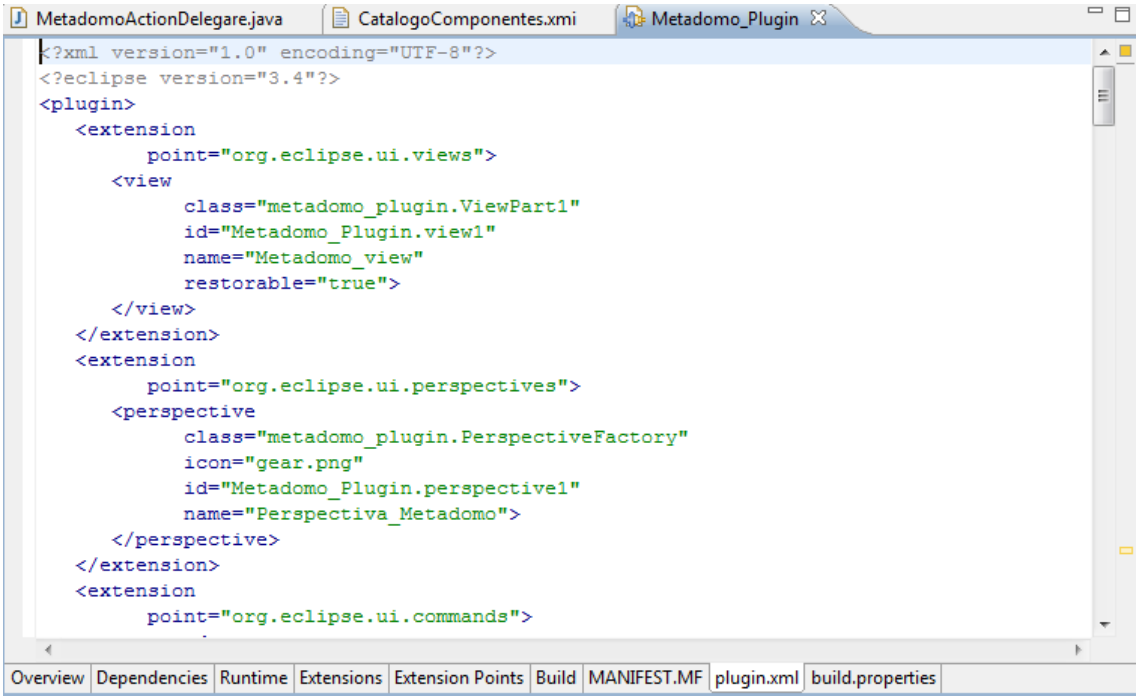

Figura 2.5: Editor Eclipse XML.

- **Barra menú principal.** Permite el acceso a diferentes funciones comunes de Eclipse (por ejemplo, abrir o crear un proyecto) y concretas de los Plug-ins cargados en ese momento (por ejemplo, para compilar de código C/C++).

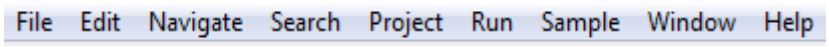

Figura 2.6: Barra de menu principal.

- **Barra de herramientas**. Al igual que el menú principal, permite el acceso a opciones comunes y, por lo tanto, compartidas por todos los Plug-ins Eclipse, y opciones específicas de cada plug-in.

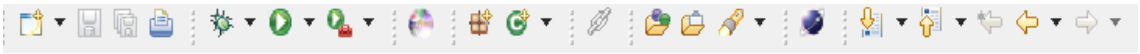

Figura 2.7.: Barra de herramientas de Eclipse

2.2.3 Java Development Tools

Las Java Development Tools (JDT) son los únicos Plug-ins dirigidos a un lenguaje de programación incluidos en el SDK de Eclipse. Herramientas de programación en Eclipse enfocadas a otros lenguajes son desarrolladas por sub-proyectos de Eclipse u otras iniciativas de la comunidad Eclipse para contribuir con nuevos Plug-ins a la plataforma. La perspectiva de desarrollo Java es la que podemos ver en la Figura 2.3. Las capacidades fundamentales proporcionadas son:

- **Editor, Outline, ayuda de contenido, plantillas, y el formato.** Estas características generales del editor se proporcionan para los archivos fuente de Java.

- **Vistas Java.** Varias Vistas se proporcionan para la navegación y la gestión de Proyectos Java. La vista Explorador de Paquetes es la piedra angular de la perspectiva de Java, vista donde podemos ver y navegar a través de los proyectos existentes así como visualizar el contenido de los mismos.

- **Configuración del Proyecto.** Se incluye un amplio soporte para la configuración de rutas de clases Java del Proyecto, las dependencias, las librerías, las opciones del compilador y muchas otras características.

- **Depurador.** Las herramientas Java proporcionan un entorno de depuración. Puede establecer puntos de interrupción, ejecución paso a paso, inspeccionar y establecer los valores de variables y cambiar el código del método durante la depuración.

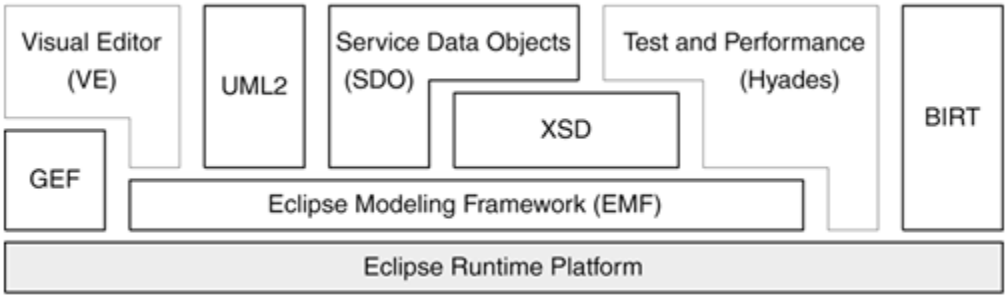

Figura 2.3: Otros Proyectos Eclipse (fuente [8])

2.2.4 Plug-in Development Enviroment

El Plug-in Development Environment (PDE) [11] suministra herramientas que automatizan la creación, manipulación, depuración y despliegue de Plug-ins. El PDE es parte del SDK de Eclipse y no es un instrumento puesto en marcha por separado. En línea con la filosofía general de la plataforma Eclipse, la PDE ofrece una gran variedad de contribuciones de la plataforma (por ejemplo, las vistas, editores, asistentes, lanzadores, etc.) que se mezclan de forma transparente con el resto del framework de Eclipse y ayuda al desarrollador en cada etapa del desarrollo del Plug-in mientras se trabaja dentro del entorno de Eclipse. Eclipse es un programa Java que hace funciones de cargador de Plug-ins. Así, Eclipse es un cargador de Plug-ins cuyo conjunto conforman la herramienta. Podemos definir un Plug-in como un programa Java que extiende la funcionalidad de Eclipse en algún sentido, cada Plug-in Eclipse puede consumir servicios proporcionados por otros Plug-ins o puede extender su funcionalidad para ser usado por otros Plug-ins. Como se ha comentado, Eclipse es una plataforma de código abierto y está diseñada para ser fácilmente extensible por terceras partes. Este mecanismo de extensión puede ser entendido, por ejemplo, si partiendo de la base de un editor textual podemos obtener un editor XML o distintos editores que posean una especialidad respecto de los otros. Podemos diferenciar los siguientes apartados del PDE:

- **Host vs. workbench de ejecución.** El entorno de trabajo Eclipse donde se está operando es donde se implementa el Plug-in, cuando llega la hora de probar el Plug-in se lanza otra instancia de Eclipse creando otro entorno de trabajo en tiempo de ejecución, en éste podremos proceder de la misma manera que si estuviéramos en el entorno de trabajo origen pero con la funcionalidad del Plug-in implementado.

- **Depuración de los Plug-ins.** El depurador de Java permite un control total mientras que se prueban los Plug-ins en una segunda instancia lanzada del workbench de Eclipse.

- **PDE perspectiva.** Una perspectiva especializada incluye vistas y accesos directos a los comandos utilizados con mayor frecuencia durante el desarrollo del Plug-in.

Un concepto imprescindible a la hora de hablar de Plug-ins y de su desarrollo es el concepto de extensión y punto de extensión. Como sabemos Eclipse es extensible y esta extensibilidad es demostrada con el concepto de extensión y punto de extensión, pensemos en punto de extensión como un enchufe que es extendido por el cable que se le conecta. Este concepto no debe sernos desconocido pues si nos paramos un poco a pensar vemos que es un concepto muy parecido al de herencia en programación, en el cual la extensión realiza una especificación/especialización del punto de extensión.

2.2.5 Otras herramientas de Eclipse

- **C/C++ Development Tools,** C/C++ Development Tools (CDT) [9] consiste en un completo y funcional IDE de C y C++ para Eclipse
- **Web Tools Platform,** La misión del proyecto Web Tools Platform (WTP) [10] es proporcionar una plataforma de herramientas genérica, extensible y basada en estándares que se fundamenta en la plataforma Eclipse y otras tecnologías del núcleo de Eclipse. El proyecto ofrece una serie de frameworks y servicios de los cuales los proveedores de software pueden beneficiarse para ofrecer soluciones de desarrollo especializadas para J2EE y Web. Los principales objetivos son

permitir la innovación de productos de forma independiente de las tecnologías propias del proveedor, al tiempo que ofrece soluciones prácticas a las preocupaciones reales de desarrollo.

- **Rich Client Platform,** Rich Client Platform (RCP) es más notable por lo que no tiene que por lo que tiene. Aunque la plataforma Eclipse está diseñada para servir como una plataforma abierta de herramientas, sus componentes puedan ser utilizados para construir casi cualquier aplicación escritorio. El conjunto mínimo de Plug-ins necesarios para construir una aplicación cliente enriquecido se conoce colectivamente como RCP. Estas aplicaciones todavía se basan en un Plug-in, y la interfaz de usuario se construye utilizando las mismas herramientas y puntos de extensión. El diseño y el funcionamiento del workbench se encuentran bajo control del Plug-in de los desarrolladores. Al contribuir al IDE, los Plug-ins están construidos sobre el SDK de Eclipse.

2.3 Otros Proyectos Eclipse

Anteriormente hemos revisado la plataforma de ejecución de Eclipse. En la comunidad Eclipse se desarrollan proyectos, que al igual que el que nos ocupa, se fundamentan en la arquitectura Eclipse para contribuir en el desarrollo de entornos de trabajo y herramientas. Algunos de estos Proyectos son maduros y de uso generalizado, mientras que otros apenas están comenzando. A su vez, estos componentes se pueden utilizar para crear nuevos Plug-ins, y muchos también se pueden ejecutar fuera del entorno de trabajo de Eclipse. Los componentes de cada Proyecto se empaquetan como un conjunto de Plug-ins que se agregan al entorno de trabajo.
Las diferentes capas en la Figura 2.4 muestran las dependencias de un componente al construirse sobre otro. En particular, muchos de los componentes se basan en las capacidades del Eclipse Modeling Framework (EMF). Describimos algunos de los componentes más importantes (mostrados en la Figura 2.4):

- **Eclipse Modeling Framework (EMF).** Constituye el soporte fundamental de lasherramientas Eclipse para el Desarrollo de Software Dirigido por Modelos (DSDM). Permite la creación y manejo de modelos y meta-modelos.

- **Graphical Editor Framework (GEF).** Permite a los desarrolladores crear un editor gráfico de modelos en el contexto de DSDM.

- **Visual Editor (VE).** Un marco para la creación de constructores de interfaz gráfica de usuario para Eclipse, incluye implementaciones de referencia a constructores GUI como Swing / JFC y SWT. Pretende ser de utilidad para la creación de constructores de interfaz gráfica de usuario para otros lenguajes como C/C++ y conjuntos alternativos de widgets, incluyendo aquellos que no son compatibles con Java.

- **Unified Modeling Language 2.0 (UML2).** Proporciona una implementación del metamodelo UML 2.0 para apoyar el desarrollo de herramientas de modelado, un esquema XML común para facilitar el intercambio de modelos semánticos, casos de prueba como medio de validación de la especificación, y reglas de validación como un medio para definir y hacer cumplir los niveles de cumplimiento.

- **XML Schema Infoset (XSD).** Una librería de referencia para su uso con cualquier código que analiza, crea o modifica los esquemas XML.

- **Service Data Objects (SDO).** Un marco que simplifica y unifica el desarrollo de datos de aplicaciones en una arquitectura orientada a servicios (SOA). Es compatible con XML e integra e incorpora los patrones de J2EE y las mejores prácticas.

- **Eclipse Test & Performance.** Marcos y servicios para las herramientas de prueba y el rendimiento que se utilizan en todo el ciclo de desarrollo, tales como pruebas, la localización, perfiles, afinación, registro, monitoreo, análisis, autonómicos, y la administración.

- **Business Intelligence and Reporting Tools (BIRT).** Infraestructura y herramientas para diseñar, implementar, generar y ver informes en una organización

Capitulo 3 Desarrollo de sistemas domóticos dirigido por modelo

3.1 Motivación

La evolución de la Ingeniería del *Software* se ha visto marcada por varios hitos a lo largo de su corta historia. Uno de los más importantes fue el giro, a principios de los años 80, que supuso el paso de la programación estructurada (lo importante eran las funciones) a la orientación a objetos (lo importante son los objetos como unión encapsulada de datos más funciones).

Pero en sus más de veinticinco años de historia la orientación a objetos ha alcanzado su madurez, dado que los requisitos impuestos por los nuevos sistemas están superando las posibilidades de este enfoque. Las plataformas evolucionan a gran velocidad, incrementándose el volumen de datos y de código, así como los aspectos funcionales y no funcionales de las aplicaciones. Por otra parte, es cada vez mayor la heterogeneidad en elementos clave como los lenguajes y paradigmas utilizados, los protocolos de acceso y de manipulación de datos, los sistemas operativos y las plataformas *middleware* empleadas. Además, aparecen nuevas tecnologías a una velocidad creciente y las previsiones indican que este crecimiento se va a mantener o incluso incrementar. Y para empeorar aún más las cosas, las tecnologías existentes no desaparecen, sino que se ocultan en capas de software más profundas.

Todos estos factores han conducido a la conclusión de que **la orientación a objeto es insuficiente,** ya que presenta deficiencias importantes en los siguientes aspectos:

- El enfoque orientado a objetos no afronta adecuadamente los requerimientos de computación presentes y futuros.

- Los lenguajes orientados a objetos han perdido la simplicidad (o "pureza") que los hacía especiales, y que era la fuente de su expresividad y potencial de desarrollo.
- Conceptos tan importantes como la encapsulación, que se introdujeron para reducir la influencia de los programadores en el desarrollo de software, fallan cuando se necesita describir propiedades globales o se requiere que el *software* evolucione o se someta a cambios importantes.
- Una de las expectativas que llevó al planteamiento de la orientación a objetos fue la reutilización.

De la promesa inicial de simplicidad (véase la Figura 3-1) se ha pasado a una complejidad creciente. La tecnología de objetos planteaba en sus inicios tres conceptos básicos: objetos, clases y métodos. Pero en su evolución a la tecnología de componentes, los conceptos manejados para satisfacer las necesidades de las aplicaciones software crecen de manera exponencial. Además, no está clara la correspondencia entre los conceptos de la orientación a objetos desde la definición de los requisitos hasta la implementación.

La tecnología de objetos ha satisfecho algunas de las expectativas que motivaron su adopción, pero ha fallado en la consecución de muchas otras. Quizás una de las causas se puede encontrar en que se ha dejado de buscar la generalidad mediante la unificación.

A principios de esta década emerge un nuevo paradigma: la ingeniería dirigida por modelos (MDE: Model Driven Engineering) con el propósito de suponer el paso definitivo hacia la industrialización del software, o al menos proporcionar mejoras significativas en la productividad y calidad. MDE se encuentra en la actualidad en la etapa inicial de generación de expectativas que prometen solucionar todas las deficiencias detectadas en la madurez de la orientación a objeto.

El enfoque MDE se basa en la utilización de modelos como elemento principal en todo el ciclo de desarrollo del software. El uso de modelos aumenta el nivel de abstracción, ya que permite centrarse en los conceptos, dejando en un segundo plano los detalles de implementación. Mediante la utilización de un conjunto de herramientas, los modelos se transforman a otros modelos y finalmente a código ejecutable, que se genera de manera automática o semiautomática.

Tal como se muestra en la Figura 3.1, mientras que la orientación a objetos ("todo es un objeto") se basa en relaciones de *instanciación* y herencia, MDE descansa sobre el principio "todo es un modelo", donde las relaciones básicas son de *representación* (un modelo *representa* un sistema) y de *conformidad* (un modelo es *conforme* a su meta-modelo).

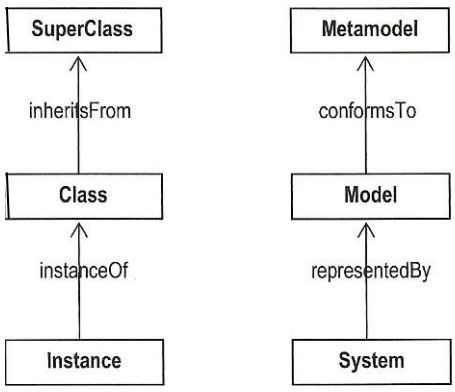

Figura 3.1: Nociones básicas en las tecnologías de objetos y modelos.

3.2 Desarrollo de Software Dirigido por Modelos (MDE)

3.2.1 Introducción

El término "desarrollo o ingeniería dirigida por modelos" (MDE) hace referencia a la técnica que hace uso, de forma sistemática y reiterada de modelos como elementos básicos a lo largo de todo el proceso de desarrollo. MDE utiliza modelos como elementos de entrada y salida en todo el proceso. El empleo de modelos, que son una representación simplificada de la realidad, permite aumentar el nivel de abstracción en la realización del diseño y la reutilización de los mismos. Así, los artefactos generados son independientes del lenguaje y paradigma de programación que se utilizará en las etapas finales para la generación de código. Además, las ideas se expresan de forma explícita en términos propios del dominio del problema, y no diluidas en el código del programa.

Según, MDE persigue, entre otros, la separación del modelo de negocio de la plataforma de implementación, la identificación, expresión precisa, separación y combinación de aspectos específicos del sistema en desarrollo con lenguajes específicos de dominio, el establecimiento de relaciones precisas entre los diferentes lenguajes en una arquitectura global, y en particular, la posibilidad de expresar transformaciones entre ellos.

3.2.2 Beneficios Esperados

Los beneficios que se esperan obtener mediante esta nueva metodología de diseño basada en modelos son:

- Mayor velocidad en la implementación.
- Mejorar la calidad del código.
- Facilitar el mantenimiento.
- Desarrollo ágil
- Incrementar la reusabilidad.
- Mejorar la flexibilidad y extensibilidad.

Una de las premisas básicas de MDE es la obtención (semi) automática de código a través de las transformaciones de modelos. Así, la cantidad de código que debe

escribirse manualmente para una aplicación se reduce drásticamente, y el desarrollador debe centrarse únicamente en el código personalizado, que no suele superar el 10% del total (en las aplicaciones, en torno al 50% del código es genérico y el 40% semi-genérico). Esto repercute directamente en un aumento de la productividad.

La fiabilidad del código es otro de los atributos que mejora con el empleo de MDE. El código generado automáticamente requiere menos depuración y verificación. También se produce una mejora en la consistencia y la calidad, ya que las transformaciones de modelos a código se basan en patrones de diseño que permiten generar un código estructurado conforme a una arquitectura.

En el software el mantenimiento constituye una parte muy importante del coste total del desarrollo. Las aplicaciones deben ser capaces de responder y adecuarse a los cambios del modelo de negocio y de la tecnología. Cambios como añadir atributos a una clase personalizada, integrar recursos externos, reprogramar modificaciones en los casos de uso, etc, suponen una gran cantidad de trabajo que se puede reducir considerablemente mediante la generación de código dirigida por modelos, ya que se reduce la cantidad de código que ha de escribirse de forma manual.

Las características intrínsecas a MDE permiten además soportar el Desarrollo Ágil (Agile Development), que se centra en dar una respuesta flexible a cambios en los requisitos, obtener software ejecutable y utilizable desde las primeras etapas y en conseguir una comunicación más estrecha entre los desarrolladores de software y el cliente. Así, con MDE se puede obtener software bien estructurado y operativo que se construirá de forma iterativa durante todo el desarrollo, y se agilizará la propagación de cambios en el código.

Por otra parte, la reusabilidad es una de las promesas de esta metodología. Los patrones de diseño pueden evolucionar en los niveles del modelo para definir como estructurar las clases del dominio y los componentes para las tareas estándar. Los modelos del dominio se pueden almacenar en librerías para ser reutilizados en nuevas aplicaciones. Además, el modelo de negocio se mantiene separado en todo momento de la arquitectura de la tecnología utilizada en la implementación.

3.2.3 Transformaciones de Modelos

MDE hace uso, de forma sistemática y reiterada, de modelos**,** entendiendo por modelo aquella representación simplificada de la realidad que muestra sólo aquellos aspectos que interesan, escondiendo los que no son de interés. En MDE un modelo se define conforme a un metamodelo que define la sintaxis abstracta de un lenguaje de modelado y establece los conceptos y relaciones entre ellos, incluyendo además las reglas que determinan cuándo un modelo está bien formado. Gracias a la utilización de los modelos como artefactos principales se consigue, entre otros, la separación del modelo de negocio de la plataforma de implementación, la identificación, expresión precisa, separación y combinación de aspectos específicos del sistema en desarrollo con lenguajes específicos de dominio (DSLs), el establecimiento de relaciones precisas entre

los diferentes lenguajes en una arquitectura global y, en particular, la posibilidad de expresar transformaciones entre ellos.

En MDE las transformaciones de modelos son esenciales para dar soporte a todo el proceso de desarrollo. Las transformaciones permiten realizar conversiones de un modelo origen a otro destino entre diferentes metamodelos, manteniéndolos sincronizados. En el ámbito del desarrollo software esto permite, en una última instancia, convertir los modelos que utilizan los desarrolladores en modelos de la plataforma de implementación en código ejecutable.

Pueden encontrar numerosas definiciones de transformación, entre ellas, las más clarificadoras son:

- Una transformación es la aplicación de una función de transformación para convertir un modelo en otro. Una función de transformación es un conjunto de reglas que definen cada uno de los aspectos de funcionamiento de una función de transformación.
- Una transformación es la generación automática de un modelo a partir de otro modelo fuente, de acuerdo con una serie de reglas. Una definición de transformación es un conjunto de reglas de transformación que juntas describen cómo se transforma un modelo descrito en el lenguaje origen a un modelo descrito en el lenguaje destino. Una regla de transformación es una descripción de cómo se transforman una o más construcciones del lenguaje origen en una o más construcciones en el lenguaje destino.
- Una transformación genera un modelo destino a partir de un modelo fuente. Una vista es un tipo de transformación restringida que impone la restricción de que el modelo obtenido no puede ser modificado independientemente del modelo fuente.

Las características más importantes que deben tener las transformaciones de modelos son las siguientes:

- **Automatización**. Es necesario definir mecanismos para programar transformaciones automáticas a partir de una serie de modelos origen, pero también debe contemplarse la posibilidad de que existan transformaciones que requieran cierta intervención manual por parte del usuario.
- **Complejidad.** Las técnicas utilizadas dependen del nivel de complejidad de la transformación.

Preservación del significado. Las transformaciones deben preservar ciertas características del modelo origen, Así, algunas deben preservar la estructura y otras el comportamiento.

Capítulo 4 HABITATION

4.1. Domótica

En los últimos años, las tecnologías de la información y las comunicaciones se están integrando en el hogar y la vida cotidiana a gran velocidad. Este proceso ha dado lugar a un nuevo tipo de sistemas reactivos: los sistemas domóticos.

Para la aparición de esta nueva tecnología han sido fundamentalmente varios factores: por un lado la disponibilidad del elemento base para el desarrollo de la informática en los últimos tiempos (el microprocesador) y por otro, la convergencia entre la informática y las telecomunicaciones, junto con la necesidad cada vez mayor de información a todos los niveles.

Asimismo, en su evolución ha tenido una gran repercusión la definición paralela de arquitecturas de comunicación de datos en el ámbito de la automatización industrial: los conocidos buses de campo, con los que los sistemas domóticos presentan grandes similitudes. De hecho, es muy difícil establecer una separación clara entre ambos campos, ya que la literatura existente incluye a muchos de los protocolos para redes de control domótico dentro de las redes de automatización industriales.

- La definición de Vivienda Domótica o Inteligente presenta múltiples versiones y matices, y son diversos los términos utilizados en distinto idioma: **casa inteligente** *(smart home),* **automatización de viviendas** *(home automation),* **domótica** *(domothique),* **sistemas domóticos** *(home systems),* etc. Hasta hoy se conocen múltiples definiciones de domótica, de las que cabe destacar las siguientes:

- La nueva tecnología de los automatismos de maniobra, gestión y control de los diversos aparatos de una vivienda, que permiten aumentar el confort del usuario, su seguridad y el ahorro del consumo energético.
- La informática aplicada a la vivienda. Agrupa el conjunto de sistemas de seguridad y de la regulación de las tareas domésticas destinadas a facilitar la vida cotidiana automatizando sus operaciones y funciones.
- Conjunto de servicios de la vivienda garantizando por sistemas que realizan varias funciones, los cuales pueden estar conectados entre sí y a redes interiores y exteriores de comunicación. Gracias a ello se obtiene un notable ahorro de energía, una gestión eficaz técnica de la vivienda, una buena comunicación con el exterior y un alto nivel de seguridad.

Pero quizás una de las más completas es: *"Sistemas de Automatización, Gestión de la Energía y Seguridad para Viviendas y Edificios: Son aquellos sistemas centralizados o descentralizados, capaces de recoger información proveniente de unas entradas (sensores o mandos), procesarlas y emitir órdenes a unos actuadores o salidas con el objeto de conseguir confort, gestión de la energía o la protección de personas, animales y bienes. Estos sistemas pueden tener la posibilidad de acceso a redes exteriores de*

comunicación, información o servicios, como por ejemplo, red telefónica conmutada, servicios INTERNET, etc".

Existe aún hoy cierta polémica en cuanto a la idoneidad del término domótica ya que el objeto de esta disciplina no es únicamente la vivienda sino cualquier tipo de edificación. Por ello, se han creado diversos términos para distinguir el alcance de las domótica según el sector de aplicación.

- Domótica, para el sector doméstico (aunque hoy día se ha generalizado además para el sector edificios).
- Inmótica, para el sector terciario (automatización de edificios como hoteles, hospitales, oficinas, etc).
- Urbótica, para las ciudades. Control de la iluminación pública, gestión de semáforos, telecomunicaciones, medios de pago, etc.

En la actualidad la arquitectura de un sistema domótico, al estar prácticamente basada en una red más o menos compleja de comunicaciones, nos lleva a tratarlo como una **Red Domótica,** a la cual conectamos dispositivos de lo más variado y a los que podemos acceder desde cualquier punto de la red. Con esta idea, se puede definir una Red Domótica como una **instalación inteligente capaz de interactuar con el medio que le rodea.**

4.1.1 Tipos de Aplicaciones de la Domótica

Las aplicaciones desarrolladas en domótica ofrecen la posibilidad de gestionar un sistema inteligente mediante la modificación local o remota de los parámetros de la instalación. Para ello ofrecen una serie de servicios realizados por un conjunto de automatismos o dispositivos con cierto grado de inteligencia (basados en microcontroladores) dirigidos a la consecución de cuatro objetivos básicos (véase Figura 4.1):

- **Gestión Energética y Recursos**: regulación de la climatización, gestión de los consumos de cada electrodoméstico y de la potencia controlada, control del suministro de recursos como electricidad, gas y agua, etc.
- **Seguridad:** custodia y vigilancia frente a la intrusión, la inundación, el fuego, los escapes de gas, etc.
- **Comunicaciones:** comunicación interna del sistema, telecontrol y telemetría, SMS, señales acústicas, etc.
- **Confort:** automatización de tareas repetitivas, programaciones horarias, escenarios luminosos, riego automático, etc.

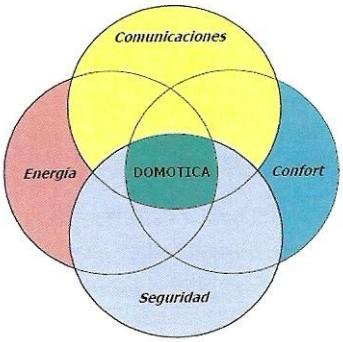

Figura 4.1: Aplicaciones de la domótica.

Las fronteras entre estos cuatro objetivos son difusas y en muchos casos un mismo dispositivo favorece el logro de varios objetivos a la vez, lo cual, por otra parte, economiza la instalación. Es precisamente esta filosofía de **integración** la que da realmente significado a la domótica, ya que de otro modo estaríamos hablando de mera automatización de una vivienda o edificio, ya que integra el control de una serie de sistemas y el uso que se hace de ellos.

4.1.2 Problemática en Domótica

Uno de los principales problemas en el desarrollo de sistemas domóticos es el hecho de que no hay un estándar para implementar estas aplicaciones. Existen varios estándares y protocolos adoptados por las empresas que lideran el mercado y es improbable que se establezca una única tecnología dominante en el campo de la domótica a corto plazo. Además, cada uno de estos estándares proporciona su propio software con el que crear las aplicaciones domóticas y programar los dispositivos. Por lo tanto se ha ce imprescindible seleccionar una tecnología en particular (plataforma específica) en la etapa de diseño inicial, puesto que las herramientas y dispositivos finales dependen de esta elección. Estos hechos hacen que el desarrollo de aplicaciones domóticas sea totalmente dependiente de la plataforma, siendo muy complicado incrementar el nivel de abstracción y trabajar con conceptos del dominio domótico en lugar de trabajar con elementos de la tecnología.

Debido a esta problematica se propone HAbitATION (development of Home Automation Aplications using a mOdel driveN aproach), una tecnología que utiliza el enfoque MDE para dar soporte completo al ciclo de vida del desarrollo de sistemas domóticos, desde la captura de requisitos hasta la implementación en una plataforma especifica

4.2 Metodología Propuesta

A continuación mostramos un esquema con los pasos a seguir para construir una aplicación por un usuario.

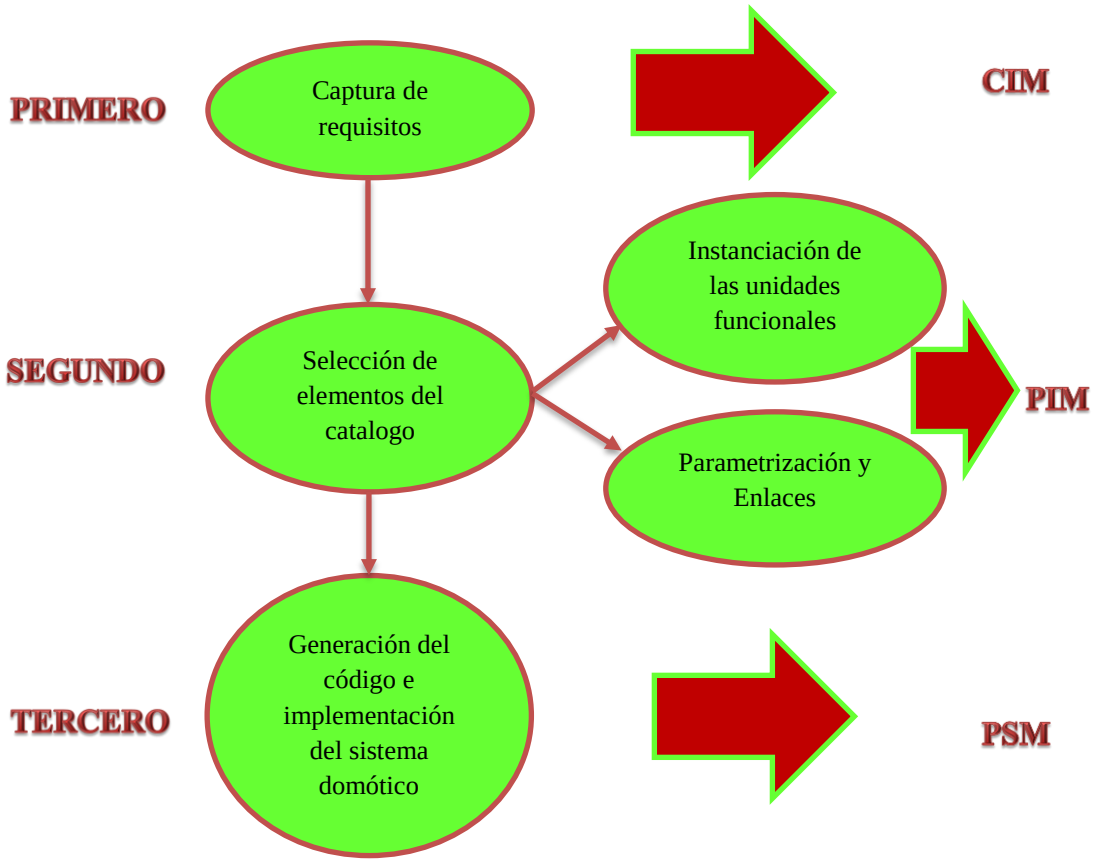

Figura 4.2: Esquema de los pasos a seguir por esta metodología.

La metodología integral de desarrollo de sistemas domóticos que se ha utilizado sigue el enfoque MDA, se pueden distinguir tres grandes fases metodológicas a tener en cuenta: (1) fase de análisis y definición de requisitos (Nivel CIM), (2) fase de diseño (Nivel PIM) y (3) fase de implementación (Nivel PSM).

A continuación se presentan cada una de estas fases y niveles con un mayor grado de detalle, comenzando con la explicación del DSL dado que es en él donde se definen los conceptos del dominio que han influenciado en el resto de la metodología.

4.2.1 Lenguajes Específicos De Dominio (DSL)

Dada la problemática que presenta el desarrollo tradicional de software para sistemas domóticos, se hace necesaria una metodología que mejore tanto la calidad como la productividad en el proceso de desarrollo. Dicha metodología debe recoger requisitos del sistema domótico. Independientemente de la plataforma o estándar que se vaya a emplear en la implementación, facilitando así la portabilidad entre sistemas. Además, las herramientas desarrolladas deben ser intuitivas y de fácil manejo, permitir la reutilización de los elementos del software y facilitar la extensibilidad de las aplicaciones.

La definición de este lenguaje tiene por objetivo ayudar a los diseñadores a describir los sistemas domóticos utilizando únicamente conceptos del dominio. En este sentido, el DSL que se presenta en este trabajo facilita la captura de requisitos propios de un sistema domótico de forma visual e intuitiva. De esta forma, los usuarios ven facilitada la posibilidad de expresar y entender su conocimiento y experiencia en el dominio. Por esta razón la primera premisa es la de disponer de una riqueza semántica importante para la visualización del conocimiento, pero a la vez concisa y común a las distintas plataformas. Antes de entrar en detalles es necesario exponer los principales conceptos con los que se trabaja en el dominio domótico y que deben tenerse en cuenta a la hora de crear el DSL.

Por esta razón es conveniente distinguir dos vistas del DSL, una para el desarrollo de aplicaciones con un desarrollador de aplicaciones como usuario y una segunda vista - para desarrollar y realizar posibles actualizaciones del catálogo.

4.2.1.1. Vista Catalogo DSL

En cualquier sistema domótico existe una serie de elementos (que denominamos "*Unidades Funcionales*") que aparecen en todas las tecnologías y estándares domóticos. Se diferencian en la arquitectura, protocolos utilizados o módulos disponibles, pero son iguales en cuanto a funcionalidad. Con el fin de promover la reutilización de estas unidades funcionales y evitar tener que definir múltiples veces la misma unidad para cada aplicación, se ha optado por utilizar un *Catálogo* de unidades funcionales reutilizable, de manera que una vez definido dicho catálogo éste se pueda utilizar en cualquier aplicación y sólo sea necesario obtener ejemplares de dicho catálogo. Estas unidades funcionales a su vez disponen de unos S*ervicios* gracias a los cuales las unidades podrán interaccionar con otras unidades. Muchos de estos servicios se repiten entre las unidades funcionales, de manera que se ha creado un *Catálogo de Servicios* con unas *Definiciones de Servicios* que puedan ser reutilizadas en cualquier unidad funcional.

4.2.1.2 Vista aplicación DSL

Esta vista del DSL será utilizada por el desarrollador**,** encargado de definir/diseñar nuevas aplicaciones y sin tener que ser un experto en el dominio domótico. En este sentido, y gracias a la existencia del catálogo, la especificación de una aplicación domótica se realiza mediante:

• La instanciación de unidades funcionales (*FUnitInstance*) que vienen definidas en el catálogo. Estas instancias de unidades funcionales deben ser configuradas añadiendo los valores necesarios a sus parámetros.

• Los enlaces (*Link*) entre unidades funcionales. Mediante enlaces se puede indicar la forma en la que las unidades funcionales van a interactuar con el resto del sistema. Al realizar un enlace se indican los servicios que se interconectan. Estos enlaces pueden comportarse como enlaces de tipo *canal* en el caso de que una de las unidades funcionales involucradas sea pasiva de manera que se modela una conexión hardware o bien como un enlace normal.

• Escenas (*Scene*). Se puede configurar la ejecución de varios servicios de unidades funcionales de forma secuencial con una única acción.

En Habitation se utiliza la propuesta MDA del enfoque MDE, que organiza el desarrollo de software en tres capas, y a su vez especifica tres niveles de abstracción que proporciona tres puntos de vista diferentes: nivel de Modelo Independiente de Computación (CIM) con modelos de mayor abstracción; nivel Modelo Independiente de Plataforma (PIM) y nivel Modelo Específico de Plataforma (PSM). De esta manera la división en distintos niveles de abstracción permite abordar el desarrollo de software desde distintos puntos de vista independientes, minimizando la complejidad del problema inicial.

4.2.2 Nivel de requisitos (CIM)

La captura de requisitos se sitúa en el nivel CIM definido por MDA y se corresponde con la primera fase de la metodología propuesta. Aquí se recogen los requisitos solicitados por el usuario y que el sistema domótico debe satisfacer.

La gestión de requisitos se ha integrado dentro de la metodología siguiendo el enfoque MDE, lo que ha permitido una reutilización sistemática de los requisitos y su posible trazabilidad.

Por otro lado, en el desarrollo de sistemas domóticos es habitual que los requisitos básicos estén repetidos o que sean muy parecidos entre distintos proyectos. Por esta razón se ha credo un catálogo que permita reutilizar los requisitos entre los distintos desarrollos de sistemas domóticos, quedando por tanto el metamodelo como se muestra en la Figura 4.2

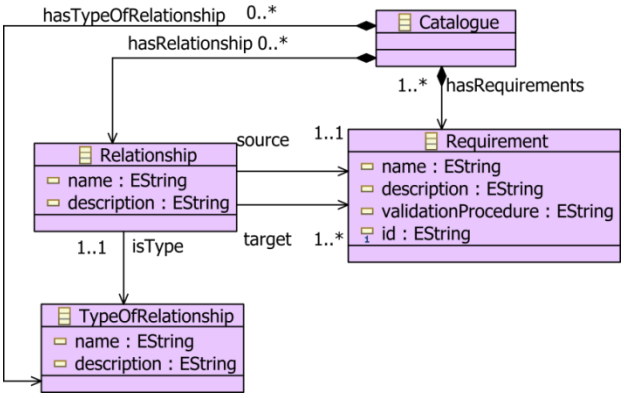

Figura 4.3: Metamodelo de requisitos.

Este metamodelo tiene una versatilidad suficiente, con una estructura simple que podría ser aplicado también a otros dominios. El elemento raíz es *Catalogue*, donde se incluyen el conjunto de requisitos. Cada uno de estos requisitos incluye un nombre, una descripción y un procedimiento de validación para el usuario con el fin de validar si el sistema cumple la funcionalidad esperada.

Los requisitos normalmente se encuentran fuertemente relacionados entre ellos, por lo que se ha incorporado la EClass *Relationship* que permite relacionar varios requisitos entre sí. El tipo de relación entre requisitos se identifica con el elemento *TypeOfRelationship*.

La Figura 4.3 muestra un ejemplo del catálogo de requisitos basados en la estructura de este metamodelo en el entorno Eclipse.

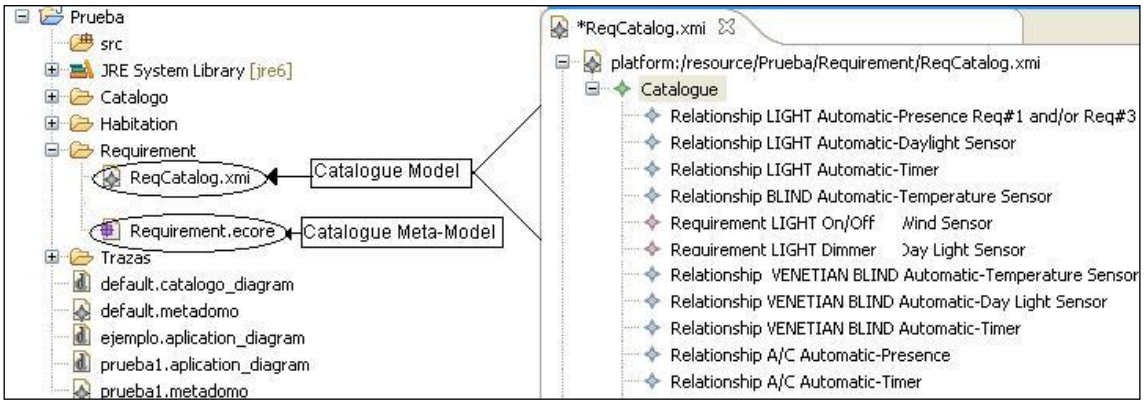

Figura 4.4: metamodelo de requisitos y modelo de requisitos.

Como primera aproximación, se ha propuesto un conjunto básico de requisitos para los sistemas domóticos (la Figura 4.4 muestra un extracto). Estos requisitos están divididos en cuatro categorías básicas: confort de iluminación (iluminación), el confort motorizado (para dispositivos movidos por motores, como puertas, persianas, proyectores, etc.), el confort de climatización, y la seguridad. Requisitos de comunicación sólo se han incluido dentro de la seguridad para obtener información acerca de los tipos de alarma enviados a móviles por SMS.

REQUIREMENTS CATALOGUE					
CONFORT- LIGHTING					
ID	NAME	DESCRIPTION	VALIDATION PROCEDURE	TARGETS	TYPE OF RELATIONSHIP
Req.1	ON/OFF	Control of Light through a push-button or any other domotic mechanism.	Apply push button PB to verify the correct functioning of the light.	None	-------
Req.2	Dimmer	Light intensity regulation through the mechanism, short push for ON/OFF, longer push for Dimmer Up or Down.	Push the mechanism (ej: push-button) short for ON/OFF and longer for regulation in order to verify correct functioning.	None	--------
Req.17	Automatic-Day Light Sensor	Automatic Light Swithching through a daylight sensor.	Wait for daylight sensor to detect the diminishing light outside, and see if the light gets turned on.	Req1, Req2	Requires at least one of them.
Req.18	Automatic-Timer	Automatic Light Swithching through time programation (clock).	Program the timer and verify if the light turns on at the programmed time.	Req1, Req2	Requires at least one of them.
Req.3	Automatic-Presence	Automatic Light Swithching through a presence detector.	Walk near the presence detector to verify the correct functioning of the light.	Req1 , Req2	Requires at least one of them.
Req Comfort Illuminat.	Automatic-Presence&Timer	Automatic Light Swithching through a presence detector when timer has been programmed to funcion	Walk near the presence detector when the time has been programmed for funciongin to verify the correct functioning of the light.	Req1, Req2, Req3, Req.18	Requires all of them.

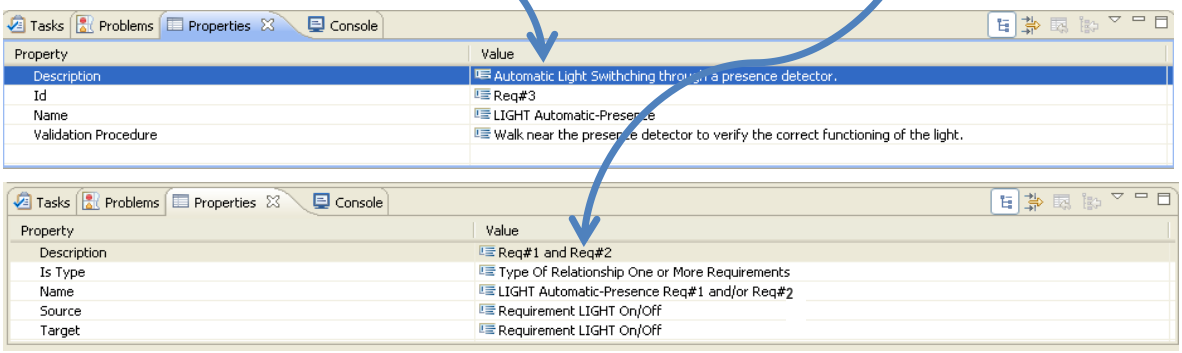

Figura 4.5: Catálogo de requisitos genérico con un pantallazo de los requisitos en la herramienta Eclipse.

Cada uno de los requisitos tiene asociado un fragmento de modelo DSL donde se representa en notación gráfica el requisito. Esta asociación se ha realizado de forma manual y la selección actual de los requisitos del catálogo también se realiza de forma manual, aunque se está desarrollando un plugin en java que permitirá en un futuro poder realizar la selección de una forma más fácil. De esta manera, a la hora de desarrollar una aplicación sólo se debe seleccionar los requisitos necesarios del catálogo y automáticamente se genera un modelo inicial de aplicación en notación DSL (véase Figura 4.5). Este modelo puede ser modificado posteriormente para refinar el modelado de la aplicación. Cada una de las correspondencias entre requisito y elementos del DSL se han trazado con el fin de poder realizar un seguimiento de estos requisitos a lo largo del ciclo de vida de desarrollo del software para el sistema domótico y poder gestionar un control de cambios exhaustivo.

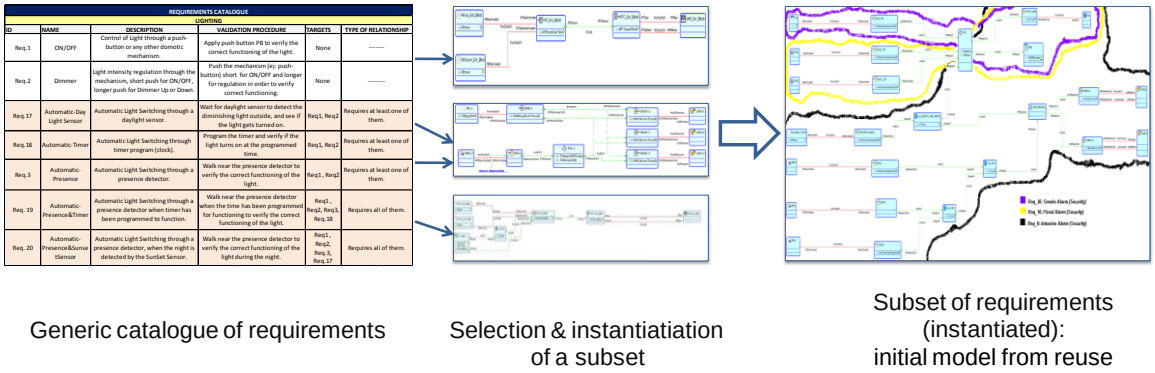

Figura 4.6: Esquema general del enfoque propuesto en la fase de requisitos.

4.2.3 Componentes (PIM)

Para el nivel PIM del *framework*, se ha decidido utilizar un modelo de componentes que sirva de nivel intermedio para el desarrollo y que haga de punto de unión con otros sistemas reactivos (por ejemplo el robótico, redes de sensores, visión artificial, …)

En esta fase se deben tomar decisiones muy importantes en cuanto a metodología, el principal problema que determina la estrategia a seguir en esta fase viene marcado por la fuerte dependencia existente en cada una de las plataformas.

Como se ha dicho en numerosas ocasiones, para asegurar la independencia de la plataforma en las primeras fases de desarrollo, se optó por utilizar unidades funcionales en lugar de dispositivos. El problema de trabajar con unidades funcionales surge en el salto de un modelo totalmente independiente de plataforma a un modelo de plataforma donde se modelan dispositivos reales.

La solución que se ha dado a este problema es disponer de un catálogo de dispositivos domóticos predefinidos para cada una de las tecnologías existentes (por lo menos de las más importantes). En este catálogo se indicará para cada dispositivo que funcionalidades lo compone. Por tanto existirá un modelo de componentes a nivel PIM y un segundo modelo de componentes, fruto de un refinamiento del primero, que se situará a nivel PSM.

Por este motivo se hace indispensable tener catalogados los distintos dispositivos que se pueden encontrar en las diferentes tecnologías teniendo en cuenta las funcionalidades que ofrecen. Y dado que las unidades funcionales son representadas en el modelo de componentes como un componente simple, la forma más sencilla de realizar esta operación de catalogación es mediante un refinamiento del modelo de componentes.

4.2.3.1 Metamodelo de Componentes

Una arquitectura de componentes se caracteriza por tener como elementos arquitectónicos componentes y conectores. Para la definición de estos componentes y conectores se han definido una serie de *Eclass* y se han tenido en cuenta las posibles relaciones de inclusión o transformación que serán necesarias. En la Figura 4.6 se puede observar dicho metamodelo en formato *EDiagram*.

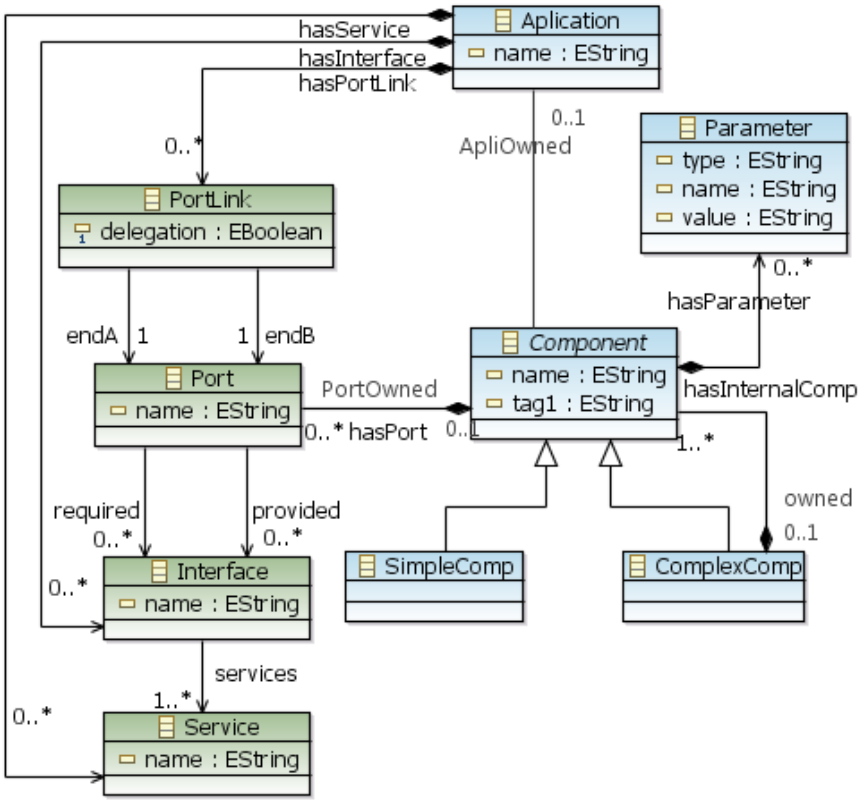

Figura 4.7: Metamodelo de componentes en formato EDiagrama .

4.2.4 Nivel especifico de la plataforma (PSM)

4.4.4.1 Plataforma de Implementación

Finalmente el proceso termina con la generación de código y la implementación real del sistema domótico. Para ello se debe seleccionar la plataforma específica de implementación que para nuestro caso se ha utilizado KNX (en la Figura 4.7 se muestra el metamodelo para KNX).

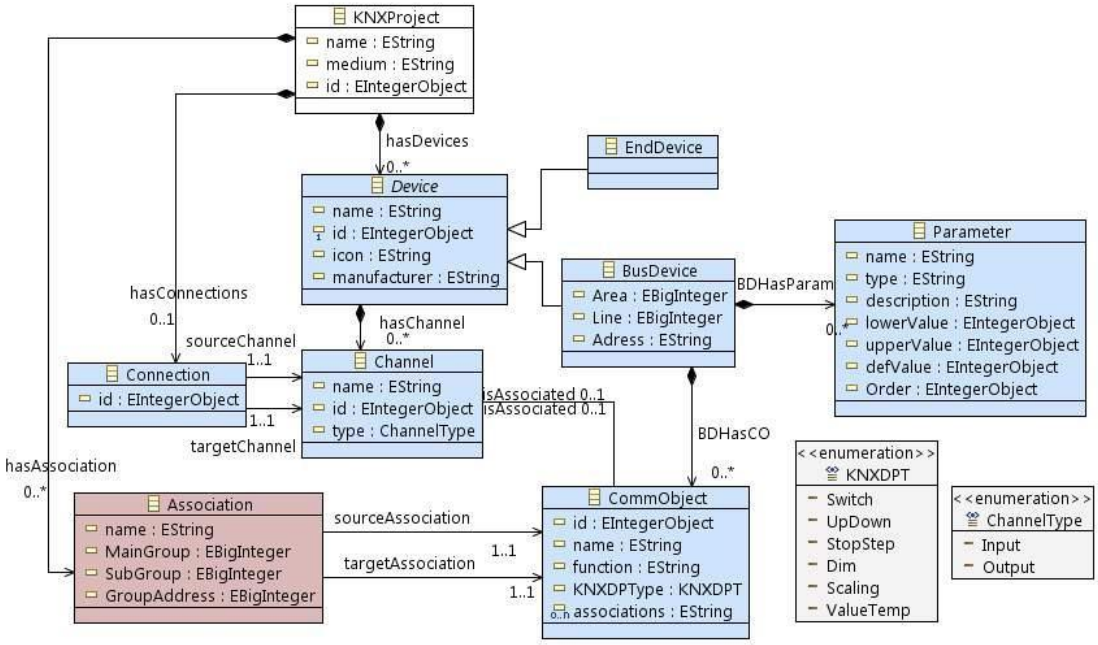

Figura 4.9: Metamodelo para la plataforma KNX.

Para el nivel específico de la plataforma EIB/KNX se ha apostado por el aprovechamiento de las herramientas disponibles en lugar de construir un conjunto de herramientas a partir de cero.

En la actualidad el punto de partida para un desarrollador tradicional sería la creación del proyecto mediante la iteración manual con la herramienta ETS (acrónimo del inglés *Engineering Tool Software*). Con la nueva metodología esa iteración desaparece y la herramienta se convierte en un soporte para la ejecución del código obtenido.

El programa ETS es la única herramienta software independiente del fabricante para diseñar y configurar instalaciones inteligentes para el control de casas y edificios hechas con el sistema KNX. La Herramienta *Software* para Ingeniería ETS3 se estructura por tanto de manera flexible, extensible y modular para poder facilitar futuras ampliaciones de la tecnología EIB/KNX. Asimismo, se ofrece al usuario una amplia ayuda en línea que facilita toda la información necesaria en este contexto. La herramienta ETS además puede ampliar su funcionalidad mediante una serie de plugins disponibles.

El plugin IT Tool de la herramienta ETS proporciona un entorno de desarrollo para escribir el código de programación de los sistemas domóticos. Este código se escribe en el lenguaje VBscript y se organiza en una serie de ficheros llamados macros. Estas macros son ejecutadas por la herramienta ETS obteniendo así la configuración final que se implementa en cada uno de los dispositivos.

Todas las operaciones realizadas por una macro pueden hacerse también “a mano” con la herramienta ETS. Sin embargo, las macros son más rápidas y cometen menos errores. De esta manera, la principal ventaja del uso de macros es que ahorran tiempo y permite automatizar la creación de un proyecto domótico.

Capítulo 5 Desarrollo de un Plug-in Eclipse para mejorar la integración y operatividad de Metadomo

En este capitulo describimos el desarrollo realizado para integrar la herramienta Metadomo en el entorno Eclipse. En primer lugar se especifican los objetivos del desarrollo, abordando cada uno de ellos a lo largo del capítulo.

5.1 Retos del desarrollo

En la revisión de la herramienta Metadomo pudimos comprobar que Metadomo es un conjunto de herramientas (editores, transformaciones, etc.) desarrolladas utilizando herramientas diferentes (JET, ATL). Así, Metadomo no proporciona un entorno totalmente integrando lo que dificulta el uso, aprendizaje y distribución de la herramienta.
El planteamiento inicial consiste en la creación de un Plug-in para Eclipse cuyo objetivo no es añadir funcionalidades a Metadomo si no que pretende:

1. Integrar los diferentes editores y transformaciones de Metadomo en un único entorno de trabajo, de manera que se vea mejorada la usabilidad de la herramienta Metadomo.
2. Mejorar la accesibilidad a las funcionalidades de Metadomo proporcionando un entorno más amigable e intuitivo.
3. Facilitar la operatividad de Metadomo, simplificando notablemente el número de operaciones realizadas para la obtención de un mismo fin respecto de Metadomo.
4. Mejorar la distribución de la herramienta y facilitar su instalación.
5. Proporcionar información de ayuda y guiar al usuario en la aplicación de la herramienta Metadomo.

Para alcanzar estos objetivos nos proponemos implementar las siguientes características:

- **Creación de una vista.** Con este instrumento mejoraremos la operatividad y la accesibilidad a Metadomo, todas las operaciones de Metadomo quedarán disponibles en una vista con una interfaz amigable y fácilmente utilizable.
- **Creación de una perspectiva para Metadomo.** Permite integrar las diferentes partes que engloban la herramienta.
- **Creación de menús.** Proporciona acceso a las funciones de la vista desde el menú principal y el explorador de paquetes.

5.2 Creación de una vista para Metadomo

Abordaremos la creación de una vista que permitirá mejorar la operatividad de Metadomo.

5.2.1 Descripción inicial

En el Capítulo II abordamos el concepto de vista como parte del IDE de Eclipse. El objetivo es crear una vista Eclipse que consista en una interfaz gráfica de usuario que contemple de una manera intuitiva toda la funcionalidad de la herramienta Metadomo. En la Figura 5.1 muestra el aspecto final de la vista desarrollada que representa un diagrama de flujo con el proceso de
transformación que siguen los modelos en Metadomo.
Se han considerados dos posibles caminos, por un lado, el paso directo de DSL a código (1). Por otro las distintas transformaciones pasando por todos los estados, DSL a Componentes (2), Componentes a Componentes (3), Componentes a KNX (4), KNX a Codigo (5).

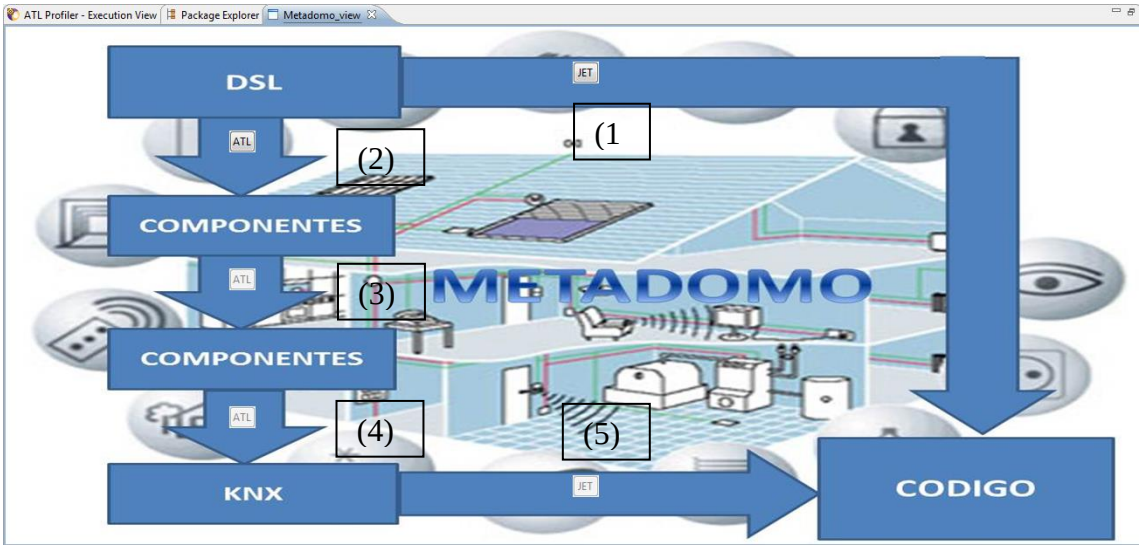

Figura 5.1: Vista Metadomo.

5.2.2 Procedimiento

Para la creación de la vista Eclipse hemos seguido el siguiente procedimiento:

1. **Adicción de dependencias necesarias para el funcionamiento del Plug-in.**

Comenzamos con la creación de un nuevo *Proyecto Eclipse* del tipo *Plugin Development*, pulsamos sobre *Plug-in.xml* en la raíz del Proyecto y nos dirigimos a la pestaña dependencias donde añadimos *org.eclipse.ui.ide*, que permite abrir los editores asociados utilizados a través de la vista, para añadir esta dependencia seleccionamos la pestaña dependencias de nuestro Proyecto y pulsamos añadir con la dependencia seleccionada (Figura 5.2).

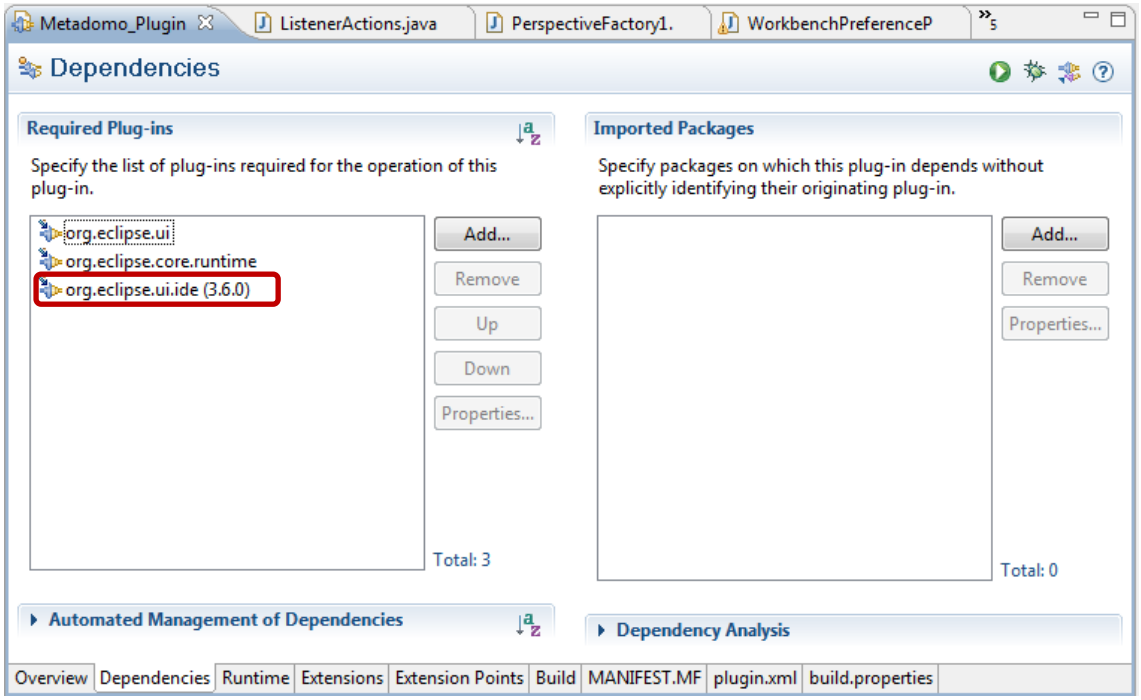

Figura 5.2: Sección de dependencias de Metadomo_Plugin.

2. Adicción de la extensión necesaria para el funcionamiento del Plug-in.

Como muestra la Figura 5.3 añadimos la extensión en la pestaña *Extensions* del Proyecto y añadimos *org.eclipse.ui.views*.

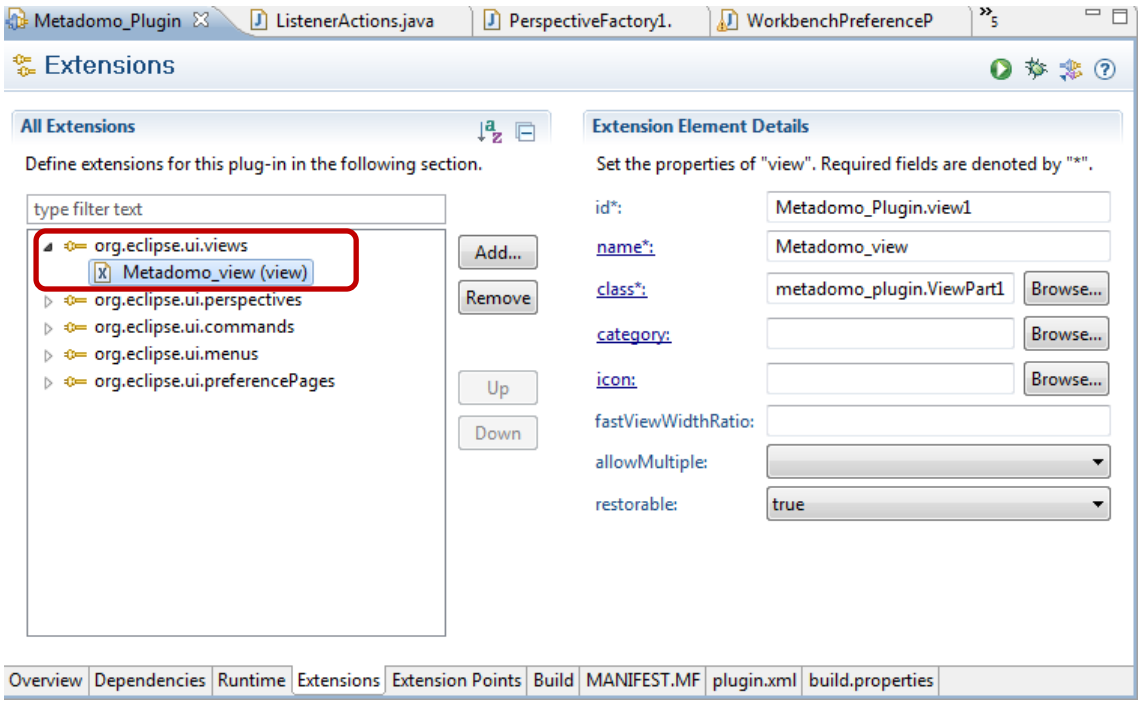

Figura 5.3: Extensión org.eclipse.ui.views.

3. Creación de la clase que gobierna la vista.

Seleccionamos con el botón derecho New View y rellenamos el campo *id* con el identificador, el nombre, la categoría y el icono no son necesarios. El apartado *class* es donde escribimos la clase Java que nos muestra la apariencia, funcionalidad, en resumen, es la clase que crea y controla la vista Metadomo.

```
"public class ViewPart1 extends ViewPart {
	MyViewLook apariencia;
	@Override
	public void createPartControl(Composite parent) {
		apariencia = new MyViewLook(parent);
		apariencia.setState(StateId.DSL);
		apariencia.initialize(new
		ListenerActions(apariencia,parent));


	}
}"
```

Para la creación de la clase de la vista y concretamente la interfaz gráfica se ha utilizado la librería SWT y ha sido implementada siguiendo el patrón *Modelo-Vista-Controlador* desarrollando así un diagrama de clases en el que la apariencia de la vista queda desacoplada. La clase que utiliza para la creación de la interfaz gráfica es MyViewLook, esta clase además inicializa los botones y se encarga del redimensionado de la vista.

```
public class MyViewLook extends AbstractViewLook implements
PaintListener{
	Image myImage;
	Composite c,composite;
	public MyViewLook(Composite com) {
		super(com);
		composite=com;
		com.setLayout(new FillLayout());
		myImage = new Image(com.getDisplay(),
		Activator.class.getResourceAsStream("../fondoV3.jpg"));
		c = new Composite(com,SWT.NONE);
		com.setParent(c);

		reset = new Button(c,SWT.PUSH);
		reset.setText("RESET");

		dslToComBtn = new Button(c,SWT.PUSH);
		dslToComBtn.setText("ATL");

		ComToComBtn = new Button(c, SWT.PUSH);
		ComToComBtn.setText("ATL");

		ComToKnxBtn = new Button(c, SWT.PUSH);
		ComToKnxBtn.setText("ATL");

		knxToCodBtn = new Button(c, SWT.PUSH);
		knxToCodBtn.setText("JET");

		dslToCodBtn = new Button(c, SWT.PUSH);
		dslToCodBtn.setText("JET");
```

```
            etiqueta = new Label(c, SWT.ARROW);

            c.addPaintListener(this);

      }
      @Override
      public void paintControl(PaintEvent pe) {
            GC gc = pe.gc;
            gc.setAntialias(SWT.ON);
            gc.setInterpolation(SWT.HIGH);



            int p = composite.getSize().x;
            int n = composite.getSize().y;
            c.setSize(new Point(p,n));

            gc.drawImage(myImage, 0, 0,
            myImage.getBounds().width,myImage.getBounds().height, 0, 0,
            p, n);
            Figure root = new Figure();
            XYLayout layout = new XYLayout();
            root.setLayoutManager(layout);
            root.setFont(composite.getFont());
            RectangleFigure rectangleFigure = new RectangleFigure();

      rectangleFigure.setBackgroundColor(ColorConstants.lightGray);
            rectangleFigure.setLayoutManager(new ToolbarLayout());
            rectangleFigure.setPreferredSize(100, 100);

            reset.setBounds((composite.getClientArea().width/27),
            composite.getClientArea().height/25, 50,35);

            dslToComBtn.setBounds((composite.getClientArea().width/4)-
            (composite.getClientArea().width/19),
            composite.getClientArea().height-
            12*composite.getClientArea().height/15, 30,25);

            dslToCodBtn.setBounds((composite.getClientArea().width)-
            (composite.getClientArea().width/2),(composite.getClientAre
            a().height-14*composite.getClientArea().height/15), 30,
            25);

      ComToComBtn.setBounds((composite.getClientArea().width/4)-
(composite.getClientArea().width/19),composite.getClientArea().height-
8*composite.getClientArea().height/15,30,25);

      ComToKnxBtn.setBounds((composite.getClientArea().width/4)-
(composite.getClientArea().width/19),composite.getClientArea().height-
4*composite.getClientArea().height/15, 30, 25);

      knxToCodBtn.setBounds((composite.getClientArea().width)-
(composite.getClientArea().width/2), composite.getClientArea().height-
2*composite.getClientArea().height/15, 30, 25);

      }

}
```

La clase MyViewLook hereda de la clase abstracta AbstracViewLook. En esta se declaran los estados por los que pueden pasar las trasformaciones y a su vez los botones que se encuentran habilitados y deshabilitados según el estado en el que se encuentra la trasformación.

```
"public class AbstractViewLook {


	public static final String DSL_TO_COMPONENTES =
"DSL_TO_COMPONENTES";
	public static final String COMPONENTES_TO_COMPONENTES =
"COMPONENTES_TO_COMPONENTES";
	public static final String COMPONENTES_TO_KNX =
"COMPONENTES_TO_KNX";
	public static final String KNX_TO_CODIGO = "KNX_TO_CODIGO";
	public static final String DSL_TO_CODIGO = "DSL_TO_CODIGO";
	public static final String PATH_FUENTE = "PATH_FUENTE";
	public static final String RESET = "RESET";


	protected static Button reset;
	protected static Button dslToComBtn;
	protected static Button ComToComBtn;
	protected static Button ComToKnxBtn;
	protected static Button knxToCodBtn;
	protected static Button dslToCodBtn;
	protected static Label etiqueta;

	Composite composite;



	public AbstractViewLook(Composite compo) {
		this.composite = compo;
	}

	public void initialize(SelectionListener se) {

		reset.addSelectionListener(se);
		reset.setData("id", RESET);
		dslToCodBtn.addSelectionListener(se);
		dslToCodBtn.setData("id", DSL_TO_CODIGO);
		knxToCodBtn.addSelectionListener(se);
		knxToCodBtn.setData("id", KNX_TO_CODIGO);
		ComToKnxBtn.addSelectionListener(se);
		ComToKnxBtn.setData("id", COMPONENTES_TO_KNX);
		ComToComBtn.addSelectionListener(se);
		ComToComBtn.setData("id", COMPONENTES_TO_COMPONENTES);
		dslToComBtn.addSelectionListener(se);
		dslToComBtn.setData("id", DSL_TO_COMPONENTES);



	}
```

```
	public void setState(StateId id) {

		switch (id) {
		case DSL:
			dslToCodBtn.setEnabled(true);
			dslToComBtn.setEnabled(true);
			ComToComBtn.setEnabled(false);
			ComToKnxBtn.setEnabled(false);
			knxToCodBtn.setEnabled(false);

			break;

		case KNX:
			dslToCodBtn.setEnabled(false);
			dslToComBtn.setEnabled(false);
			ComToComBtn.setEnabled(false);
			ComToKnxBtn.setEnabled(false);
			knxToCodBtn.setEnabled(true);

			break;
		case COMPONENTES:
			dslToCodBtn.setEnabled(false);
			dslToComBtn.setEnabled(false);
			ComToComBtn.setEnabled(true);
			ComToKnxBtn.setEnabled(false);
			knxToCodBtn.setEnabled(false);

			break;
		case COMPONENTES2:
			dslToCodBtn.setEnabled(false);
			dslToComBtn.setEnabled(false);
			ComToComBtn.setEnabled(false);
			ComToKnxBtn.setEnabled(true);
			knxToCodBtn.setEnabled(false);

			break;
		case CODIGO:
			dslToCodBtn.setEnabled(false);
			dslToComBtn.setEnabled(false);
			ComToComBtn.setEnabled(false);
			ComToKnxBtn.setEnabled(false);
			knxToCodBtn.setEnabled(false);

			break;
		case RESET:
			dslToCodBtn.setEnabled(true);
			dslToComBtn.setEnabled(true);
			ComToComBtn.setEnabled(false);
			ComToKnxBtn.setEnabled(false);
			knxToCodBtn.setEnabled(false);

		}


	}

}"
```

Al pulsar cada uno de los botones en la clase ListenerAction se ejecuta la trasformación que corresponda y se cambia el estado.

```
"public class ListenerActions implements SelectionListener {

	Composite parent;
	AbstractViewLook a;

	public ListenerActions(AbstractViewLook a, Composite parent) {
		this.a = a;
		this.parent = parent;
	}

	@Override
	public void widgetSelected(SelectionEvent e) {
		Button bo;
		bo = (Button) e.getSource();
		a.setState(StateId.DSL);
		System.out.println(mensajeEstado(bo));
	}

	public String mensajeEstado(Button bo){
		String sms = null;
	if(bo.getData("id")==AbstractViewLook.DSL_TO_COMPONENTES){
			a.setState(StateId.COMPONENTES);

	MetaDomoTrans.runDSL2Component("C:/Users/marcoantonio/Documents/
Proyecto/Workspace/modelos/kk.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/component.
xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/dslToCompo
nent.xmi");
			sms="Componentes";
		}
	else if(bo.getData("id")==AbstractViewLook.DSL_TO_CODIGO){
			a.setState(StateId.CODIGO);

	MetaDomoTrans.runDSL2Component("C:/Users/marcoantonio/Documents/
Proyecto/Workspace/modelos/kk.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/component.
xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/dslToCompo
nent.xmi");

	MetaDomoTrans.runComponent2Component("C:/Users/marcoantonio/Docu
ments/Proyecto/Workspace/modelos/Prueba.metadomo","C:/Users/marcoanton
io/Documents/Proyecto/Workspace/modelos/Prueba.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/component2
.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/ComponentT
oComponent.xmi");

	MetaDomoTrans.runComponent2KNX("C:/Users/marcoantonio/Documents/
Proyecto/Workspace/modelos/Prueba.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/knx.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/ComponentT
oKnx.xmi");
```

```
		MetaDomoTrans.runKNX2Code("C:/Users/marcoantonio/Documents/Proye
cto/Workspace/modelos/Prueba.metadomo","Proyecto");
                        sms="Codigo";
                }
        else
if(bo.getData("id")==AbstractViewLook.COMPONENTES_TO_COMPONENTES){
                        a.setState(StateId.COMPONENTES2);

        MetaDomoTrans.runComponent2Component("C:/Users/marcoantonio/Docu
ments/Proyecto/Workspace/modelos/CatalogoComponentes.xmi","C:/Users/ma
rcoantonio/Documents/Proyecto/Workspace/modelos/component.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/component2
.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/ComponentT
oComponent.xmi");
                        sms="componentes";
                }
                else
if(bo.getData("id")==AbstractViewLook.COMPONENTES_TO_KNX){
                        a.setState(StateId.KNX);

        MetaDomoTrans.runComponent2KNX("C:/Users/marcoantonio/Documents/
Proyecto/Workspace/modelos/component2.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/knx.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/ComponentT
oKnx.xmi");
                        sms="knx";
                }
        else if(bo.getData("id")==AbstractViewLook.KNX_TO_CODIGO){
                        a.setState(StateId.CODIGO);

        MetaDomoTrans.runKNX2Code("C:/Users/marcoantonio/Documents/Proye
cto/Workspace/modelos/knx.xmi","Proyecto");
                        sms="codigo";

        }else if (bo.getData("id")==AbstractViewLook.RESET) {
                        a.setState(StateId.DSL);
                        sms="inicio";

        }
                return sms;

        }
}"
```

Como podemos observar la vista Metadomo Plug-in (Figura 5.1) abarca todas las funcionalidades de Metadomo de manera intuitiva y dando transparencia al usuario respecto de la herramienta.

4. **Dificultades encontradas.**
   Al realizar la vista nos encontramos con la problemática del redimensionamiento de los elementos de la vista. Al introducir los botones y redimensionar la vista se descuadraban por lo que hemos optado por utilizar elementos proporcionales al ancho y alto de la ventana en cada momento.

```
"reset.setBounds((composite.getClientArea().width/27),
composite.getClientArea().height/25, 50,35);

dslToComBtn.setBounds((composite.getClientArea().width/4)-
(composite.getClientArea().width/19),
composite.getClientArea().height-
12*composite.getClientArea().height/15, 30,25);

dslToCodBtn.setBounds((composite.getClientArea().width)-
(composite.getClientArea().width/2),(composite.getClientArea().height-
14*composite.getClientArea().height/15), 30, 25);

ComToComBtn.setBounds((composite.getClientArea().width/4)-
(composite.getClientArea().width/19),composite.getClientArea().height-
8*composite.getClientArea().height/15,30,25);

ComToKnxBtn.setBounds((composite.getClientArea().width/4)-
(composite.getClientArea().width/19),composite.getClientArea().height-
4*composite.getClientArea().height/15, 30, 25);

knxToCodBtn.setBounds((composite.getClientArea().width)-
(composite.getClientArea().width/2),  composite.getClientArea().height-
2*composite.getClientArea().height/15, 30, 25);"
```

5.3 Accesibilidad a las funcionalidades de Metadomo

En este punto veremos como a parte de la vista Metadomo, también utilizando el mecanismo de extensión, hemos creado un menú Metadomo con todas las operaciones ya vistas e implementadas en la vista, en la barra de herramientas y en el menú del explorador de paquetes así como un botón de habilitación/des habilitación de la vista.

5.3.1 Adicción de nuevas opciones en la barra principal

El objetivo es crear un botón en la barra de herramientas el cual oculte/muestre la vista Metadomo (véase Figura 5.4). Para ello introducimos el concepto de action como un evento que se produce al actuar sobre el botón. Así el método action asociado a un botón en la barra de herramientas es llamado cada vez que el usuario hace clic. A continuación se describe el proceso seguido.

Figura 5.4: Botón action.

1. **Adicción de dependencias necesarias para el funcionamiento del Plug-in.**
   Con la dependencia añadida previamente para el funcionamiento de la vista sería suficiente, si queremos realizar este Plug-in previo a la construcción de la vista deberíamos añadir dicha dependencia.

2. **Adicción de la extensión necesaria para el funcionamiento del Plug-in.**
   Añadimos la extensión *org.eclipse.ui.actionSets* en la pestaña extensión de Plug-in.xml como muestra la Figura 5.5.

3. **Configuración de la extensión.**
   Con el botón derecho sobre la extensión pulsamos New->ActionSet , lo rellenamos escribiendo "Metadomo_Plugin.actionSet" y en la opción "visible" seleccionamos true pues en caso contrario si estaría creado pero no sería visible. Pulsando botón derecho sobre el *actionSet* creamos New->Action como vemos en la Figura 5.6. Campos a tener en cuenta: (1) el identificador que hemos rellenado con "Metadomo_Plugin.action1"; (2) el campo *toolbarPath* en el que indicamos que nuestro botón lo queremos anclado a la barra principal para ello escribimos "normal/additions"; (3) el campo *icons* donde deberemos pasarle la ruta del icono que queremos que muestre nuestro botón, en nuestro caso es el símbolo de una herramienta y por último, el campo *class* donde pulsamos y creamos la clase Java que gobierna el *action*.

La clase la hemos llamado MetadomoActionDelegare e implementa la interfaz *IWorkbenchWindowActionDelegate*, en el método principal de la clase llamado *run* pasamos el identificador de la vista dado en Plug-in.xml y le decimos que lo abra cada vez que el botón *action* sea pulsado. Tenemos un nuevo botón a*ction* que habilita/deshabilita la vista Metadomo.

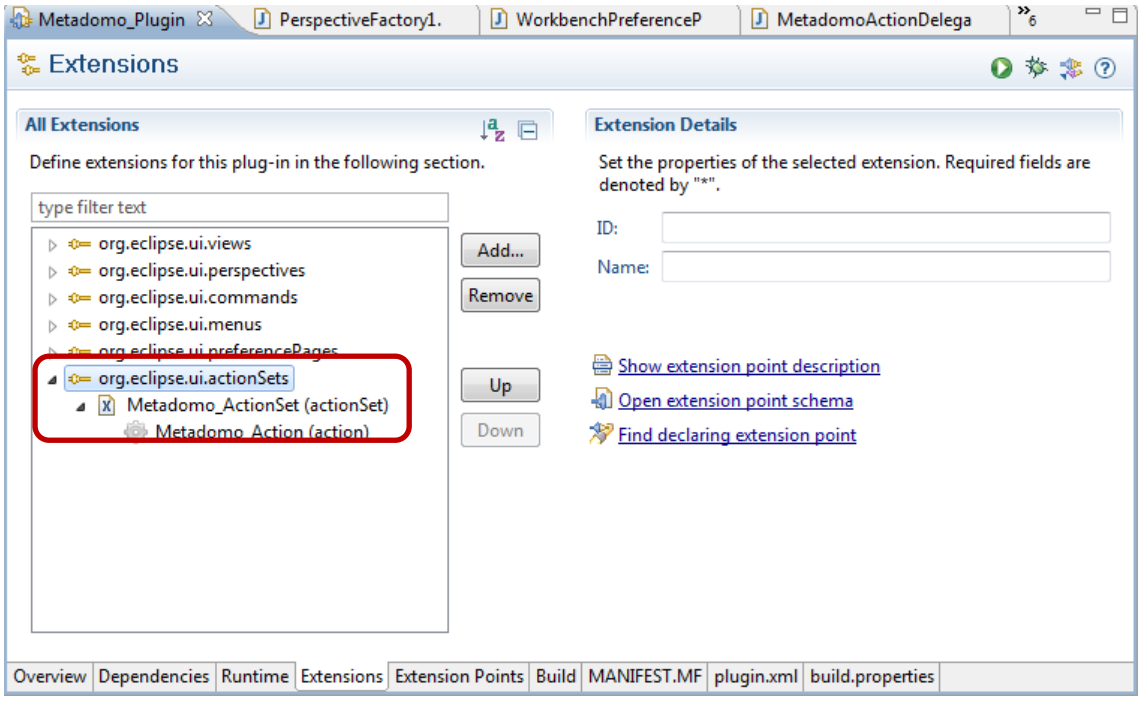

Figura 5.5: Adición de la extensión.

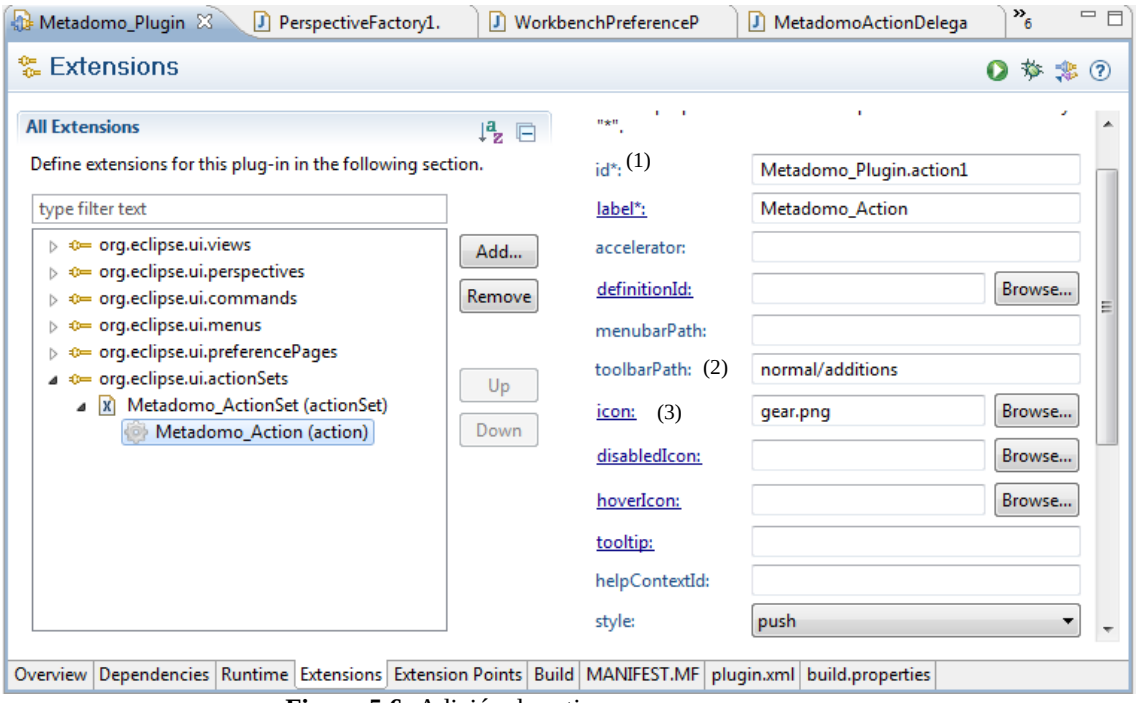

Figura 5.6: Adición de action y campos.

5.3.2 Adicción de nuevas opciones en el menú principal

En este punto abordamos la creación de un menú en la barra principal para permitir al usuario el acceso a las transformaciones definidas en Metadomo.

5.3.2.1 Descripción inicial

El objetivo es crear un menú llamado Metadomo en la barra principal que permita aplicar las distintas transformaciones que se pueden realizar en nuestra aplicacion. En la Figura 5.7 puede observarse la disposición de este menú.

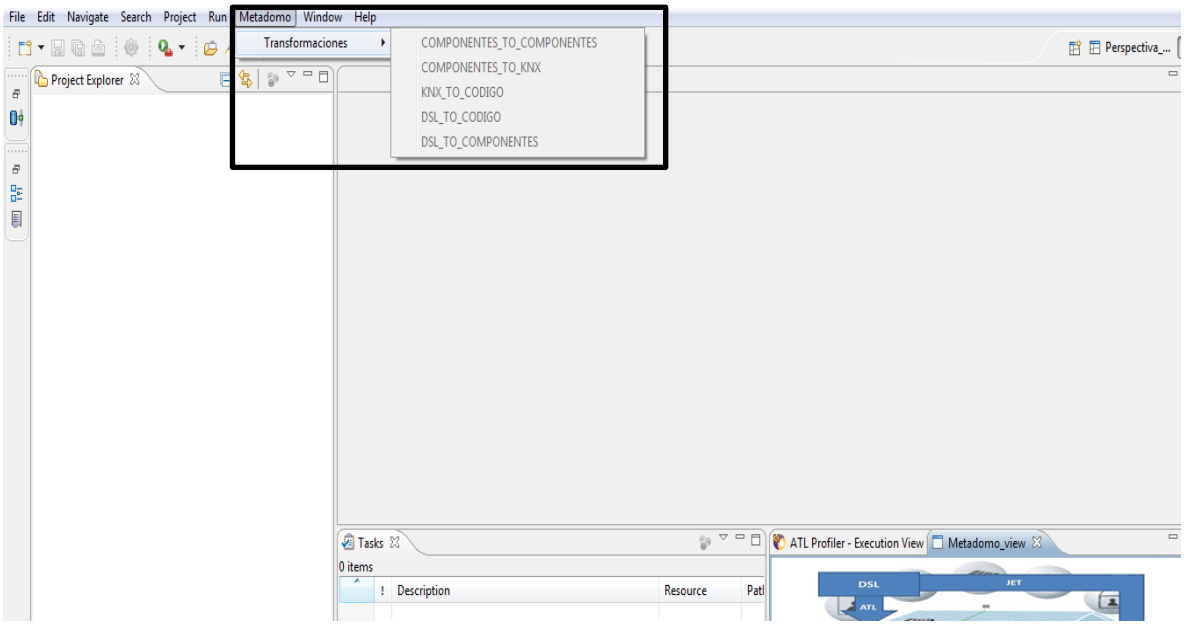

Figura 5.7: Menu Metadomo.

5.3.2.2 Procedimiento

Pasos a realizar para la construcción del menú:

1. **Adicción de dependencias necesarias para el funcionamiento del Plug-in**.
   Con la dependencia añadida previamente para el funcionamiento de la vista sería suficiente, si queremos realizar este Plug-in previo a la construcción de la vista deberíamos añadir dicha dependencia.

2. **Adicción de la extensión necesaria para el funcionamiento del Plug-in**.
   Para poder crear el menú Metadomo debemos añadir (Figura 5.8) las extensiones *org.eclipse.ui.commands* y *org.eclipse.ui.menus*. En la primera extensión que añadimos (*org.eclipse.ui.commands*) pulsamos botón derecho y creamos cinco nuevos *commands* (New->Command).

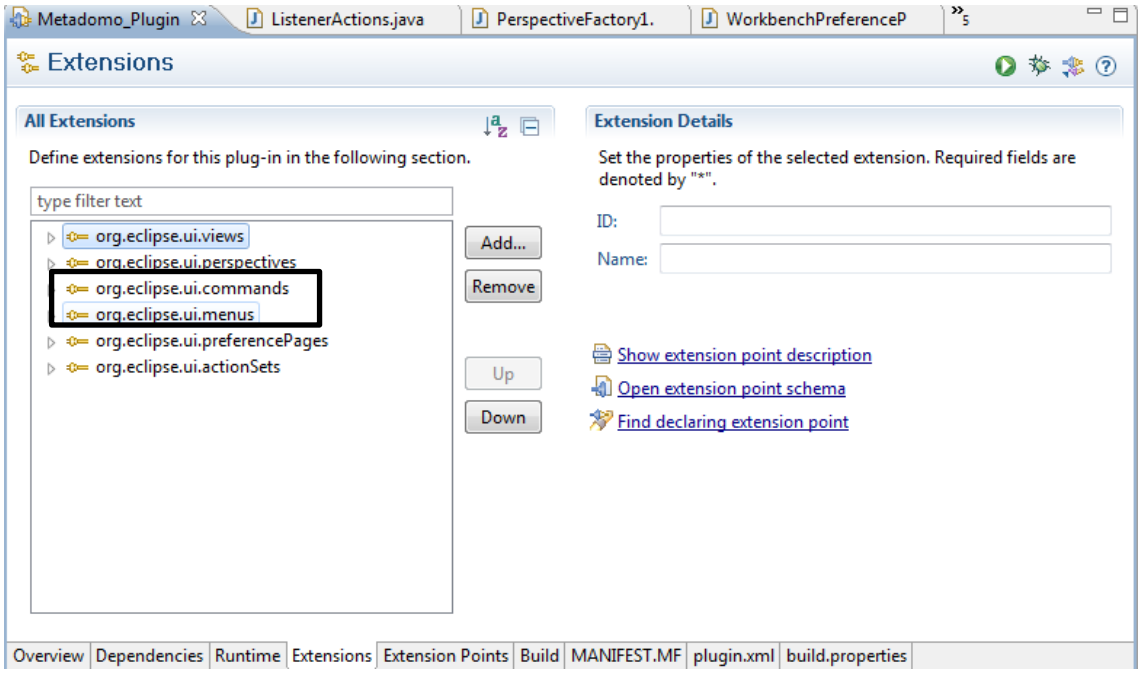

Figura 5.8: Adición de extensiones.

Como podemos observar en la Figura 5.9 hemos añadido cinco *command*, en el campo y hemos rellenado el campo id con los identificadores.

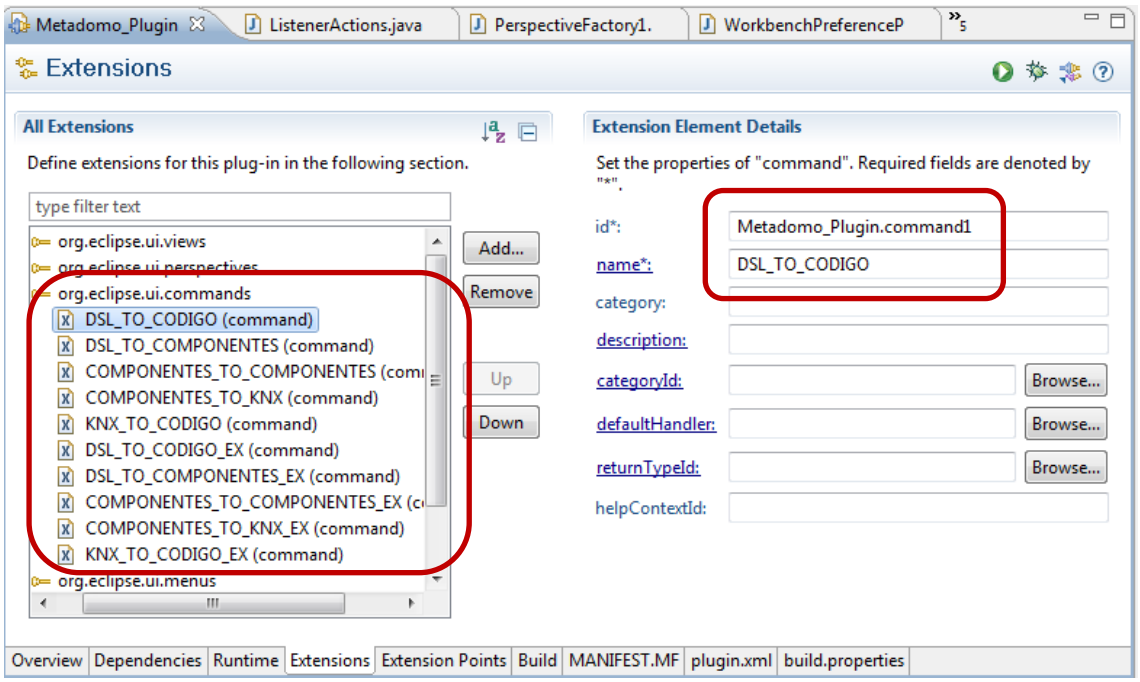

Figura 5.9: Adición de commands y campos rellenos.

En el apartado defaultHandler introducimos el nombre de cada una de las clases que lanza las transformaciones. Estas clases son:

- **Trasformación de DSL a COMPONENTES:**

```
public class DslToComponent extends AbstractHandler {
	AbstractViewLook a;
	@Override
	public Object execute(ExecutionEvent event) throws
ExecutionException {
		a.dslToCodBtn.setEnabled(false);
		a.dslToComBtn.setEnabled(false);
		a.ComToComBtn.setEnabled(true);
		a.ComToKnxBtn.setEnabled(false);
		a.knxToCodBtn.setEnabled(false);

	MetaDomoTools.runDSL2Component("C:/Users/marcoantonio/Documents/
Proyecto/Workspace/modelos/Prueba.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/component.
xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/dslToCompo
nent.xmi");
		return null;
	}
}
```

- **Trasformación de COMPONENTES A COMPONENTES:**

```
public class ComponentToComponent extends AbstractHandler {

	AbstractViewLook a;
	@Override
	public Object execute(ExecutionEvent event) throws
ExecutionException {
		a.dslToCodBtn.setEnabled(false);
		a.dslToComBtn.setEnabled(false);
		a.ComToComBtn.setEnabled(false);
		a.ComToKnxBtn.setEnabled(true);
		a.knxToCodBtn.setEnabled(false);

	MetaDomoTools.runComponent2Component("C:/Users/marcoantonio/Docu
ments/Proyecto/Workspace/modelos/Prueba.metadomo","C:/Users/marcoanton
io/Documents/Proyecto/Workspace/modelos/Prueba.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/component2
.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/ComponentT
oComponent.xmi");
		return null;
	}
}
```

- **Trasformación de COMPONENTES A KNX:**

```
public class ComponentToKnx extends AbstractHandler {
	AbstractViewLook a;
	@Override
	public Object execute(ExecutionEvent event) throws
ExecutionException {
		a.dslToCodBtn.setEnabled(false);
		a.dslToComBtn.setEnabled(false);
		a.ComToComBtn.setEnabled(false);
		a.ComToKnxBtn.setEnabled(false);
		a.knxToCodBtn.setEnabled(true);

	MetaDomoTools.runComponent2KNX("C:/Users/marcoantonio/Documents/
Proyecto/Workspace/modelos/Prueba.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/knx.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/ComponentT
oKnx.xmi");
		return null;
	}
}
```

- **Trasformación de KNX a CODIGO:**

```
public class KnxToCodigo extends AbstractHandler {
	AbstractViewLook a;
	@Override
	public Object execute(ExecutionEvent event) throws
ExecutionException {
		a.dslToCodBtn.setEnabled(false);
		a.dslToComBtn.setEnabled(false);
		a.ComToComBtn.setEnabled(false);
		a.ComToKnxBtn.setEnabled(false);
		a.knxToCodBtn.setEnabled(false);

	MetaDomoTools.runKNX2Code("C:/Users/marcoantonio/Documents/Proye
cto/Workspace/modelos/Prueba.metadomo","Proyecto");
		return null;
	}
}
```

- **Trasformación de DSL A CODIGO:**

```
public class DslToCodigo extends AbstractHandler {

	AbstractViewLook a;
	@Override
	public Object execute(ExecutionEvent event) throws
ExecutionException {
		a.dslToCodBtn.setEnabled(false);
		a.dslToComBtn.setEnabled(false);
		a.ComToComBtn.setEnabled(false);
		a.ComToKnxBtn.setEnabled(false);
		a.knxToCodBtn.setEnabled(false);

	MetaDomoTools.runDSL2Component("C:/Users/marcoantonio/Documents/
Proyecto/Workspace/modelos/Prueba.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/component.
xmi",
```

```
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/dslToCompo
nent.xmi");

		MetaDomoTools.runComponent2Component("C:/Users/marcoantonio/Docu
ments/Proyecto/Workspace/modelos/Prueba.metadomo","C:/Users/marcoanton
io/Documents/Proyecto/Workspace/modelos/Prueba.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/component2
.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/ComponentT
oComponent.xmi");

		MetaDomoTools.runComponent2KNX("C:/Users/marcoantonio/Documents/
Proyecto/Workspace/modelos/Prueba.metadomo",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/knx.xmi",
"C:/Users/marcoantonio/Documents/Proyecto/Workspace/modelos/ComponentT
oKnx.xmi");

		MetaDomoTools.runKNX2Code("C:/Users/marcoantonio/Documents/Proye
cto/Workspace/modelos/Prueba.metadomo","Proyecto");
			return null;
		}
}
```

Estas funciones además de lanzar las transformaciones habilitan y deshabilitan los botones de la vista para que desde el menú y la vista se realicen las transformaciones indistintamente y evitar errores. Para la creación del menú debemos crear un menuContribution pulsando botón derecho sobre la extensión y New->MenuContribution, dentro de menuContribution esta el campo locationURI en el cual para que nuestro menú se ancle a la barra principal tenemos que escribir "menu:org.eclipse.ui.main.menu" y en el campo allPopups escribimos false. Pulsando botón derecho sobre menuContribution creamos un menú con etiqueta Metadomo, este será la primera parte visual anclada a mi menú, pulsando botón derecho sobre el creamos otro menú New->Menú que será el que agrupe las transformaciones a realizar.
En el menú transformaciones creamos cinco nuevos commands (New->Command) los llamaremos según el nombre de la transformación que vayan a ejecutar y en el campo commandId le pasamos el identificador del command previamente creado en la extensión org.eclipse.ui.commands, a continuación creamos un separador con New->Separator y en el campo visible establecemos true y con esto ya tendríamos acabado, como muestra la Figura 5.7, el menú Metadomo en la barra principal.

5.3.3 Adicción del menú contextual en el explorador de paquetes

El objetivo es poder realizar las operaciones Metadomo sobre el archivo que seleccionemos en el explorador de paquetes (Figura 5.10), así la manera de proceder será muy similar a la creación del menú Metadomo en la barra principal, a excepción del campo locationURI que en este caso para que el menu quede anclado al explorador de paquetes debemos rellenarlo con "popup:org.eclipse.jdt.ui.PackageExplorer".Al seleccionar un archivo en el explorador de paquetes podremos realizar las operaciones transformación de nuestra aplicación.

Pasos para la realización del Plug-in:

1. **Adicción de dependencias necesarias para el funcionamiento del Plug-in.**
   Con la dependencia añadida previamente para el funcionamiento de la vista sería suficiente, si queremos realizar este Plug-in previo a la construcción de la vista deberíamos añadir dicha dependencia.

2. **Adicción de la extensión necesaria para el funcionamiento del Plug-in.**
   Como hemos mencionado la manera de proceder a la hora de crear este menú es análoga a la manera de proceder al crear el menú en la barra principal así que si no las tenemos añadidas previamente añadimos las extensiones *org.eclipse.ui.command* y *org.eclipse.ui.menus*.
3. **Creación de la clase que gobierna el Plug-in.**
   Seleccionado botón derecho sobre la extensión *org.eclipse.ui.command* creamos cinco nuevos commands los llamaremos según el nombre de la transformación que vayan a ejecutar en el campo identificador (*id*) le pasamos el identificador del command previamente creado en la extensión org.eclipse.ui.commands, a continuación creamos un separador con New->Separator y en el campo visible establecemos true y con esto ya tendríamos acabado, como muestra la Figura 5.10 el menú en el explorador de paquetes.

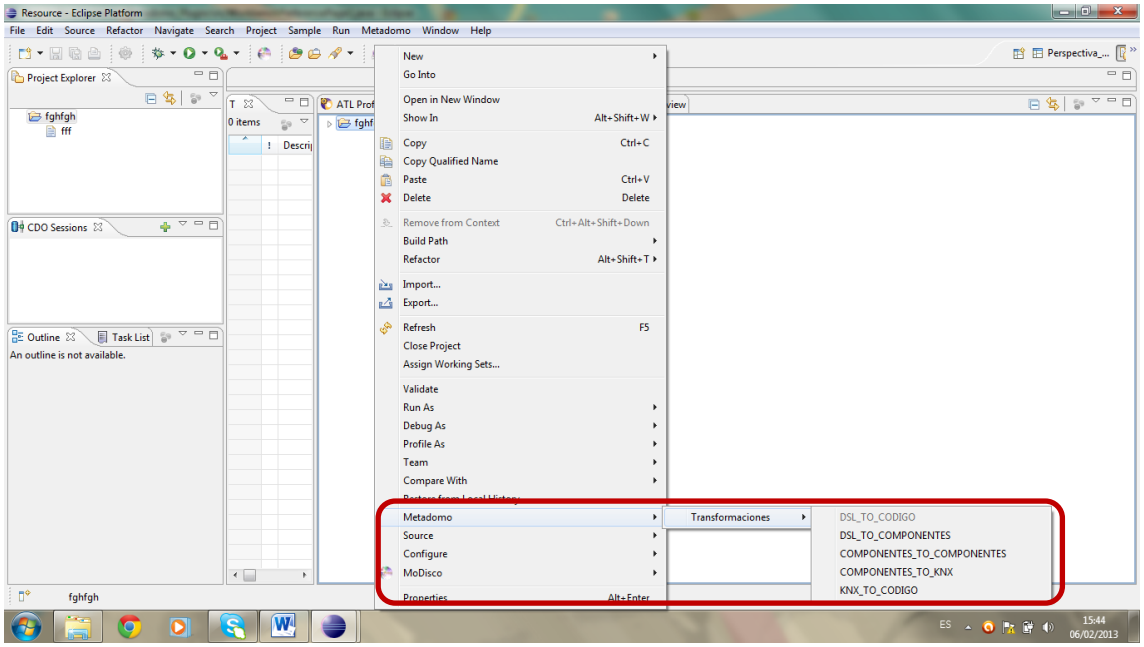

Figura 5.10: Adición de commands y campos rellenos.

Seleccionando botón derecho sobre org.eclipse.ui.menus creamos un nuevo menuContribution dentro del cual creamos un menú que llamamos Metadomo, esta será la etiqueta de principio de menú que veremos cuando seleccionemos botón derecho sobre el explorador de paquetes. A continuación creamos otro menú llamado transformaciones que será el que agrupe estas mismas, sobre transformaciones creamos un nuevo command que llamamos como las transformaciones que vamos a realizar y le pasamos el identificador de cada transformación correspondiente.

5.3.4 Creación de una Perspectiva

Una perspectiva es una organización de varias vistas de Eclipse alrededor del área del editor.

Como desarrolladores de Plug-in Eclipse podemos crear una perspectiva desde cero o extender una perspectiva existente. La Figura 5.11 muestra la perspectiva desarrollada para Metadomo, ésta contendrá los siguientes elementos: la vista Metadomo (descrita en la sección 5.2), botón de habilitación de la vista en la barra de herramientas (descrito en la sección 5.3.1), el menú Metadomo en la barra principal (descrito en la sección 5.3.2), así como el Explorador de Paquetes y los demás elementos comunes del entorno Eclipse.

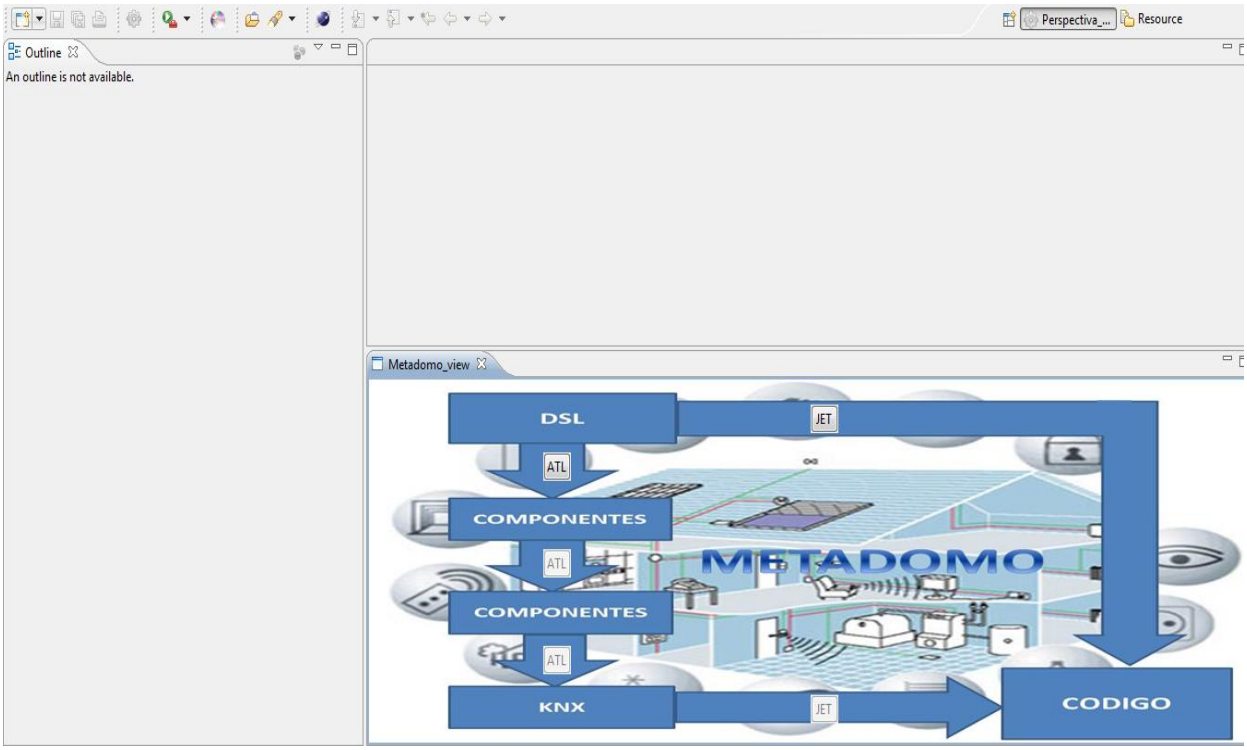

Figura 5.11: Perspectiva Metadomo

Para la creación del Plug-in seguimos lo siguientes pasos:

1. Adicción de dependencias necesarias para el funcionamiento del Plug-in. Con la dependencia añadida previamente para el funcionamiento de la vista sería suficiente, si queremos realizar este Plug-in previo a la construcción de la vista deberíamos añadir dicha dependencia.
2. Adicción de la extensión necesaria para el funcionamiento del Plug-in. Para crear una nueva perspectiva debemos añadir la extensión org.eclipse.ui.perspectives, como muestra la Figura 5.12, el campo id lo rellenamos con el identificador de la vista, en el campo icon seleccionamos el icono que muestre nuestra perspectiva y en el campo class escribimos la clase Java que gobierna la perspectiva.
3. Creación de la clase que gobierna el Plug-in. Esta clase la llamamos PerspectiveFactory e implementa la interfaz IPerpectivefactory por lo que tenemos que implementar el método createInitialLayout (IPageLayout layout) en el cual lo único que hacemos es añadir la vista Metadomo a la perspectiva. El resultado es que cada vez que abrimos la perspectiva Metadomo como muestra la Figura 5.13 en Windows ->Open Perspective. Obtenemos la perspectiva Metadomo con la vista asociada, Figura 5.11

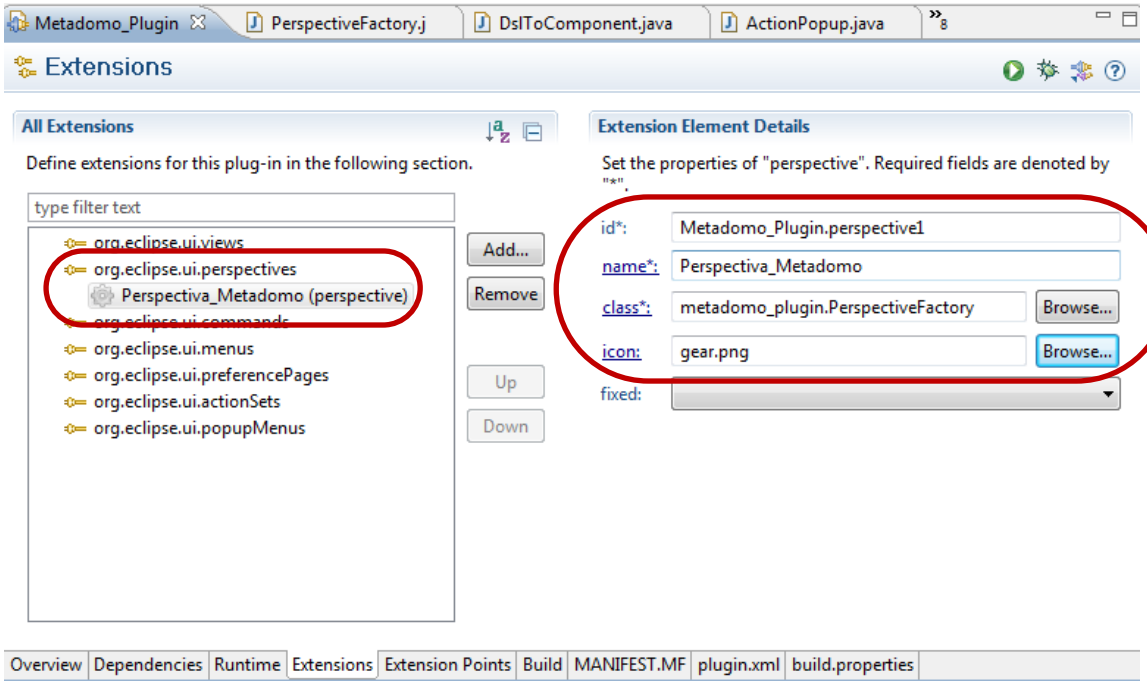

Figura 5.12: Adición de org.eclipse.ui.perspective

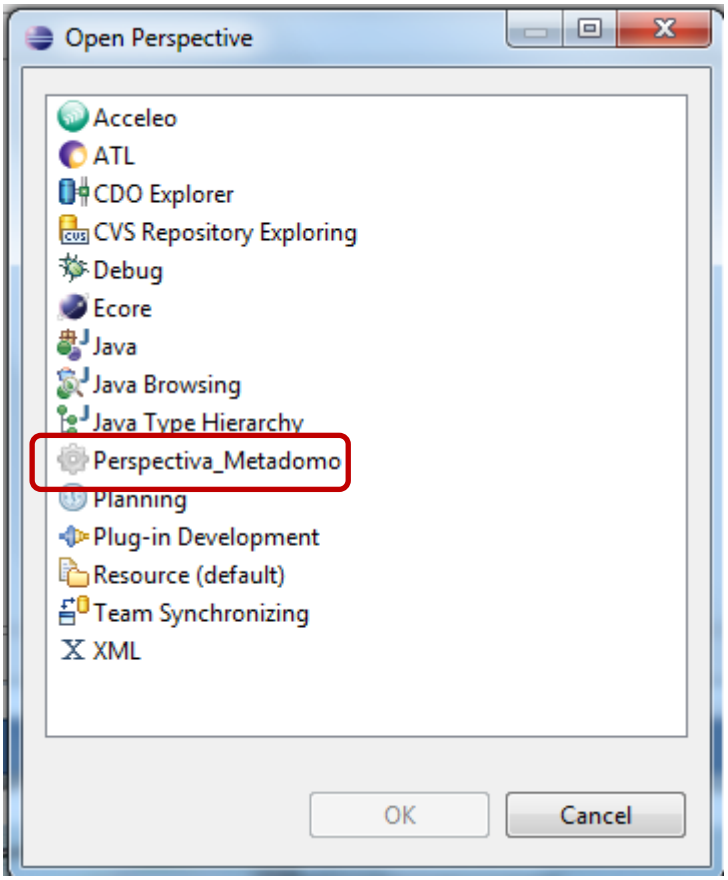

Figura 5.13: Abrir perspectiva Metadomo

5.4 Exportación del Plug-in desarrollado

En este apartado vamos a describir el proceso de exportaciones realizadas. En primer lugar, abordamos la exportación del plug-in desarrollado como un archivo *.jar, de forma que un usuario pueda instalar el plug-in en su entorno Eclipse. Esta opción implica que el usuario se hará responsable de la instalación de aquellos Plug-ins necesarios para el funcionamiento de
Metadomo. En segundo lugar, abordamos la exportación del plug-in como producto Eclipse, es decir, se proveerá un ejecutable con el entorno Eclipse configurado para la herramienta Metadomo.

5.4.1 Creación de un fichero JAR

Para la creación del archivo *Metadomo_Plug-in_1.0.0.jar* nos situamos en la pestaña Overview del Plug-in concretamente a la sección titulada Exporting, como vemos en la Figura 5.14, y realizamos los pasos que se presentan.

1. Organizar y limpiar los Proyectos Eclipse.
2. Externalizar los "*strings*" dentro del plug-in utilizando el asistente correspondiente.
3. Definir las librerías, especificar el orden en que deben construirse y dar una lista de las carpetas originales que deben ser compiladas en la librería seleccionada.
4. Exportar los Proyectos seleccionados en una forma adecuada para la implementación de un producto Eclipse. Concretamente en la Figura 5.15 puede verse la selección de los Plug-ins a exportar en el caso de este Proyecto es Metadomo. Por último, se introduce el directorio de salida donde queremos que se cree el archivo *.jar e indicamos su nombre.

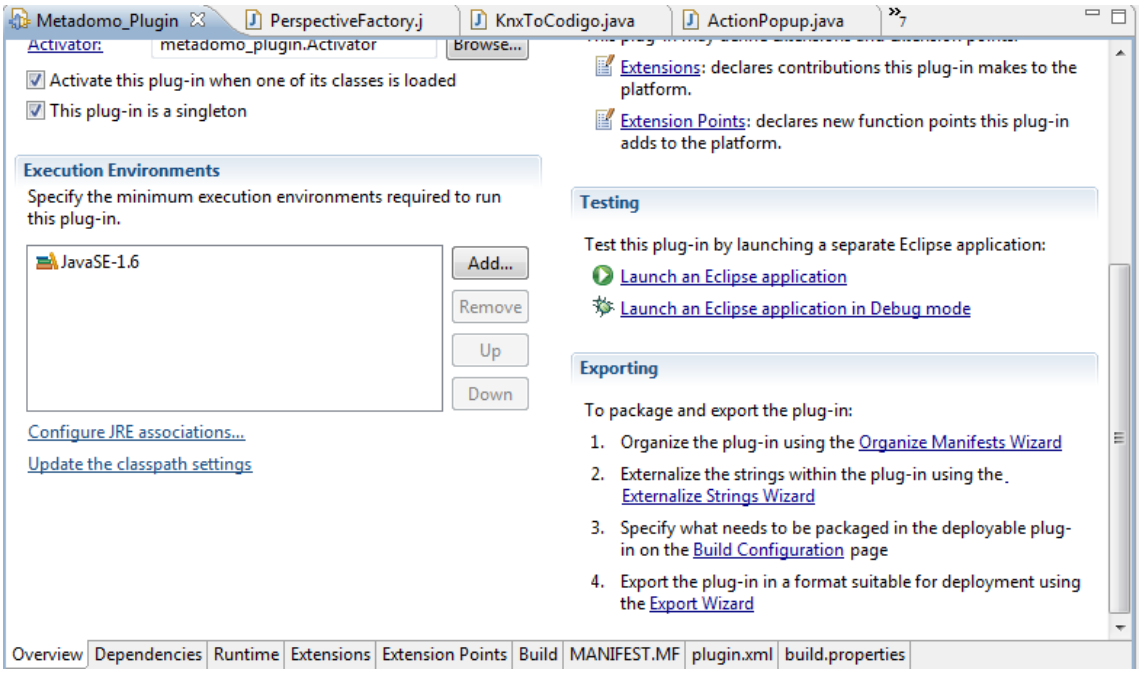

Figura 5.14: Exportación del producto.

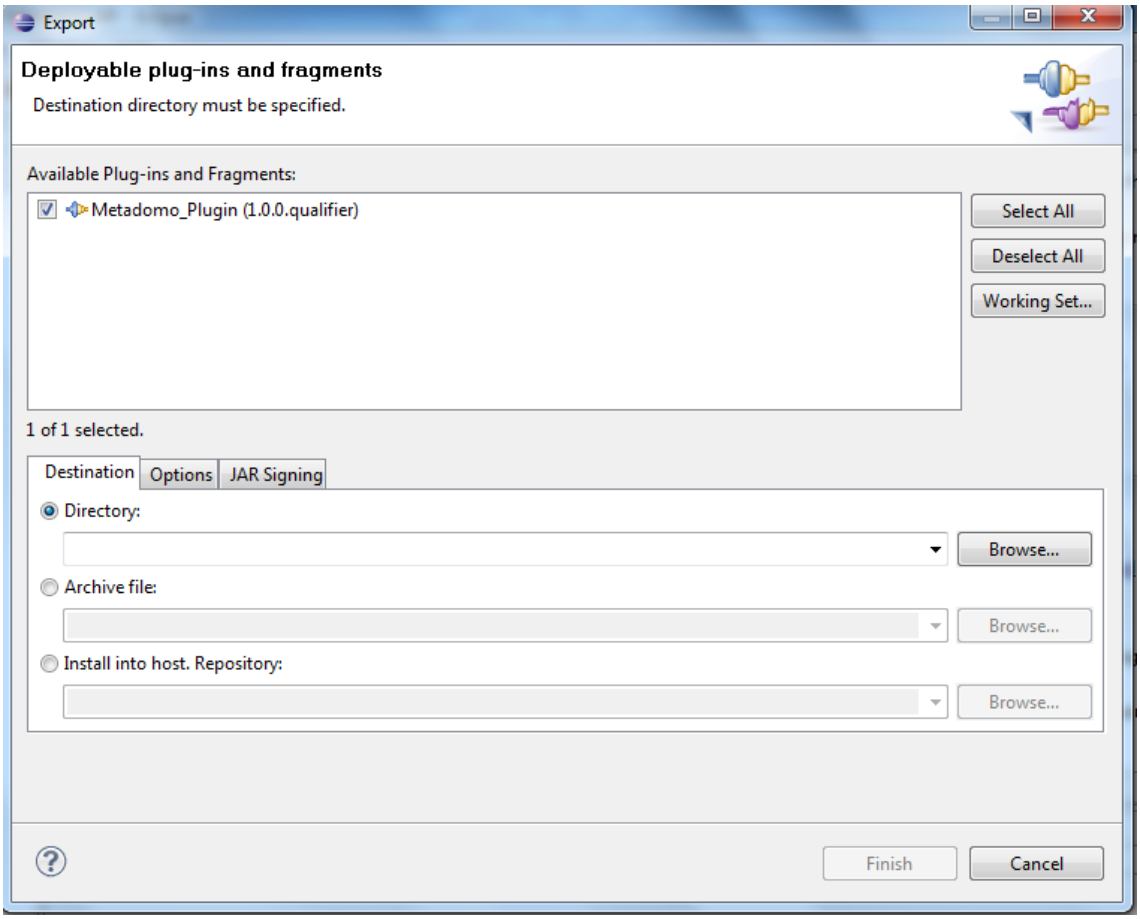

Figura 5.15: Directorio salida del fichero

Capitulo 6 Conclusiones y Lineas de trabajo futuras

6.1. Conclusiones

A lo largo del desarrollo de este Proyecto se han alcanzado varias conclusiones, entre las que cabe destacar las siguientes:

- Se ha desarrollado un Plug-in Eclipse para la herramienta Metadomo aprovechando la capacidad de Eclipse para ser extendido usando el mecanismo de extensión y puntos de extensión.

- Metadomo se fundamenta en el paradigma de Desarrollo de Software Dirigido por Modelos que permite elevar el nivel de abstracción del desarrollo software, de forma que el usuario sólo ha de ser experto en su ámbito de aplicación obviando los conocimientos técnicos involucrados y ajenos a dicho ámbito

- Se ha desarrollado una vista, con una interfaz amigable y sencilla, que proporciona la representación y seguimiento del flujo de trabajo típico de usuario en Metadomo, mejorando la operatividad y uso de esta herramienta. Además, la vista ha simplificado el número de operaciones que el usuario debe hacer para realizar las mismas tareas que en el entorno original.

- Se ha añadido un nuevo botón en la barra de herramientas que permite activar o desactivar la vista desarrollada.

- Se han añadido nuevas opciones de menú en la barra principal y en el menú contextual que mejoran la accesibilidad a las funcionalidades de Metadomo.

- Se ha desarrollado una perspectiva para integrar los diferentes editores que conforman Metadomo y todos los elementos desarrollados en el presente Proyecto, consiguiendo un entorno totalmente unificado.

- Se ha realizado un producto Eclipse para la herramienta Metadomo, lo que mejora su distribución e instalación. Evita, por lo tanto, complicaciones instalando Eclipse y todos aquellos Plug-ins requeridos para el correcto funcionamiento de Metadomo.

6.2. Líneas de trabajo futuras

En este apartado se incluyen algunas posibles extensiones y mejoras que consideramos que sería interesante abordar de cara al futuro.

- Dado que, en ciertos aspectos, el desarrollo de Plug-ins Eclipse es repetitivo y, a veces, rutinario, por ejemplo, los pasos para la creación de menús o vistas constan de elementos comunes que se suelen repetir independientemente de la aplicación. Se podría plantear el desarrollo de una herramienta para automatizar el diseño e implementación de Plug-ins dirigidos a la integración de otras herramientas existentes que utilicen Eclipse.
  .
- Actualizar el Plug-in desarrollado en el presente Proyecto para la última versión de la herramienta Metadomo.

Bibliografía

1. https://www.**eclipse**.org/
2. Manuel Jiménez Buendía, Tesis Doctoral Desarrollo de sistemas domóticos utilizando un enfoque dirigido por modelos
3. Francisca Rosique, Manuel Jiménez Buendía y Andrés Iborra Un lenguaje de modelado gráfico para sistemas domóticos
4. Francisca Rosique, Pedro Sánchez, Manuel Jiménez, Bárbara Álbarez, Un marco integral para el desarrollo de sistemas domoticos.