UNIVERSIDAD POLITÉCNICA DE CARTAGENA
E. T. S. Ingeniería de Telecomunicaciones

INTEGRACIÓN DE SOFTWARE PARA EL DESARROLLO DE APLICACIONES ROBÓTICAS BASADAS EN PLUG-INS ECLIPSE

José Fermín Díaz Amado

Título del Proyecto

INTEGRACIÓN DE SOFTWARE PARA EL DESARROLLO DE APLICACIONES ROBÓTICAS BASADAS EN PLUG-INS ECLIPSE

Autor

José Fermín Díaz Amado

Titulación

Ingeniero Técnico de Telecomunicación especialidad Telemática

Directores

Francisca Rosique Contreras
Juan Francisco Inglés Romero

Defensa

En Cartagena, Junio de 2012

ÍNDICE

CAPÍTULO I

Introducción

Eclipse [1] es una plataforma de desarrollo open source basada en Java, muy extendida y valorada en el ámbito del desarrollo de software. En sí mismo, Eclipse es framework destinado a construir Entornos de Desarrollo Integrados (IDE, Integrated Development Environment) a través de la implementación de Componentes (Plug-ins) que extienden la funcionalidad del entorno Eclipse. Así pues, podemos encontrar Plug-ins para el desarrollo de aplicaciones Java (JDT, Java Development Tools), C/C++ (CDT, C/C++ Development Tools), COBOL, etc.

El entorno de modelado HuRoME (Humanoid Robot Modeling Environment) [2-5] integra un conjunto de herramientas diseñadas para facilitar el modelado de coreografías y la modernización del software existente para el robot humanoide Robonova [6]. HuRoME permite a los numerosos usuarios de Robonova: (1) modelar gráficamente y validar formalmente las secuencias de movimientos del robot (coreografías), (2) generar automáticamente la implementación asociada a cada coreografía en el lenguaje RoboBASIC, y (3) modernizar y reutilizar el software ya existente, permitiendo la obtención de los modelos equivalentes a cualquier programa RoboBASIC existente.

El entorno HuRoME ofrece una aproximación al desarrollo de software para robótica utilizando un enfoque de Desarrollo de Software Dirigido por Modelos [7]. HuRoME no sólo facilita la generación automática de código a partir de modelos, sino también al proceso inverso, permitiendo la transformación de programas ya existentes (legacy code) en modelos, facilitando así su tratamiento (depuración, extensión, reutilización, análisis, simulación, etc.) con un nivel de abstracción mayor que el que proporciona el código fuente. No obstante, HuRoME no proporciona un entorno totalmente unificado lo que dificulta 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 HuRoME persiguiendo, por una parte, mejorar la usabilidad y operatividad del entorno para el desarrollo de aplicaciones robóticas, y por otra, facilitar su distribución y extensión.

1.1 Objetivos

Los objetivos que se pretenden abordar en el presente Proyecto son los siguientes:

1. Conocer los fundamentos que rigen los entornos IDE. En el caso de este Proyecto, nos centramos en el estudio del entorno de desarrollo Eclipse, haciendo especial énfasis en el funcionamiento basado en extensiones y la creación de Plug-ins

2. El objetivo principal de este Proyecto es la creación de un Plug-in Eclipse que integre la herramienta HuRoME, creando así un entorno de trabajo orientado a la robótica mediante el cual podamos realizar todas las operaciones que permite HuRoME, pero mejorando la usabilidad, operatividad y distribución de la herramienta. Para ello el Plug-in desarrollado en este Proyecto se compone de los siguientes elementos: (1) Una vista Eclipse que representa de forma gráfica el flujo de trabajo típico que un usuario ha de seguir en HuRoME y que permite realizar las operaciones de manera más intuitiva. (2) Un botón en la barra de herramientas que habilita/deshabilita la vista en el entorno. (3) Un panel de preferencias para configurar ciertos aspectos de la herramienta. (4) Nueva opciones de menús para mejorar la accesibilidad de las funcionalidades de HuRoME. (5) Una ayuda que expone el funcionamiento del Plug-in. (6) Una perspectiva que engloba todo el entorno de trabajo de el Plug-in.

3. Obtención de resultados y extracción de conclusiones sobre la herramienta desarrollada. Planteamiento de mejoras y líneas de trabajo futuras.

1.2 Fases de desarrollo del Proyecto

El desarrollo de este Proyecto se ha llevado a cabo siguiendo las etapas que se resumen a continuación:

a) Estudio de los fundamentos de los entornos IDE. En esta fase se han abordado los fundamentos básicos del presente Proyecto. En concreto, se ha realizado el estudio de la arquitectura software de la plataforma Eclipse y su extensión basada en Plug-ins***.***

b) Estudio de la herramienta HuRoME dirigida al desarrollo de aplicaciones software para un robot humanoide. Se han analizado las posibilidades que nos ofrece la herramienta HuRoME, así como su posible integración en el entorno Eclipse

c) Diseño e implementación de un Plug-in Eclipse para construir un entorno IDE dirigido a la robótica. En esta fase, se ha desarrollado un Plug-in Eclipse para facilitar la usabilidad de la herramienta HuRoME. Así pues, se ha abordado la implementación de una vista para representar el flujo de trabajo con HuRoME, un panel de preferencias de configuración, un menú de ayuda, el despliegue de nuevas opciones de menú en el explorador de paquetes y en la barra

principal, la inclusión de nuevos elementos en la barra de herramientas y la creación de una perspectiva Eclipse con todas estas nuevas características desarrolladas.

d) Evaluación del entorno desarrollado, extracción de conclusiones y redacción de la memoria.

1.3 Organización de la memoria

El resto de la memoria se organiza como se indica a continuación:

Capítulo 2: Breve revisión del entorno de desarrollo Eclipse

Capítulo 3: Revisión de la herramienta HuRoME y su uso en el desarrollo de aplicaciones robóticas

Capítulo 4: Desarrollo de un Plug-in Eclipse para mejorar la integración y operatividad de la herramienta HuRoME.

Capítulo 5: Guía de uso del Plug-in desarrollado para el entorno HuRoME.

Capítulo 6: Exposición de resultados, conclusiones y trabajos futuros.

Bibliografía

Anexo I: Descripción de la clase HuromeTools

Anexo II: Código del Plug-in implementado

CAPÍTULO II

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), como 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.

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para *VisualAge*. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Eclipse fue liberado originalmente bajo la *Common Public License*, pero después fue re-licenciado 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).

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, HP-UX 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.

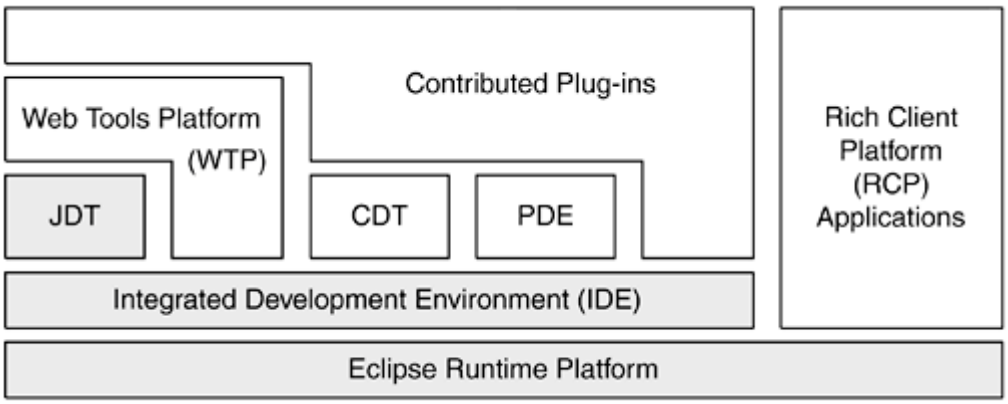

Figura 2.1.: IDE Eclipse (fuente [8])

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.

- **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.
- **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.
- **Editores.** En la Figura 2.2 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.
- **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++).
- **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 especificas de cada plug-in.

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.

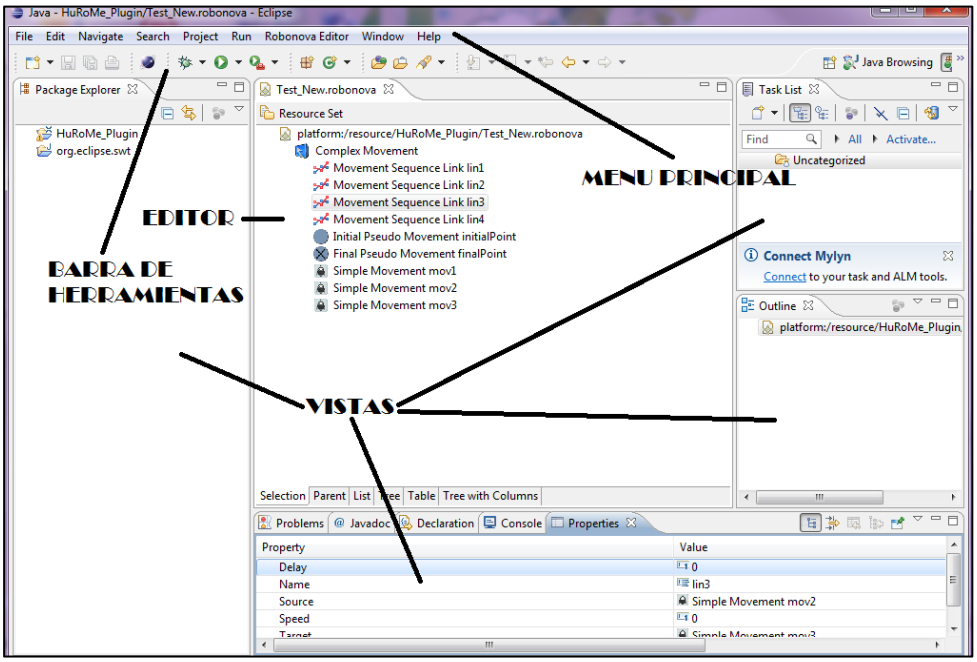

Figura 2.2: IDE Eclipse

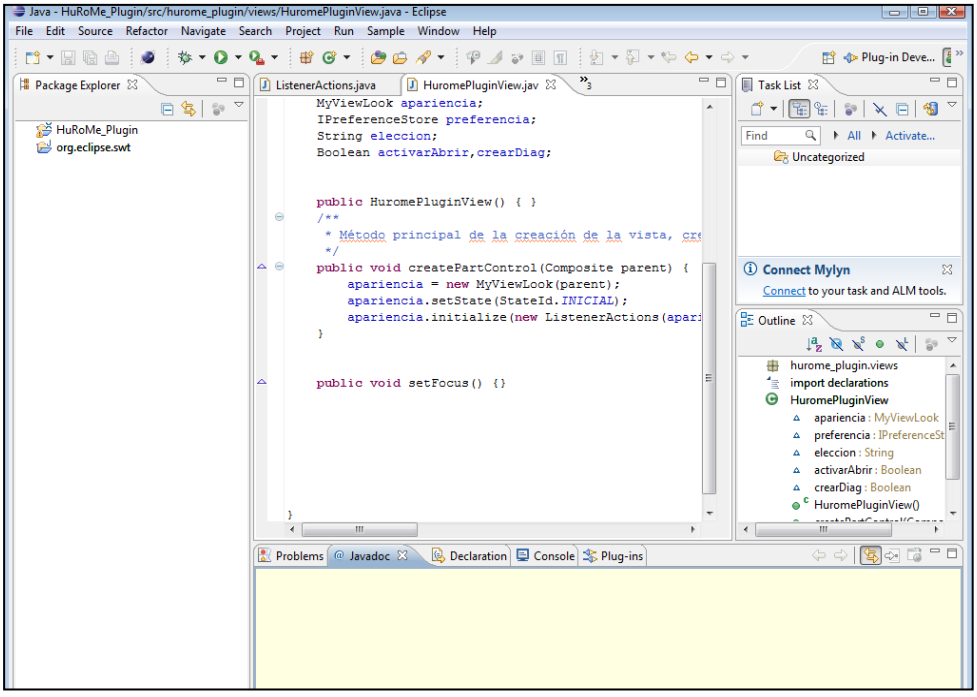

Figura 2.3.: Perspectiva Java

2.2.4 C/C++ Development Tools

C/C++ Development Tools (CDT) [9] consiste en un completo y funcional IDE de C y C++ para Eclipse, este incluye:

- **C/C++ editor**. Editor de texto especializado para C y C++ con formato, coloreado de la sintaxis y ayuda de contenido
- **C/C++ debugger.** Debugador por defecto.
- **C/C++ program launcher.** Similar al lanzador de Java pero con opciones de debugación**.**
- **C/C++ parser and syntax API.** Esencial para el funcionamiento de otros Plug-in.
- **Search engine.** Especializado en localizar con precisión códigos y referencias.
- **Makefile generator.** Necesario para compilar los archivos de C y C++.

2.2.5 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.

El Proyecto WTP se compone de dos partes. El sub-proyecto *Web Standard Tools* que provee una infraestructura común dirigida a aplicaciones Web. Esto incluye herramientas para desarrollar aplicaciones de tres niveles (presentación, lógica de negocio, y datos). En esta parte las herramientas proporcionadas son:

- **Standard languages.** Se prestará apoyo para HTML / XHMTL, XML, XML Schema XML, Servicios Web, XQueries, SQL y otros lenguajes utilizados por aplicaciones web.
- **Editors.** Los editores soportan distintos lenguajes, incluyendo características consistentes para los esquemas, el asistente de contenido, plantillas, y el formato.
- **Validators.** Lenguajes basados en XML.
- **Server publication.** El entorno de trabajo contiene comandos y puntos de vista para iniciar, detener, publicar y ejecutar las aplicaciones en varios servidores de destino.

Por otro lado, el sub-proyecto *J2EE Standard Tools* que provee una infraestructura común para desarrollar aplicaciones basadas en J2EE. En esta parte las herramientas proporcionadas son:

- **J2EE artifacts.** Soporte para EHB, Servlet, JSP, JCA, JDBC, JTA, JMS, JMX, JNDI y Web Services.
- **JSP editor.** Los editores soporta etiquetas en HTML y JSP.
- **JSP refactoring.**
- **Search Facilities & Comparison of syntax.** Permite criterios de búsqueda en la sintaxis de JSP, XML y otros tipos de documentos. Comparación de ficheros JSP o XML.

2.2.6 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.

2.2.7 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 las herramientas 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 meta-modelo 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

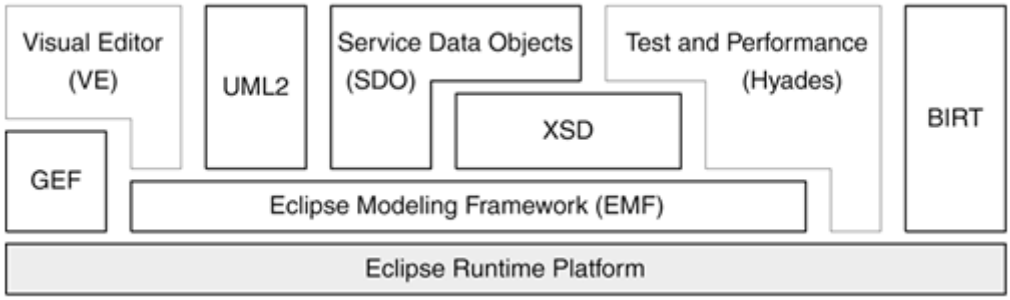

Figura 2.4.: Otros Proyectos Eclipse (fuente [8])

CAPÍTULO III

Humanoid Robot Modeling Environment (HuRoME)

En el presente capítulo presentamos una breve revisión de las características y funcionalidades de la herramienta HuRoME [2-5].

3.1 Descripción general de la herramienta

El entorno de modelado HuRoME (*Humanoid Robot Modeling Environment*) integra un conjunto de herramientas diseñadas para facilitar el modelado de coreografías y la modernización del software existente para el robot humanoide Robonova [6]. HuRoME permite a los numerosos usuarios de Robonova: (1) modelar gráficamente y validar formalmente las secuencias de movimientos del robot (coreografías), (2) generar automáticamente la implementación asociada a cada coreografía en el lenguaje RoboBASIC, y (3) modernizar y reutilizar el software ya existente, permitiendo la obtención de los modelos equivalentes a cualquier programa RoboBASIC existente.

El entorno HuRoME ofrece una aproximación al desarrollo de software para robótica utilizando un enfoque de Desarrollo de Software Dirigido por Modelos. HuRoME no sólo facilita la generación automática de código a partir de modelos, sino también al proceso inverso, permitiendo la transformación de programas ya existentes (legacy code) en modelos, facilitando así su tratamiento (depuración, extensión, reutilización, análisis, simulación, etc.) con un nivel de abstracción mayor que el que proporciona el código fuente. En el siguiente apartado describimos brevemente los fundamentos del *Desarrollo de Software Dirigido por Modelos*.

3.2 Breve revisión del Desarrollo de Software Dirigido por Modelos

El *Desarrollo de Software Dirigido por Modelos* (DSDM) [7] comprende un conjunto de técnicas y herramientas que permiten modelar formalmente los sistemas que se quieren desarrollar para, posteriormente, aplicando una serie de transformaciones automáticas, obtener el código final de las aplicaciones. Así, el DSDM gira entorno a la definición y el uso sistemático de modelos y de transformaciones de modelos a lo largo de todo el ciclo de vida del desarrollo de software.

La noción de modelo es muy antigua y puede definirse como "una abstracción o simplificación de la realidad con un cierto propósito" [12]. Más concretamente, de acuerdo a la definición proporcionada por Bézivin y Gerbé en [13], "un modelo es una simplificación de un sistema construido con un objetivo definido. El modelo ha de ser capaz de responder a las preguntas que se le formulen como si se tratara del sistema real. Las respuestas que proporciona el modelo deberán ser exactamente las mismas que si respondiera el sistema, con la condición de que las preguntas se formulen en los mismos términos en los que se definió el modelo". Así, se puede decir que un modelo es una simplificación de un sistema que proporciona información sobre el mismo en el contexto de unos ciertos objetivos.

Un sistema puede estar representado por uno o más modelos, cada uno de ellos centrado en representar un determinado aspecto de interés con un cierto grado de detalle (nivel de abstracción). Así, es posible crear modelos de análisis, de diseño, e incluso modelos de implementación, muy próximos a la plataforma de desarrollo que se empleará para codificar el sistema final. Los modelos de más alto nivel pueden evolucionar a otros de más bajo nivel mediante la aplicación de transformaciones automáticas de modelos, hasta obtenerse un modelo lo suficientemente detallado como para poder generar, a partir de él, el código final del sistema. Así, el uso de modelos y de transformaciones modelo-a-modelo y modelo-a-código permiten automatizar, en gran medida, el proceso de desarrollo de software.

Para definir cualquier modelo es necesario contar con un lenguaje de modelado. Los meta-modelos definen formalmente la sintaxis abstracta de los lenguajes de modelado [14], recogiendo sus conceptos (palabras del lenguaje), así como las reglas que indican cómo se pueden combinar dichos conceptos para formar modelos válidos. De este modo, como se indica en [13], "un modelo solamente será válido si es conforme a su meta-modelo". Por otra parte, la representación (ya sea gráfica o textual) asociada a cada uno de los elementos del meta-modelo (conceptos y relaciones) constituye la sintaxis concreta del lenguaje.

Por último, la semántica asociada a los elementos de cada meta-modelo, esto es, su significado o interpretación, tiene impacto fundamentalmente en las transformaciones de modelos, en las que se define cómo transformar (interpretar) cada concepto del modelo de partida, en términos otro lenguaje (ya sea de modelado o de programación).

Actualmente, el DSDM es uno de los paradigmas de desarrollo de software más en boga en el ámbito de la Ingeniería del Software. Los fundamentos sobre los que asienta el DSDM fueron establecidos hace ya un par de décadas. Sin embargo, el auge de este enfoque sólo ha sido posible en los últimos años gracias, por una parte, a la encomiable labor de estandarización que, en el ámbito del DSDM, ha llevado a cabo la OMG (*Object Management Group*) con su iniciativa MDA (*Model-Driven Architectura*) [15], y por otra, a la aparición en el mercado de las primeras herramientas que dan soporte a este nuevo enfoque, permitiendo explotar todo su potencial.

Entre las herramientas que actualmente dan soporte al DSDM, cabe mencionar las siguientes: DSL Tools (de Microsoft) [16], Meta-Edit+ (de la empresa *Meta-Case*) [17] y Eclipse. En los últimos años, Eclipse se ha convertido en el estándar de facto para la comunidad de DSDM, ya que implementa las principales tecnologías estandarizadas por el OMG para dar soporte a este enfoque.

3.3 Descripción de las funcionalidades de HuRoME

A continuación se describen las tres herramientas que conforman HuRoME (ver Figura 3.1), todas ellas desarrolladas utilizando las facilidades para DSDM que ofrece la plataforma Eclipse. En primer lugar, la Sección 2.1 describe la herramienta gráfica de modelado y validación de coreografías, implementada con GMF (*Graphical Modeling Framework*) [18]. A continuación, la Sección 2.2, describe la transformación modelo-a- texto (M2T), implementada con JET (*Java Emitter Templates*) [19], para generar código RoboBASIC y/o RoboScript [20] a partir de los modelos anteriores. Por último, la Sección 2.3, describe la transformación texto-a-modelo (T2M), implementada con Xtext [21] y ATL (*ATLAS Transformación Lenguaje*) [22], para obtener modelos a partir de código RoboScript.

3.3.1 Editor gráfico de modelos

HuRoME ofrece una herramienta gráfica de modelado que permite diseñar, en un entorno amigable e intuitivo, coreografías de movimientos para robots Robonova. Así, cualquier usuario puede especificar, de manera totalmente 'visual', las coreografías del robot sin necesidad de conocer su lenguaje de programación.

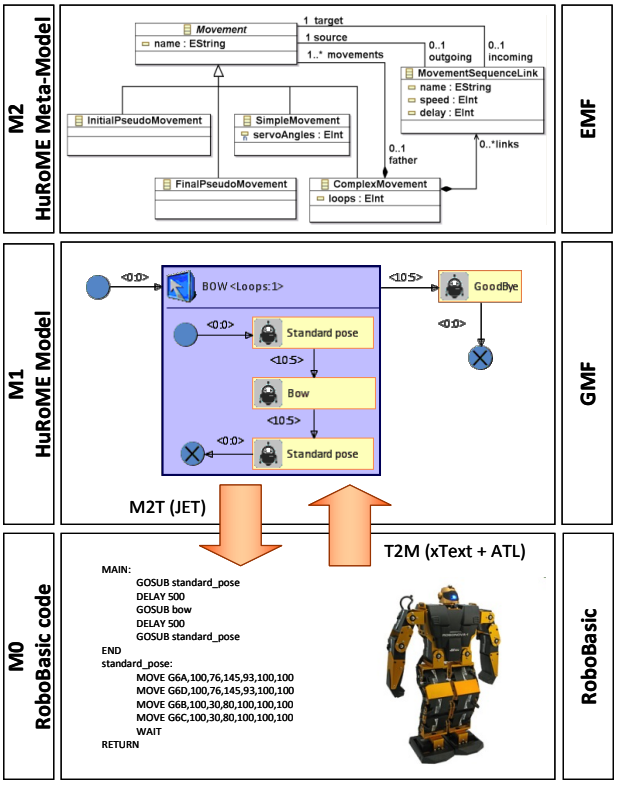

Figura 3.1.: Herramientas integradas en HuRoME (fuente [2])

El meta-modelo sobre el que se ha construido esta herramienta, incluye los conceptos necesarios para modelar las coreografías de movimientos de los robots Roblonaba, en particular: (1) *SimpleMovement*, modela los cambios de postura logrados mediante el accionamiento de uno o más de sus actuadores mecánicos; (2) *ComplexMovement*, modela la composición jerárquica de movimientos; (3) *PseudoMovement*, modela los puntos de control que establecen el inicio y el final de cada secuencia; y (4) *MovementSequenceLink*, enlaza parejas de movimientos indicando el orden en que éstos se ejecutan. Entre los atributos asociados a cada uno de estos conceptos, encontramos los valores angulares de las articulaciones en cada SimpleMovement, el número de veces que se repite cada *ComplexMovement*, o el retardo y la velocidad asociada a los *MovementSequenceLink*.

La herramienta de modelado proporciona una sintaxis concreta (en este caso gráfica), para cada uno de los conceptos del meta-modelo (sintaxis abstracta). En la Figura 3.2 se muestra el editor gráfico de modelos desarrollado como parte de HuRoME. Esta editor consta de tres partes: (1) una paleta de herramientas (panel derecho) desde la que el usuario puede seleccionar los conceptos que desea incorporar a sus modelos, (2) un área de trabajo (panel central) donde podrá modelar las coreografías de movimientos del robot, y (3) una vista de propiedades (panel inferior) para añadir y completar la información asociada a los distintos elementos del modelo (por ejemplo, el valor angular de cada uno de los servomotores del robot en cada *SimpleMovement*).

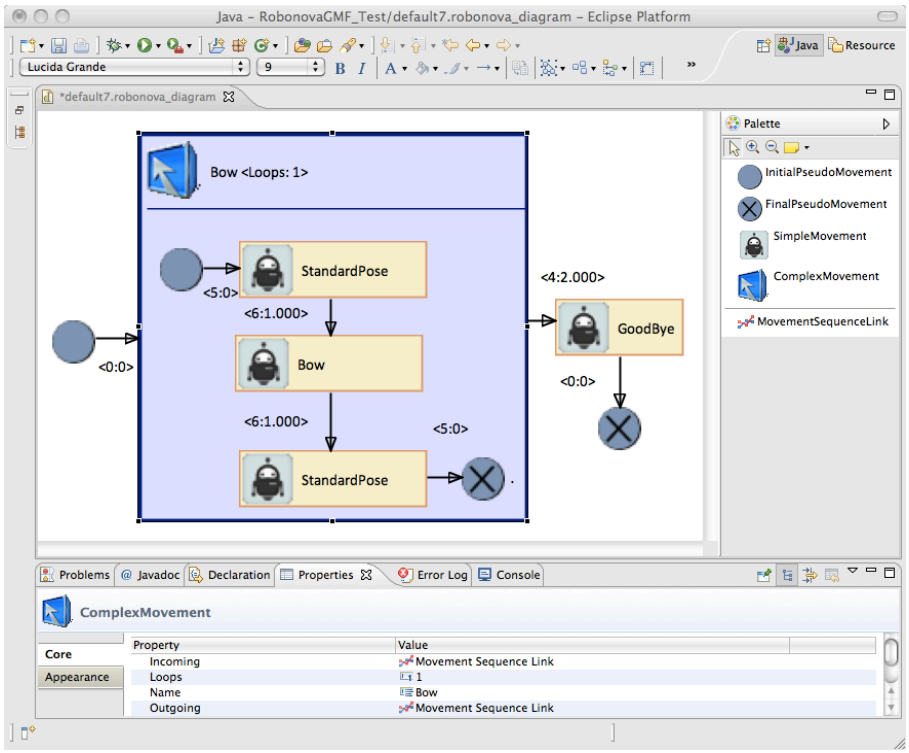

Figura 3.2.: Apariencia del editor gráfico de modelos de HuRoME

La herramienta de modelado, implementada como parte de HuRoME, incorpora facilidades de validación, que permiten comprobar formalmente la corrección sintáctica de los modelos, comprobando si éstos son conformes al meta- modelo y a una serie de restricciones adicionales, escritas en lenguaje OCL y añadidas al editor. Entre las restricciones incluidas en este caso, cabe destacar las siguientes: (1) no se permiten las conexiones reflexivas (conectar un

movimiento consigo mismo); (2) sólo se pueden enlazar movimientos del mismo nivel de composición; (3) el elemento inicial de una secuencia tiene sólo un link de salida y el final sólo un link de entrada; (4) todos los movimientos, simples o compuestos, tienen un único link de entrada y otro de salida, salvo la excepción descrita en 3.

Cuando se detecta una violación de estas reglas o de las recogidas explícitamente en el meta-modelo, los elementos afectados son marcados con un círculo rojo y una cruz (ver Figura 3.2). Sólo cuando los modelos están totalmente libres de errores de validación, puede aplicárseles la transformación que se describe a continuación para generar el código correspondiente para el robot.

3.3.2 Generación de código a partir de modelos

El objetivo de esta herramienta es convertir, mediante una transformación automática modelo- a-texto (M2T), los modelos diseñados y validados con el editor gráfico descrito en la Sección 2.1, en código RoboBASIC que pueda ser ejecutado en un robot Robonova. La Tabla 1 refleja las estructuras del lenguaje RoboBASIC a las que se traducen los distintos conceptos definidos en el meta-modelo descrito en la sección anterior.

El procedimiento de transformación lo dirige el motor de ejecución de JET. Éste recorre el modelo especificando qué plantilla de código se debe invocar para cada tipo de elemento encontrado.

Así pues, según se indica en la Tabla 1, las instancias de la clase *ComplexMovement* son traducidas en bucles que se repiten el número de veces indicado por el atributo *loops*. El código que se incluirá en cada bucle vendrá determinado por el resultado de procesar los elementos contenidos en cada *ComplexMovement*. Por otro lado, los elementos SimpleMovement se transforman en una o más sentencias de movimiento simultáneo de los servomotores, delimitadas con la clave WAIT. De este modo, el robot transiciona a una pose determinada por el valor de la propiedad *servoAngles*. Por último, las instancias de la clase *MovementSequenceLink* se traducen a sentencias que controlan la velocidad y el retardo de los movimientos, de acuerdo a los valores establecidos en las propiedades *speed* y *delay*.

TRANSFORMACIÓN MODELO A CÓDIGO ROBOSCRIPT

Conceptos del dominio	Código RoboScript
ComplexMovement	*Código elementos contenidos repetido loops veces*
SimpleMovement	PTP [ALLON | ALLOFF] MOVE24 *servoAngles*
MovementSequenceLink	SPEED *speed* DELAY *delay*

Tabla 1.: Correspondencia entre los conceptos del meta-modelo y las estructuras de código RoboScript (fuente [2])

3.3.3 Generación de modelos a partir de código

Con el objetivo de soportar el proceso inverso al descrito en el apartado anterior, la versión actual de HuRoME incorpora una transformación texto-a-modelo (T2M) que permite obtener

modelos de coreografías a partir de código RoboBASIC v1.0 ya existente (bien generado automáticamente por la herramienta anterior o codificado manualmente por un desarrollador). En una primera fase, partiendo del código RoboBASIC v1.0, se realiza una transformación T2M para obtener un modelo conforme al meta-modelo de Xtext. Asociado a esta fase se ha desarrollado un editor textual para el lenguaje RoboBASIC, que permite el formateo léxico y la validación sintáctica del código (ver Figura 3.3). En una segunda fase, se ha implementado en ATL una transformación M2M con el fin de obtener modelos conformes al meta- modelo de HuRoME a partir de los modelos Xtext obtenidos en la etapa anterior.

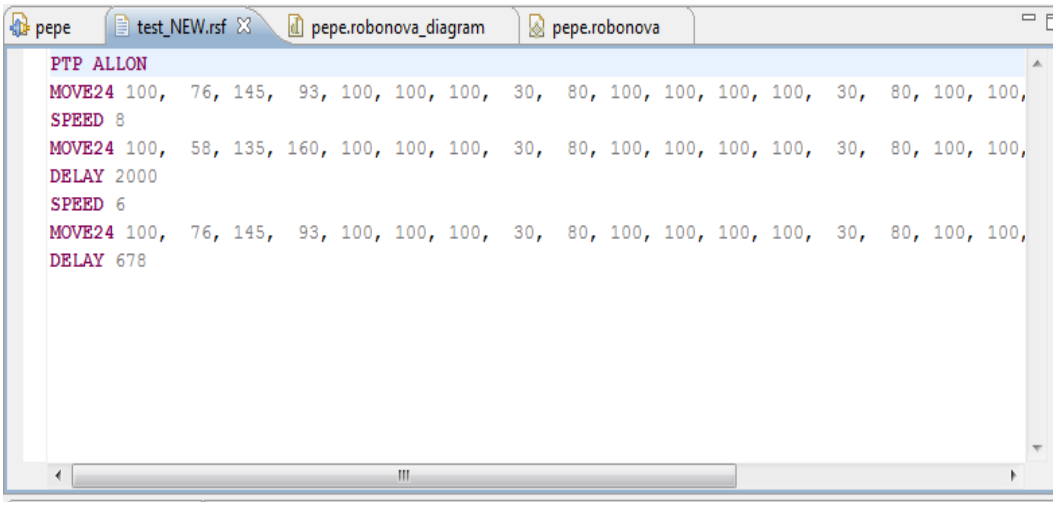

Figura 3.3.: Editor textual XTEXT

3.4 Ejemplo I: obteniendo código desde un modelo

En este apartado ilustramos el uso de la herramienta HuRoME para obtener código RoboBASIC y RoboScript a partir de un modelo de ejemplo de secuencia de movimientos del robot.

3.4.1 Diseño de un modelo

A modo de ejemplo, se propone realizar el modelo correspondiente a una coreografía sencilla. En la secuencia que planteamos, el robot parte de una posición estándar, se encuentra alzado sobre sus piernas y con los brazos relajados, tras ello, se inclinará a modo de reverencia y volverá a su posición, por último, termina levantando un brazo como despedida. Para ello, creamos un nuevo Proyecto Java en Eclipse (File→New→Java Project) y tras ello, un nuevo modelo HuRoME (New→Other→Robonova Diagram). Tras esta operación se han creado dos ficheros, uno con la extensión *.robonova (continente de la información del modelo) y otro con *.robonova_diagram (continente de la información gráfica del diagrama). La Figura 3.2 muestra el ejemplo realizado utilizando el editor gráfico de modelos HuRoME.

3.4.2 Validación del modelo

Validamos el modelo representado en la Figura 3.2 seleccionando la opción *Validate* en el menú principal. Obtendremos un diálogo que nos indica la correcta validación del modelo, en caso de erros obtendremos la información detallada en la vista *Problems*.

3.4.3 Generación de código

Para lanzar la generación automática de código RoboBASIC o RoboScript desde el modelo creado y validado, se han seguido los siguientes pasos:

1. Copiamos el modelo (archivo *.robonova) y lo movemos a la carpeta llamada model localizada en el Proyecto JET (este Proyecto forma parte de la herramienta HuRoME y contiene la implementación de la transformación M2T).

2. Abrimos la ventana Run Configurations (seleccionando Run→Run Configurations). Véase la Figura 3.4. En esta ventana se crea una nueva configuración haciendo doble clic en JET Transformation de la lista (sólo en el caso de que no exista ya una instancia creada) y escribimos la ruta al modelo de entrada de la transformación. Finalmente pulsamos Run.

3. Aparecerá dos ficheros de salida, uno con código roboBASIC (*.bas) y otro con RoboScript (*.rsf), en la raíz del Proyecto (Figura 3.5).

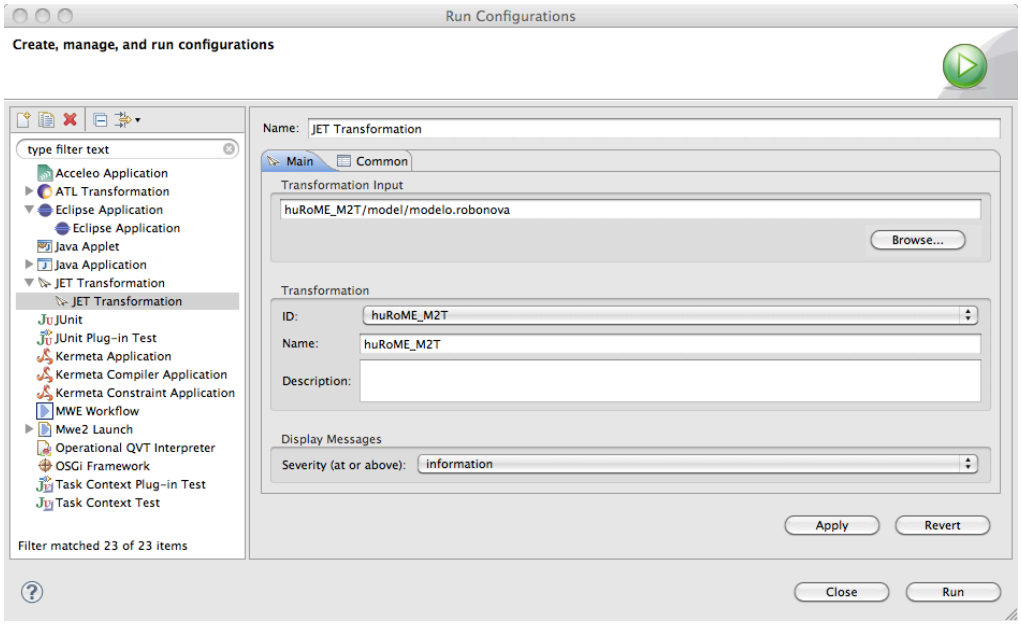

Figura 3.4.: Ventana Run Configurations

3.5 Ejemplo II: obteniendo el modelo desde código RoboScript

En este apartado partimos del código RoboScript generado anteriormente para ilustrar el procedimiento para obtener el correspondiente modelo gráfico, que deberá ser equivalente al de la Figura 3.2.

3.5.1 Serialización del código RoboScript

En primer lugar, tras abrir el código con el editor textual Xtext de HuRoME, ejecutamos el proceso de serialización para obtener la representación del código en formato XMI (extensión *.xmi). En la figura 3.7 puede verse el modelo obtenido desde código RoboScript.

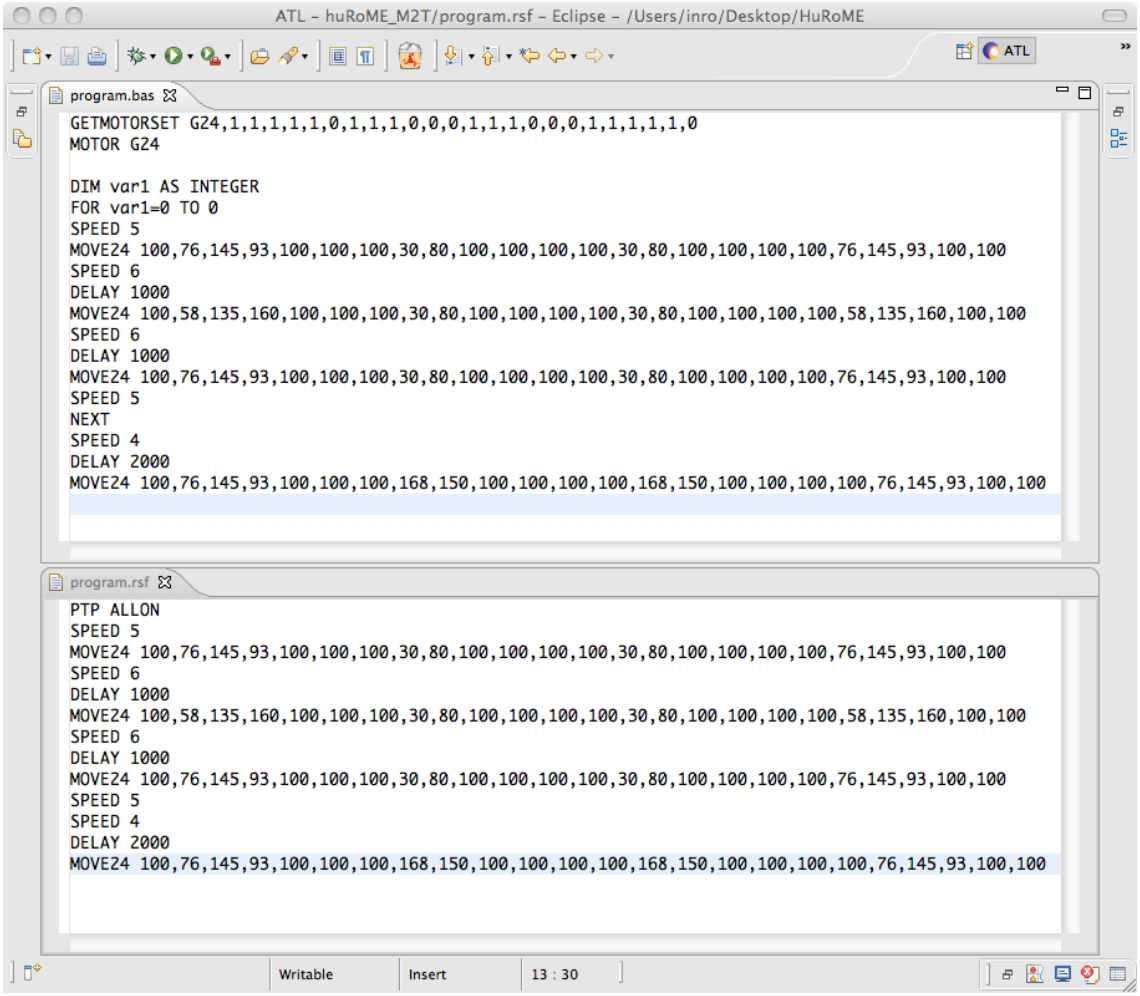

Figura 3.5.: Código generado. La parte superior corresponde a RoboBASIC, la parte inferior a RoboScript

3.5.2 Ejecución de la transformación

Para la obtención del modelo a partir de la representación XMI de código, se siguen los siguientes pasos:

4. Abrimos la ventana Run Configurations (seleccionando Run→Run Configurations).En esta ventana se crea una nueva configuración haciendo doble clic en ATL Transformation de la lista (sólo en el caso de que no exista ya una instancia creada) y revisamos o escribimos la ruta a los meta-modelos y modelos de entrada y salida. Finalmente pulsamos Run.

5. Se creará un nuevo modelo con extensión *.robonova que podrá abrirse con el Tree Editor.

6. Para conseguir una representación gráfica del modelo y poder abrirlo utilizando el editor GMF, debemos copiar el fichero *.robonova en el Proyecto Java que fue creado para albergar el plug-in del editor gráfico. Tras ello, seleccionar el fichero y desplegar su menú contextual haciendo clic sobre la opción “Inicialize robonova_diagram diagram file”, ésta generará forma automática el fichero *.robonova_diagram. En la Figura 3.8 puede verse el modelo del ejemplo en sus distintas representaciones.

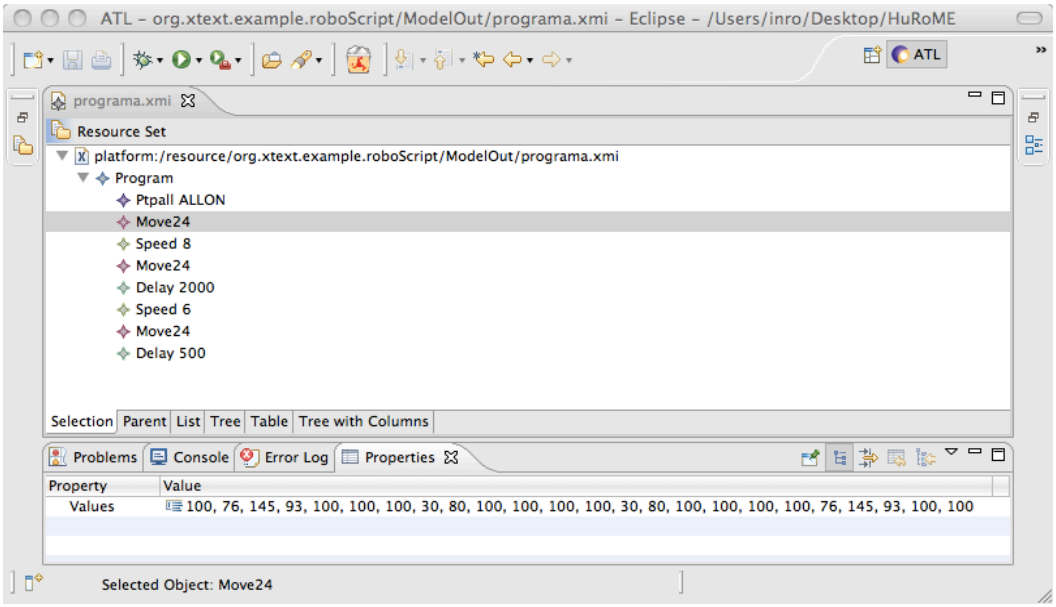

Figura 3.7.: Modelo tras el proceso de serialización

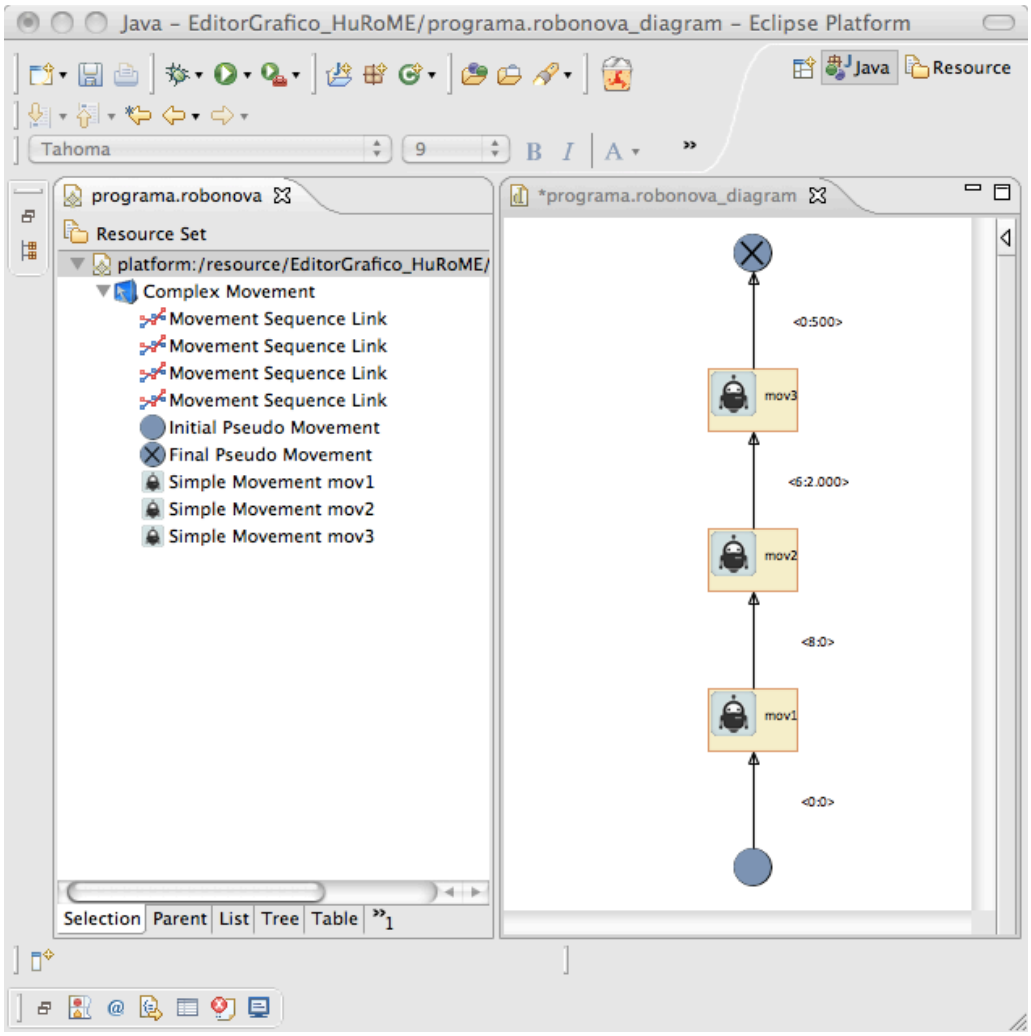

Figura 3.8.: Modelo final tras la transformación ATL

CAPÍTULO IV

Desarrollo de un Plug-in Eclipse para mejorar la integración y operatividad de HuRoME

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

4.1 Retos del desarrollo

En la revisión de la herramienta HuRoME pudimos comprobar que HuRoME es un conjunto de herramientas (editores, transformaciones, etc.) desarrolladas utilizando herramientas diferentes (EMF, JET, GMF, ATL y Xtext). Así, HuRoME 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 HuRoME si no que pretende:

1. Integrar los diferentes editores y transformaciones de HuRoME en un único entorno de trabajo, de manera que se vea mejorada la usabilidad de la herramienta HuRoME.
2. Mejorar la accesibilidad a las funcionalidades de HuRoME proporcionando un entorno más amigable e intuitivo.
3. Facilitar la operatividad de HuRoME, simplificando notablemente el número de operaciones realizadas para la obtención de un mismo fin respecto de HuRoME.
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 HuRoME.

Para alcanzar estos objetivos nos proponemos implementar las siguientes características:

- **Creación de una vista.** Con este instrumento mejoraremos la operatividad y la accesibilidad a HuRoME, todas las operaciones de HuRoME quedarán disponibles en una vista con una interfaz amigable y fácilmente utilizable.
- **Creación de botones en la barra de herramientas.** Facilita la accesibilidad a las funcionalidades de HuRoME.
- **Creación de una perspectiva para HuRoME.** 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.
- **Creación de una ayuda.** Menú de ayuda con documentación sobre la utilización del plug-in desarrollado.
- **Creación de un panel de preferencias.** Para seleccionar diferentes configuraciones de la herramienta HuRoME.

Por último, comentar que el Plug-in se construirá sobre la clase *HuromeTools* cuyos métodos conforman un API de alto nivel para acceder a las funcionalidades de HuRoME (ver Anexo I).

4.2 Creación de una vista para HuRoME

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

4.2.1 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 HuRoME. En la Figura 4.1 muestra el aspecto final de la vista desarrollada que representa un diagrama de flujo con el proceso de transformación que siguen los modelos en HuRoME. Se han considerados dos posibles caminos, por un lado, siguiendo la numeración de izda. a dcha., (1) se abre un modelo de secuencia de movimientos del robot, (2) es validado y (3) se aplica una transformación M2T a código RoboBASIC y/o RoboScript. Por otro lado, siguiendo la numeración de dcha. a izda., (1) se abre un fichero con código RoboScript, (2) es validado y (3) se aplica una transformación T2M obteniendo el modelo de secuencia de movimientos del robot. La Figura 4.2 muestra la especificación básica de este proceso como diagrama de estados.

Figura 4.1.: Vista HuRoME

4.2.2 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 *Plug-in Development*, pulsamos sobre *Plug-in.xml* en la raíz del Proyecto y nos dirigimos a la pestaña dependencias sonde 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 4.3).
2. **Adicción de la extensión necesaria para el funcionamiento del Plug-in.** Como muestra la Figura 4.4 añadimos la extensión en la pestaña *Extensions* del Proyecto y añadimos *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 (Figura 4.5). 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 HuRoME (Figura 4.5). 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* [23] desarrollando así un diagrama de clases en el que la apariencia de la vista queda desacoplada (Figura 4.6). Como podemos observar la vista HuRoME Plug-in (Figura 4.1) abarca todas las funcionalidades de HuRoME de manera intuitiva y dando transparencia al usuario respecto de la herramienta. Tenemos opciones de abrir un modelo o un código, mostrar el nombre del archivo sobre el cual se van a ejecutar las operaciones, validarlo, si valida realizar la correspondiente transformación M2T o T2M.

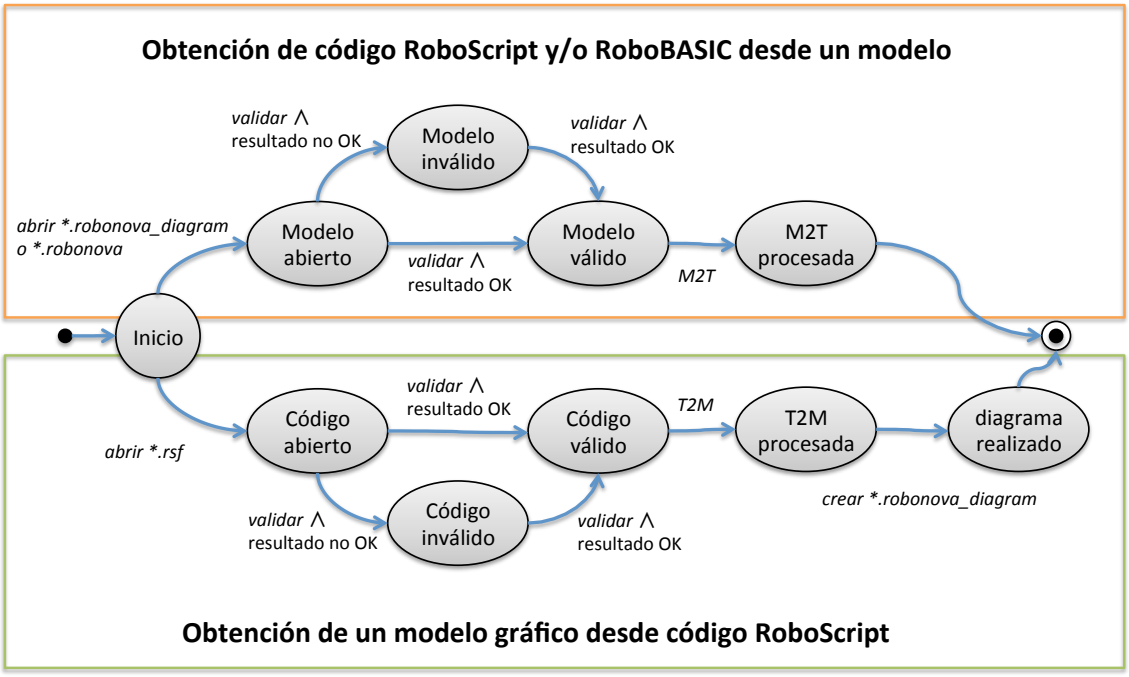

Figura 4.2.: Diagrama de estados que define el proceso de transformación en HuRoME

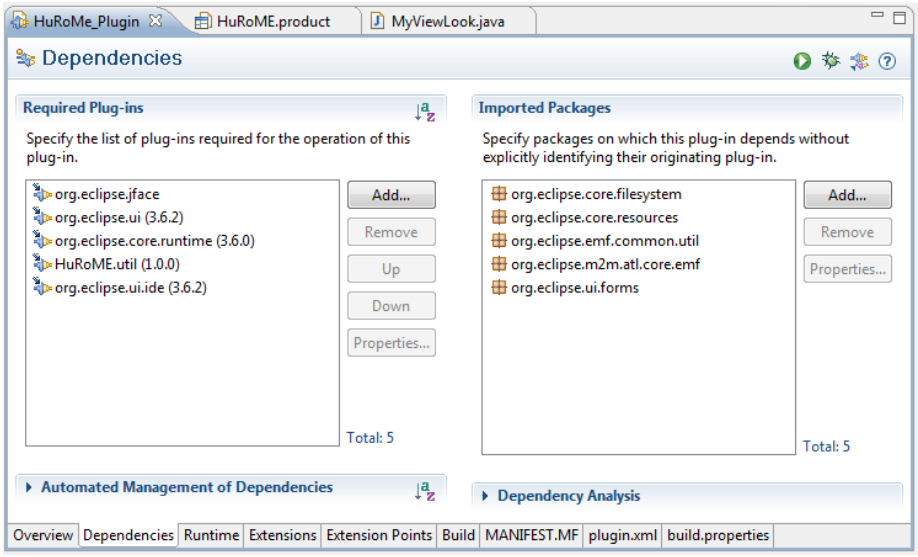

Figura 4.3.: Sección dependencias de HuRoME_Plug-in

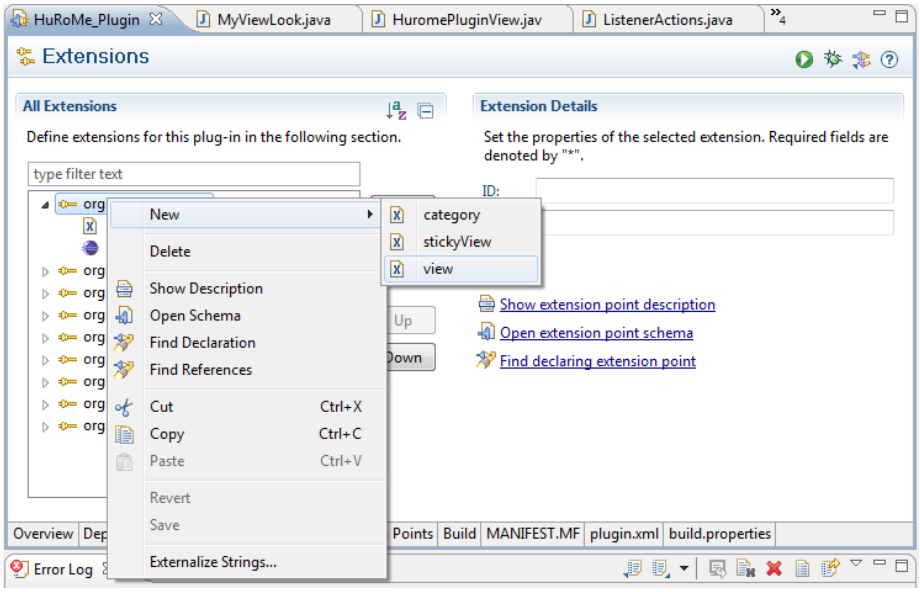

Figura 4.4.: Extensión org.eclipse.ui.views

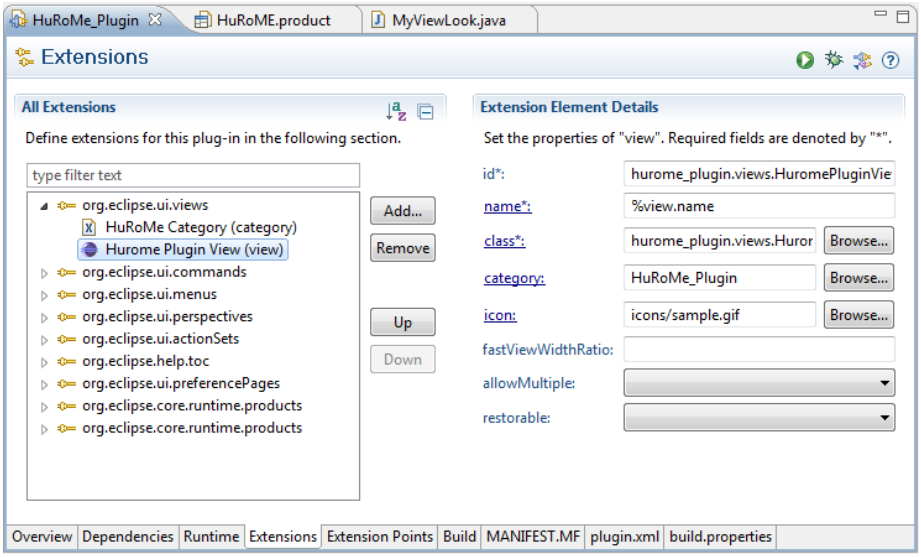

Figura 4.5.: Extensión org.eclipse.ui.views

4.2.3 Detalles del desarrollo

El diagrama UML de clases necesario para la implementación de la vista ha sido desarrollado siguiendo el patrón *Modelo-Vista-Controlador* (Figura 4.6) de manera que se desacopla la apariencia gráfica y las funcionalidades de la vista. Tenemos la clase principal de la vista HuromePlug-inView, que hereda a su vez de *ViewPart*, en la cual se crea una nueva apariencia con un objeto MyViewLook y seguidamente la inicializamos con un objeto *ListenerActions*, todo esto debe ser creado dentro del método *public void createPartControl(Composite parent)* el cual es el método que gobierna la vista, todo lo referente a ella debe ser creado o referenciado dentro de este método. La clase *ListenerAction* implementa la interfaz *SelectionListener*. Esta clase contiene el código que define la máquina de estados basada en el diagrama mostrado en la Figura 4.2. El método *public void widgetSelected (SelectionEvenet e),* donde realizamos el tratamiento de eventos que ocurren sobre los botones de la vista y realiza operaciones según que botón recibió el evento, estas operaciones, en realidad, gestionan las transiciones de los estados y mensajes de error o de confirmación según si la operación fue satisfactoria o no lo fue. Está compuesta por un objeto de la clase *CommonActions* que es la clase la cual recoge acciones comunes como transformaciónM2T o transformaciónT2M entre otras que son reutilizadas en la clase *MenúActions* que gobierna el menú de la barra principal y el menú del explorador de paquetes. La clase *AbstractViewLook* es una abstracción de la interfaz gráfica que nos presenta la vista, es decir, es la clase donde se crean los botones y etiquetas y además se crean las configuraciones de apariencia que mostrará la vista según el estado en el que se encuentre. La clase *MyViewlook* hereda de *AbstractViewLook* e implementa la interfaz *PaintListener* donde implementamos el método *public void PaintControl(PaintEvent pe)* el cual es el encargado de capturar cualquier variación en el tamaño de la vista y actuar en consecuencia recalculando las posición de todos sus elementos con el fin de que no sufran variación alguna en la distribución sobre la misma y es la encargada de personalizar la imagen de fondo que tendrá nuestra vista, colocar el texto y la creación e ubicación de botones y etiquetas en la vista.

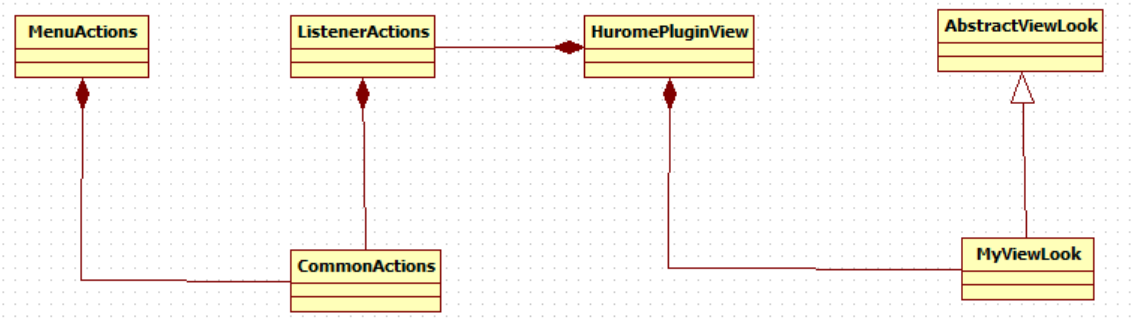

Figura 4.6.: Diagrama de clases UML de la vista HuRoME

4.3 Accesibilidad a las funcionalidades de HuRoME

En este punto veremos como a parte de la vista HuRoME, también utilizando el mecanismo de extensión, hemos creado un menú HuRoME 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.

4.3.1 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 HuRoME (véase Figura 4.7). 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.

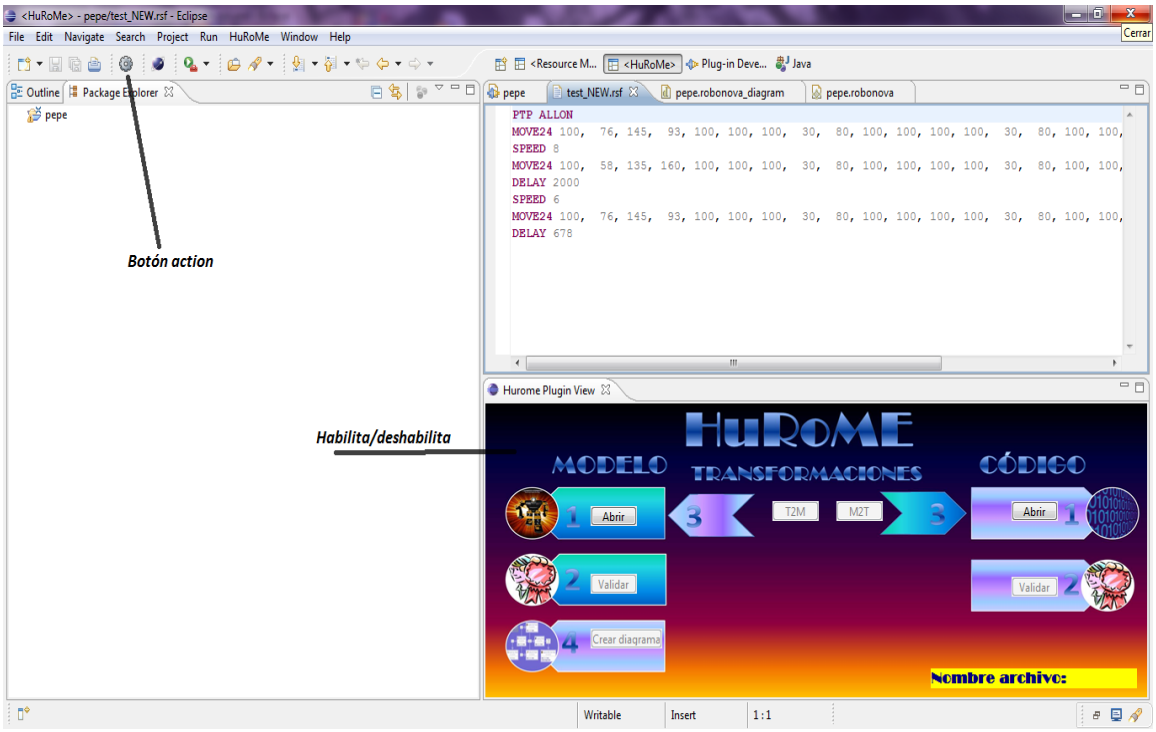

Figura 4.7.: Botón action

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 4.8.
3. **Configuración de la extensión.** Con el botón derecho sobre la extensión pulsamos New→ActionSet (Figura 4.9), lo rellenamos escribiendo "HuRoME_Plug-in.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 4.10. Campos a tener en cuenta: (1) el identificador que hemos rellenado con "HuRoME_Plug-in.actionSet.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, (4) el campo *class* donde pulsamos y creamos la clase Java que gobierna el *action*.

La clase la hemos llamado *HuromeActionDelegare* 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 *action* que habilita/deshabilita la vista HuRoME.

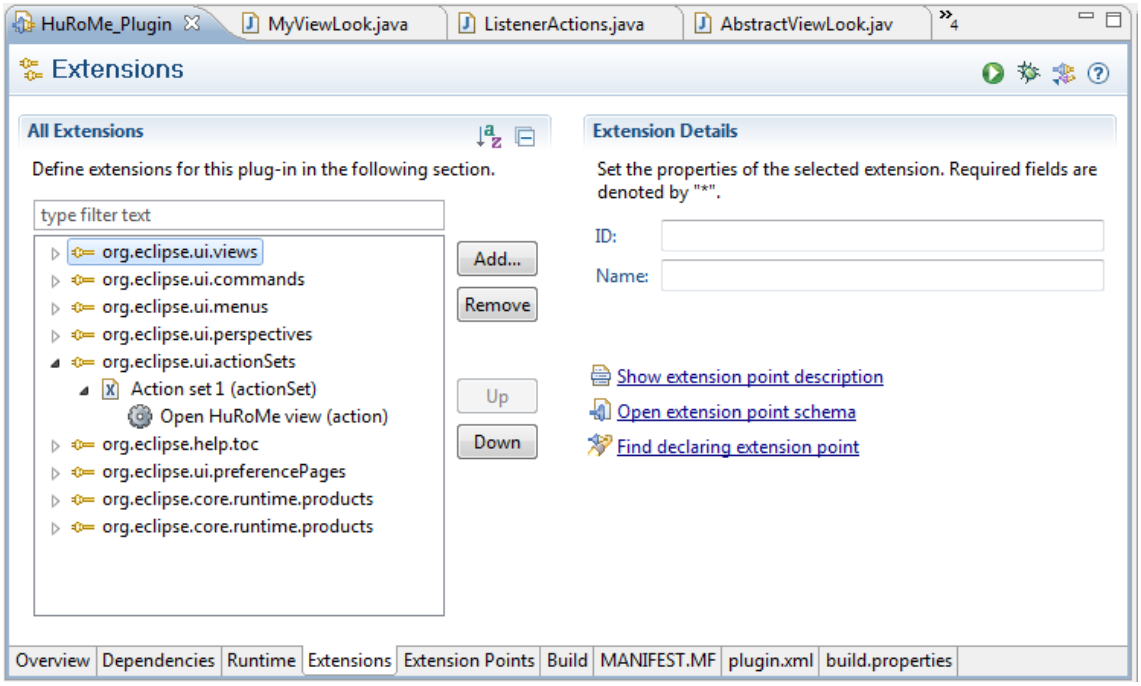

Figura 4.8.: Adicción de la extensión

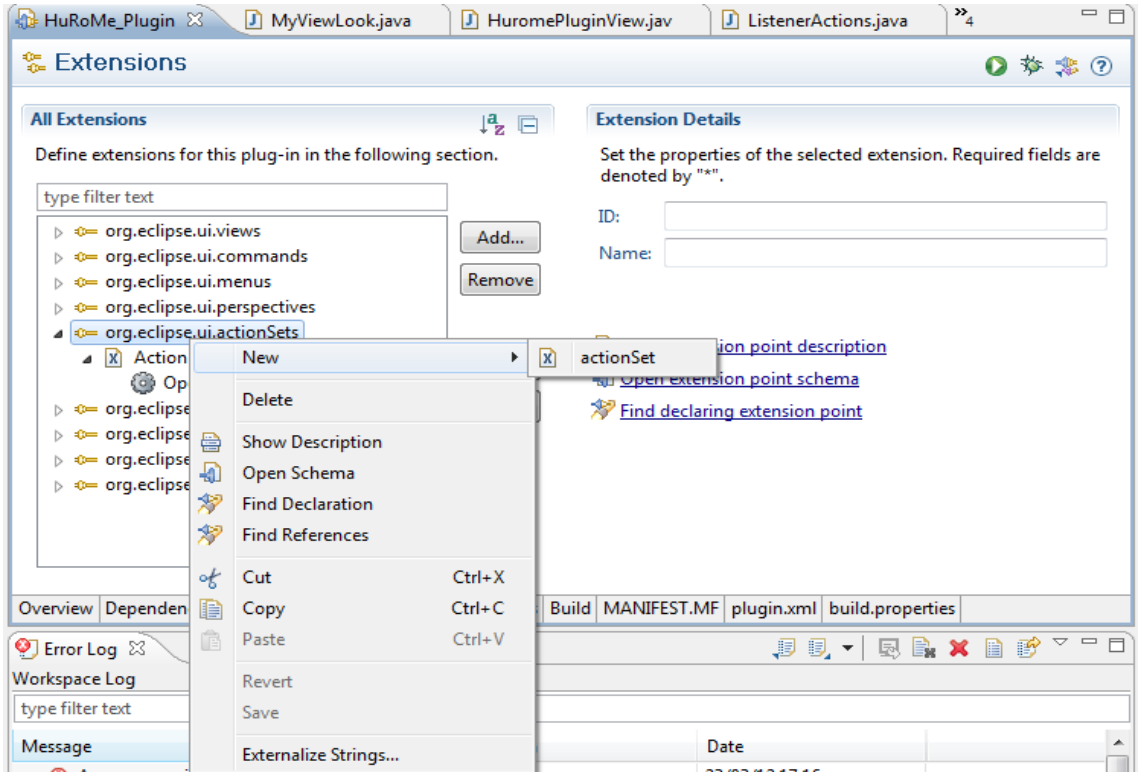

Figura 4.9.: Adición actionSet

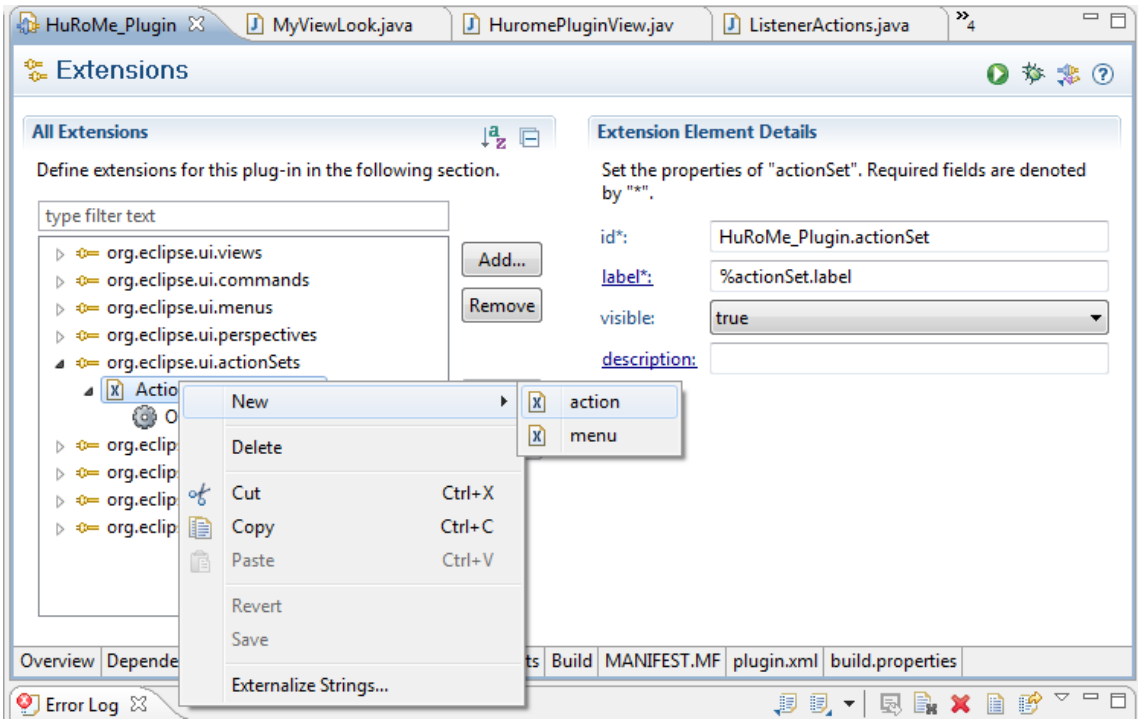

Figura 4.10: Adición action

4.3.2 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 HuRoME.

4.3.2.1 Descripción inicial

El objetivo es crear un menú llamado HuRoME en la barra principal que permita aplicar las transformaciones modelo-a-texto (M2T) y texto-a-modelo (T2M) sobre el archivo que se encuentre actualmente abierto en el editor. En la Figura 4.11 puede observarse la disposición de este menú.

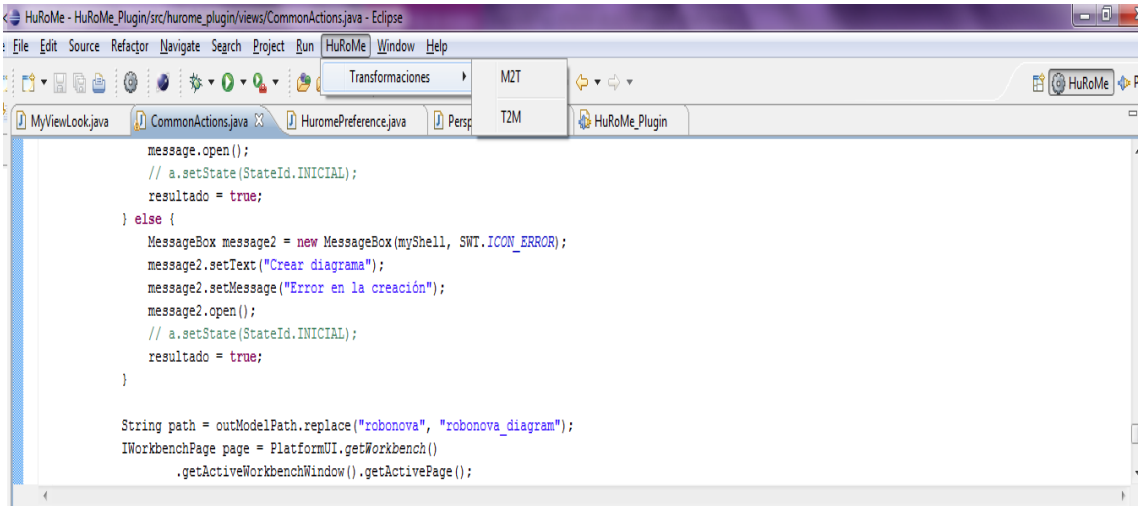

Figura 4.11.: Menú HuRoME

Las operaciones para validar el modelo y el código son realizadas de forma transparente al usuario, de tal manera que cuando ejecutamos la operación de transformación desde este menú, se valida previamente el modelo o el código mostrando un mensaje con el resultado de la operación, en el caso de obtener un resultado negativo la transformación el proceso se interrumpe y la transformación no se ejecuta.

4.3.2.2 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 HuRoME debemos añadir (Figura 4.12) 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 cuatro nuevos *commands* (New→Command), dos serán para el explorador de paquetes que veremos más adelante y los otros dos para nuestro menú en la barra principal.

Como podemos observar en la Figura 4.13 hemos añadido cuatro *command*, en el campo *id* de los correspondientes al explorador de paquetes hemos añadido M2T y T2M y en los correspondientes a los de la barra de herramientas hemos añadido *M2T_ToolBar* y *T2M_ToolBar* con el fin de diferenciarlos a la hora de programar a través de ese campo (campo *id*).

4.3.2.3 Detalles de desarrollo

En el diagrama UML expuesto anteriormente (Figura 4.6) podemos apreciar la existencia de una clase llamada *MenuActions* la cual al igual que *ListenerActions* posee un objeto del tipo *CommonActions*, esto es debido a que nuestro menú de la barra principal tanto como el menú del explorador de paquetes tienen que realizar las operaciones comunes como validar modelo, validar texto, transformación modelo-a-texto y texto-a-modelo.

Una vez creado los *command* seleccionamos *M2T_TololBar* y *T2M_ToolBar* y accedemos a los campos *defaultHandler* donde le diremos que la clase a ejecutarse cada vez que presionen una de estas opciones será *MenuActions*, clase la cual hereda de *AbstractHandler* y donde implementamos el método *public Object execute (ExecutionEvent event)* y capturamos la excepción con *throws ExecutionException*.

Dentro del método *execute* identificamos mediante el campo id que elemento del menú produjo el evento y al tener un objeto de la clase *CommonActions* realizamos las operaciones pertinentes a la acción. Para que toda la funcionalidad de los *command* quede anclada a un menú tenemos la extensión *org.eclipse.ui.menus*. Para la creación del menú debemos crear un menuContribution como muestra la Figura 4.14 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 HuRoME, 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 un nuevo *command* (New→Command) el cual llamamos M2T y en el campo *commandId* le pasamos el identificador del *command* previamente creado en la extensión *org.eclipse.ui.commands*, concretamente le pasamos el M2T_toolBar, a continuación creamos un separador con New→Separator y en el campo *visible* establecemos *true* y para terminar creamos un nuevo *command* el cual llamamos T2M y le pasamos en el campo *commandId* el identificador previamente creado *T2M_ToolBar*. Añadiendo los *id* de los *command* a los *command* creados en la extensión *org.eclipse.ui.menus* ya tendríamos acabado, como muestra la Figura 4.11, el menú HuRoME en la barra principal.

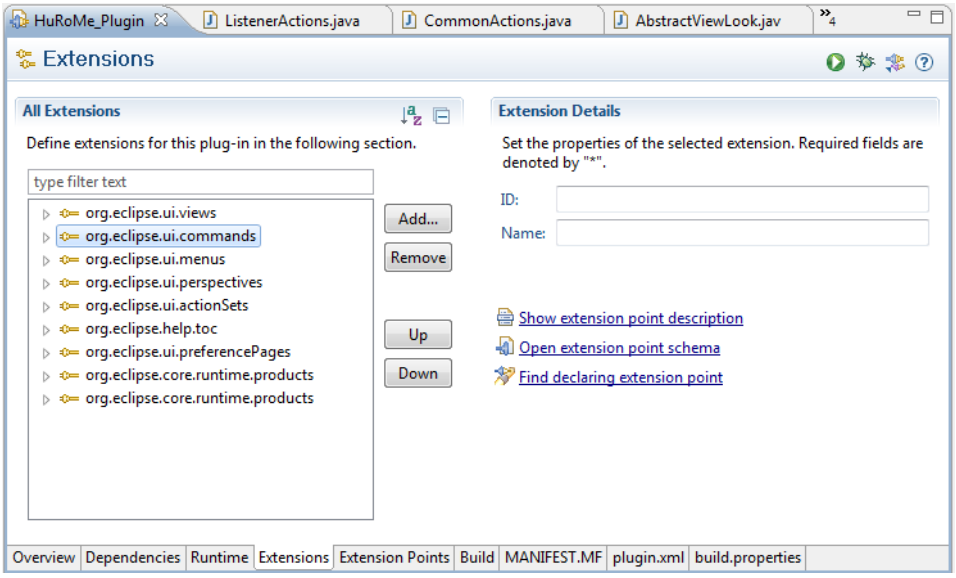

Figura 4.12.: Extensiones org.eclipse.ui.commands y org.eclipse.ui.menus

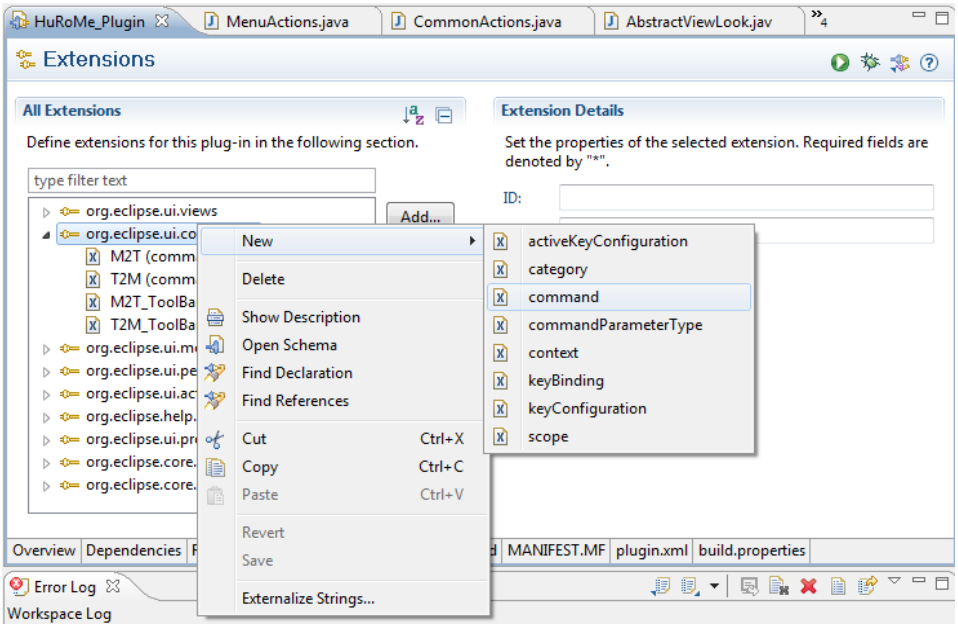

Figura 4.13.: Extensiones org.eclipse.ui.commands y org.eclipse.ui.menus

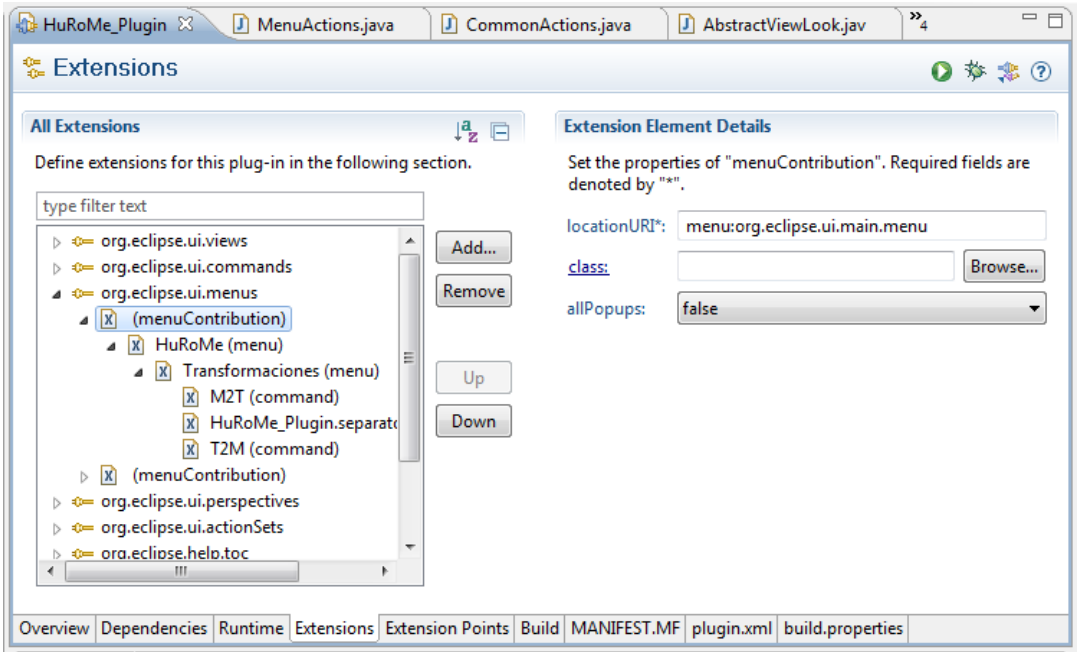

Figura 4.14: Árbol de menuContribution.

4.3.3 Adicción del menú contextual en el explorador de paquetes

El objetivo es poder realizar las operaciones HuRoME sobre el archivo que seleccionemos en el explorador de paquetes (Figura 4.15), así la manera de proceder será muy similar a la creación del menú HuRoME en la barra principal, al seleccionar un archivo en el explorador de paquetes podremos realizar las operaciones transformar modelo-a-texto (M2T) y texto-a-modelo (T2M). Al igual que en la sección 4.3.2, el proceso de validación se realizará de forma transparente cuando se ejecute una transformación. 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 como muestra la Figura 4.16 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 dos nuevos *command* llamados M2T y T2M en el campo identificador (*id*). En el campo *defaultHandler* le decimos que la clase será *MenúActions*, descrito su funcionamiento anteriormente. Seleccionando botón derecho sobre *org.eclipse.ui.menus* creamos un nuevo *menuContribution* dentro del cual creamos un menú que llamamos HuRoME, 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 M2T y le pasamos

el identificador (campo *id*) creado en la extensión *org.eclipse.ui.commands*, creamos un separador para las transformaciones con New→Separator y para finalizar creamos el *command* T2M al cual le pasamos el identificador T2M creado en la extensión mencionada, todo debe quedar como la Figura 4.17. El resultado es un menú HuRoME con sus funciones en el explorador de paquetes (Figura 4.15).

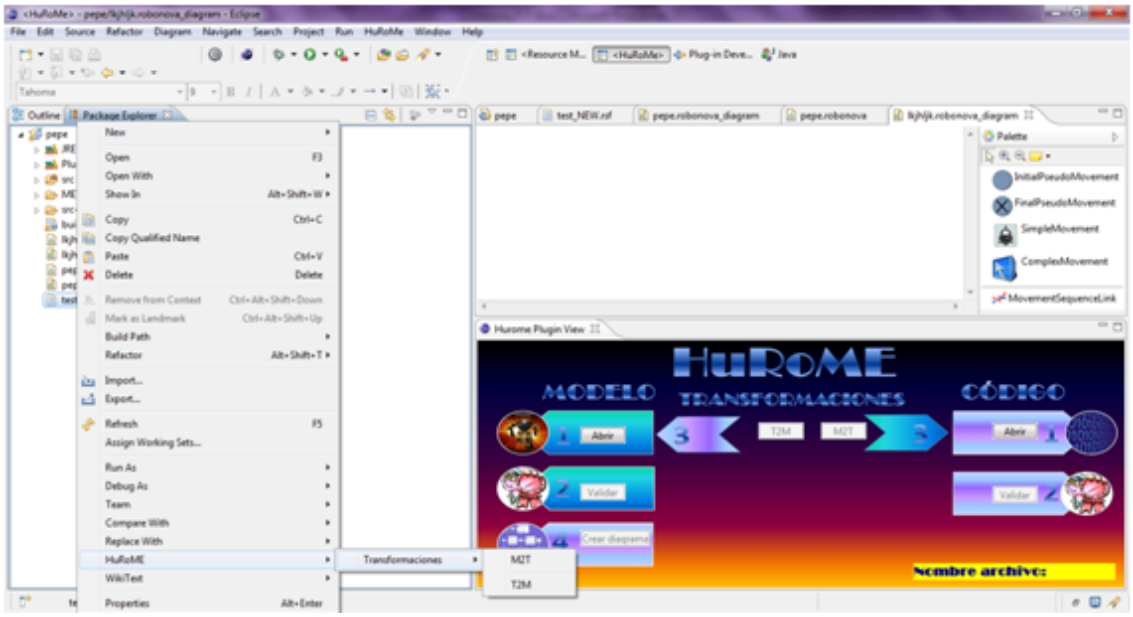

Figura 4.15.: Menú en el Package Explorer

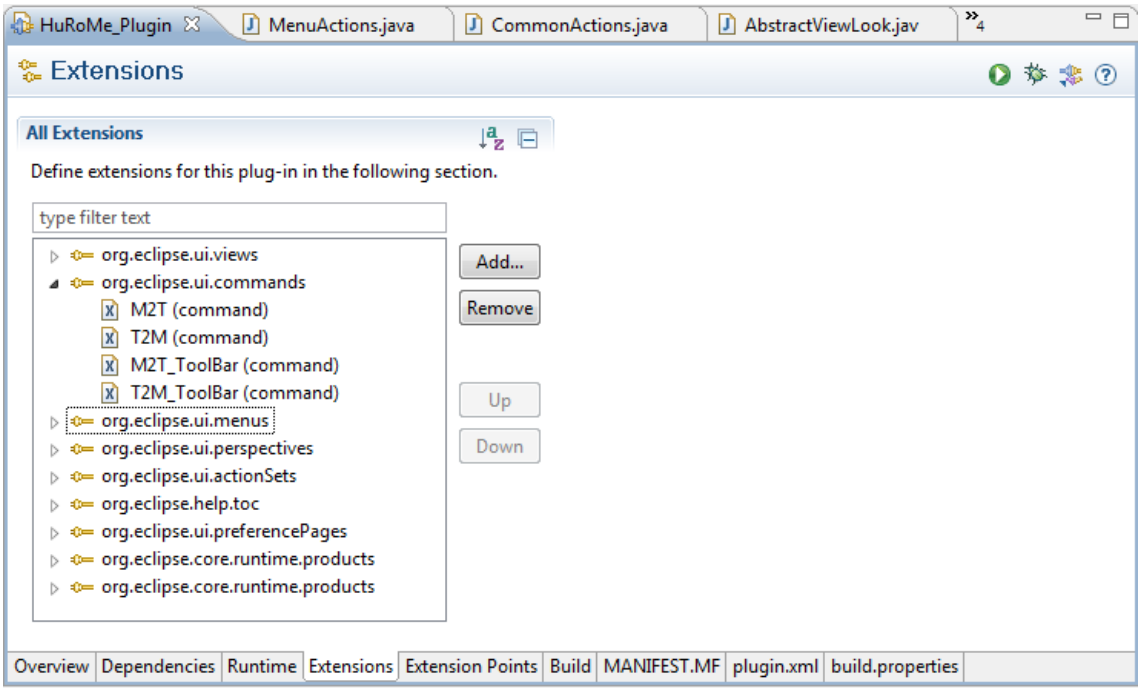

Figura 4.16.: Extensiones previas añadidas

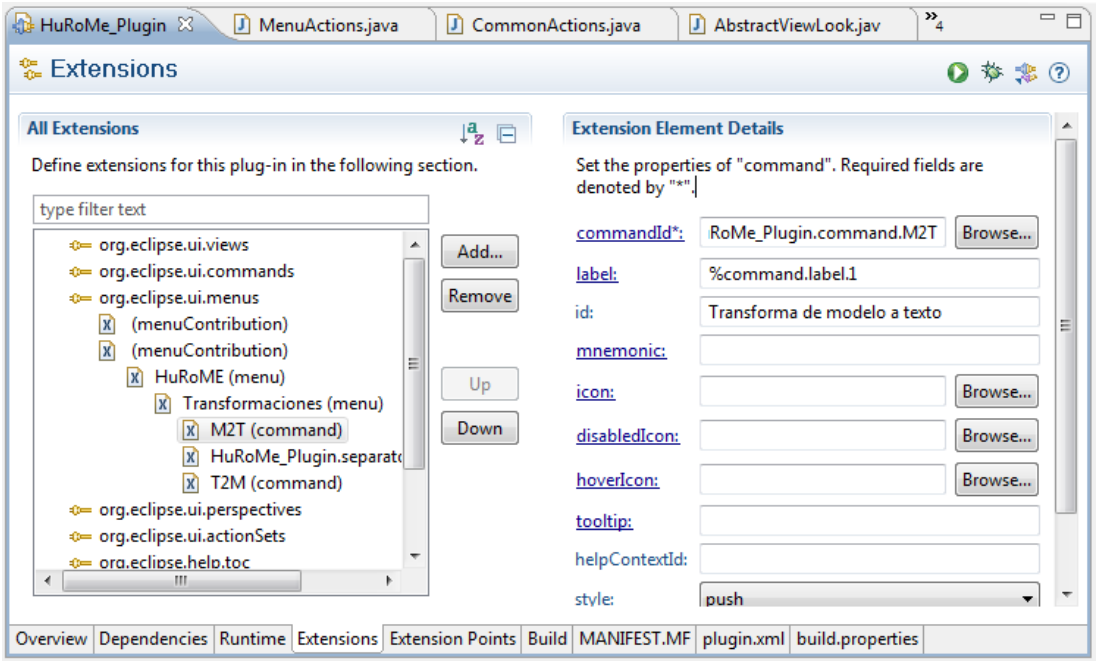

Figura 4.17.: Árbol menú Package Explorer

4.3.3 Creación de un panel de preferencias

El objetivo es crear un panel de preferencias en el que un usuario pueda establecer ciertas configuraciones de la herramienta. La Figura 4.18 muestra el panel de preferencias desarrollado, según puede observarse, éste permite (1) seleccionar el fichero de salida de la transformación M2T, (2) habilitar la apertura automática del archivo resultante de las transformaciones en el correspondiente editor y (3) crear automáticamente el diagrama tras finalizar la transformación T2M.

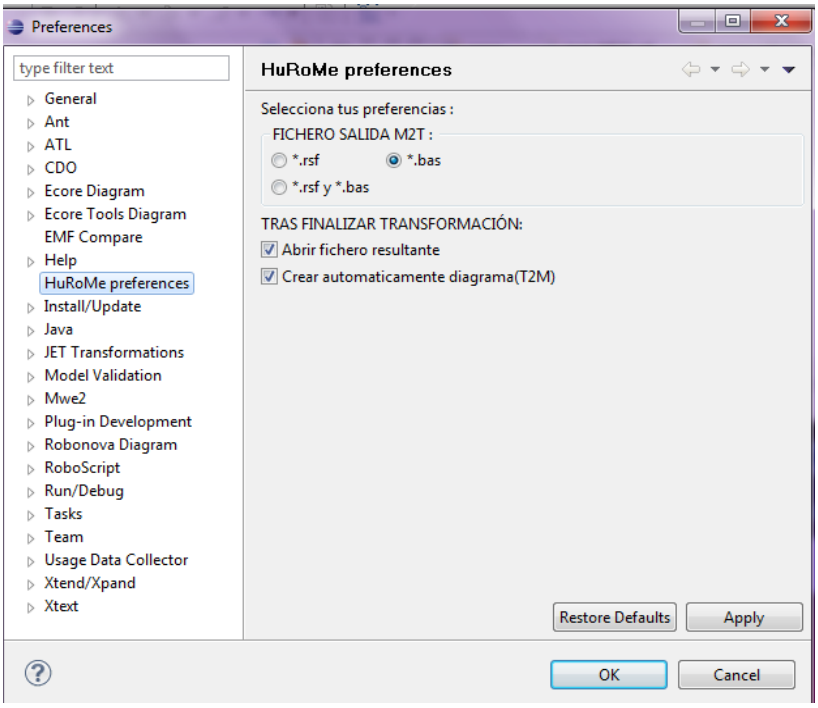

Figura 4.18.: Panel de preferencias HuRoME

Para la creación del Plug-in realizamos las siguientes operaciones:

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 la creación de dicho panel debemos añadir la extensión *org.eclipse.ui.preferencePages* en la pestaña *Extensions* del Plug-in.xml como muestra la Figura 4.19. En el campo *id* añadimos el identificador de la preferencia, en este caso HuRoME_Plug-in.PreferencePage y en el apartado *class* pinchamos sobre el enlace y escribimos la clase que controla la preferencia la cual hemos nombrado como *HuromePreference*.
3. **Creación de la clase que gobierna el Plug-in.** HuromePreference clase hereda de *FieldEditorPreferencePage* e implementa la interfaz *IWorkbenchPreferencePage* por lo que debemos implementar el método *createFieldEditors()*, en este método es donde añadimos los ítems que mostraremos en el panel de preferencias, en nuestro panel hemos añadido un *RadioGroupFieldEditor* que se encarga se seleccionar la preferencia del fichero de salida tras la transformación modelo a texto y dos booleanos que se encargan de gestionar la apertura y creación del diagrama tras finalizar las operaciones.

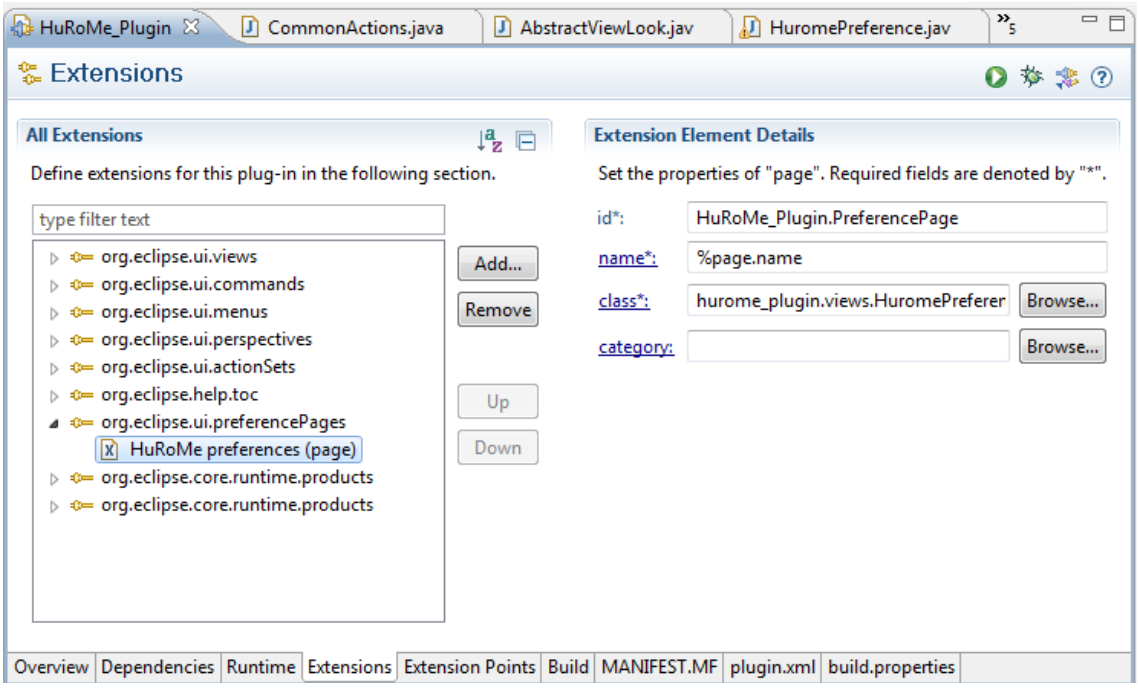

Figura 4.19.: Adición de la extensión org.eclipse.ui.preferencePages

4.3.4 Creación de un menú de ayuda.

El objetivo es crear un menú de ayuda Help → Help Contents (tal y como se muestra en la Figura 4.20) de manera que el usuario de la herramienta HuRoME pueda consultar la documentación para solventar problemas o aclarar dudas.

Para la realización de Plug-in realizamos los 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.** Como muestra la Figura 4.21 añadimos la extensión *org.eclipse.help.toc* en la pestaña *Extension* del Plug-in, a continuación observamos que tenemos dos archivos llamados *toc.xml* y *tocgettingstarted.xml* creados.
3. **Creación de la clase que gobierna el Plug-in.** El archivo *toc.xml* representa la tabla primaria del contenido de los archivos. Si expandimos los archivos *toc.xml* y *toggettingstarted.xml* podemos observar que están compuestos por sub-apartados con campos como *anchorID* que es el identificador del recurso, o *location* donde se especifica que archivo html será visualizado cuando el usuario acceda a esa sección de la ayuda, pulsando la opción *add Topic* podemos añadir un nuevo recurso a la ayuda donde añadir un nuevo documento html como muestra la Figura 4.22.

Como vemos en la Figura 4.22 hemos creado tres secciones en la ayuda, una para la vista explicando su uso, otra para el menú HuRoME en la barra de herramientas y la última para el menú HuRoME en el Explorador de Paquetes, con su correspondiente *topic* cada una en el cual hemos rellenado el campo nombre con el texto que queramos aparezca en la ayuda y el campo *location* con el archivo *xml* que mostrará el contenido de la misma. Como vemos en la Figura 4.23 accedemos al menú de ayuda dirigiéndonos a la pestaña Help→Help Contents.

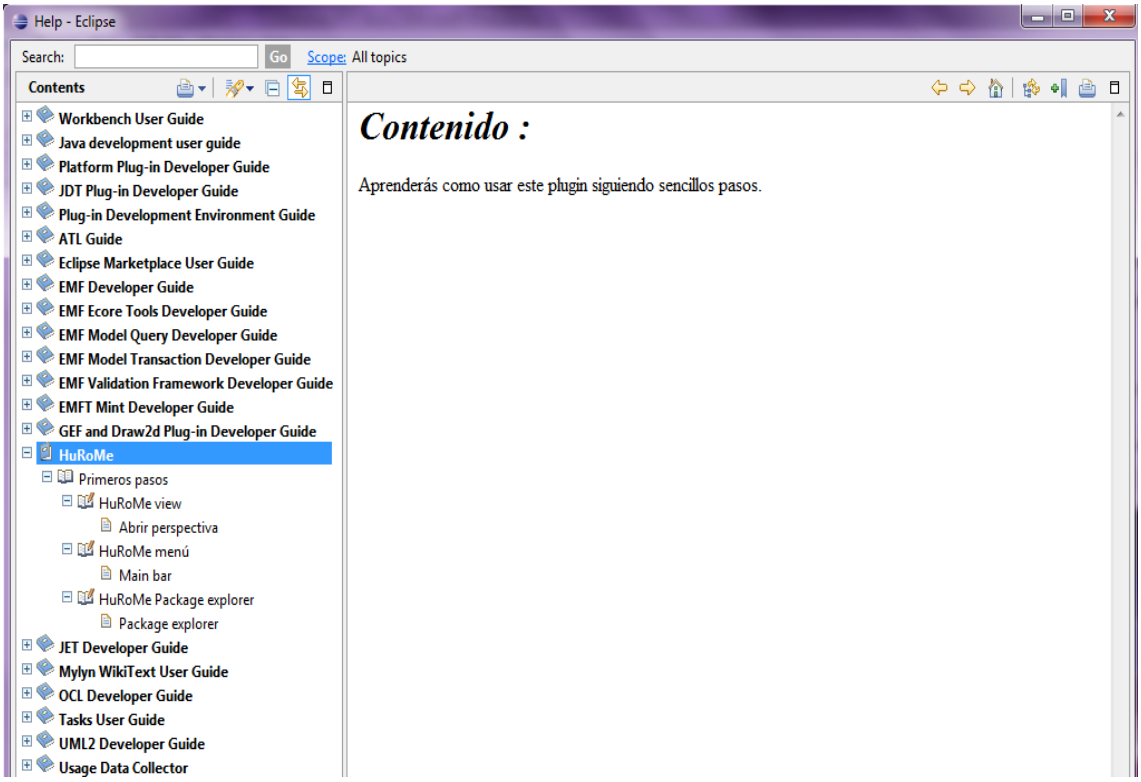

Figura 4.20.: Árbol de Ayuda HuRoME

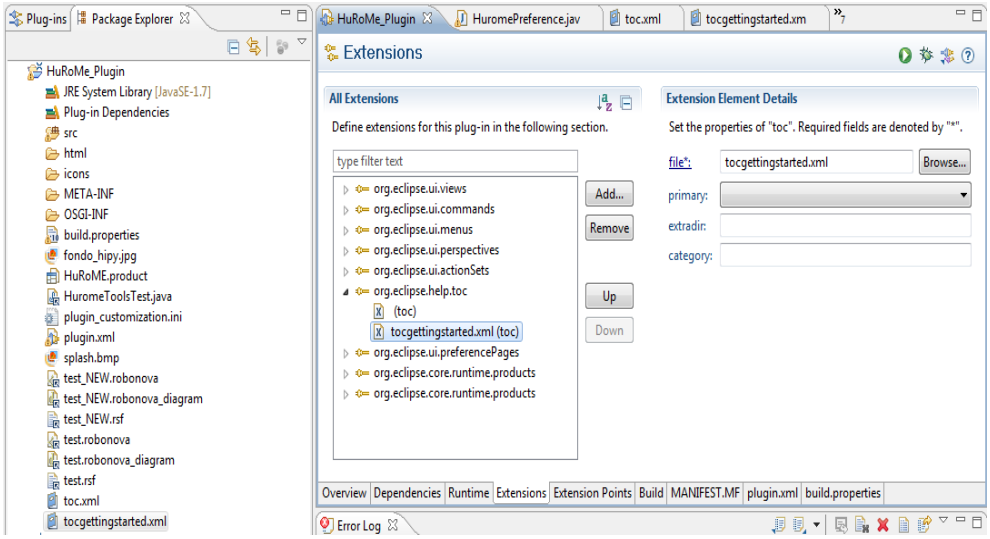

Figura 4.21.: Adición de org.eclipse.help.toc

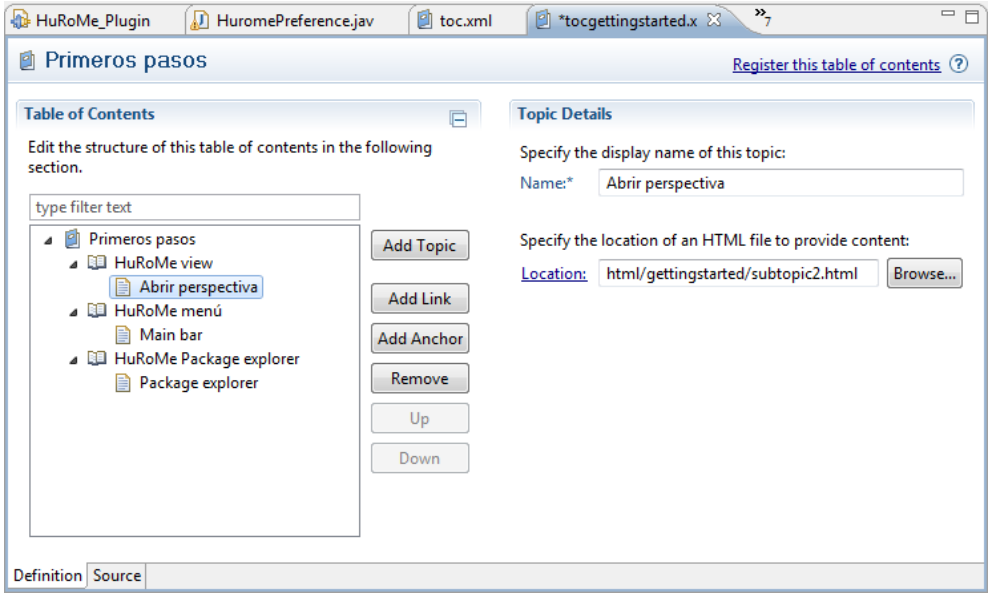

Figura 4.22.: Ayuda HuRoME

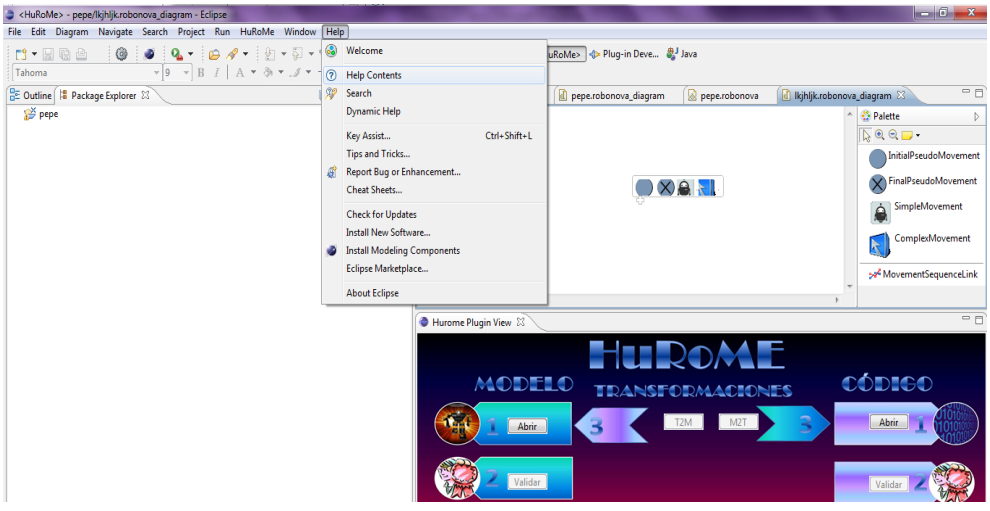

Figura 4.23.: Ayuda HuRoME

4.3.5 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 4.24 muestra la perspectiva desarrollada para HuRoME, ésta contendrá los siguientes elementos: la vista HuRoME (descrita en la sección 4.2), botón de habilitación de la vista en la barra de herramientas (descrito en la sección 4.3), el menú HuRoME en la barra principal y en el menú contextual (descrito en las sección 4.4 y 4.5 respectivamente), así como el Explorador de Paquetes y los demás elementos comunes del entorno Eclipse.

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 4.25, 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 HuRoME a la perspectiva. El resultado es que cada vez que abrimos la perspectiva HuRoME como muestra la Figura 4.26 en Window→Open Perspective. Obtenemos la perspectiva HuRoME con la vista asociada, Figura 4.24.

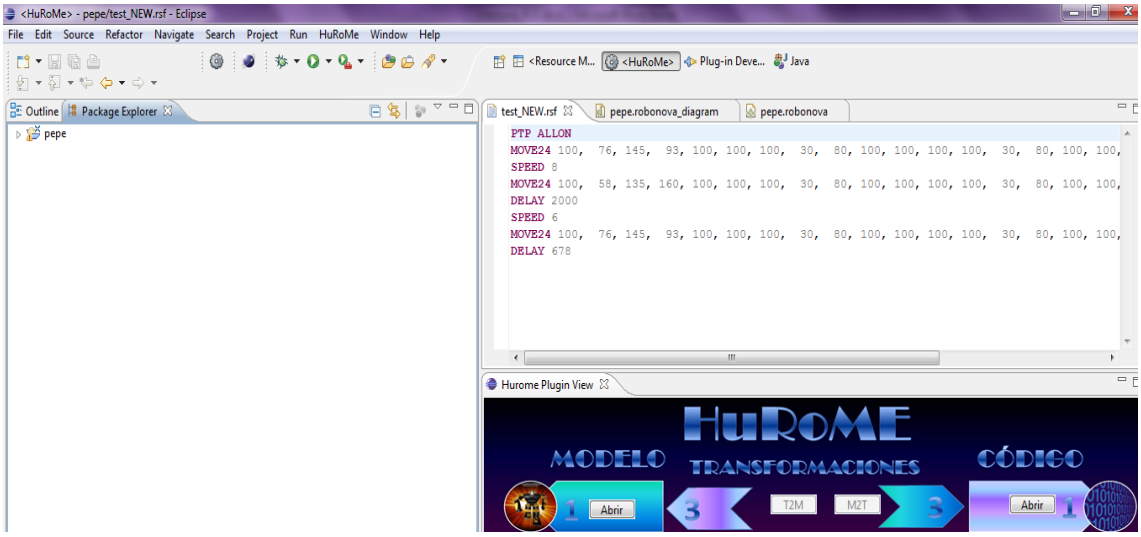

Figura 4.24.: Perspectiva HuRoME

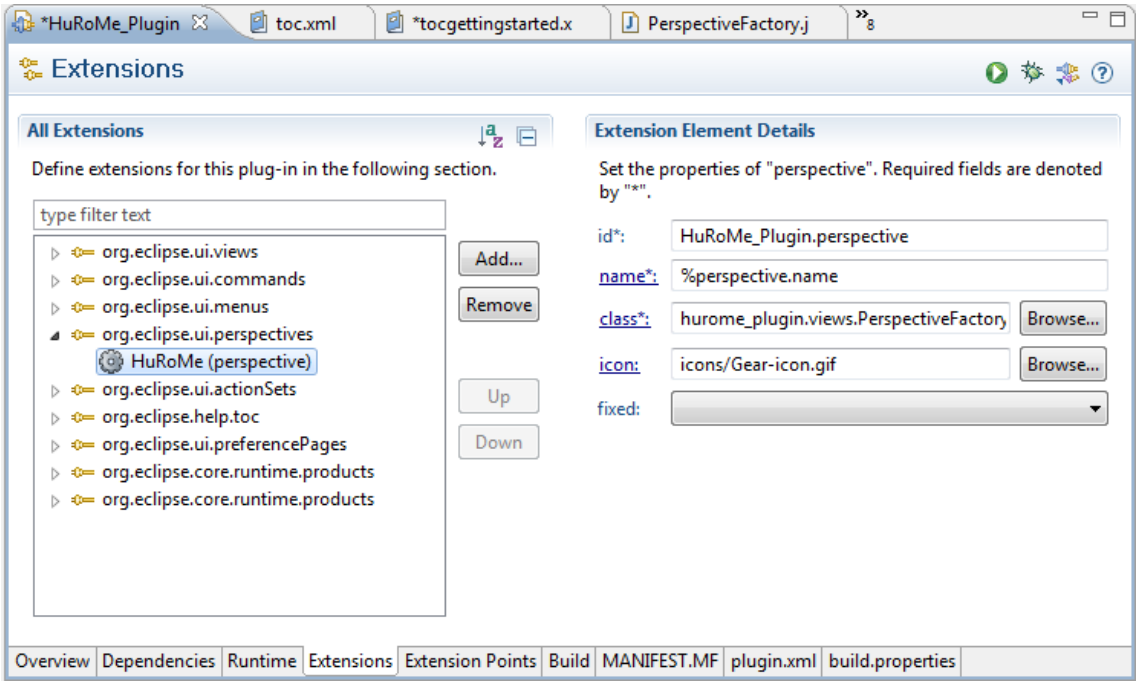

Figura 4.25.: Adicción de org.eclipse.ui.perspetives

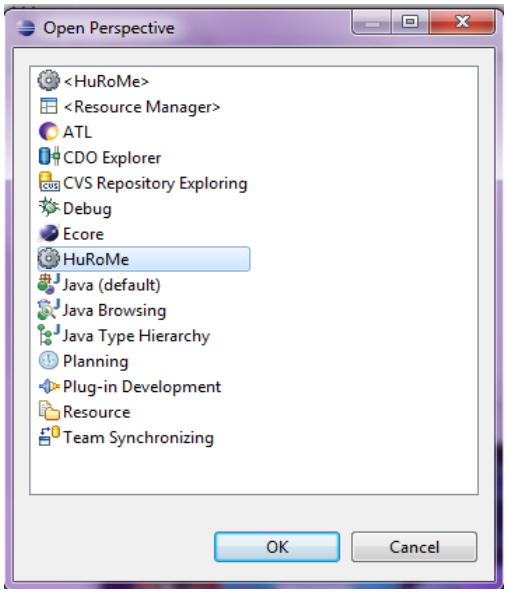

Figura 4.26. Abrir perspectiva HuRoME

4.3.6 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 HuRoME. 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 HuRoME.

4.3.6.1 Creación de un fichero JAR

Para la creación del archivo *HuRoME_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 4.27, 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 4.28 puede verse la selección de los Plug-ins a exportar en el caso de este Proyecto es HuRoME. Por último, se introduce el directorio de salida donde queremos que se cree el archivo *.jar e indicamos su nombre.

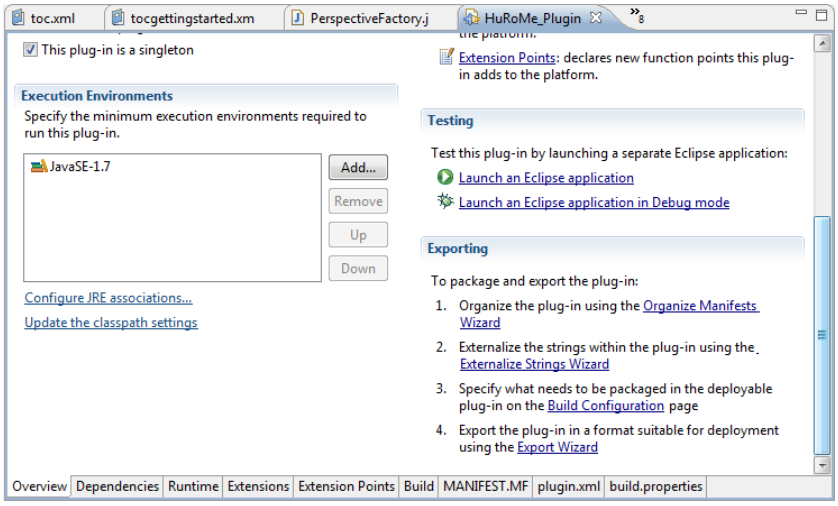

Figura 4.27.: Exportación del producto

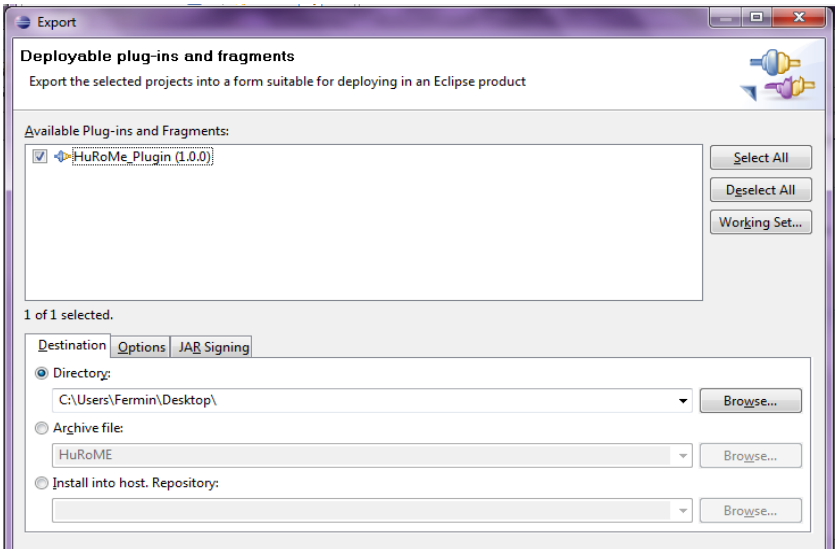

Figura 4.28.: Directorio salida fichero

4.3.6.2 Creación del producto Eclipse

Un Producto Eclipse se crea como un RCP (*Rich Client Platfform*) obteniendo un archivo ejecutable HuRoME.exe. Al hacer doble clic se abre un entorno Eclipse mínimo totalmente configurado para la herramienta HuRoME que sólo contiene aquellos Plug-ins necesarios para su funcionamiento.

Para la creación de una Producto Eclipse realizamos el siguiente procedimiento:

1. Pulsamos botón derecho sobre nuestro plug-in en el explorador de paquetes y seleccionamos New→Product Configuration. Introducimos el nombre que tendrá el archivo y dejamos el resto de opciones por defecto (véase Figura 4.29). Tras finalizar se habrá creado un archivo *.product.

2. Aparece un nuevo diálogo para configurar el Producto, en la pestaña *Overview* (Figura 2.30) completamos los siguientes campos: (1) el identificador de producto, se recomienda dejarlo vacío ya que puede generar errores; (2) el número de versión del producto (inicialmente configurado como "1.0.0"); (3) el nombre del producto (HuRoME); (4) en "Product Definition", pulsar *New*, rellenar "*Product id*" (HuRoME), en "*definition plug-in*" seleccionar el plug-in sobre el cual construimos el producto final. Por último, configurar "Application" como "*org.eclipse.ui.ide.workbench*" y finalizar. (5) Comprobar que "*Application*" es "*org.eclipse.ui.ide.workbench*" y que está marcada la opción "*Plug-ins*".

3. En la pestaña *Dependences*, especificamos que Plug-ins son necesarios para el correcto funcionamiento de nuestra herramienta. Para ello, pulsando la opción *Add* buscamos y añadimos estos Plug-ins. La opción *Add Requires Plug-in* automáticamente añade todas las dependencias secundarias de los Plug-ins ya añadidos. Es importante resaltar que para que el Producto funcione adecuadamente siempre debemos añadir como dependencia "*org.eclipse.ui.ide.application*".

4. En la pestaña *Launching*, se especifica la plataforma de ejecución del producto y el nombre ejecutable.

5. En la pestaña *Splash*, especificamos que imagen será visualizada inicialmente mientras Eclipse se está abriendo. La imagen debe encontrarse en el directorio del plug-in especificado en el campo "plug-in". La Figura 4.31 muestra la imagen de splash utilizada.

6. En la pestaña *Branding*, especificamos los iconos que se asocian al producto y que se utilizarán para adornar la ventana de la aplicación, concretamente en el borde superior izquierdo y en la barra de inicio cuando el programa esté en ejecución.

7. En la pestaña *Liscensing*, puede especificarse la URL y el texto con la licencia del producto.

8. Para finalizar la creación del producto pulsamos el asistente de exportación en "*exporting*" de la pestaña *Overview* e indicamos la ruta donde se creará el producto.

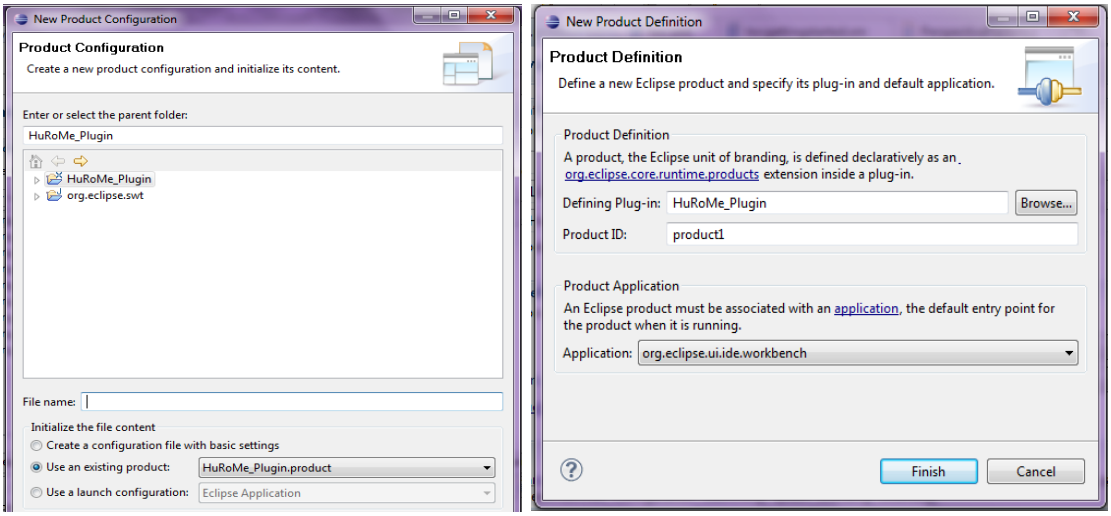

Figura 4.29.: (Izda.) Creación de nuevo producto. **(Dcha.)** Definición del producto

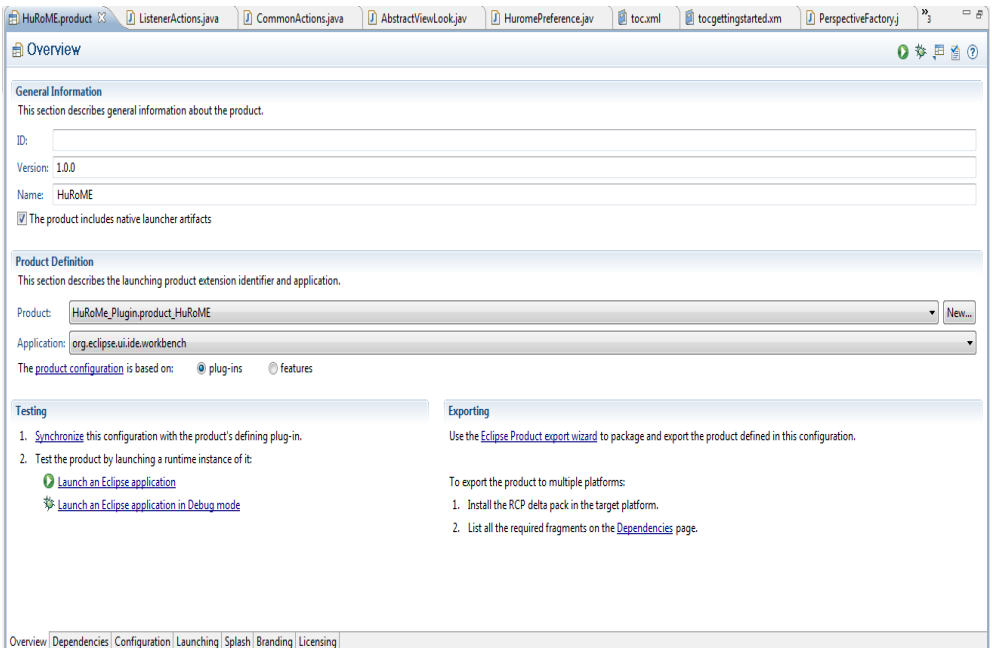

Figura 4.30.: Creación de nuevo producto

Figura 4.31.: Imagen Splash

CAPÍTULO V

Guía de uso del Plug-in

En este capítulo se describe la instalación y uso de las distintas herramientas desarrolladas que componen el Plug-in HuRoME. Concretamente para ilustrar el uso de la herramienta se utiliza el ejemplo introducido en la sección 3.4.1 de modo que podrá comprobarse las ventajas que proporciona el Plug-in desarrollado respecto a la herramienta original.

5.1 Instalación del Plug-in

Los pasos descritos a continuación describen el proceso de instalación de Eclipse, los Plug-ins necesarios para su funcionamiento y el JAR HuRoME creado en la sección 4.3.6.1:

1. Descargar *Eclipse Helios Modeling Tools* [24] y descomprimir el fichero.

2. Instalar los Plug-ins requeridos para el funcionamiento de HuRoME, estos son: JET, Xtext, ATL y GMF. Para instalar estos Plug-ins basta con dirigirnos a Help→Install Modelling Components y seleccionarlos.

3. Instalar los Plug-ins de la herramienta HuRoME (véase [5] para más detalle).

4. Copiar el archivo HuRoME_Plug-in_1.0.0.jar en la carpeta “Plug-ins” ubicada en el directorio de instalación de Eclipse y reiniciar el entorno Eclipse.

En la Figura 4.6 vemos el aspecto resultante de la instalación de las herramientas en este apartado descritas.

Cabe reseñar que el Producto Eclipse creado en la sección 4.3.6.2 no requiere instalación alguna, por lo que tan sólo habrá que copiarlo en la ubicación deseada y ejecutar el archivo HuRoME.exe. Con ello, directamente en el propio ejecutable dispondremos de la versión de *Eclipse Helios* con las funcionalidades mínimas, con los Plug-ins de HuRoME y el Plug-in desarrollado en este Proyecto, todo listo para ser utilizado por el usuario.

5.2 Obteniendo código desde un modelo

El objetivo es ilustrar la obtención de código desde un modelo utilizando el ejemplo I descrito en 3.4.1. De modo que podremos comparar las ventajas que ofrece el Plug-in desarrollado frente a la herramienta HuRoME original. La coreografía utilizada en el ejemplo I plantea que el robot parte de una posición estándar, se encuentra alzado sobre sus piernas y con los brazos relajados, tras ello, se inclinará a modo de reverencia y volverá a su posición, por último, termina levantando un brazo como despedida (puede verse el modelo en la Figura 3.2). A continuación se describen los pasos seguidos:

1. Comenzamos abriendo la perspectiva HuRoME con la vista desarrollada, usamos Window→Show View→Other, como vemos en la Figura 5.1, seleccionamos Hurome Plug-in View. Esta operación es equivalente a pulsar el correspondiente botón añadido en la barra de herramientas (véase la sección 4.3.1).

2. Creamos un nuevo Proyecto (File→New→Project→General→Project) y lo nombramos en el cuadro de diálogo (mostrado en la Figura 5.2 Izda.).

3. En la vista de HuRoME que representa el flujograma de uso de la herramienta, pulsamos el botón abrir en el campo modelo y seleccionamos el archivo *.robonova o *.robonova_diagram que contiene el modelo con la coreografía (véase Figura 5.2 Dcha.).

4. Como vemos en la Figura 5.3, al abrir el archivo éste es visualizado en el editor gráfico de HuRoME. En el flujograma, observamos que queda habilitada la opción validar dentro del campo modelo (esta opción aparecía deshabilitada inicialmente). La opción validar realiza una validación del archivo y verifica si es apto o no para realizar las transformaciones.

5. Pulsamos sobre validar obteniendo un mensaje que indica que el archivo es válido para realizar la transformación modelo a texto, como vemos en la Figura 5.4. En el caso de que no hubiera sido así, obtendríamos un mensaje de error y el botón de transformación permanecería deshabilitado.

6. El botón modelo-a-texto (M2T), en el flujograma, queda habilitado, lo pulsamos y rellenamos cuadro el dialogo mostrado en la Figura 5.5. Configuramos el nombre del Proyecto, el nombre del fichero/os de salida y el tipo de código que queremos que se genere, pudiendo puede ser: (1) RoboBASIC (*.bas) y/o (2) RoboScript (*.rsf). Seleccionamos generar el código del robot en ambos lenguajes y aceptamos. Una vez realizada la transformación, aparece un mensaje que muestra que la transformación modelo-a-texto ha sido realizada satisfactoriamente. Además, podemos observar en el Explorador de Paquetes que se han creado los archivos con el nombre y en el Proyecto especificado (Figura 5.6), estos archivos coinciden con los mostrados en la Figura 3.3.

De manera alternativa, también podemos realizar el procedimiento anterior: (1) sobre un modelo abierto en el editor gráfico accediendo al menú HuRoME→Transformaciones→M2T; y (2) seleccionando un modelo en el Explorador de Paquetes y pulsando en su menú contextual la opción HuRoME→Transformaciones→M2T.

Como mencionamos en la sección 4.3.3 hemos creado un panel de preferencias (Figura 5.7) para la herramienta Plug-in, con este panel podemos pre-seleccionar el tipo de fichero de salida tras aplicar M2T. Además, podemos configurar la apertura automática del fichero resultante y la creación automática del diagrama tras finalizar la transformación texto a modelo.

Tras la realización de la transformación M2T observamos que el número de acciones a ejecutar se ve considerablemente reducido respecto del Ejemplo I realizado en el capítulo III (sección 3.4). La vista desarrollada, así como los menús incluidos facilitan y simplifican el uso de la herramienta HuRoME.

5.3 Obteniendo el modelo desde código RoboScript

Para obtener el modelo gráfico de secuencias de movimientos del robot descrito por código RoboScript, se han de realizar los siguientes pasos:

1. Inicialmente, podemos observar en la vista HuRoME que solo tenemos habilitados los botones para “abrir” tanto en modelo como en código. Así, seleccionamos “abrir” (en el campo código) un archivo RoboScript (extensión *.rsf) (Figura 5.8). Como vemos en la Figura 5.12, al abrir el archivo éste es visualizado en el editor de código de HuRoME.

2. Observamos que se habilita el botón validar lo que indica que la operación validar código está disponible, validamos el fichero y visualizamos un mensaje de validación correcta (Figura 5.10).

3. La opción transformar texto-a-modelo (T2M) queda habilitada para su uso, presionamos sobre T2M (véase Figura 5.10), Escribimos el nombre que queramos que tenga el archivo, como muestra la Figura 5.14, resultante tras la transformación T2M que tendrá extensión *.robonova. Visualizaremos un dialogo mostrándonos si la operación ha sido realizada satisfactoriamente o no. Observamos que ha sido creado satisfactoriamente el archivo test1.robonova (Figura 5.15) y que se nos habilita la opción crear diagrama, pulsamos sobre ella para obtener *.robonoba_diagram. Los archivos .robonova y .robonova_diagram quedan creados en el “*workspace*”.

De manera alternativa, también podemos realizar el procedimiento anterior: (1) sobre un fichero RoboScript abierto en el editor accediendo al menú HuRoME→Transformaciones→T2M; y (2) seleccionando el fichero RoboScript en el Explorador de Paquetes y pulsando en su menú contextual la opción HuRoME→Transformaciones→T2M.

A modo de conclusión y en comparativa con las secciones 3.4 y 3.5 podemos decir que el Plug-in desarrollado mejora la usabilidad y la accesibilidad de HuRoME facilitando las operaciones y mostrándolas de una manera más intuitiva e amigable.

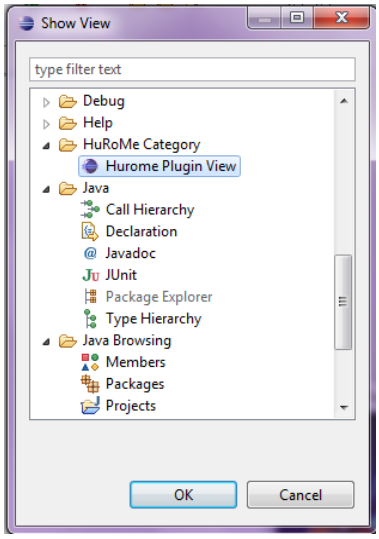

Figura 5.1.: Izda. HuRoME Plug-in view

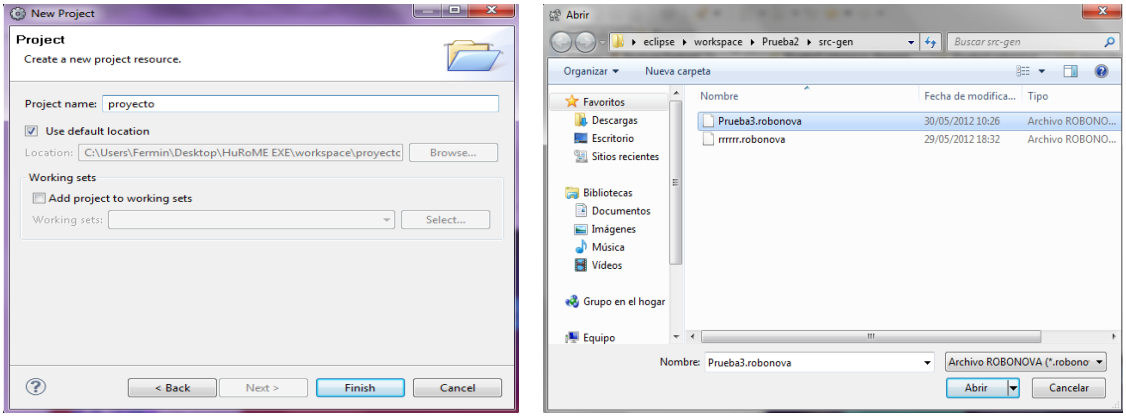

Figura 5.2.: (Izda) Creación del proyecto. **(Dcha.)** Dialogo abrir el archivo *.robonova

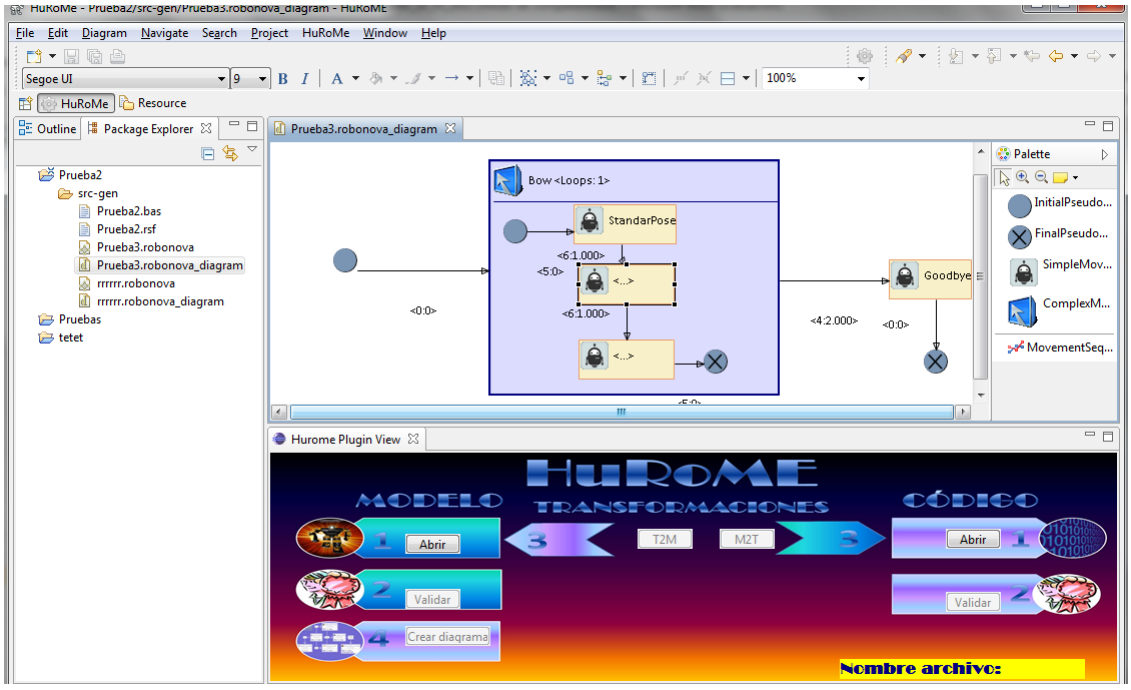

Figura 5.3.: Modelo gráfico *.robonova_diagram

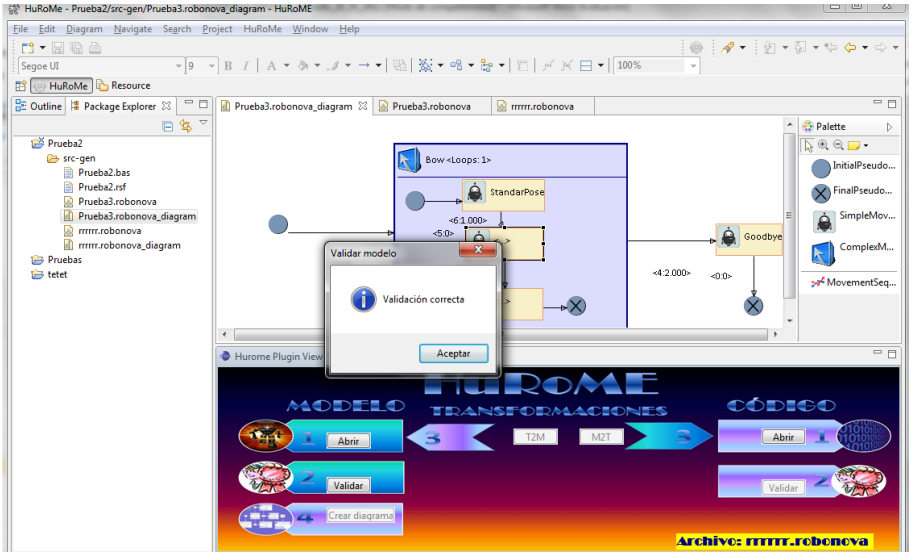

Figura 5.4.: Validación correcta

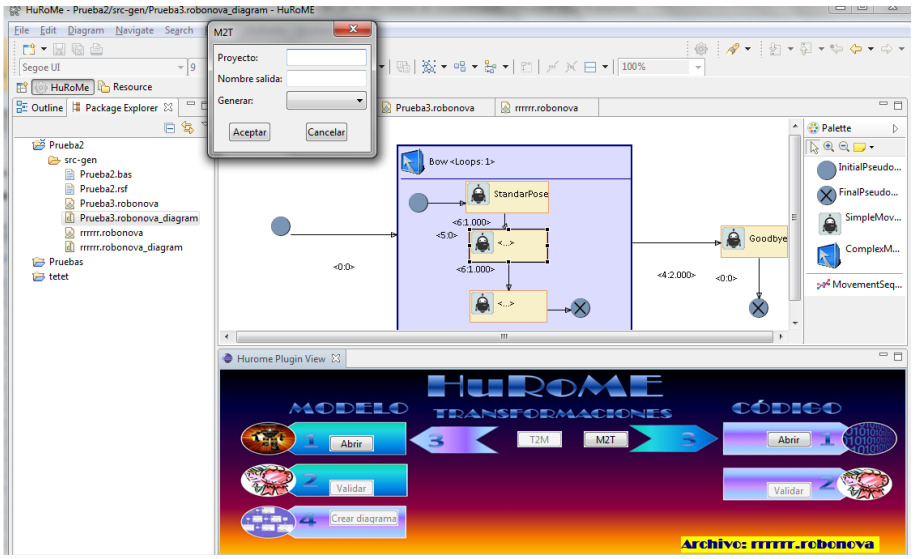

Figura 5.5.: Dialogo guardar M2T

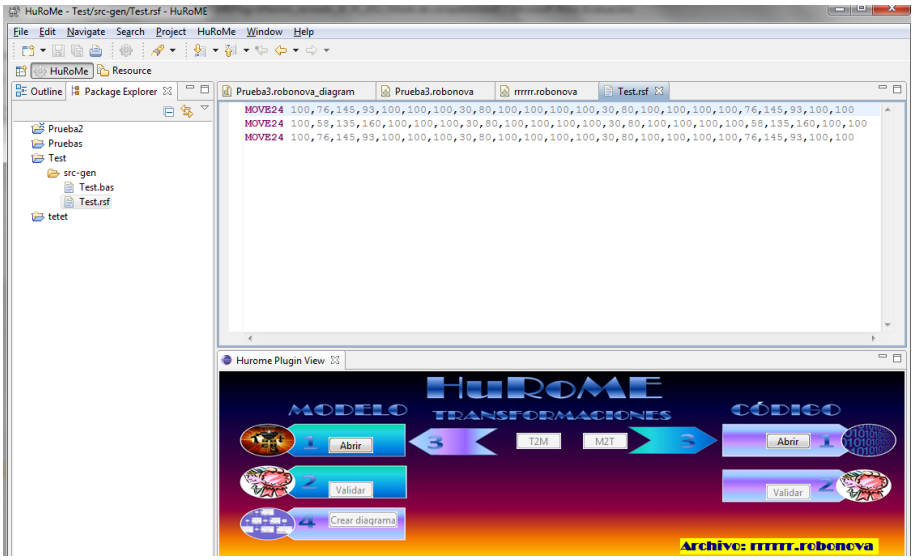

Figura 5.6. Código RoboScript.

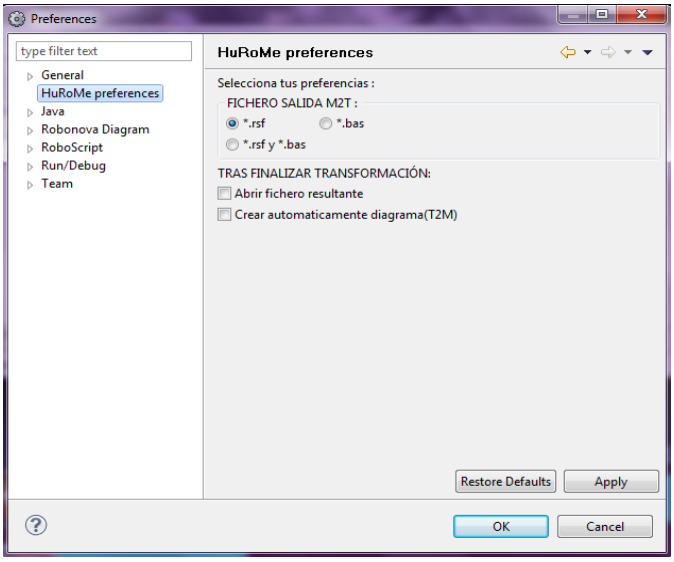

Figura 5.10.: Panel de preferencias HuRoME

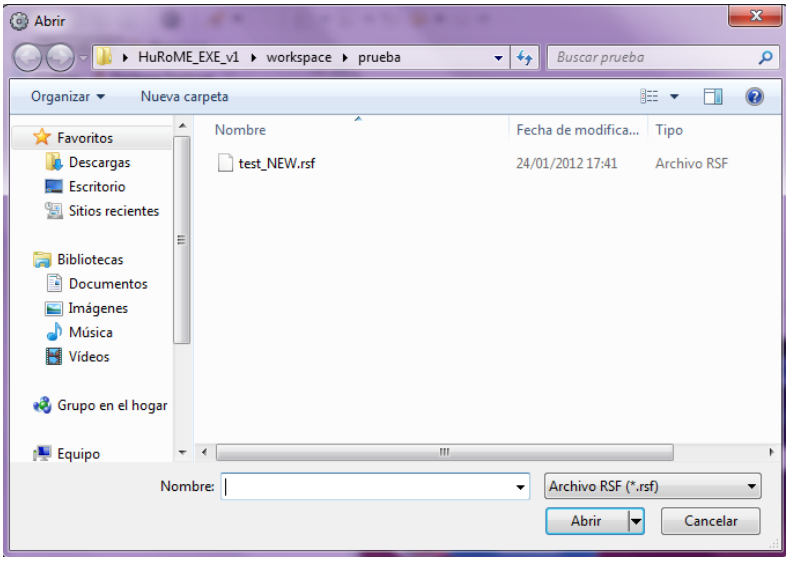

Figura 5.11.: Diálogo abrir RoboScript

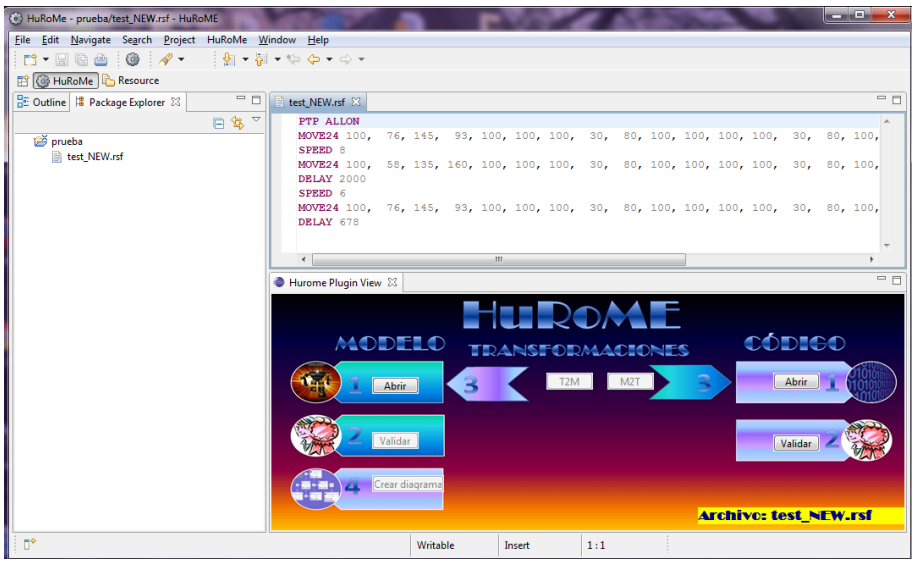

Figura 5.12.: Editor Xtext

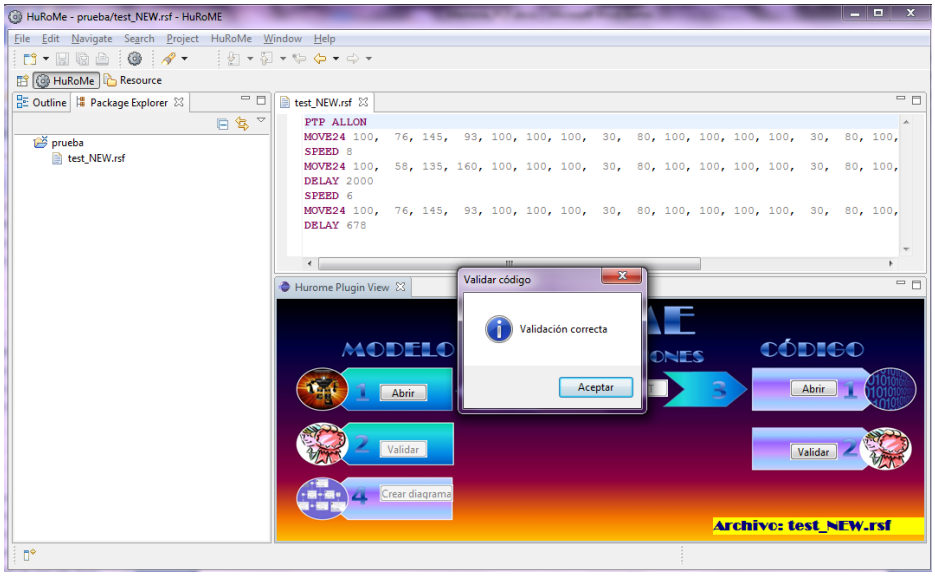

Figura 5.13.: Diálogo validación correcta

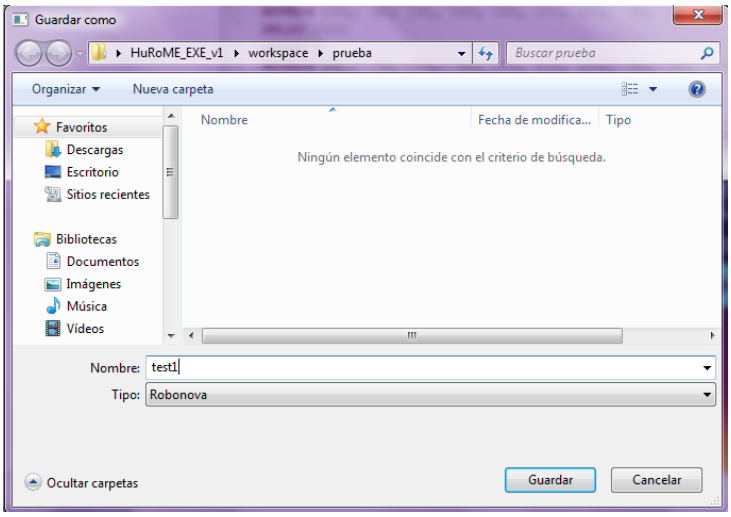

Figura 5.14.: Diálogo guardar archivo *.robonova

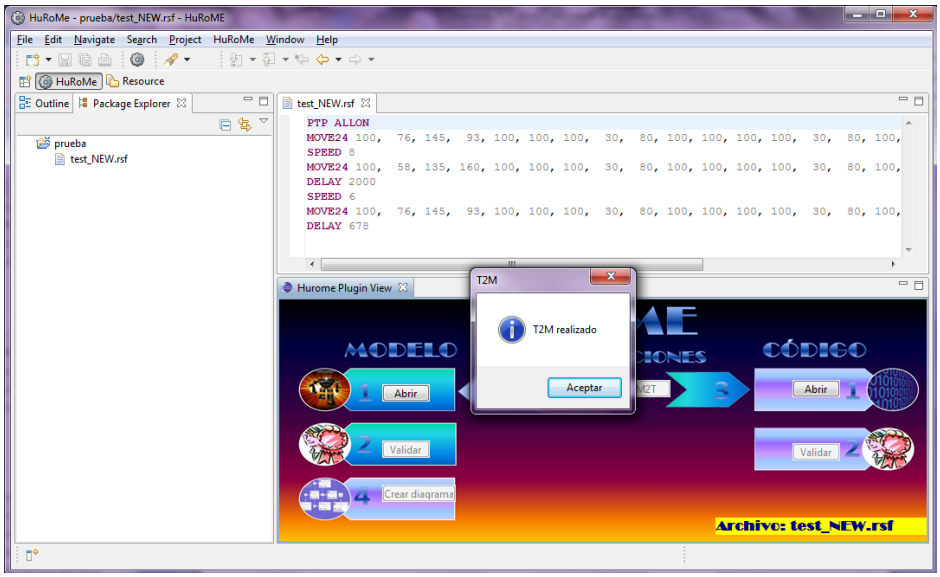

Figura 5.15.: Proceso T2M realizado

CAPÍTULO VI

Conclusiones y Líneas de Trabajos Futuras

En este capítulo se exponen las conclusiones más relevantes del Proyecto alcanzadas durante su realización. Para terminar, se introducen algunas posibles líneas de trabajo futuro.

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 HuRoME aprovechando la capacidad de Eclipse para ser extendido usando el mecanismo de extensión y puntos de extensión.

- ✓ El presente Proyecto proporciona una guía sobre el desarrollo de Plug-ins Eclipse especificando paso por paso el procedimiento a seguir para la creación de un nuevo Plug-in.

- ✓ HuRoME 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. No obstante, HuRoME no proporciona un entorno totalmente unificado lo que dificulta su uso, aprendizaje y distribución.

- ✓ 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 HuRoME,

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 HuRoME.

- ✓ Se ha desarrollado un panel de preferencias que agrupa y simplifica la configuración de la herramienta HuRoME.

- ✓ Se ha desarrollado un menú de ayuda con documentación de uso de la herramienta facilitando el aprendizaje de los nuevos usuarios.

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

- ✓ Se ha realizado un producto Eclipse para la herramienta HuRoME, 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 HuRoME.

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 la apariencia de la vista desarrollada está desacoplada de la funcionalidad, se plantea el desarrollo de un mecanismo para personalizar su aspecto visual según las preferencias del usuario.

- 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.

- Explorar el desarrollo de IDEs basados en tecnologías Web. Por ejemplo, para permitir que un usuario de HuRoME pueda utilizar la herramienta desde su Navegador Web.

- Actualizar el Plug-in desarrollado en el presente Proyecto para la última versión de la herramienta HuRoME.

Bibliografía

[1] Eclipse. http://www.eclipse.org/

[2] J.F. Inglés-Romero, C. Vicente-Chicote, D. Alonso, *Entorno para Coreografiar Movimientos en un Robot Humanoide*, III Jornadas de Jóvenes Investigadores de la Universidad Politécnica de Cartagena, Cartagena (Spain), May 2010. ISSN 1888 8356.

[3] J.F. Inglés-Romero, C. Vicente-Chicote, D. Alonso, *Desarrollo de Software Aplicado a la Robótica: Un Caso Práctico*, I Jornadas de Jóvenes Investigadores de la Universidad de Extremadura, Cáceres (Spain), April 2010. ISBN 978-84-693-1707-5.

[4] J.F. Inglés-Romero, C. Vicente-Chicote, D. Alonso, *HuRoME: Entorno para Modelado de Coreografías y Modernización de Código para un Robot Humanoide*, XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD'10), Valencia (Spain), September 2010. ISBN 978-84-92812-51-6.

[5] J.F. Inglés-Romero*: Humanoid Robot Modeling Environment (HuRoME)*, Trabajo Final de Máster, Máster en Tecnologías de la Información y Comunicaciones, Universidad Politécnica de Cartagena, September 2010.

[6] Robonova 1, HITEC Robotics, http://www.robonova.com/

[7] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering*, Management, ed. Wiley, 2006, ISBN: 978-0-470-02570-3.

[8] D. Carlson, *Eclipse Dustilled*. Addison Wesley Professional, 2005, ISBN: 0-321-28815-7.

[9] C/C++ Development Tools (CDT), www.eclipse.org/cdt

[10] Web Tools Project (WTP), www.eclipse.org/webtools

[11] E.Clayberg and D. Rubel, Building Commercial-Quality Plug-Ins, Addison Wesley Professional, 2004, ISBN: 978-0321228475.

[12] Booch, G., Rumbaugh, J. and Jacobson, I.: Unified Modeling Language User Guide. 2nd Edition. Addison-Wesley, 2005.

[13] Bézivin, J. and Gerbé, O.: "Towards a Precise Definition of the OMG/MDA Framework." En Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE'01), Noviembre 2001.

[14] Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled. Principles of Model-Driven Architecture. Addison-Wesley, 2004.

[15] OMG. "Model Driven Architecture." Document ormsc/2001-07-01, Julio 2001. Accesible desde http://www.omg.org/

[16] DSL Tools. http://msdn.microsoft.com/en-us/library/bb126235.aspx

[17] Meta Edit. http://www.metacase.com/

[18] The Eclipse Graphical Modeling Framework (GMF), http://www.eclipse.org/modeling/gmf/

[19] JET, http://www.eclipse.org/modeling/m2t/?project=jet#jet

[20] RoboBASIC Command Instruction Manual v2.10, http://www.hitecrobotics.com

[21] Xtext, http://www.eclipse.org/Xtext/

[22] ATLAS Transformation Language (ATL), http://www.eclipse.org/atl/

[23] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented. Addison Wesley Professional (1994)

[24] Eclipse Helios Modeling Tools, http://www.eclipse.org/modeling/

Anexo I. Descripción de la Clase HuromeTools

Este anexo describe el procedimiento de uso y la API de la clase HuromeTools, clase que permite el acceso a las funcionalidades de la herramienta HuRoME, y que, por lo tanto, resulta de vital importancia para el Plug-in desarrollado pues permite manejar modelos de manera transparentemente a las tecnologías basadas en el framework EMF.

A1.1 Procedimiento para utilizar la clase hurome.util.HuromeTools

1. Es necesaria la versión Modeling Tools de Eclipse (http://www.eclipse.org/downloads/). Una vez descargada y descomprimida, se requieren los siguientes plugins: ATL, xText, GMF y JET. Para ello, seleccionar en el menú: Help→Install Modeling Components, marcar en la listas los plugins anteriores y finalizar.

2. Instalación de HuRoME. Para ello descomprimir el archivo Hurome.zip y mover el contenido de la carpeta plugins a la carpeta plugins de Eclipse (con la aplicación Eclipse cerrada).

3. Crear un proyecto tipo Plug-in Project y seguir la guía para añadir una nueva opción en el menú contextual de la vista PackageExplorer (pag. 49). Esto nos servirá ejecutar el código de ejemplo cada vez que hagamos click sobre la correspondiente opción del menú.

4. Crear un paquete hurometest donde añadir la clase definida en el fichero HuromeToolsTest.java. Esta clase deberá añadirse en el campo “Default Handler” para que sea asociada al nuevo menú.

5. Abrir la pestaña Dependences de plugin.xml y añadir en Imported Package los siguientes paquetes: hurome.util (en este paquete se define la clase HuromeTools) y org.eclipse.emf.common.util (incluye la definición de algunas clases que se emplean en el código de ejemplo, como BasicDiagnostic y Diagnostic).

6. Copiar los ficheros de ejemplo (*.robonova, *.robonova_diagram, *.rsf) en el sistema de ficheros de su ordenador. Leer el código proporcionado como ejemplo, incluido en la clase HuromeToolsTest, y modificar las rutas a los ficheros de entrada según donde hayan sido copiados.

7. Lanzar la ejecución en un nueva instancia de Eclipse. Quizás la primera vez sea necesario acceder a Run→Run Configurations…→Doble click sobre “Eclipse application” (para crear un nuevo perfil)→Pulsar Run.

8. Al pulsar el segundo botón de Eclipse sobre el Package Explorer debería aparecer la nueva opción de menú, al hacer click en ella se ejecutará el código de ejemplo.

A1.2 Descripción de la clase HuromeTools

`public static final String HUROME_EDITOR_ID`

Constante que proporciona el ID del editor gráfico de modelos HuRoME. Los identificadores de los editores son necesario para, por ejemplo, abrir un fichero con el editor de forma programada o para recuperar información sobre un documento abierto en el editor (como la ruta del fichero).

`public static final String HUROME_TREE_EDITOR_ID`

Constante que proporciona el ID del editor en árbol (TreeEditor) de modelos HuRoME.

`public static final String ROBOSCRIPT_EDITOR_ID`

Constante que proporciona el ID del editor textual de código RoboScript.

`public static final int M2T_GENERATE_ONLY_ROBOBASIC`

Posible valor config en el metodo runHurome2Code.

`public static final int M2T_GENERATE_ONLY_ROBOSCRIPT`

Posible valor config en el metodo runHurome2Code.

`public static final int M2T_GENERATE_ONLY_ROBOSCRIPT`

Posible valor config en el metodo runHurome2Code.

`public static boolean validateHuromeModel(String inFilePath, BasicDiagnostic diagnostics)`

Comprueba la corrección del modelo de secuencia de movimientos de HuRoME. Rellena un objeto Diagnostic con los detalles de la validación.

Parametros:

- **inFilePath** Ruta absoluta al modelo de entrada (fichero *.robonova).
- **diagnostics** Referencia a un objeto creado BasicDiagnostic, éste puede contener una colección de diagnósticos (para más información puede consultarse la clase org.eclipse.emf.common.util.BasicDiagnostic).

Retorno:
Devuelve true si la validación se ha ejecutado, false si ha habido alguna incidencia en los argumentos de entrada o si el proceso a fallado.

Nota:
En Windows los path absolutos han de comenzar con una barra ('/'). Por ejemplo: "/C:/Documents and Settings/Administrador/Escritorio/default.robonova"

`public static boolean runHurome2Code(String inFilePath, String outProjectName, String outFilesRelativePath, String outFilesName, int config)`

Ejecuta la transformación de un modelo HuRoME de secuencia de movimientos a código para el robot Robonova. Los ficheros de código son creados en el proyecto y path indicados, utilizando el nombre y la configuración pasada.

Parametros:

- **inFilePath** Ruta absoluta al modelo de entrada (fichero *.robonova).
- **outProjectName** Nombre del proyecto de salida, si no existe es creado. Si se pasa una cadena vacía el proyecto es "RobonobaCode".
- **outFilesRelativePath** Ruta dentro del proyecto donde se crearan los ficheros de salida, si no existe se crea. Si se pasa una cadena vacía la ruta es "src-gen".
- **outFilesName** Nombre de los ficheros de salida. Si se pasa una cadena vacía el nombre es "program".
- **config** Configuracion de la transformacion. (Valores: ver constantes arriba).

Retorno:
true si la transformación se ha ejecutado con éxito, false si ha habido alguna incidencia en los argumentos de entrada o el proceso ha fallado.

Nota:
En Windows los path absolutos han de comenzar con una barra ('/'). Por ejemplo: "/C:/Documents and Settings/Administrador/Escritorio/default.robonova"

---

public static boolean validateRoboScript(String inFilePath, BasicDiagnostic diagnostics)

Comprueba la corrección del fichero de código RoboScript. Devuelve un objeto que ofrece un diagnostico tras la validación, retorna null cuando el path de entrada es vacío.

Parámetros:
- **inFilePath** Ruta absoluta al modelo de entrada (fichero *.rsf).
- **diagnostics** Referencia a un objeto creado BasicDiagnostic, éste puede contener una colección de diagnósticos (para más información puede consultarse la clase org.eclipse.emf.common.util.BasicDiagnostic).

Retorno:
Devuelve true si la validación se ha ejecutado, false si ha habido alguna incidencia en los argumentos de entrada o si el proceso a fallado.

Nota:
En Windows los path absolutos han de comenzar con una barra ('/'). Por ejemplo: "/C:/Documents and Settings/Administrador/Escritorio/default.robonova"

---

public static boolean runCode2Hurome(String inFilePath, String outFilePath)

Ejecuta la transformación de un modelo RoboScript a un modelo HuRoME de secuencia de movimientos.

Parámetros:
- **inFilePath** Ruta absoluta al modelo de entrada (fichero *.rsf)
- **outFilePath** Ruta absoluta al modelo de salida (fichero *.robonova)

Retorno:
true si la transformación se ha ejecutado con éxito, false si ha habido alguna incidencia en los argumentos de entrada o el proceso ha fallado.

Nota:
En Windows los path absolutos han de comenzar con una barra ('/'). Por ejemplo: "/C:/Documents and Settings/Administrador/Escritorio/default.robonova"

---

public static boolean initializeDiagramFile(String inFilePath, String outDiagramName)

Crea la vista gráfica (fichero *.robonova_diagram) de un modelo de secuencias de movimiento (fichero *.robonova).

Parámetros:
- **inFilePath** Ruta absoluta al modelo de entrada (fichero *.robonova).

- **outDiagramName** Nombre del fichero *.robonova_diagram de salida. Si se pasa null o una cadena vacia el fichero es nombrado igual que el fichero pasado *.robonova.

Retorno:
true si el proceso se ha ejecutado con éxito, de lo contrario, false.

Nota:
En Windows los path absolutos han de comenzar con una barra ('/'). Por ejemplo: "/C:/Documents and Settings/Administrador/Escritorio/default.robonova"

Anexo II. Código del Plug-in implementado

Este anexo muestra el código desarrollado para la implementación del Plug-in.

A2.1 Clase AbstractViewLook

```
Public abstract class AbstractViewLook {

	public static final String OPEN_MODEL_BTN_ID = "OPEN_MODEL_BTN";
	public static final String OPEN_CODE_BTN_ID = "OPEN_CODE_BTN";
	public static final String CREATE_DIAG_BTN_ID = "CREATE_DIAG_BTN";
	public static final String VALIDATE_MODEL_BTN_ID = "VALIDATE_MODEL_BTN";
	public static final String VALIDATE_CODE_BTN_ID = "VALIDATE_CODE_BTN";
	public static final String MT2_BTN_ID = "MT2_BTN";
	public static final String T2M_BTN_ID = "T2M_BTN";


	protected Button openModelBtn;
	protected Button openCodeBtn;
	protected Button createDiagramBtn;
	protected Button validateModelBtn;
	protected Button validateCodeBtn;
	protected Button m2tBtn;
	protected Button t2mBtn;
	protected Label labFile;
	Composite composite;

	public Button getOpenMode(){
		return openModelBtn;
	}

	/**
	 * @método le pasamos un objeto de la clase composite a nuestro
	 * constructor ya que nuestra vista extiende a ViewPart cuyo método a
	 * extender requiere un Composite
	 *
	 */
	public AbstractViewLook(Composite compo){
		this.composite=compo;
	}
	/**
	* @metodo inicializa a cada botón un identificador y su correspondiente
	* escuchador
	* @parametros: le pasamos el escuchador a los respectivos botones.
	*/
	public void initialize(SelectionListener se){
		m2tBtn.addSelectionListener(se);
		m2tBtn.setData("id", MT2_BTN_ID);
		t2mBtn.addSelectionListener(se);
		t2mBtn.setData("id", T2M_BTN_ID);
		createDiagramBtn.addSelectionListener(se);
		createDiagramBtn.setData("id",CREATE_DIAG_BTN_ID);
		openModelBtn.addSelectionListener(se);
		openModelBtn.setData("id", OPEN_MODEL_BTN_ID);
		validateModelBtn.addSelectionListener(se);
		validateModelBtn.setData("id",VALIDATE_MODEL_BTN_ID);
		validateCodeBtn.addSelectionListener(se);
		validateCodeBtn.setData("id",VALIDATE_CODE_BTN_ID);
		openCodeBtn.addSelectionListener(se);
		openCodeBtn.setData("id",OPEN_CODE_BTN_ID);
	}
	public void setState(StateId id){
		switch (id){
			case INICIAL:
				openModelBtn.setEnabled(true);
				openCodeBtn.setEnabled(true);
				validateModelBtn.setEnabled(false);
				validateCodeBtn.setEnabled(false);
				m2tBtn.setEnabled(false);
				t2mBtn.setEnabled(false);
```

```
                        createDiagramBtn.setEnabled(false);
                        break;
                    case MODEL_OPEN:
                        openModelBtn.setEnabled(true);
                        openCodeBtn.setEnabled(true);
                        validateModelBtn.setEnabled(true);
                        validateCodeBtn.setEnabled(false);
                        m2tBtn.setEnabled(false);
                        t2mBtn.setEnabled(false);
                        reateDiagramBtn.setEnabled(false);
                        break;
                    case CODE_OPEN:
                        openModelBtn.setEnabled(true);
                        openCodeBtn.setEnabled(true);
                        validateModelBtn.setEnabled(false);
                        validateCodeBtn.setEnabled(true);
                        m2tBtn.setEnabled(false);
                        t2mBtn.setEnabled(false);
                        createDiagramBtn.setEnabled(false);
                        break;
                    case CODE_VALIDATED_OK:
                        openModelBtn.setEnabled(true);
                        openCodeBtn.setEnabled(true);
                        validateModelBtn.setEnabled(false);
                        validateCodeBtn.setEnabled(false);
                        m2tBtn.setEnabled(false);
                        t2mBtn.setEnabled(true);
                        createDiagramBtn.setEnabled(false);
                        break;
                    case MODEL_VALIDATED_OK:
                        openModelBtn.setEnabled(true);
                        openCodeBtn.setEnabled(true);
                        validateModelBtn.setEnabled(false);
                        validateCodeBtn.setEnabled(false);
                        m2tBtn.setEnabled(true);
                        t2mBtn.setEnabled(false);
                        createDiagramBtn.setEnabled(false);
                        break;
                    case MODEL_VALIDATED_FAILED:
                        openModelBtn.setEnabled(true);
                        penCodeBtn.setEnabled(true);
                        validateModelBtn.setEnabled(false);
                        validateCodeBtn.setEnabled(false);
                        m2tBtn.setEnabled(false);
                        t2mBtn.setEnabled(false);
                        createDiagramBtn.setEnabled(false);
                        break;
                    case CODE_VALIDATED_FAILED:
                        openModelBtn.setEnabled(true);
                        openCodeBtn.setEnabled(true);
                        validateModelBtn.setEnabled(false);
                        validateCodeBtn.setEnabled(false);
                        m2tBtn.setEnabled(false);
                        t2mBtn.setEnabled(false);
                        createDiagramBtn.setEnabled(false);
                        break;
                    case T2M_PROCCESSED:
                        openModelBtn.setEnabled(true);
                        openCodeBtn.setEnabled(true);
                        validateModelBtn.setEnabled(false);
                        validateCodeBtn.setEnabled(false);
                        m2tBtn.setEnabled(false);
                        t2mBtn.setEnabled(false);
                        createDiagramBtn.setEnabled(true);
                        break;
                }
        }
        /**
         * @descripción: única funcion es la de cambiar el nombre del archivo
         * cada vez que sea requerido
         */
        public void changeArchiveName(String name){
                labFile.setText("Archivo: "+name);
        }
}
```

A2.2 Clase MyViewLook

```
public class MyViewLook extends AbstractViewLook implements PaintListener {

	Image myImage,error,ok;
	String M_TYPE;
	String extension;
	File myArchivo;
	FileReader fr;
	BufferedReader br;
	GC fondo;
	Composite composite;
	Composite c;

	/**
	 * Configuramos colores,tamaños y disposiciones de elementos en nuestra
	 * vista
	 * @param com, necesario para el metodo createPartControl que
	 * implementamos de ViewPart
	 */

	public MyViewLook (Composite com) {
		super(com);

		composite = com;
		com.setLayout(new FillLayout());
		myImage = new Image(com.getDisplay(),
		Activator.class.getResourceAsStream("fondo_hipy.jpg"));
		c = new Composite(com,SWT.NONE);
		com.setParent(c);
		Color color =
		com.getShell().getDisplay().getSystemColor(SWT.COLOR_DARK_BLUE);
		Color color1 =
		com.getShell().getDisplay().getSystemColor(SWT.COLOR_YELLOW);
		labFile = new Label(c,SWT.NULL);
		labFile.setText("Nombre archivo:");
		labFile.setBackground(color1);
		labFile.setForeground(color);
		m2tBtn = new Button(c,SWT.PUSH);
		m2tBtn.setText("M2T");

		t2mBtn = new Button(c,SWT.PUSH);
		t2mBtn.setText("T2M");

		createDiagramBtn =new Button(c,SWT.PUSH);
		createDiagramBtn.setText("Crear diagrama");

		openModelBtn = new Button(c,SWT.PUSH);
		openModelBtn.setText("Abrir");

		validateModelBtn = new Button(c,SWT.PUSH);
		validateModelBtn.setText("Validar");

		validateCodeBtn = new Button(c,SWT.PUSH);
		validateCodeBtn.setText("Validar");

		openCodeBtn = new Button(c,SWT.PUSH);
		openCodeBtn.setText("Abrir");

		c.addPaintListener(this);
	}
	@Override
	/**
	 * Controla y reubica los elementos según cambios en el tamaño de la
	 * ventana en mi vista
	 */
	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);

        if(composite.getClientArea().width!=1298){
                labFile.setBounds(composite.getClientArea().width-
                composite.getClientArea().width/3+5, composite.getClientArea().height-
                composite.getClientArea().height/10, 245,20);

                m2tBtn.setBounds((composite.getClientArea().width/2)+(composite.getClient
                Area().width/32), (composite.getClientArea().height/2)-
                (composite.getClientArea().height/6), 55, 20);
                t2mBtn.setBounds((composite.getClientArea().width/2)-
                (composite.getClientArea().width/15),(composite.getClientArea().height/2)
                -(composite.getClientArea().height/6),55,20);

                createDiagramBtn.setBounds((composite.getClientArea().width/2)-
                (composite.getClientArea().width/3+6),composite.getClientArea().height-
                (composite.getClientArea().height/5+10), 85, 20);
                openModelBtn.setBounds((composite.getClientArea().width/2)-
                (composite.getClientArea().width/3+6),
                (composite.getClientArea().height/2)-(composite.getClientArea().height/6-
                5), 55, 20);

                validateModelBtn.setBounds((composite.getClientArea().width/2)-
                (composite.getClientArea().width/3+6),
                (composite.getClientArea().height/2)+(composite.getClientArea().height/30
                +15), 55, 20);

                validateCodeBtn.setBounds((composite.getClientArea().width/2)+(composite.
                getClientArea().width/3-30),
                (composite.getClientArea().height/2)+(composite.getClientArea().height/30
                +18), 55, 20);

                openCodeBtn.setBounds((composite.getClientArea().width/2)+(composite.getC
                lientArea().width/3-30), (composite.getClientArea().height/2)-
                (composite.getClientArea().height/6), 55, 20);
                        labFile.setFont(new
                Font(c.getShell().getDisplay(),"Broadway",12,SWT.BOLD));
        }else{
                labFile.setBounds(composite.getClientArea().width-
                composite.getClientArea().width/3+5, composite.getClientArea().height-
                composite.getClientArea().height/10, 265,20 );
                labFile.setFont(new
                Font(c.getShell().getDisplay(),"Broadway",16,SWT.BACKGROUND));

                m2tBtn.setBounds((composite.getClientArea().width/2)+(composite.getClient
                Area().width/32), (composite.getClientArea().height/2)-
                (composite.getClientArea().height/6), 75, 40);
                t2mBtn.setBounds((composite.getClientArea().width/2)-
                (composite.getClientArea().width/15),(composite.getClientArea().height/2)
                -(composite.getClientArea().height/6),75,40);

                createDiagramBtn.setBounds((composite.getClientArea().width/2)-
                (composite.getClientArea().width/3+6),composite.getClientArea().height-
                (composite.getClientArea().height/5+10), 85, 40);

                openModelBtn.setBounds((composite.getClientArea().width/2)(composite.getC
                lientArea().width/3+6), (composite.getClientArea().height/2)-
                (composite.getClientArea().height/6-5), 75, 40);
                validateModelBtn.setBounds((composite.getClientArea().width/2)-
                (composite.getClientArea().width/3+6),
                (composite.getClientArea().height/2)+(composite.getClientArea().height/30
                +15), 75, 40);

                validateCodeBtn.setBounds((composite.getClientArea().width/2)+(composite.
                getClientArea().width/3-30),
                (composite.getClientArea().height/2)+(composite.getClientArea().height/30
                +18), 75, 40);

                openCodeBtn.setBounds((composite.getClientArea().width/2)+(composite.getC
                lientArea().width/3-30), (composite.getClientArea().height/2)-
                (composite.getClientArea().height/6), 75, 40);

                }
        }
}
```

A2.3 Clase CommonActions

```
public class CommonActions {

	public boolean error;
	public CommonActions(){}

	/**
	 * Se encarga de la validación de el código seleccionado
	 * @param codePathForHurome, es la ruta donde se encuentra nuestro
	 * código al que realizamos la operación de validar
	 * @return true si la validación ha sido correcta o false en caso
	 * contrario
	 */
	public boolean validateCode(String codePathForHurome){
		Shell myShell = new Shell();
		BasicDiagnostic diagnostics;
		diagnostics = new BasicDiagnostic();
		error = !HuromeTools.validateRoboScript(codePathForHurome,
			diagnostics);

		if (error != false) {
			MessageBox message = new MessageBox(myShell,
				SWT.ICON_ERROR);
			message.setText("Validar código");
			message.setMessage("Error: código no valido" + "\n" +
			diagnostics);
			message.open();
			return false;

		} else {
			MessageBox message = new MessageBox(myShell,
			SWT.ICON_INFORMATION);
			message.setText("Validar código");
			message.setMessage("Validación correcta");
			message.open();
			return true;

		}
	}
	/**
	 * Se encarga de la validación de nuestro modelo, si seleccionamos un
	 * diagrama aplicara la acción de validar al mismo archivo pero con
	 * extensión .robonova
	 * @param modelPathForHurome, ruta donde se encuentra nuestro modelo a
	 * validar
	 * @return true en caso de que la validacción sea satisfactoria, false
	 * en caso contrario
	 */
	public boolean validateModel(String modelPathForHurome){
		Shell myShell = new Shell();
		BasicDiagnostic diagnostics;
		if(modelPathForHurome.contains("robonova_diagram")){

		modelPathForHurome=modelPathForHurome.replace("robonova_diagram",
		"robonova");
		}
		diagnostics = new BasicDiagnostic();
		error = !HuromeTools.validateHuromeModel(modelPathForHurome,diagnostics);

		if (error || diagnostics.getSeverity() == Diagnostic.ERROR ) {
			MessageBox message = new MessageBox(myShell,SWT.ICON_ERROR);
			message.setText("Validar modelo");
			if(error)
				message.setMessage(
					"Error: el modelo no existe o es invalido"
					+ "\n" + "Detalles: "
					+ "Error en los argumentos de entrada");
			else
				message.setMessage(
					"Error: el modelo no existe o es invalido"
					+ "\n" + "Detalles: " + diagnostics);
			message.open();
			return false;
```

```
		} else {
			MessageBox message3 = new
			MessageBox(myShell,SWT.ICON_INFORMATION);
			message3.setText("Validar modelo");
			message3.setMessage("Validación correcta");
			message3.open();
			return true;
		}
	}
	/**
	 * Realiza la transformación modelo a código
	 * @param modelPathForHurome, ruta donde se encuentra el modelo a
	 * transformar
	 * @return true en caso de transformación correcta, false en otro caso
	 */
	public boolean transformacionM2T(String modelPathForHurome){
		String seleccion;
		boolean abrir,hacerAmbos;
		int value;
		IPreferenceStore preferencia;
		Shell myShell = new Shell();
		boolean marcador = false;
		MyDialogoM2T dialogo;
		dialogo = new MyDialogoM2T(myShell);
		preferencia = Activator.getDefault().getPreferenceStore();
		seleccion = preferencia.getString("CHOICE");
		abrir = preferencia.getBoolean("BOOLEAN_VALUE_ABRIR");

		if(preferencia.getString("CHOICE").contains("choice1")){

			dialogo.setValorTransformacion(
				HuromeTools.M2T_GENERATE_ONLY_ROBOSCRIPT);

		}else if (preferencia.getString("CHOICE").contains("choice2")){

			dialogo.setValorTransformacion(
				HuromeTools.M2T_GENERATE_ONLY_ROBOBASIC);

		}else if(preferencia.getString("CHOICE").contains("choice3")){

			dialogo.setValorTransformacion(
				HuromeTools.M2T_GENERATE_ALL);
		}
		value=dialogo.open();
		if(value==1 && dialogo.solucion==1){

			if(modelPathForHurome.contains(".robonova_diagram")){
				modelPathForHurome =
					modelPathForHurome.replace(
					".robonova_diagram
					", ".robonova");
				System.out.println(
					"la ruta para el diagrama es:  "
					+modelPathForHurome);
				error =
					!HuromeTools.runHurome2Code(
					modelPathForHurome,
					dialogo.proyecto, "", dialogo.nombre,
					dialogo.valorTransformacion);

				if (error != true) {
					MessageBox message = new MessageBox(myShell,
							SWT.ICON_INFORMATION);
					message.setText("M2T");
					message.setMessage("M2T realizado");
					message.open();
					marcador=false;
					//a.setState(StateId.INICIAL);
				} else {
					MessageBox message2 = new MessageBox(myShell,
							SWT.ICON_ERROR);
					message2.setText("M2T");
					message2.setMessage("Error en la transformación" );
					message2.open();
					marcador=false;
					//a.setState(StateId.INICIAL)}
```

```
if(preferencia.getBoolean(
        "BOOLEAN_VALUE_ABRIR")==true){
        hacerAmbos=false;
        IPath workspace =
                ResourcesPlugin.getWorkspace()
                .getRoot().getLocation();
        File file;
        IFileStore fileStore;

        Path rutaArchivo = new
        Path(workspace+"/"+dialogo.proyecto
                +"/"+"src-gen"+"/"
                +dialogo.nombre+".rsf");
        file = new File(workspace
                +"/"+dialogo.proyecto+"/"
                +"src-gen"+"/"+dialogo.nombre+".bas");
        fileStore = EFS.getLocalFileSystem()
                .getStore(file.toURI());

        IFile xtextFile =
                ResourcesPlugin.getWorkspace()
                .getRoot()
                .getFileForLocation(rutaArchivo);

        if (xtextFile == null) {
                MessageBox message = new
                        MessageBox(myShell,
                        SWT.ICON_ERROR);
                message.setText("Abrir código");
                message.setMessage("Archivo debe estar en
                WORKSPACE");
                message.open();
        }
        IEditorInput editorInput =
                new FileEditorInput(xtextFile);
        IWorkbenchPage page =
                PlatformUI.getWorkbench()
                .getActiveWorkbenchWindow()
                .getActivePage();

        if(dialogo.valorTransformacion ==
                HuromeTools
                .M2T_GENERATE_ONLY_ROBOSCRIPT){

                page.openEditor(editorInput,
                        HuromeTools.ROBOSCRIPT_EDITOR_ID);
                marcador=true;

        }else if(dialogo.valorTransformacion ==
                HuromeTools.M2T_GENERATE_ONLY_ROBOBASIC){

                IDE.openEditorOnFileStore(page,
                        fileStore);
                marcador=true;

        } else if(dialogo.valorTransformacion ==
                HuromeTools.M2T_GENERATE_ALL){

                page.openEditor(editorInput,
                HuromeTools.ROBOSCRIPT_EDITOR_ID);
                marcador=true;

                IDE.openEditorOnFileStore(
                        page, fileStore);
        }
}


}else if(value==3 && dialogo.solucion==1){

        MessageBox message = new MessageBox(
                myShell,SWT.ICON_INFORMATION);
                message.setText("M2T");
        message.setMessage("Debe rellenar los campos");
        message.open();
```

```
		}else {
			dialogo.close();
		}
		return marcador;
	}

	public void widgetDefaultSelected(SelectionEvent e) {}

	/**
	 * Realiza la transformación texto a modelo
	 * @param codePathForHurome, ruta donde se encuentra el código a transformar
	 * @return true en caso de transformación correcta, false en caso contrario
	 */
	public boolean transformacionT2M(String codePathForHurome){
		String outModelPath;
		boolean resultado = false;
		boolean abrir;
		Shell myShell = new Shell();
		FileDialog myDialog = new FileDialog(myShell,SWT.SAVE);
		myDialog.setFilterNames(new String[]{"Robonova"});
		myDialog.setFilterExtensions(new String[]{"*.robonova"});
		myDialog.setFilterPath("/D:/");
		outModelPath = myDialog.getFilterPath();
		outModelPath = myDialog.open();
		outModelPath =  "/" + outModelPath.replace("\\", "/");

		error =  !HuromeTools.runCode2Hurome(codePathForHurome,outModelPath);

		if (error != true) {
			MessageBox message = new MessageBox(myShell,SWT.ICON_INFORMATION);
			message.setText("T2M");
			message.setMessage("T2M realizado");
			message.open();
			resultado=false;
			//a.setState(StateId.T2M_PROCCESSED);
		} else {
			MessageBox message2 = new MessageBox(myShell,SWT.ICON_ERROR);
			message2.setText("T2M");
			message2.setMessage("Error en la transformación");
			message2.open();
			resultado=true;
			//a.setState(StateId.INICIAL);
		}
		IPreferenceStore preferencia;
		preferencia = Activator.getDefault().getPreferenceStore();
		abrir = preferencia.getBoolean("BOOLEAN_VALUE_ABRIR");

		if(abrir==true && !preferencia.getBoolean("BOOLEAN_VALUE_DIAG")){
			File f = new File(outModelPath);
			IFileStore fileStore =
				EFS.getLocalFileSystem().getStore(f.toURI());
			IWorkbenchPage page =
				PlatformUI.getWorkbench().getActiveWorkbenchWindow()
				.getActivePage();

			IDE.openEditorOnFileStore(page, fileStore);

		}
		if(preferencia.getBoolean("BOOLEAN_VALUE_DIAG")==true){
			error = !HuromeTools.initializeDiagramFile(outModelPath,null);

			if (error != true) {
				MessageBox message = new MessageBox(myShell,
					SWT.ICON_INFORMATION);
				message.setText("Crear diagrama");
				message.setMessage("Diagrama realizado");
				message.open();
				//a.setState(StateId.INICIAL);
				resultado=true;
			} else {
				MessageBox message2 = new MessageBox(myShell,
					SWT.ICON_ERROR);
				message2.setText("Crear diagrama");
				message2.setMessage("Error en la creación");
					message2.open();
					//a.setState(StateId.INICIAL);
```

```
                                        resultado=true;
                                }

                                String path = outModelPath.replace("robonova",
                                        "robonova_diagram");
                                IWorkbenchPage page = PlatformUI.getWorkbench()
                                        .getActiveWorkbenchWindow().getActivePage();
                                IFile modelFile =
                                        ResourcesPlugin.getWorkspace().getRoot()
                                        .getFileForLocation(new Path(path));
                                IEditorInput editorInput = new FileEditorInput(modelFile);
                                page.openEditor(editorInput, HuromeTools.HUROME_EDITOR_ID);
                        }

                return resultado;
        }
}
```

A2.4 Clase ListenerActions

```
public class ListenerActions implements SelectionListener {
        String archivo, MODEL_PATH, CODE_PATH, modelPathForHurome,
                codePathForHurome,selecion;
        Composite parent;
        AbstractViewLook a;
        boolean error,abrir,crearDia,hacerAmbos;
        String roboScriptPath, outModelPath,seleccion;
        IFileStore fileStore;
        File file2Open;
        MyDialogoM2T dialogo;
        int value,tipeToGenerate;
        IPreferenceStore preferencia;
        CommonActions actions = new CommonActions();
        public ListenerActions(){}

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


        /**
         * Método principal de la clase, controlando SelectionEvent conocmos el widget
         * que produjo el evento y realizamos las operaciones correspondientes
         */

        public void widgetSelected(SelectionEvent e) {
                try{
                        Button bo;
                        bo = (Button) e.getSource();
                        if (bo.getData("id") == AbstractViewLook.OPEN_MODEL_BTN_ID) {
                                FileDialog f = new FileDialog(parent.getShell(), SWT.OPEN);
                                f.setFilterNames(new String[] {
                                        "Archivo ROBONOVA (*.robonova)",
                                        "Archivo DIAGRAMA(*.robonova_diagram)", "Todos" });
                                f.setFilterExtensions(new String[] { "*.robonova",
                                        "*.robonova_diagram", "*.*" });
                                MODEL_PATH = f.open();
                                archivo = f.getFileName();
                                a.changeArchiveName(archivo);
                                if (MODEL_PATH != null) {
                                        file2Open = new File(MODEL_PATH);
                                        IFile modelFile = ResourcesPlugin.getWorkspace()
                                                .getRoot().getFileForLocation(
                                                new Path(MODEL_PATH));
                                        System.out.println("modelFile:" + modelFile);
                                        if (modelFile == null) {
                                                MessageBox message = new
                                                MessageBox(parent.getShell(),
                                                        SWT.ICON_ERROR);
                                                message.setText("Abrir modelo");
                                                message.setMessage(
                                                "Archivo debe estar en el WORKSPACE");
```

```
                        message.open();
                    }
                    IWorkbenchPage page = PlatformUI.getWorkbench()
                            .getActiveWorkbenchWindow().getActivePage();
                    IEditorInput input = new FileEditorInput(modelFile);

                    if (archivo.contains(".robonova")) {

                        page.openEditor(input,
                                HuromeTools.HUROME_TREE_EDITOR_ID);
                        a.setState(StateId.MODEL_OPEN);

                    } else if (archivo.contains(".robonova_diagram")) {
                        page.openEditor(input,
                                HuromeTools.HUROME_EDITOR_ID);
                        a.setState(StateId.MODEL_OPEN);
                        MODEL_PATH.replace(".robonova_diagram",
                                ".robonova");

                    } else {
                        MessageBox message = new MessageBox(
                                parent.getShell(),
                                SWT.ICON_ERROR);
                                message.setText("Abrir modelo");

                        message.setMessage("Archivo no válido");
                        message.open();
                        a.setState(StateId.INICIAL);
                    }

                    modelPathForHurome = "/" + MODEL_PATH.replace("\\", "/");
                }

            } else if (bo.getData("id") == AbstractViewLook.OPEN_CODE_BTN_ID) {
                FileDialog f = new FileDialog(parent.getShell(), SWT.OPEN);
                f.setFilterNames(new String[] { "Archivo RSF (*.rsf)", "Todos" });
                f.setFilterExtensions(new String[] { "*.rsf", "*.*" });
                CODE_PATH = f.open();
                archivo = f.getFileName();
                a.changeArchiveName(archivo);
                if (archivo.contains(".rsf"))
                    a.setState(StateId.CODE_OPEN);

                if (CODE_PATH != null) {
                    IWorkbenchPage page = PlatformUI.getWorkbench()
                            .getActiveWorkbenchWindow().getActivePage();

                    Path rutaArchivo = new Path(CODE_PATH);
                    System.out.println("pasando a PATH()" + rutaArchivo);
                    if (rutaArchivo.isAbsolute() == true) {
                        System.out.println("es absoluta");
                    }
                    if (rutaArchivo.isValidPath(CODE_PATH) == true) {
                        System.out.println("es path valido");
                    }
                    codePathForHurome = "/" + CODE_PATH.replace("\\", "/");


                    IFile xtextFile = ResourcesPlugin.getWorkspace().getRoot()
                            .getFileForLocation(new Path(CODE_PATH));
                    if (xtextFile == null) {
                        MessageBox message =
                                new MessageBox(parent.getShell(),
                                SWT.ICON_ERROR);
                        message.setText("Abrir código");
                        message.setMessage(
                                "Archivo debe estar en WORKSPACE");
                        message.open();
                    }

                    IEditorInput editorInput = new FileEditorInput(xtextFile);
                    page.openEditor(editorInput,

                    HuromeTools.ROBOSCRIPT_EDITOR_ID);
                    a.setState(StateId.CODE_OPEN);
                }
```

```
		} else if(bo.getData("id") == AbstractViewLook.VALIDATE_CODE_BTN_ID) {
			boolean smsState = actions.validateCode(codePathForHurome);
			if (smsState==false){
				a.setState(StateId.CODE_VALIDATED_FAILED);
			}else{
				a.setState(StateId.CODE_VALIDATED_OK);
			}

		} else if (bo.getData("id")==AbstractViewLook.VALIDATE_MODEL_BTN_ID) {
			boolean smsState = actions.validateModel(modelPathForHurome);
			if(smsState==false){
				a.setState(StateId.MODEL_VALIDATED_FAILED);
			}else{
				a.setState(StateId.MODEL_VALIDATED_OK);
			}

		} else if (bo.getData("id") == AbstractViewLook.MT2_BTN_ID) {
			boolean smsState = actions.transformacionM2T(modelPathForHurome);
			if(smsState==true){
				a.setState(StateId.CODE_OPEN);
			}else{
				a.setState(StateId.INICIAL);
			}

		} else if (bo.getData("id") == AbstractViewLook.T2M_BTN_ID) {
			boolean smsState = actions.transformacionT2M(codePathForHurome);
			if(smsState==true){
			a.setState(StateId.INICIAL);

			}else{
				a.setState(StateId.T2M_PROCCESSED);
			}

	} else if (bo.getData("id") == AbstractViewLook.CREATE_DIAG_BTN_ID) {
			error = !HuromeTools.initializeDiagramFile(outModelPath,null);

			if (error != true) {
				MessageBox message = new MessageBox(parent.getShell(),
					SWT.ICON_INFORMATION);
				message.setText("Crear diagrama");
				message.setMessage("Diagrama realizado");
				message.open();
				a.setState(StateId.INICIAL);
			} else {
				MessageBox message2 = new MessageBox(parent.getShell(),
					SWT.ICON_ERROR);
				message2.setText("Crear diagrama");
				message2.setMessage("Error en la creación");
				message2.open();
				a.setState(StateId.INICIAL);
			}
	}
}catch(Exception excep){
	System.out.println("entra en excep");
	excep.printStackTrace();
	MessageBox message = new MessageBox(parent.getShell(),SWT.ICON_ERROR);
	message.setText("Excepción");
	message.setMessage(excep.getMessage());
	message.open();
}}

public boolean validateCode(){
	BasicDiagnostic diagnostics;
	diagnostics = new BasicDiagnostic();
	error = !HuromeTools.validateRoboScript(codePathForHurome,diagnostics);

	if (error != false) {
		MessageBox message = new MessageBox(parent.getShell(),
			SWT.ICON_ERROR);
		message.setText("Validar código");
		message.setMessage("Error: código no valido" + "\n" + diagnostics);
		message.open();
		return false;

	} else {
		MessageBox message = new MessageBox(parent.getShell(),
```

```
                        SWT.ICON_INFORMATION);
                message.setText("Validar código");
                message.setMessage("Validación correcta");
                message.open();
                return true;
        }
}
@Override
public void widgetDefaultSelected(SelectionEvent e) {}
}
```

A2.5 Clase HuromePluginView

```
public class HuromePluginView extends ViewPart {

        MyViewLook apariencia;
        IPreferenceStore preferencia;
        String eleccion;
        Boolean activarAbrir,crearDiag;


        public HuromePluginView() {   }

        /**
         * Método principal de la creación de la vista, creamos la apariencia sobre el
         * Composite,configuramos el estado inicial y añadimos los escuchadores a los
         * widgets de la misma
         */
        public void createPartControl(Composite parent) {
                apariencia = new MyViewLook(parent);
                apariencia.setState(StateId.INICIAL);
                apariencia.initialize(new ListenerActions(apariencia,parent));
        }

        public void setFocus() {}
}
```