Un entorno de modelado y desarrollo para sistemas sociales

Sistemas Informáticos
2008 – 2009

Facultad de Informática
Universidad Complutense de Madrid

Francisco Domenech Marín <franciscodomenechmarin@gmail.com>
Roberto Jiménez Domínguez <rjd_84@hotmail.com>
Jorge Jiménez Rodríguez <frosklyss6@gmail.com>

Dirigido por:
Rubén Fuentes Fernández

Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código desarrollados en éste proyecto.

Francisco Domenech Marín Roberto Jiménez Domínguez Jorge Jiménez Rodríguez

Resumen. El modelado de aplicaciones es una pieza clave para el posterior desarrollo de las mismas, y más aún en aquellas situaciones en las que el contexto social toma parte activa. El proceso de modelado de este tipo de aplicaciones se hace difícil si no se cuenta con unos cimientos y un lenguaje adecuado que logren capturar todos los requisitos e interacciones existentes en el sistema. La Teoría de Actividad junto con el lenguaje UML-AT nacen para completar este proceso, definiendo para ello una serie de clases y relaciones que ayudan y facilitan el modelado de este tipo de aplicaciones. El objetivo de éste proyecto es el desarrollo de una herramienta de modelado que permita la elaboración de diagramas UML-AT, así como la generación automática de reglas de transformación ATL entre modelos, las cuales pueden utilizarse posteriormente en otras aplicaciones ya existentes.
Palabras clave: Teoría de Actividad, Arquitectura dirigida por modelos, Ecore, EMF, GEF, ATL.

Abstract. Modeling applications is a key element for further development, and even more in those situations in which the social context is active. The modeling process of such applications is difficult if you do not have a foundation and a language appropriate to capture all requirements and interactions in the system. Activity theory with the language UML-AT begin to complete this process for defining a series of classes and relationships that help and facilitate the modeling of such applications. The goal of this project is to develop a modeling tool that enables the development of UML-AT diagrams, as well as the automatic generation of ATL transformation rules between models, which can be subsequently used in other existing applications.
Keywords: Activity Theory, Model Driven Architecture, ECore, EMF, GEF, ATL.

Índice de contenido

1. Introducción

Hoy en día cada vez adquiere más importancia el contexto social en el proceso de desarrollo de aplicaciones software, es por ello que surge cada vez más la incesante necesidad de disponer de herramientas que ayuden y faciliten el trabajo de los diseñadores.

Bajo esta situación tiene lugar el presente proyecto, cuyo fin es desarrollar una herramienta de modelado que ayude al diseño de modelos basados en UML-AT [3], un lenguaje centrado en los conceptos de la Teoría de Actividad y de dominio procedente de las Ciencias Sociales.

1.1. Objetivos del proyecto

El objetivo del proyecto es el desarrollo de dos módulos Eclipse [1] (i.e. plug-in).
Por una parte, un primer módulo que permita el diseño de modelos UML-AT, los cuales ofrezcan todos los conceptos y relaciones presentes en la Teoría de Actividad.
Y por otra parte un segundo módulo que posibilite llevar a cabo transformaciones entre modelos, generando para ello las reglas de transformación necesarias en ATL [6] [7], un lenguaje específicamente concebido para definir transformaciones y que está soportado por Eclipse.

Para el desarrollo del proyecto se ha hecho uso de las siguientes tecnologías y entornos:

- *Eclipse* [1]: entorno de desarrollo muy extendido en la comunidad software debido a su alta calidad como tal y su alta versatilidad en cuanto a frameworks.

- *Ecore* [4]: lenguaje de definición de metamodelos soportado por Eclipse. Permite la creación de metamodelos de forma sencilla y eficiente, y junto con el EMF forman una gran herramienta para la construcción de metamodelos.

- *Eclipse Modeling Framework (EMF)* [2]: framework de modelado soportado por Eclipse y pensado para el desarrollo de aplicaciones software basadas en modelos de datos estructurados. Ofrece librerías y plug-ins que permiten la generación automática de clases Java a partir de metamodelos Ecore, facilitando el desarrollo de editores que permitan la creación y visualización del correspondiente lenguaje de modelado especificado.

- *Graphical Editing Framework (GEF)* [8]: framework de Eclipse que facilita el desarrollo del aspecto gráfico de una aplicación, ofreciendo una gran diversidad de posibilidades al respecto a la vez que hace sencilla su implementación.

- *Atlas Transformation Language (ATL)* [6] [7]: lenguaje de transformación entre modelos soportado por Eclipse y dentro del campo de la Ingeniería dirigida por modelos (MDE). Es el estándar de los lenguajes de transformación debido a su

simplicidad, y existe una gran cantidad de documentación al respecto lo que facilita su estudio e implementación.

1.2. Planificación del trabajo

Primeramente nuestra labor consistió en conocer y comprender aquellas tecnologías que debíamos usar en la elaboración de los plug-in.

Acto seguido y una vez conocido todo lo referente a la Teoría de Actividad [9], el primer paso fue la creación de un metamodelo que representase por completo dicho entorno; para ello se utilizó Ecore [4], un lenguaje de definición de metamodelos soportado por el Eclipse Modeling Framework (EMF) [2].

En lo referente a la elaboración del primer plug-in inicialmente tuvo lugar la configuración del mismo, cargando todos los elementos necesarios para su desarrollo y configurando las distintas opciones de identificación.
A continuación y sirviéndonos de la tecnología EMF generamos nuestro modelo para después ampliarlo convenientemente según nuestras necesidades, creando la estructura de paquetes y clases básicas necesarias para la implementación del mismo.

Añadidas las restricciones propias del lenguaje UML-AT e implementados los métodos necesarios hasta el momento tuvo lugar el desarrollo de la clase principal del modelo (Diagrama), aquella que contendría todos los elementos de los modelos UML-AT creados por el usuario, decidiéndose como estructura principal un *array* de entidades, ya que el resto de información nos vendría dada propiamente por dichas entidades.

En este punto, el siguiente paso fue definir la parte gráfica de nuestra aplicación haciendo uso del Graphical Editing Framework (GEF) [8] a la vez que se definía la comunicación entre dicha interfaz y nuestro modelo mediante el controlador.
El desarrollo de ambas partes tuvo lugar en paralelo ya que resultaba más cómodo a la hora de ir creando los componentes del editor e ir relacionándolos con las partes del modelo correspondientes. Para ello primeramente se elaboró todo lo referente a las entidades y después lo concerniente a los distintos tipos de relaciones existentes, de forma que cada nuevo cambio en la parte gráfica se enlazaba con el modelo mediante el desarrollo de los elementos pertinentes en el controlador.

Tras finalizar esta parte con la realización de las opciones de guardado y cargado de los modelos UML-AT, se realizaron diversas pruebas para comprobar la existencia o no de errores en las tres partes (modelo, vista y controlador) y llevar a cabo su corrección en caso de ser necesario.

Al final del desarrollo también introducimos un archivo de opciones para cargar las entidades y distintos tipos de relaciones a partir de un fichero XML.

En cuanto al desarrollo del segundo plug-in, se comenzó igualmente con el estudio de las tecnologías involucradas en el mismo, así como con su configuración inicial.

En esta segunda aplicación se reutilizó el código del primer plug-in, ya que las únicas diferencias con este último son por una parte los cambios de la interfaz gráfica y por otra la generación de las reglas de transformación ATL [6] [7] entre modelos.

Es por ello que su desarrollo tuvo lugar en dos fases, una primera fase en la que nuestro trabajo consistió en adecuar la parte gráfica a las necesidades existentes, creando un nuevo aspecto del editor que consiste en dos editores del primer plug-in separados por una serie de botones (un editor para el modelo fuente y otro para el modelo destino), y una segunda fase relacionada con la generación de las reglas de transformación ATL.

Para esta segunda fase el procedimiento seguido fue en un principio generar reglas de transformación entre modelos sencillos, únicamente con entidades, y posteriormente tener en cuenta modelos con relaciones, terminando por implementar el hecho de que en el modelo destino deben aparecer las descripciones de las entidades del modelo origen en caso de que exista coincidencia de identificadores.

Por último, al igual que en el primer plug-in, se llevaron a cabo toda una serie de pruebas para verificar la corrección de la aplicación.

La memoria del proyecto se empezó durante las últimas semanas del desarrollo del segundo plug-in, y partiendo de una estructura inicial que fue evolucionando tuvo lugar su elaboración punto por punto.

1.3. Contenido de la memoria

Tras una pequeña introducción acerca del contexto del proyecto, la sección 2 describe los conceptos en los que se basa la Teoría de Actividad, necesarios para el posterior desarrollo del metamodelo. A continuación en la sección 3 se hace hincapié en la creación del metamodelo mediante el lenguaje Ecore, para luego continuar en la sección 4 con la generación del modelo utilizando la tecnología EMF y el desarrollo de la parte gráfica del plug-in a través de la tecnología GEF. La sección 5 engloba todo lo referente al segundo plug-in, es decir, su diseño y cómo se lleva a cabo la generación del código de transformación ATL entre modelos. Las conclusiones finales tienen lugar en la sección 6, seguida de los agradecimientos en la sección 7, y de un anexo acerca de las equivalencias existentes en la Teoría de Actividad. Por último tienen lugar las referencias bibliográficas utilizadas así como un índice de figuras con el propósito de agilizar futuras búsquedas al respecto.

2. Teoría de Actividad

2.1. Introducción

La Teoría de Actividad (AT) es una teoría que proviene del ámbito de las Ciencias Sociales y cuyo diseño se centra en el lenguaje natural, a diferencia de muchos de los diseños de lenguajes de modelado.

El lenguaje de modelado elegido para su representación es el lenguaje UML [17]. El principal criterio para elegir este lenguaje es la gran difusión que tiene en la comunidad de Ingeniería de Software y su disponibilidad para definir nuevos elementos con restricciones específicas, en concreto en los dominios de la aplicación. Estos mecanismos son los estereotipos usados en los perfiles UML, éstos son un subtipo de los elementos básicos de un modelo pero con un significado y uso específico. Pueden usar valores etiquetados para almacenar valores que no pertenecen a sus elementos base. Suelen ir marcados entre "<<" y ">>".
La explicación estándar de esta teoría se encuentra descrita en la referencia Engeström 1987 [9].

Gracias a las similitudes existentes entre la AT y la Ingeniería de Software orientado a Agentes (AOSE), estos conceptos pueden ser descritos en dicho paradigma. Como resultado de esto los conceptos de la AT pueden ser formalizados en UML y así obtener lo que se conoce como UML-AT, el cual se describe en el siguiente apartado.

2.2. Vocabulario

A continuación se procede a la descripción del lenguaje UML-AT. Como hemos dicho anteriormente todo el conjunto de roles se encuentra descrito en la referencia de Engeström 1987 [9]. Se han añadido algunos roles nuevos como *activity system*, *objective* o *artifact*:

- *Activity system* (sistema de actividad), comprende todo el contexto de una actividad, es decir, todos los elementos relacionados con esa actividad y en otras especificaciones.

- *Objective* (objetivo), representa las necesidades satisfechas por el resultado de una actividad.

- *Artifact* (artefacto), es un comodín ya que puede representar cualquier concepto de la AT.

A continuación se muestra una figura que describe como los estereotipos están relacionados con las entidades en el lenguaje UML. Los estereotipos de las entidades del metamodelo son clases y relaciones:

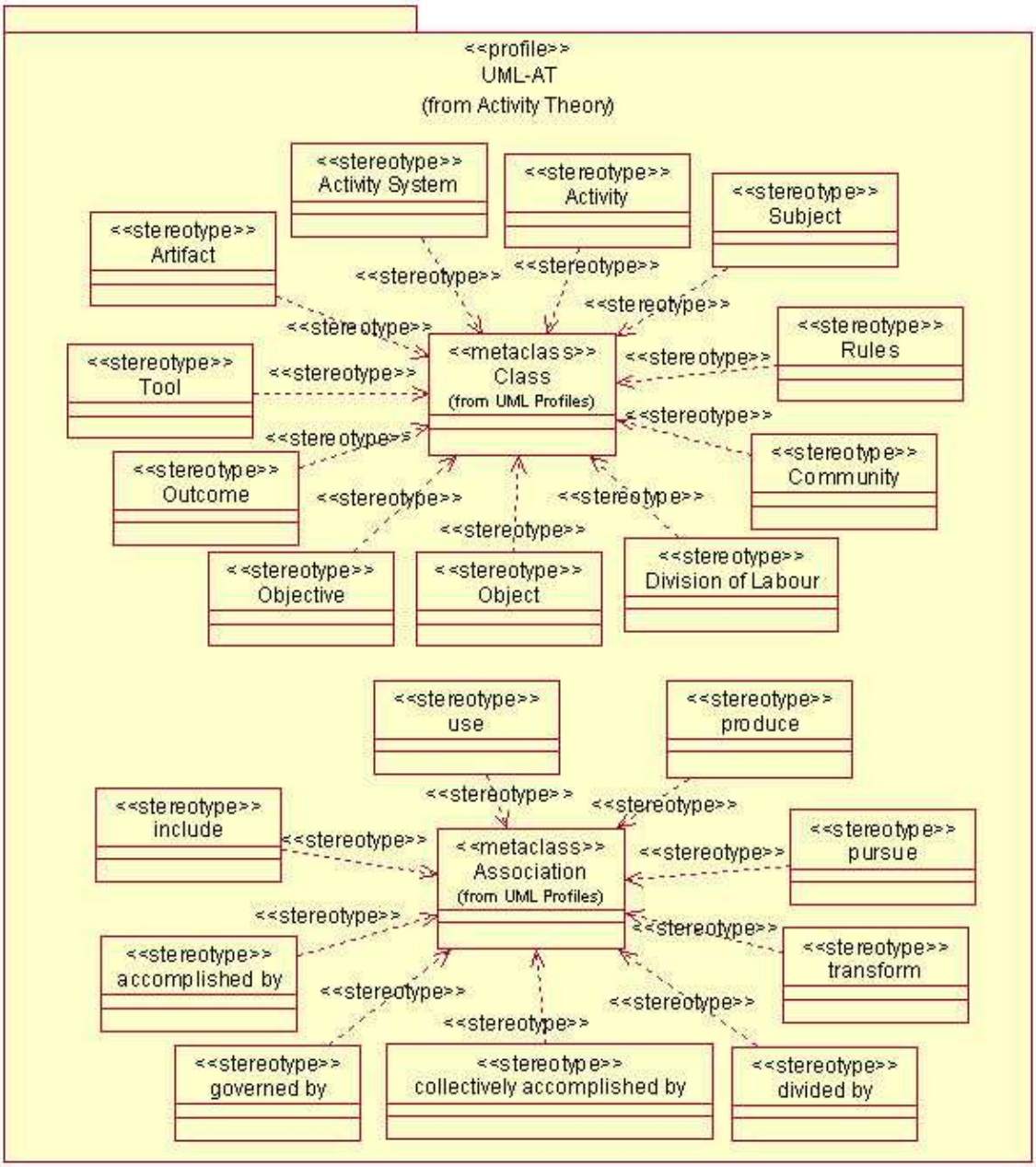

Fig. 2.1: Definición de lenguaje UML-AT mediante perfiles UML

En la siguiente figura se muestra como se unen los conceptos de la AT usando las relaciones descritas:

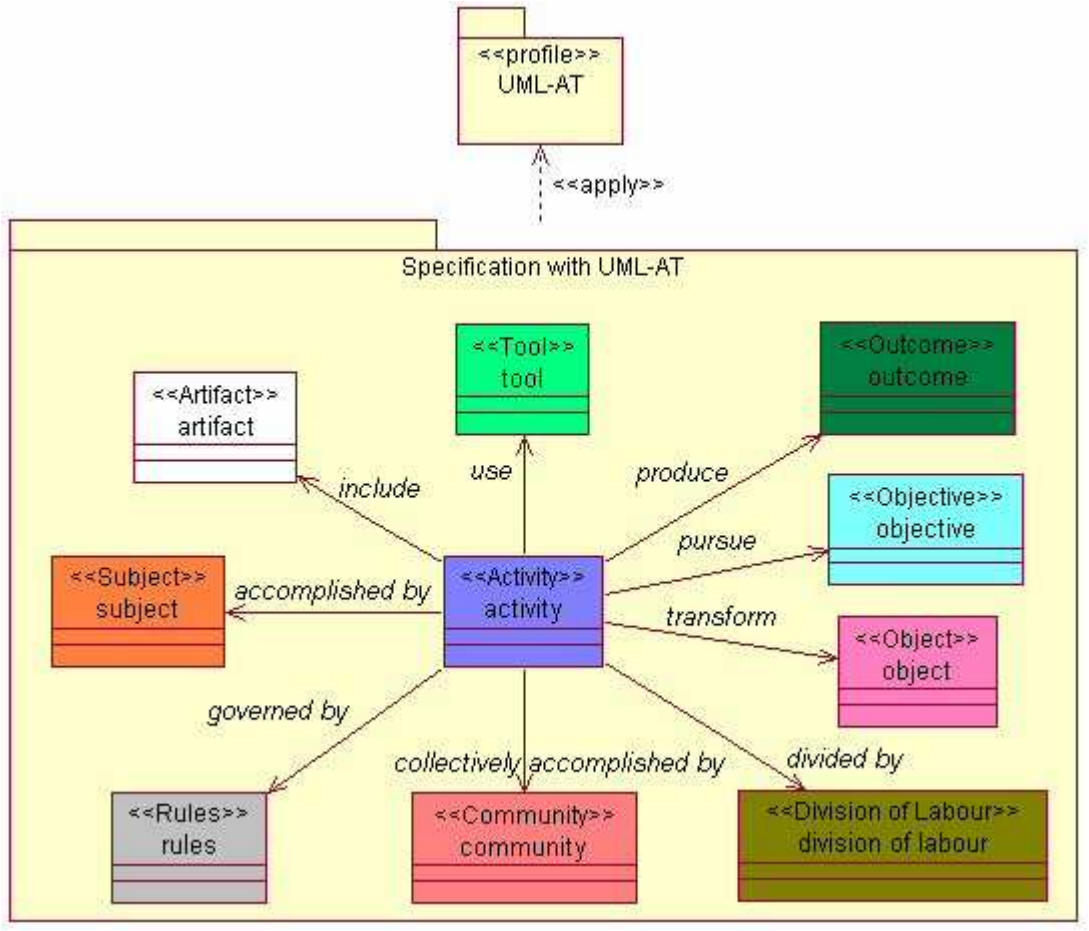

Fig. 2.2: Conceptos de la Teoría de Actividad y sus relaciones representadas con UML

La representación con estereotipos está complementada con un código de colores que facilita distinguir visualmente los roles.

Todos los roles y relaciones que se han descrito hasta ahora no son suficientes para describir y razonar sobre las actividades, por ejemplo, no se permiten relaciones de contribución, descomposición o transformación. A continuación se muestran la adicción de nuevos conceptos para solucionar estas carencias.

2.2.1. Relaciones con adornos

Tres son las posibles relaciones de este tipo que pueden aplicarse a cualquier relación del vocabulario. Estas relaciones permiten establecer el número de instancias en la cual se da la relación. Son las siguientes:

1. *ALL*, el patrón descrito en el diagrama significa que debe ser satisfecho para cualquier relación del tipo dado entre los conceptos dados.

2. *NOT*, se puede considerar como la negación de la anterior relación, y significa que no existe ninguna relación del tipo dado entre los conceptos dados.

3. *ANY*, indica que hay al menos una instancia de la relación descrita entre los conceptos dados.

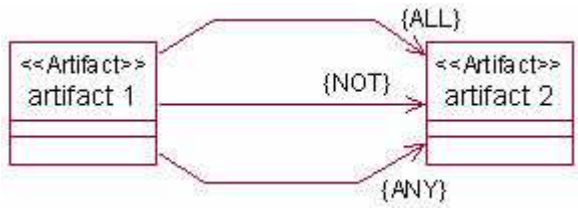

Fig. 2.3: Relación con *adornos*

Se usa el rol artefacto ya que estas relaciones se pueden dar en cualquier tipo de relación y los artefactos pueden representar cualquiera de los roles del sistema.

2.2.2. Relación *Change of Role* (Cambio de Rol)

Esta relación indica que un elemento de la especificación adopta un nuevo rol en la actividad del sistema, diferente al que ha llevado a cabo hasta el momento. Es el caso de artefactos entre actividades vecinas.

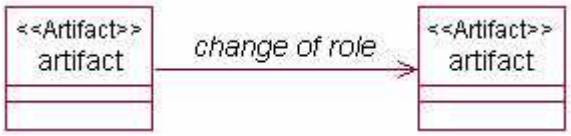

Fig. 2.4: Relación *cambio de rol*

2.2.3. Relación *Inheritance* (Herencia)

Cuando un concepto 1 hereda de otro concepto 2, significa que el primero de ellos puede substituir al segundo donde éste aparezca y además puede participar en cualquiera de sus relaciones.

Fig. 2.5: Relación de *herencia*

2.2.4. Relación *Play*

Este tipo de relación solo se puede dar entre los roles de *Subject* (sujeto), y puede considerarse como una especialización de la relación de herencia. Un sujeto 1 que hereda mediante la relación play de otro sujeto 2 quiere decir que el sujeto 1 persigue los mismos objetivos, lleva las mismas actividades, y pertenece a las mismas comunidades que el sujeto 2.

Fig. 2.6: Relación *play* entre sujetos

2.2.5. Relación *Pursue* (Perseguir)

Esta relación ya viene dada en la especificación estándar y se da entre los roles de actividad y objetivo. Esta relación conecta un sujeto con uno de los objetivos a través de la actividad que satisface la necesidad representada por ese objetivo. Con lo cual se incorpora a la especificación la tupla de roles (sujeto, objetivo), que se puede dar con esta relación. También se puede dar entre una comunidad y sus objetivos, ya que una comunidad representa un conjunto de sujetos que se dedican a una misma actividad.

Fig. 2.7: Relación *persigue* para un sujeto

2.2.6. Relaciones de Contribución

Las relaciones de contribución muestran como diferentes artefactos influyen entre sí en el cumplimiento de sus objetivos. Algunos ejemplos de esto se dan en la satisfacción o no de objetivos, la construcción de objetos, la ejecución de actividades etc.
Las relaciones de contribución consideradas en UML-AT son:

- *Guarantee*: una contribución *guarantee* implica que en presencia del Artefacto 1, el Artefacto 2 siempre realiza su rol.

- *Essential*: en este caso la presencia del Artefacto 1 es obligatoria para que el Artefacto 2 pueda realizar su rol, sin embargo esta presencia por si sola no garantiza dicha realización.

- *Contribute positively*: una contribución positiva indica que el Artefacto 1 puede ayudar a que el Artefacto 2 cumpla su rol, aunque no llega a ser una garantía de su satisfacción.

- *Contribute negatively*: por el contrario una contribución negativa indica que el Artefacto 1 dificulta el cumplimiento de su rol al Artefacto 2, no haciéndolo imposible.

- *Undefined*: el Artefacto 1 afecta al cumplimiento del rol asignado al Artefacto 2, todo ello sin conocerse como o siendo dependiente de las circunstancias.

- *Impede*: una contribución de impedimento implica que el Artefacto 1 inhibe completamente la posibilidad de que el Artefacto 2 satisfaga su rol.

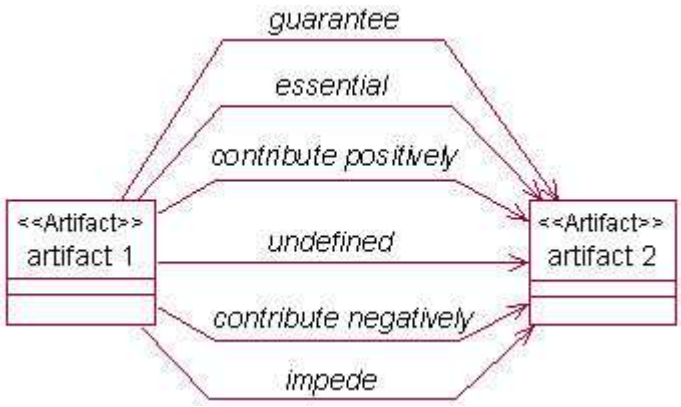

Fig. 2.8: Relaciones de *contribución* entre artefactos

En aquellos casos en los que no se diga lo contrario, la presencia o no del Artefacto 1 no tiene efecto en el cumplimiento del rol del Artefacto 2.

2.2.7. Relaciones *Satisfy* y *Fail*

Los modeladores deben considerar el hecho de que las relaciones de contribución pueden ser implícitas.
Una relación "persigue" entre una actividad y su objetivo no lleva implícito que el objetivo siempre se satisfaga cuando la actividad se ha realizado. Es por ello que se necesitan las relaciones *satisfy* y *fail* para distinguir estos dos casos:

- *Satisfy*: esta relación describe un contexto en el que la ejecución de la actividad ha generado los resultados necesarios para satisfacer las necesidades de su sujeto.

- *Fail*: por otra parte, una relación *fail* se corresponde con un contexto en el que la actividad no podía satisfacer su objetivo. Esta imposibilidad del cumplimiento del objetivo puede deberse a múltiples razones, entre ellas la no finalización de la actividad o la destrucción, por parte de otras actividades, de objetos requeridos.

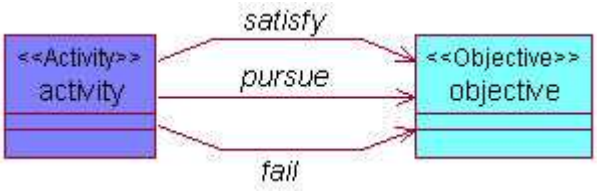

Fig. 2.9: Posibles relaciones de una actividad con el objetivo que persigue

2.2.8. Relación *Connect*

Debido a que puede haber una serie de actividades vecinas conectadas mediante asociaciones productor-consumidor, donde una actividad produce un artefacto que es parte del sistema de otra actividad, puede darse el caso en el cual un artefacto conecte actividades que no son relevantes para el análisis. Es por ello que se necesita una nueva relación para evitar la introducción de artefactos solo para conectar actividades.

La relación *connect* entre dos actividades significa que la Actividad 1 genera algún artefacto que aparece en el sistema de la Actividad 2. Así mismo implica cierta precedencia en el tiempo por parte de la Actividad 1 frente a la Actividad 2.

Fig. 2.10: Relación *connect* entre actividades

2.2.9. Relación *Consume*

La ejecución de una actividad implica la aplicación de diferentes artefactos a la hora de transformar objetos o usar herramientas. Como presunción general se supone que los recursos utilizados por una actividad continúan existiendo tras su utilización. Sin embargo una actividad a veces puede destruir el recurso tras utilizarlo. La relación consume describe esta situación, y únicamente puede aplicarse sobre artefactos que tengan el rol de objeto o herramienta.

Fig. 2.11: Relación *consume*

2.2.10. Relación *Decompose*

Otro aspecto a considerar son las relaciones de descomposición entre artefactos. Los conceptos de la AT pueden ser descritos a diferentes niveles de detalle, considerando solo la información relevante sobre una actividad en cada momento, de ahí este tipo de relaciones. Existen tres tipos de descomposiciones:

- *Decompose-AND*: el artefacto se compone de la suma de los Artefactos 1 y 2.
- *Decompose-OR*: el artefacto puede ser sustituido por el Artefacto 1 ó 2.
- *Decompose*: finalmente esta relación representa cualquiera de las dos anteriores.

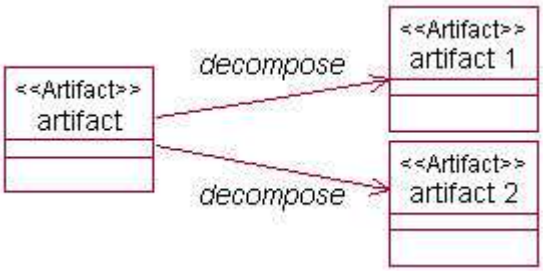

Fig. 2.12: Relación *decompose* entre artefactos

Un aspecto importante de las relaciones de descomposición es que dependiendo del rol al que se aplique las restricciones y la semántica es muy distinta. En el caso de objetos, objetivos, resultados, herramientas, sujetos y actividades, las relaciones de descomposición solo pueden conectar entidades del mismo rol, es decir, un objetivo solo puede descomponerse en otros objetivos, etc.
Sin embargo en el caso de comunidades, reglas o división de labor dicha semántica y restricciones cambian, de forma que una comunidad puede descomponerse en otras comunidades o en sujetos, las reglas pueden descomponerse en otras reglas o sistemas de actividad, y la división de labor puede descomponerse en otras divisiones de labor o en sistemas de actividad.

Otro aspecto a considerar es que estas relaciones pueden ser implícitas, tal y como ocurre en las relaciones de contribución, ya que si un artefacto se descompone en un segundo artefacto y éste último en un tercero, existe implícitamente una relación de descomposición entre el primer y tercer artefacto.

2.2.11. Relación *Surmount*

El último elemento del vocabulario de UML-AT es la relación *surmount* (superar) entre objetivos.
Esta relación establece prioridades entre objetivos, de forma que el Objetivo 1 es más importante que el Objetivo 2. De hecho, en caso de conflicto el sujeto debería intentar satisfacer el Objetivo 1 y después el Objetivo 2, centrándose sobre este último solo cuando el primero ha sido satisfecho e intentando siempre no perjudicarle.

Fig. 2.13: Relación *surmount* entre objetivos

Con este vocabulario queda descrito todo el conocimiento de la Teoría de Actividad con patrones estructurales, cuya representación no es única sino que depende del modelador, dándose el caso de múltiples representaciones de la misma información.
Los patrones estructurales pueden tener variables y valores fijos como propiedades de los roles y relaciones. Los valores fijos han de ir con dobles comillas y las variables sin nada.

Finalmente mencionar que el principal objetivo de este nuevo lenguaje no es introducir nuevas primitivas de modelado de agentes, sino permitir el uso de técnicas analíticas procedentes de las Ciencias Sociales en las ya existentes metodologías orientadas a agentes, al ser posible la traducción entre UML-AT y otros lenguajes.

3. Metamodelo Ecore

3.1. Introducción

Un metamodelo consiste en un modelo de un lenguaje de modelado, describiendo el conjunto de modelos admisibles.
Ecore [4] es el metamodelo usado por Eclipse Modeling Framework (EMF) [2], razón por la cual se escogió este tipo de metamodelo dado que trabajamos con EMF en una parte del proyecto.

Una característica importante de Ecore es que los metamodelos y modelos se representan en archivos XML [18] y además genera código automático a partir del metamodelo definido.

El lenguaje Ecore tiene los siguientes elementos:

- *EClass*: que contiene *EAttributes* y *EReferences*. Pueden extender de otras instancias *EClass*, heredando sus atributos.

- *EAttribute*: representa los atributos. Sus valores son de tipos primitivos: enteros, booleanos, cadenas de caracteres, etc.

- *EReference*: representa relaciones binarias entre dos instancias de *EClass*. La primera instancia de *EClass* contiene una instancia de *EReference* que señala a la otra instancia de *EClass*. Si la instancia de *EReference* es múltiple, quiere decir que en M1 se podrán instanciar varias veces en la *EClass* contenedora.

- *EPackage*: representa un paquete que agrupa elementos del metamodelo.

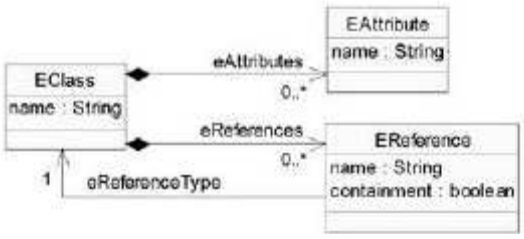

Fig. 3.1: Metamodelo Ecore

La representación de los metamodelos mediante el EMF es de la siguiente manera: los *EPackages* tienen *EClasses*, y estos a su vez tienen *EReferences* y *EAttributes*.
En la aplicación Eclipse el icono gráfico para cada uno de ellos tiene el siguiente aspecto:

- *EPackage*: cuadrado de color oscuro dividido en cuatro cuadrados iguales.

- *EClass*: rectángulo de color claro dividido por dos líneas horizontales. A la derecha del nombre, se indican las EClasses de las que heredan.

- *EReference*: rectángulo de color claro con flecha.

- *EAttribute*: rectángulo de color claro.

Además, contiene cuatro paquetes que son los siguientes:

1. *Entities*: contiene todas las entidades del lenguaje de modelado. Posee una entidad abstracta como raíz que se denomina *GeneralEntity*.

2. *Relations*: contiene los cuerpos de las relaciones. La raíz de este paquete es la entidad abstracta *GeneralRelation*.

3. *Association_end*: contiene los extremos de las relaciones. La raíz es la entidad abstracta *GeneralAssociationEnd*.

4. *Specification*: contiene la *Specification EClass*. Sus instancias son todas las raíces de los elementos del modelo. Contiene dos *EReferences*, *SEntities* y *SRelations* que contienen todas las relaciones y entidades del modelo.

3.2. Diseño del metamodelo

Para el diseño de la Teoría de Actividad usando Ecore se ha elegido la siguiente representación:

1. Los conceptos de la teoría se definirán como entidades, dentro del paquete *entities*.
2. Las relaciones, contribuciones y demás aportaciones se definirán como relaciones, dentro del paquete *relations*.
3. La multiplicidad de las relaciones vendrá especificado en el paquete *association_end*.
4. El paquete *specification* se deja como se genera.
5. Creamos un nuevo paquete llamado punto que contendrá las coordenadas (x,y) de los distintos puntos por los que vayan pasando las relaciones que se vayan generando en el editor.

El paquete raíz del cual cuelgan todos los demás se llama “umlat”. La vista del árbol queda de la siguiente manera:

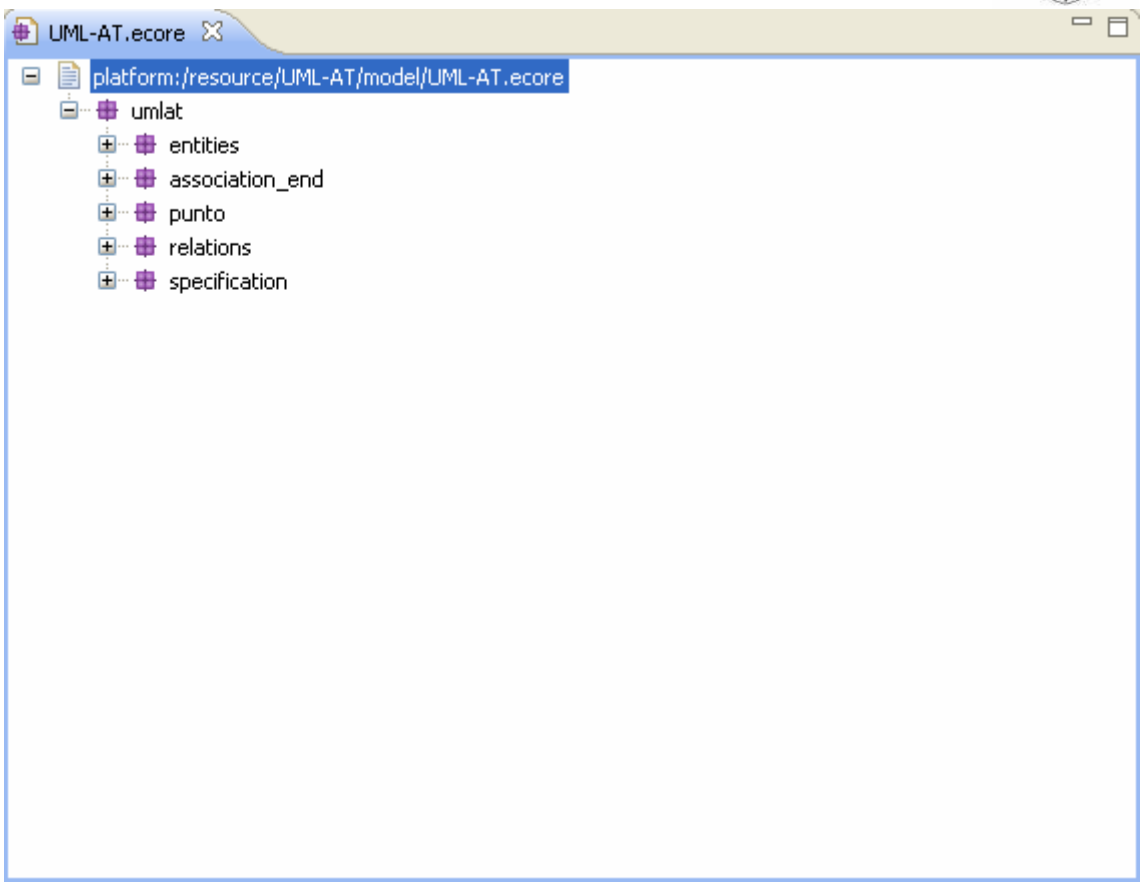

Fig. 3.2: Árbol jerárquico del metamodelo en Eclipse

3.2.1. Conceptos. Paquete *entities*.

Dentro de este paquete se encuentran todos los conceptos de la Teoría de Actividad, organizados de la siguiente forma:

- *GeneralEntity*: entidad abstracta que es la raíz de todos los demás elementos. Como todos van a heredar de esta entidad se han de poner todos los atributos comunes para los conceptos, como son:
  - *id*: identificador, tiene que ser único, no pueden existir dos entidades con el mismo identificador. Es de tipo *String*.
  - *Description*: es de tipo *String* y se utiliza para poner una breve descripción de la entidad.
  - *xPos*: posición de la entidad en el eje x del editor. Es de tipo entero.
  - *yPos*: posición de la entidad en el eje y del editor. Es de tipo entero.
  - *xSize*: anchura de la entidad. Es de tipo entero.
  - *ySize*: altura de la entidad. Es de tipo entero.

- *Artifact*: este concepto destaca sobre los demás, ya que como se ha explicado en el apartado correspondiente a la TA, cualquier concepto puede heredar de este; así por lo tanto *Artifact* hereda de *GeneralEntity* para que posteriormente y de forma implícita todos los demás al heredar de *Artifact* hereden automáticamente de *GeneralEntity*.

- Resto de los demás conceptos, entre los que se incluyen: *Activity_System*, *Activity*, *Subject*, *Rules*, *Community*, *Division_of_Labour*, *Object*, *Objective*, *Outcome* y *Tool*. Todos ellos heredan de *Artifact*.

En la siguiente figura se muestra como quedaría el paquete *entities*:

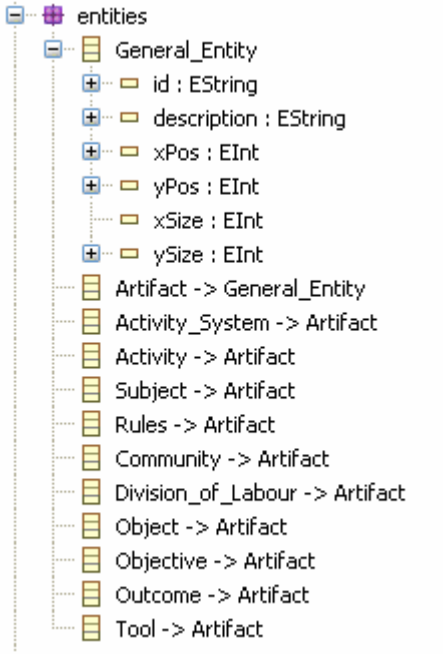

Fig. 3.3: Paquete *entities*

3.2.2. Relaciones. Paquete *relations*.

Dentro de este paquete se encuentran todas las relaciones del metamodelo, incluidas las de contribución y las de herencia. Su estructura es la siguiente:

- *Connection*: esta entidad la utilizamos como la raíz de todas las relaciones y en ella definimos tres tipos de referencias:

    - *entitySource*: que referencia a *association_end*, y refleja la entidad origen de la relación.
    - *entityTarget*: que también referencia a *association_end* y refleja la entidad destino de la relación.

- *bendPoints*: que referencia al paquete “punto”, para definir los puntos por los que pasa la relación en el editor a la hora de la visualización.

- *GeneralRelation*: entidad que hereda de *Connection*. Posee como atributo:
  - *adornment*: es de tipo *String*, y como se explicó en el punto correspondiente a la TA, refleja cuando se cumple esa relación (ALL, ANY, NOT.)

- Relaciones que heredan de *Connection*, entre ellas se encuentran: *Inheritance* (herencia), *UDecompose*, *UChangeOfRole*. Estas relaciones heredan de *Connection* y no de *GeneralRelation* porque no tienen la propiedad de *adornment*.

- Relaciones que heredan de *GeneralRelation*, entre ellas se encuentran: *UUse*, *UProduce*, *UTransform*, *UDividedBy*, *UCollectivelyAccomplishedBy*, *UGovernedBy*, *UInclude*, *UConnect*, *UConsume*, *USurmount*, *UPursue*, *UPlay* y *Contribution*. Como *GeneralEntity* hereda de *Connection*, estas relaciones tienen todas las referencias también.

- *USatisfy* y *UFail* heredan de *UPursue*.

- *Guarantee*, *Essential*, *ContributePositively*, *ContributeNegatively*, *Impede* y *Undefined* heredan de *Contribution* porque son los distintos tipos de relación que se pueden dar en contribución, ya que la relación Contribución no existe propiamente ella sola.

En la siguiente figura se muestra como quedaría el paquete *relations*:

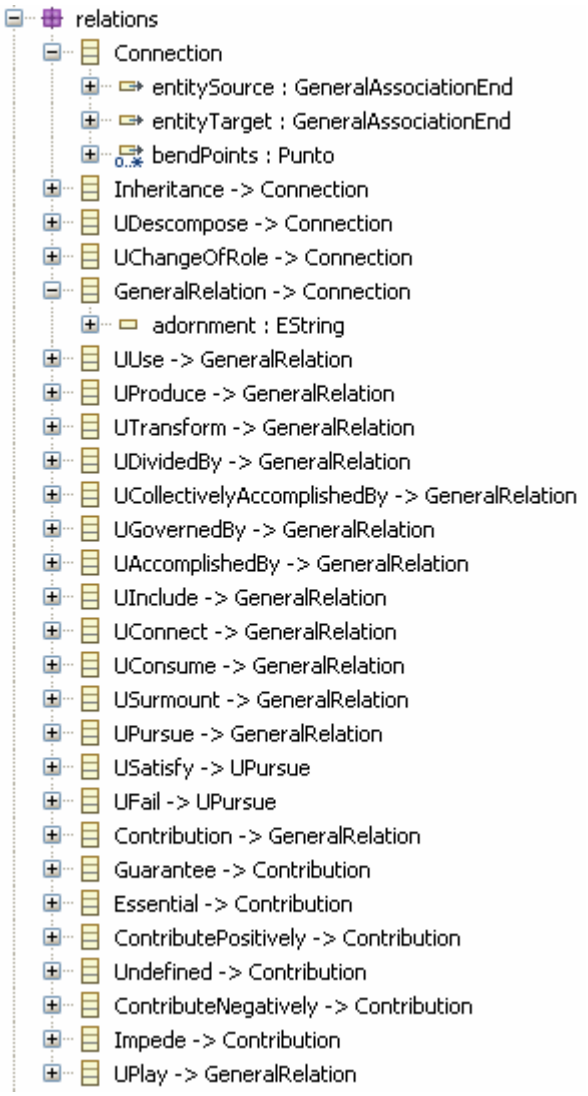

Fig. 3.4: Paquete *relations*

Al principio del desarrollo del metamodelo se especificó en el mismo las restricciones de los tipos de entidades posibles para cada tipo de relación, pero dados los problemas que surgieron (como cuando una relación puede tener varias entidades origen), se decidió introducirlo dentro de la parte del modelo.

3.2.3. Extremos de las relaciones. Paquete *association_end*.

Dentro de este paquete se encuentra la información referente a los extremos de una relación. Contiene la entidad *GeneralAssociationEnd* con los siguientes elementos:

- *multiplicityDown*: es de tipo entero y especifica el valor mínimo de la multiplicidad para ese extremo de la relación.

- *multiplicityUp*: es de tipo entero y especifica el valor máximo de la multiplicidad para ese extremo de la relación.

- *entity*: es una referencia a *GeneralEntity*, con lo que refleja todos los conceptos del sistema, y especifica la entidad a la que se une el extremo de la relación.

En la siguiente figura se muestra como quedaría el paquete *association_end*:

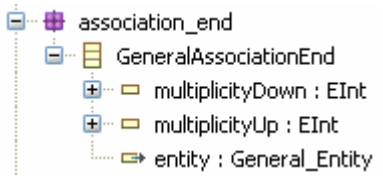

Fig. 3.5: Paquete *association_end*

Coordenadas de las relaciones. Paquete *punto*.

Dentro de este paquete se encuentra la información de las coordenadas de los puntos por los que pasa una relación en el editor. Esto se da cuando a la hora de establecer una relación entre dos entidades se establecen varios puntos por el camino de tal forma que no es una línea recta la que los une. Tiene el siguiente elemento:

- *Punto*: que contiene los atributos *x* e *y* que son de tipo entero y reflejan las coordenadas del punto de apoyo de la relación.

La siguiente figura muestra el contenido del paquete punto:

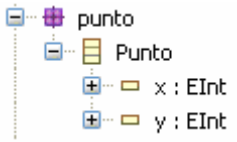

Fig. 3.6: Paquete *punto*

Paquete *specification*.

Este paquete almacena todas las entidades y relaciones (de los paquetes *entities* y *relations*), que se van generando en el editor. Contiene dos referencias, una a cada uno de los paquetes nombrados anteriormente, *SEntities* y *SRelations*:

Fig. 3.7: Paquete *specification*

3.3. Generación del metamodelo con Eclipse

Para poder introducir todo lo diseñado anteriormente en la aplicación Eclipse, estuvimos trabajando con la herramienta *Ecore Diagram*, proporcionada por el propio Eclipse. Mediante esta herramienta se llevó a cabo toda la especificación vista anteriormente.

Ecore Diagram ofrece dos posibles maneras de trabajo que explicamos a continuación, con las cuales trabajamos para poder reproducir el metamodelo. Las dos posibles maneras son las siguientes:

1. La utilización de un editor en el que aparece una ventana de edición y una paleta con todos los elementos que se pueden introducir. Con ella se trabaja todo gráficamente, añadiendo clases, enlazándolas, etc.

   La vista de la paleta es la siguiente:

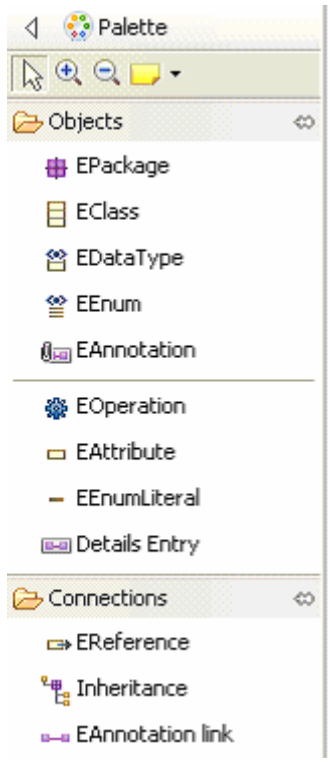

Fig. 3.8: Paleta del editor de *Ecore Diagram*

El procedimiento a seguir fue ir creando los paquetes necesarios (*EPackage*), dentro de los cuáles establecíamos las clases correspondientes (*EClass*), y en el caso de que una clase tuviera atributos especificábamos éstos mediante los *EAttribute*.
Para las relaciones usamos las conexiones de *EReference* e *Inheritance*.

2. La segunda opción consiste en una vista en árbol, en la que se define como está siendo diseñado el metamodelo:

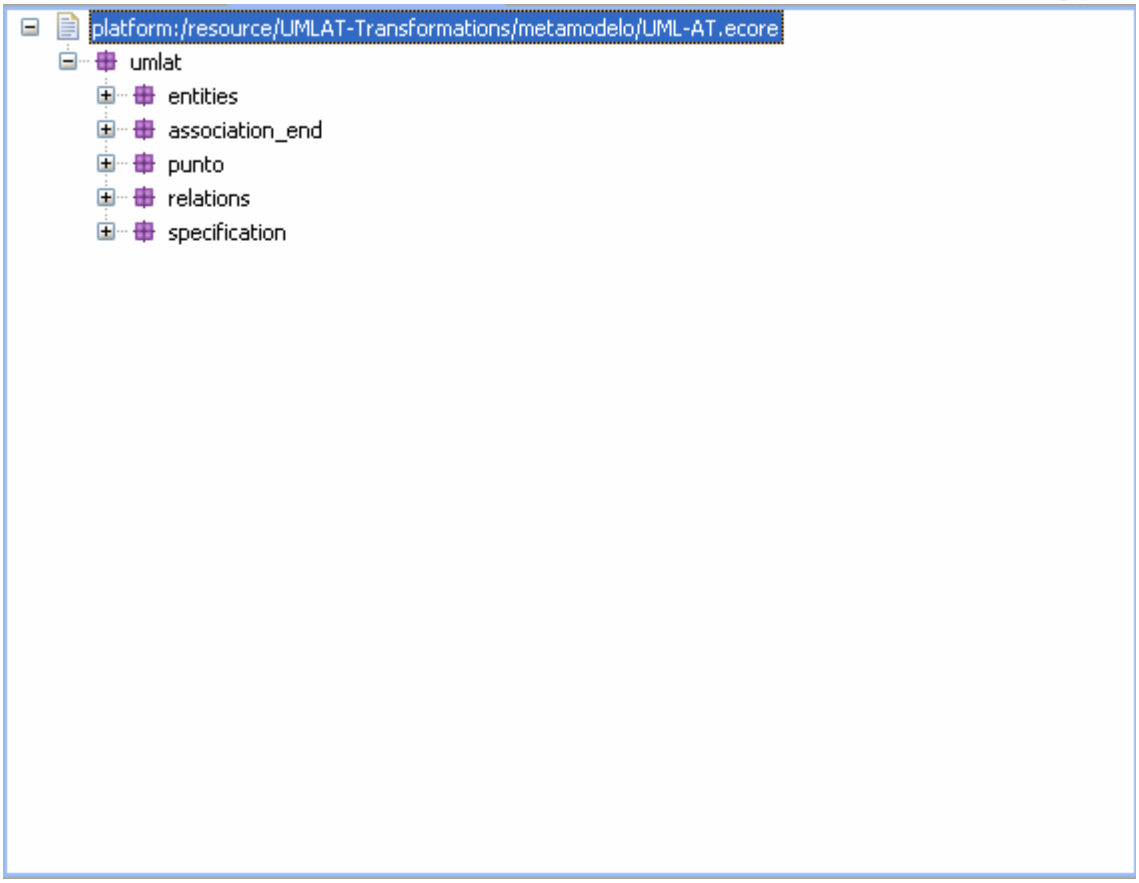

Fig. 3.9: Vista en árbol del metamodelo según *Ecore Diagram*

Esta vista ha sido con la que más hemos trabajado ya que la vista anterior era muy dificultosa a la hora de trabajar cómodamente.

En ésta vista en árbol se van mostrando los paquetes creados dentro del paquete principal. A la hora de querer añadir algún elemento a un determinado paquete, simplemente pulsando con el botón derecho en el paquete correspondiente teníamos la posibilidad de añadirle algún componente.

Dentro de cada paquete se encuentran los elementos que hemos definido:

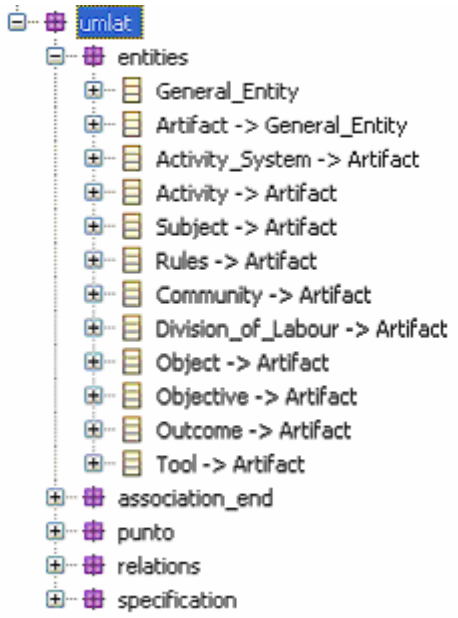

Fig. 3.10: Paquete *entities* con sus correspondientes elementos

En este caso se muestran las clases correspondientes al paquete *entities*. Para cada clase se muestra su nombre y un símbolo "→" que indica de quien hereda. Si no aparece dicho símbolo es que no hereda de nadie. A su vez podemos añadir a cada clase algún componente al igual que en el caso de los paquetes. Los componentes a añadir en los paquetes son diferentes a los que se añaden en las clases.

Si desplegamos lo que tiene cada clase podemos ver los elementos que lo conforman:

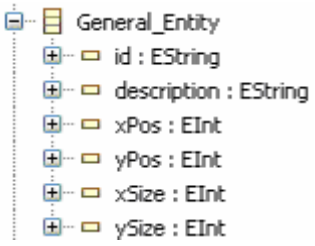

Fig. 3.11: Atributos de *General_Entity*

En este caso aparecen los atributos de *General_Entity* identificados por el nombre y el tipo que tienen.
Entre los elementos que pueden aparecer en las clases no solo están los atributos sino también las referencias a otras clases del metamodelo:

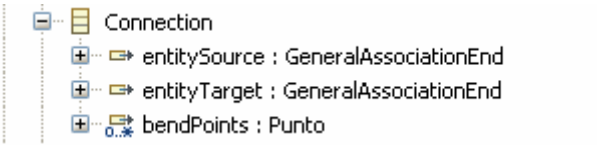

Fig. 3.12: Referencias de la clase *Connection*

Aquí tenemos un ejemplo para la clase *Connection*, que tiene tres referencias, dos a la clase *GeneralAssociationEnd* y una a la clase *Punto*.

Finalmente cabe destacar que la utilización de Eclipse para poder desarrollar el metamodelo ha sido de gran ayuda, ya que genera muchas facilidades para su diseño. Es por ello que estamos muy satisfechos con esta herramienta.

4. Diseño del primer plug-in: Editor UML-AT

4.1. Introducción

En este primer plug-in se ha desarrollado un editor que permite crear diagramas basados en la Teoría de Actividad. Los conceptos están representados por cajas con estereotipos y se pueden enlazar con las relaciones pertinentes.

El modelo tiene en cuenta la definición del lenguaje para no permitir insertar conceptos no válidos o realizar conexiones incorrectas entre estos elementos.
Para guardar los modelos creados hemos usado el soporte que proporciona el EMF que nos permite guardar los modelos en un fichero XML.

4.2. Configuración del plug-in:

Para la creación del plug-in hemos utilizado el *Plug-in Project* dentro de la carpeta *Plug-in Development*:

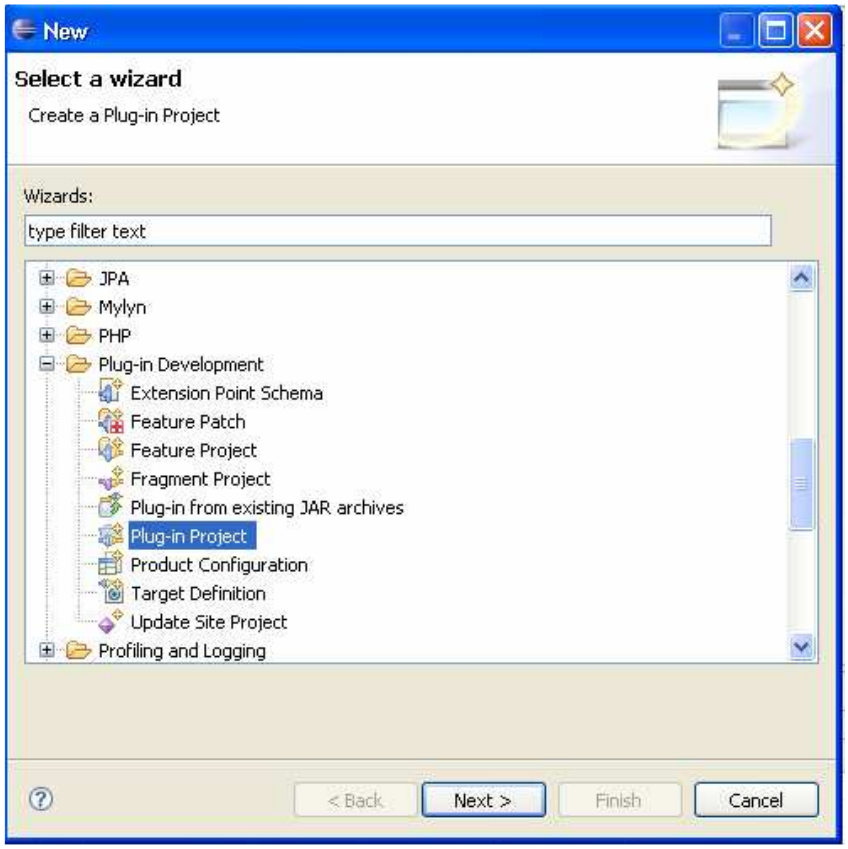

Fig. 4.1: Selección tipo de Plug-in

En las siguientes ventanas de configuración se han dejado las opciones que venían por defecto.

Dentro de la ventana principal de configuración del plug-in hemos configurado dos de las pestañas que se muestran que son *Dependencies* y *Extensions*.

En la pestaña *Dependencies* se importan los siguientes plug-in que se requieren para el funcionamiento del editor:

- *org.eclipse.emf.ecore*, para la utilización en la parte del modelo de la herramienta.
- *org.eclipse.gef*, para la utilización en la parte de la interfaz de la herramienta.
- *org.eclipse.core.resources*, para la utilización de los recursos.
- *org.eclipse.ui.views*, *org.eclipse.ui.ide*, *org.eclipse.ui.editors* y *org.eclipse.jface.text*, necesarios para trabajar con la parte visual.
- *org.eclipse.core.runtime* y *org.eclipse.ui* vienen importados directamente al crear el plug-in.

En cuanto a los paquetes que importamos se encuentran los siguientes:

- *org.eclipse.emf.ecore.xmi.impl* y *org.eclipse.emf.ecore.xmi.util* que utilizamos para salvar los modelos en formato XML.

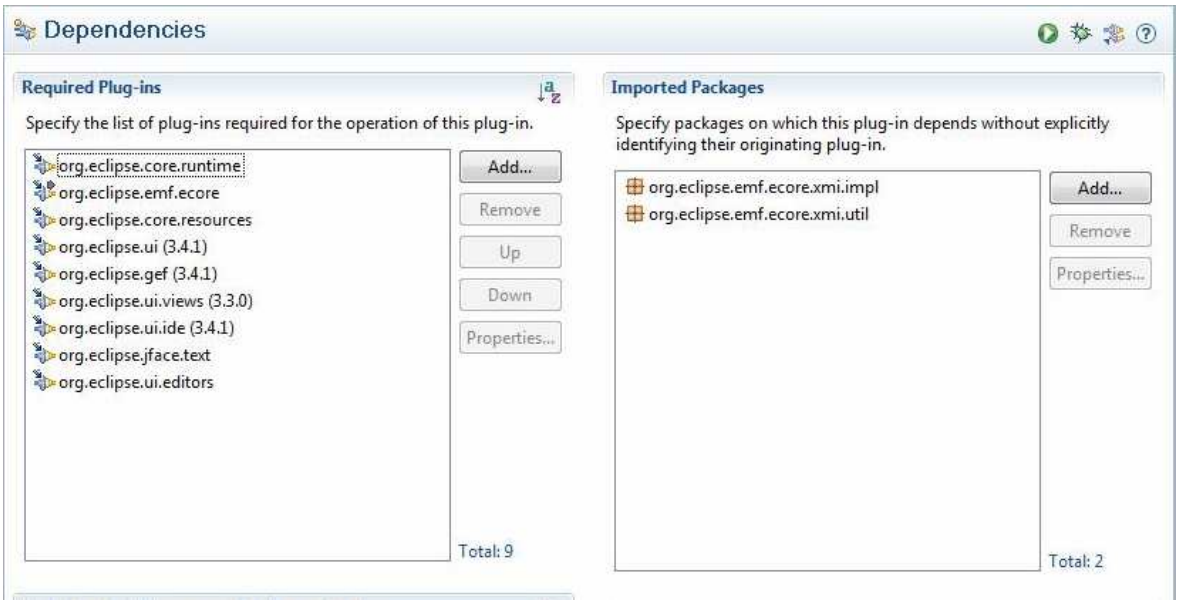

Fig. 4.2: Configuración *Dependencies*

La otra pestaña que modificamos es la de *Extensions*, de tal manera que modificamos lo siguiente:

- Introducimos el editor dentro de la extensión *org.eclipse.ui.editors* y de *org.eclipse.ui.newWizards*.
- El resto de extensiones que se muestran las introduce automáticamente el plug-in del EMF.

Al seleccionar el editor UML-AT podemos configurar las distintas opciones del mismo. En nuestro caso hemos introducido un identificador y un nombre para el editor así como la posibilidad de cargar un icono para que se visualice en el editor.

Lo más importante de está configuración son los siguientes apartados:

- *Extensions*: *umlat*, esta extensión es la que se utilizará a la hora de crear un Editor UML-AT.
- *Class: umlat.presentations.UMLATEditor*, es la clase principal a la que llama el editor.

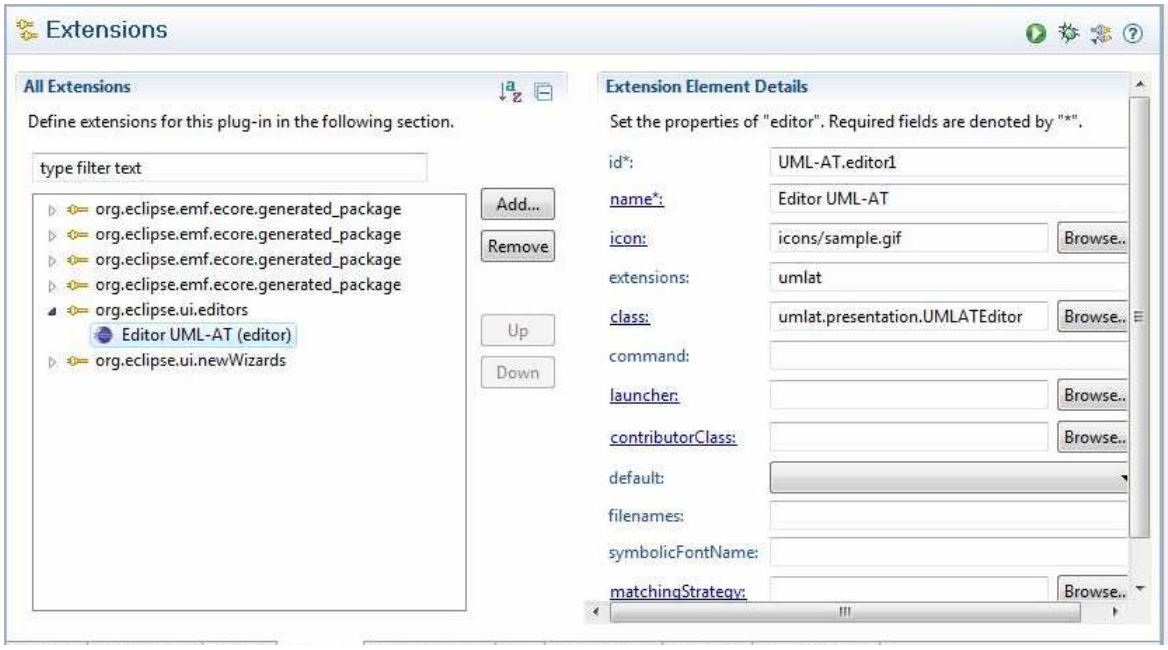

Fig. 4.3: Configuración *Extensions*

4.3. Casos de uso

Un usuario que utilice el editor de diagramas UML-AT puede realizar las siguientes funciones:

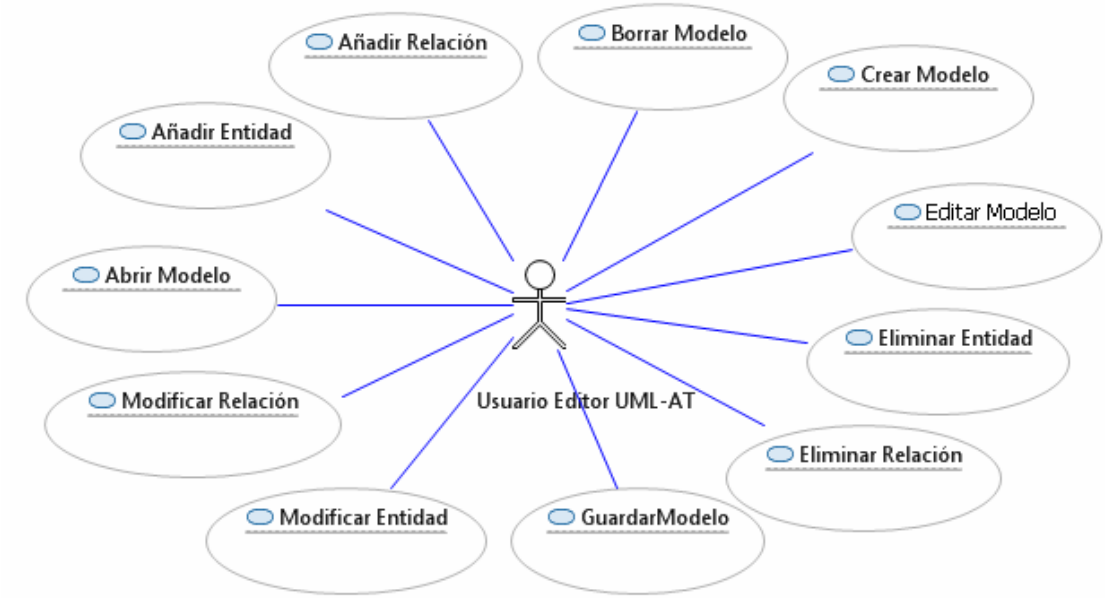

Fig. 4.4: Casos de uso

1. *Abrir Modelo*: el usuario para abrir un modelo de un diagrama debe tener previamente cargado un proyecto. Si el modelo no pertenece a ese proyecto deberá abrirse seleccionando el fichero a través de la ruta donde esté guardado.

   En cambio si el modelo que se quiere abrir pertenece al proyecto, ya estará accesible en el panel de la izquierda:

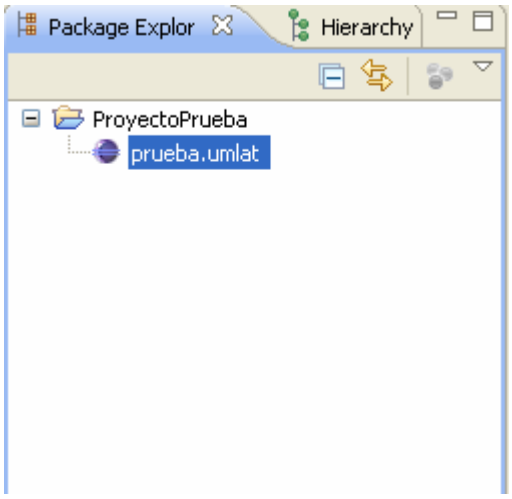

Fig. 4.5: Modelo UML-AT dentro del proyecto actual

En este caso simplemente se deberá seleccionar el modelo que se quiera abrir y aparecerá automáticamente en la ventana de edición.

También es posible abrir el modelo en el formato XML en que se guarda, para ello el usuario debe hace clic con el botón derecho del ratón en el fichero y seleccionar abrir como texto de tal forma que se muestra en dicho formato:

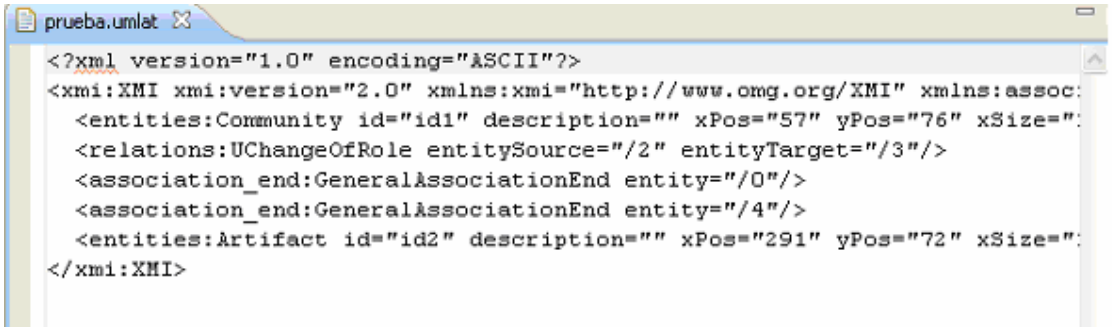

Fig. 4.6: Modelo en formato XML

2. *Añadir Entidad*: el usuario puede añadir cualquier entidad de las que se muestran en la paleta de la ventana de edición. Para ello simplemente deberá seleccionar primero la entidad deseada en la paleta y acto seguido hacer clic en la localización de la ventana de edición donde desee añadirlo.

   Entonces aparecerá una ventana donde se le pedirá al usuario que introduzca un identificador y una descripción para esa entidad:

Fig. 4.7: Ventana de propiedades de una entidad

Una vez aceptada la creación de la entidad se mostrará en la ventana de edición. A continuación sigue un ejemplo con la entidad *Activity System*:

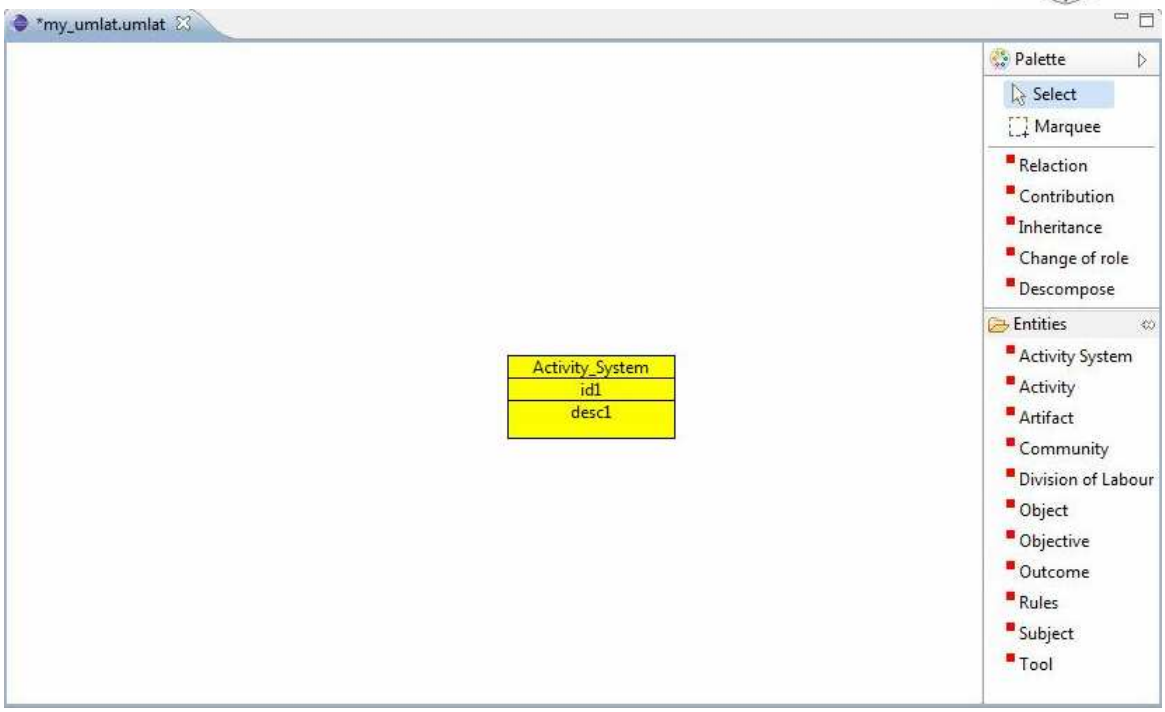

Fig. 4.8: Editor con la entidad *Activity System*

Si se intenta añadir una entidad con un identificador ya existente en el diagrama se muestra un mensaje avisando de que ya existe ese identificador e impidiendo la creación de la nueva entidad:

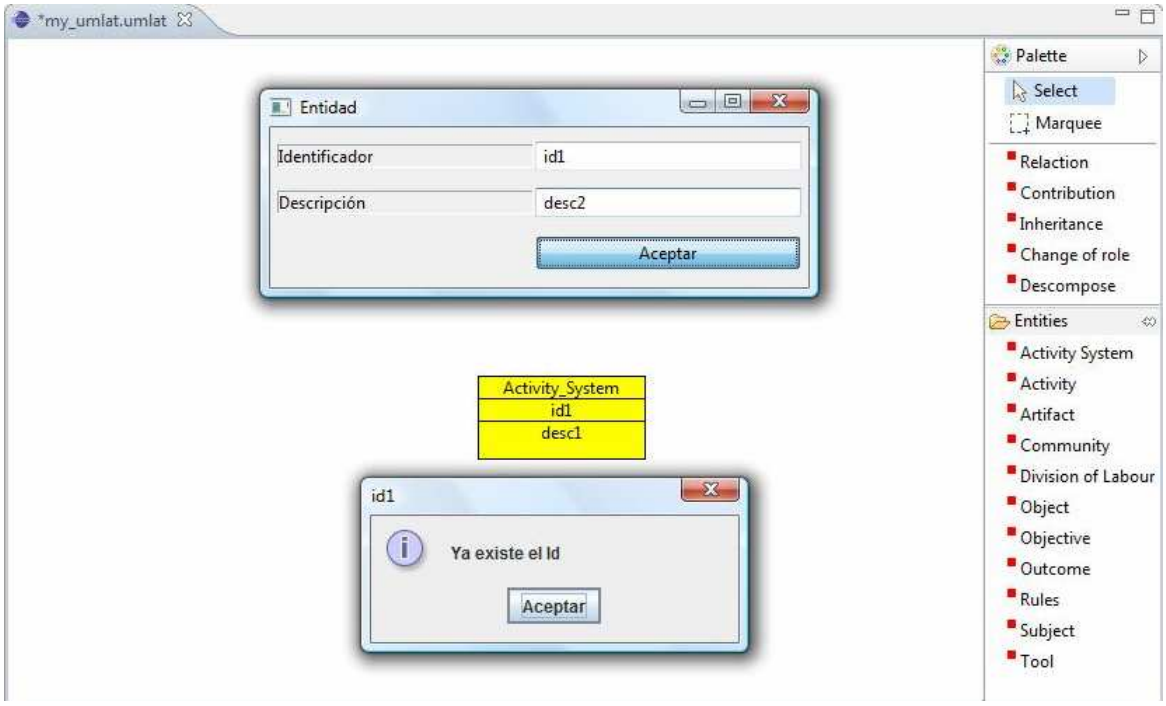

Fig. 4.9: Creación de una entidad con un identificador ya existente

3. *Añadir Relación*: el usuario para poder añadir una relación debe seleccionar en primer lugar el tipo de relación, y acto seguido hacer clic en las dos entidades que quiere que estén unidas por esa relación. Destacar que dependiendo del tipo de relación y de los tipos de entidades seleccionadas pueden ocurrir varias cosas. En la paleta se pueden seleccionar cinco tipos de relaciones:

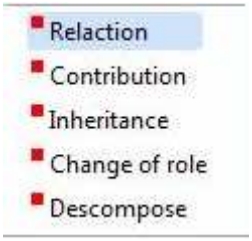

Fig. 4.10: Tipos de relaciones

a. *Change of Role*: el usuario puede seleccionar dos entidades cualesquiera para unir con esta relación.

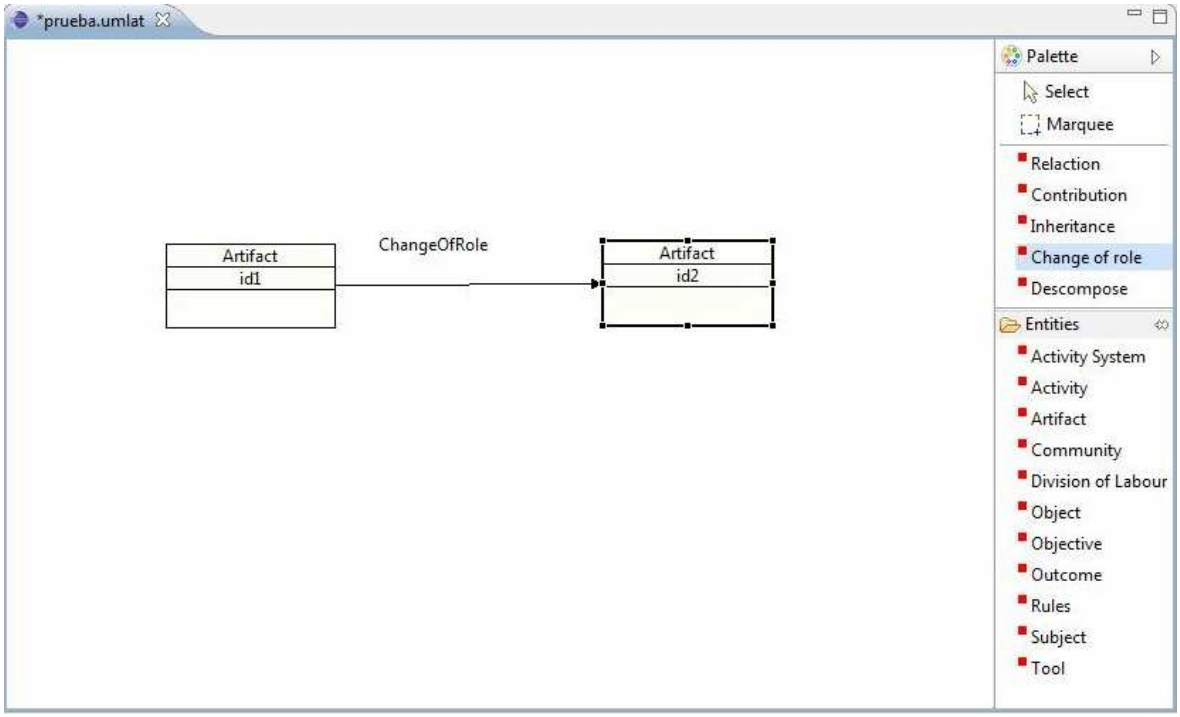

Fig. 4.11: Creación de la relación *Change Of Role*

b. *Contribution*: el usuario puede seleccionar dos entidades cualesquiera para unir con esta relación. Una vez seleccionadas aparecerá una ventana donde tendrá que especificar el tipo de contribución y la multiplicidad de esta:

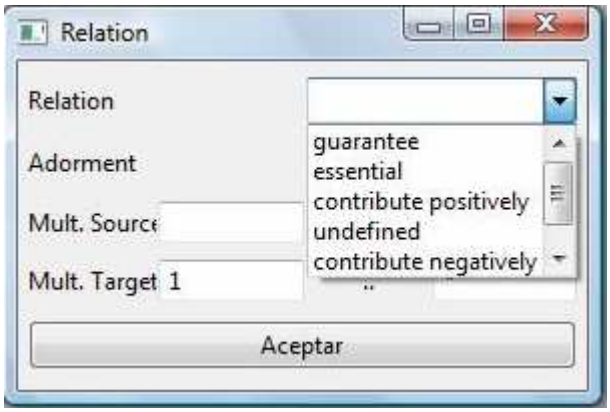

Fig. 4.12: Especificación del tipo de contribución

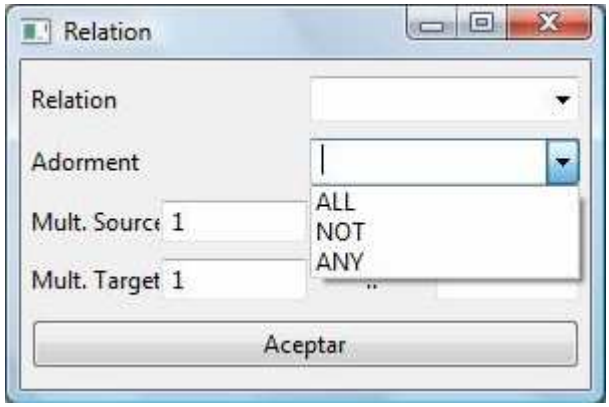

Fig. 4.13: Especificación del adorno de la relación

Fig. 4.14: Especificación de la multiplicidad

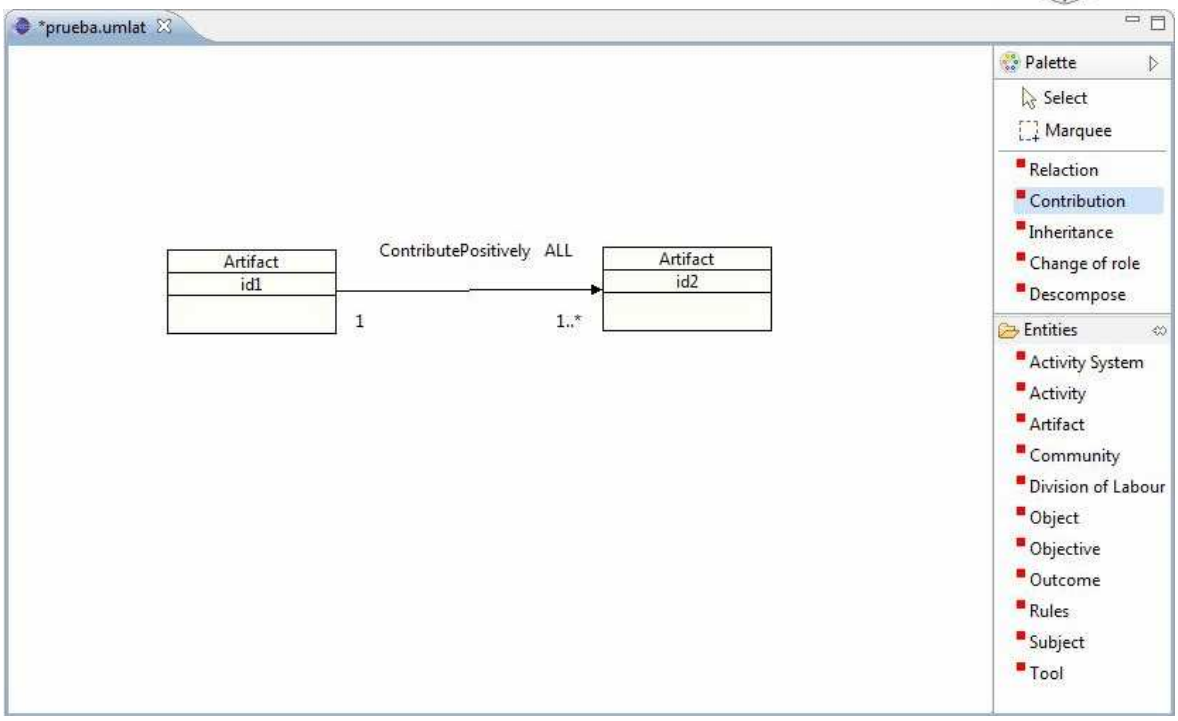

Fig. 4.15: Ejemplo final de relación de contribución *Contribute Positively*

c. *Descompose*: el usuario puede seleccionar dos entidades cualesquiera para unir con esta relación.

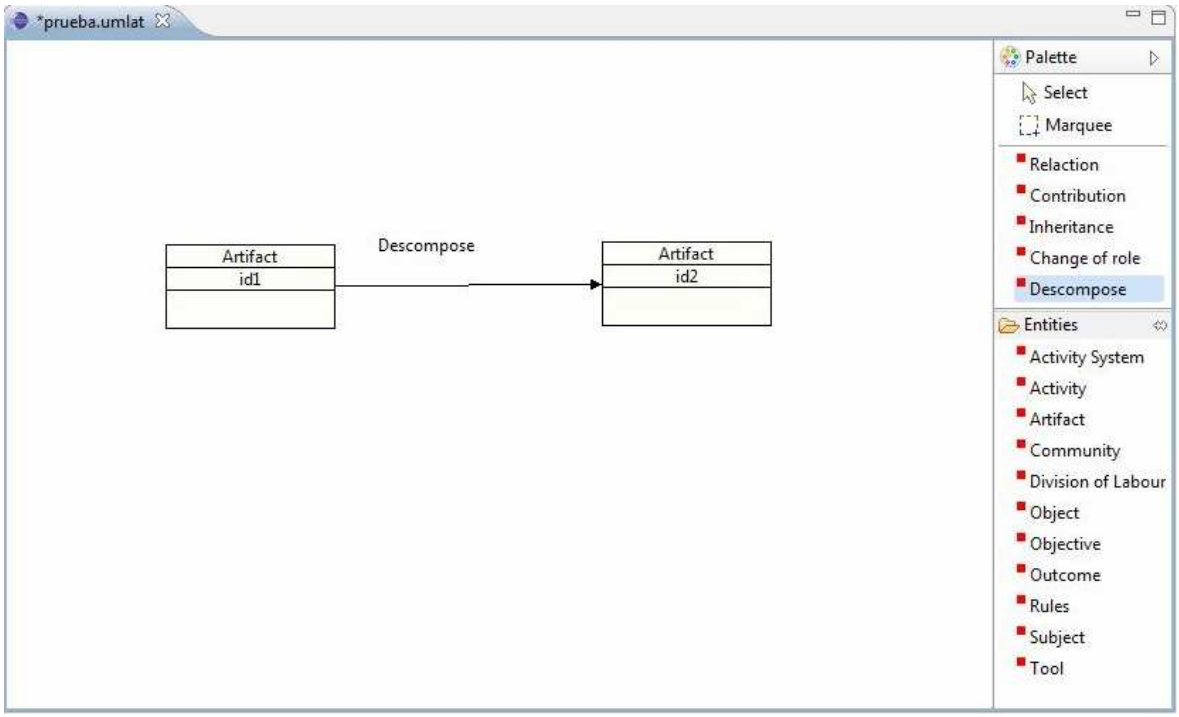

Fig. 4.16: Relación *Descompose*

d. *Inheritance*: el usuario puede seleccionar dos entidades cualesquiera para unir con esta relación. La primera entidad en seleccionar será la que herede de la segundad entidad seleccionada.

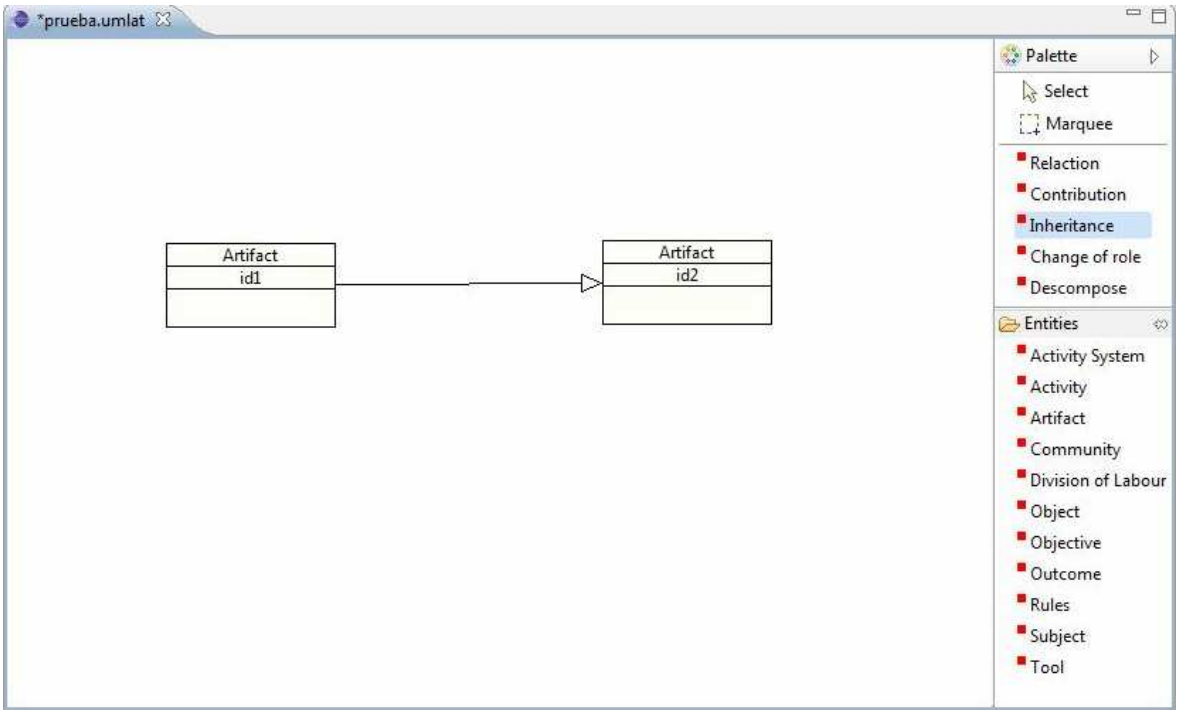

Fig. 4.17: Relación *Inheritance*

e. *Relation*: esta opción se utiliza para el resto de relaciones que no se cubren en los apartados anteriores; el usuario puede seleccionar dos entidades cualesquiera y aparecerá una ventana en la que dependiendo del tipo de entidades seleccionadas aparecerán las posibles relaciones entre ellas según el lenguaje UML-AT (se puede dar el caso de que no aparezca ninguna relación). También se deberá especificar la multiplicidad.

Fig. 4.18: Posibles relaciones *Artifact-Activity*

4. *Borrar Modelo*: el usuario puede borrar cualquier modelo simplemente seleccionado el fichero donde éste se encuentre y seleccionando la opción de eliminar tras hacer clic con el botón derecho:

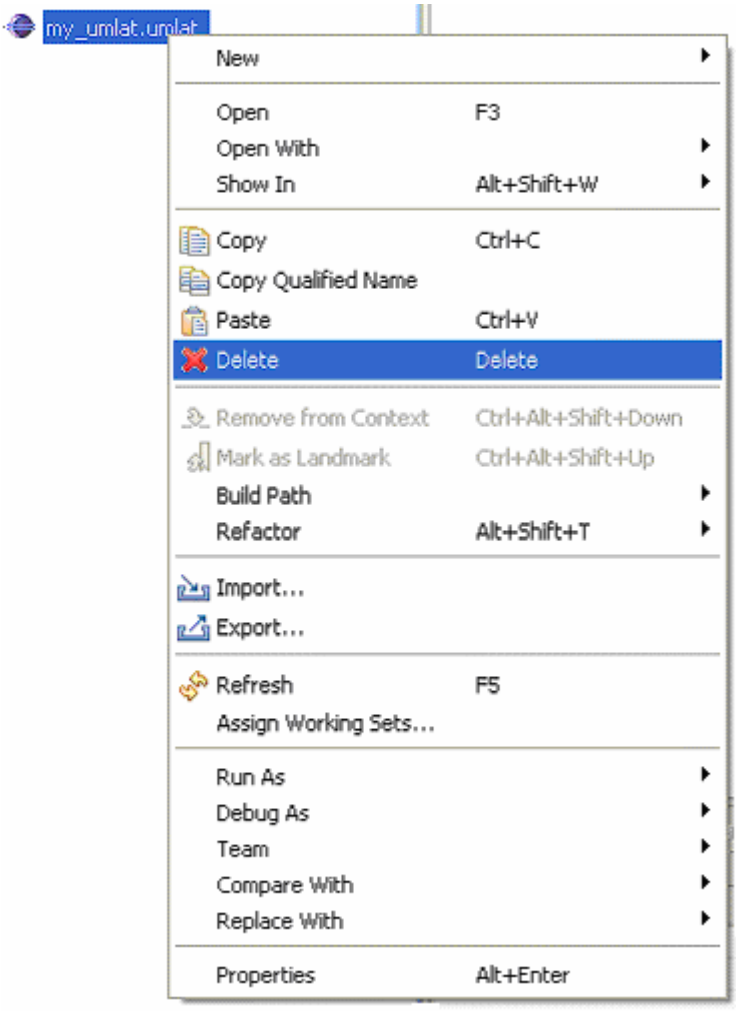

Fig. 4.19: Eliminar modelo

Aparecerá una ventana de confirmación para eliminar totalmente el modelo:

Fig. 4.20: Ventana de confirmación al eliminar un modelo

5. *Crear Modelo*: para la realización de esta función el usuario deberá en primer lugar crear un proyecto. Una vez creado podemos ejecutar el editor de dos maneras diferentes:

   a. Creando un fichero con la extensión "umlat":

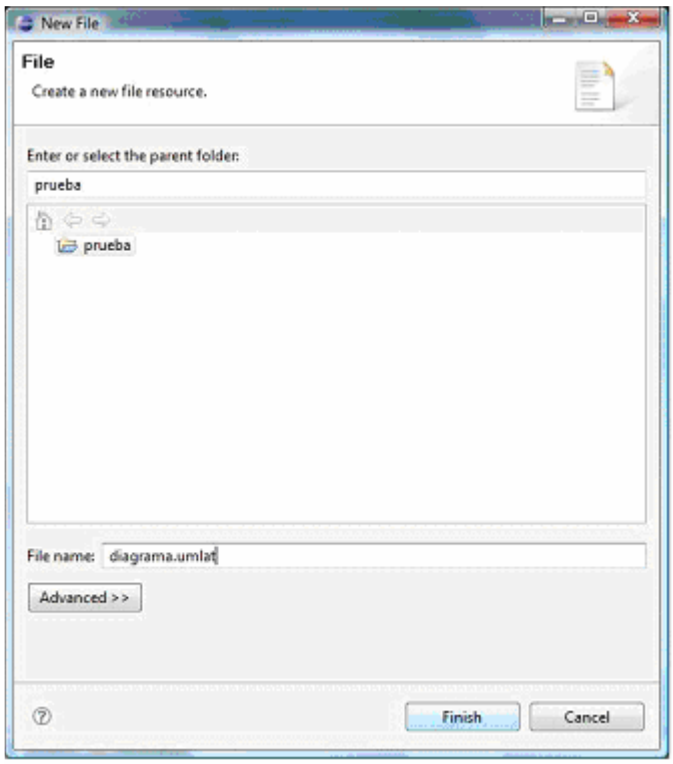

Fig. 4.21: Creación del Editor UML-AT mediante la extensión del fichero

b. Seleccionando el editor dentro de la carpeta UML-AT: para ello navegamos a través de la ruta *File->New->Other*, apareciendo lo siguiente:

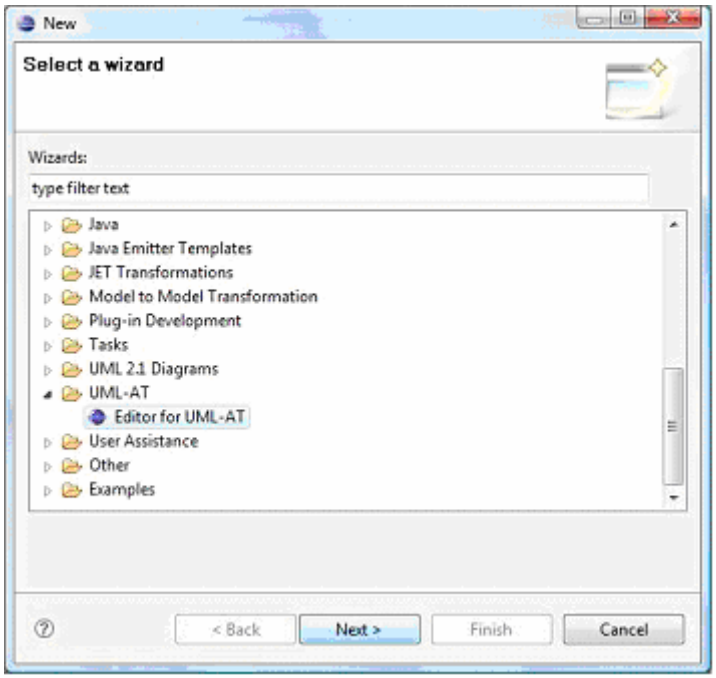

Fig. 4.22: Selección del Editor dentro de la carpeta UML-AT

A continuación se selecciona el proyecto en el que queremos incluir el modelo a crear. Por defecto como nombre del fichero aparece *my_umlat*, pero se puede modificar a gusto del usuario:

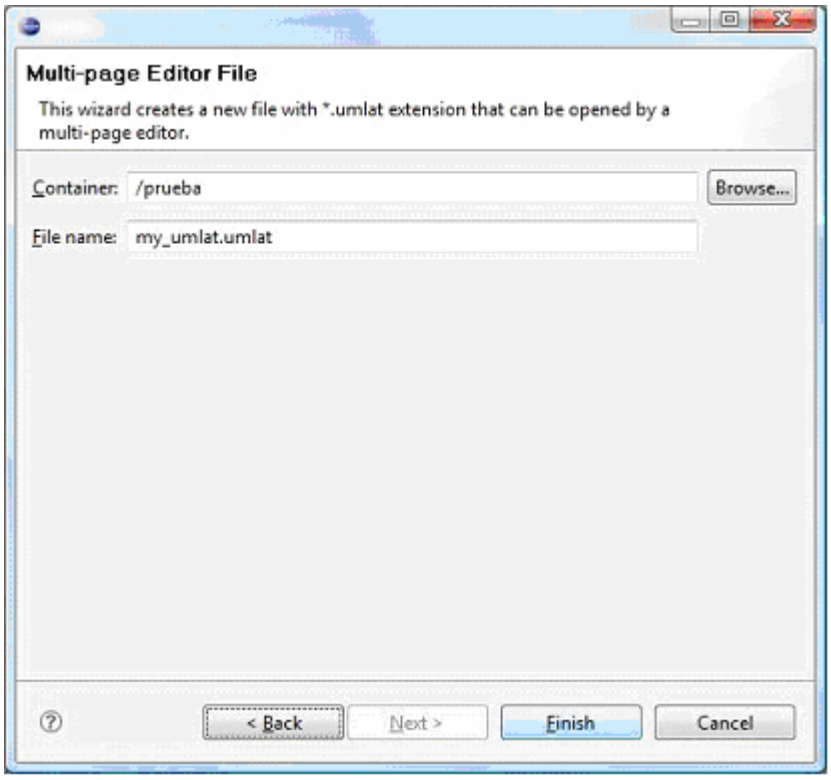

Fig. 4.23: Selección del proyecto y nombre del fichero

Una vez realizados todos estos pasos aparecerá una ventana mostrando el editor con los distintos componentes que tiene, como son la paleta y la ventana de edición:

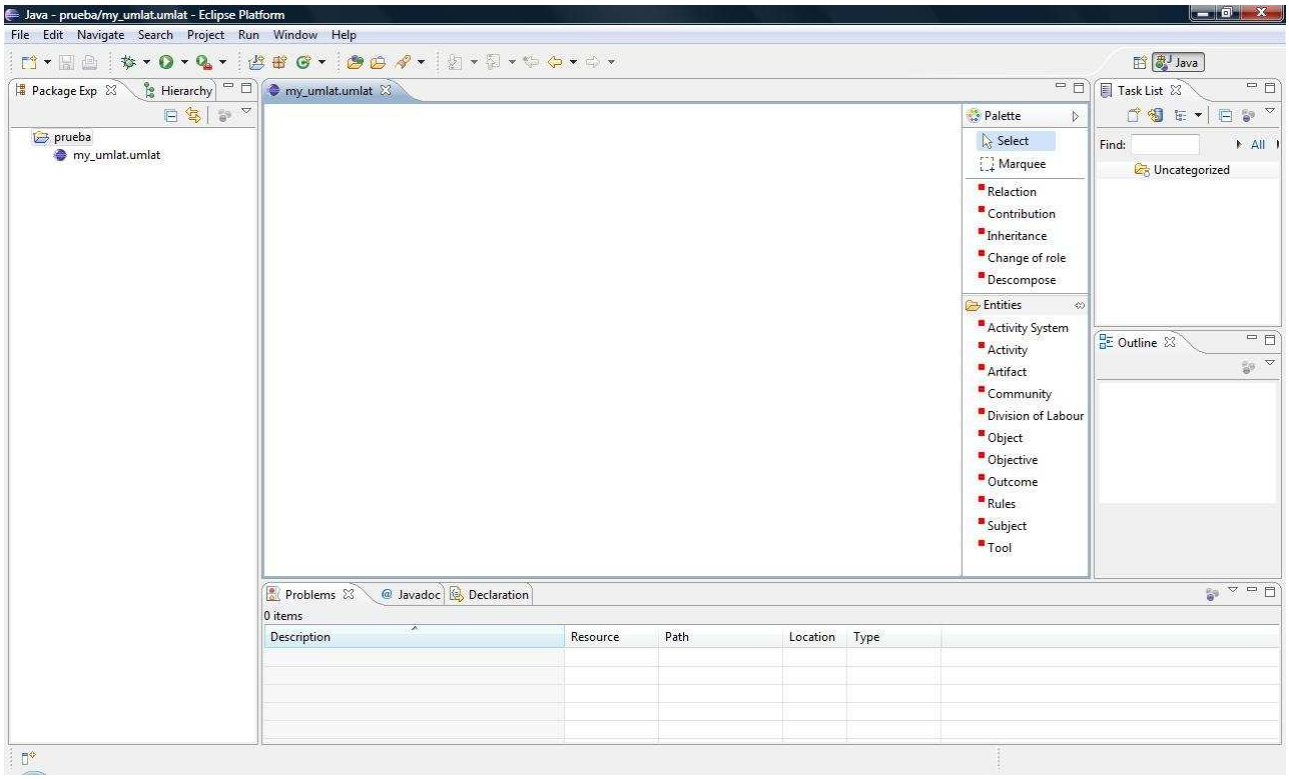

Fig. 4.24: Aspecto inicial del editor UML-AT

6. *Editar Modelo*: el usuario puede editar gráficamente la posición tanto de las entidades como de las relaciones de la siguiente manera:

   a. Para editar la posición de las entidades se debe tener seleccionado en la paleta el estado del ratón como “Select”:

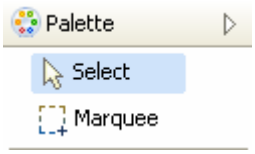

Fig. 4.25: Selección del estado del ratón

Una vez en este estado simplemente se ha de seleccionar la entidad que se quiere cambiar de posición y arrastrarla por la ventana de edición hasta la nueva posición que se desee. Las flechas de las relaciones se irán adaptando automáticamente a los cambios de posición.

   b. Para editar la posición de las relaciones, se debe proceder de forma análoga a lo anterior, seleccionando y arrastrando la relación correspondiente hasta la nueva posición deseada. En este caso van apareciendo puntos de apoyo en mitad de cada una de las rectas que componen la relación. Inicialmente solo hay un punto de apoyo.

   Aquí tenemos un caso de una relación inicial:

Fig. 4.26: Estado inicial de la representación de una relación

Y en la siguiente ilustración podemos ver los cambios hechos en las posiciones de la relación:

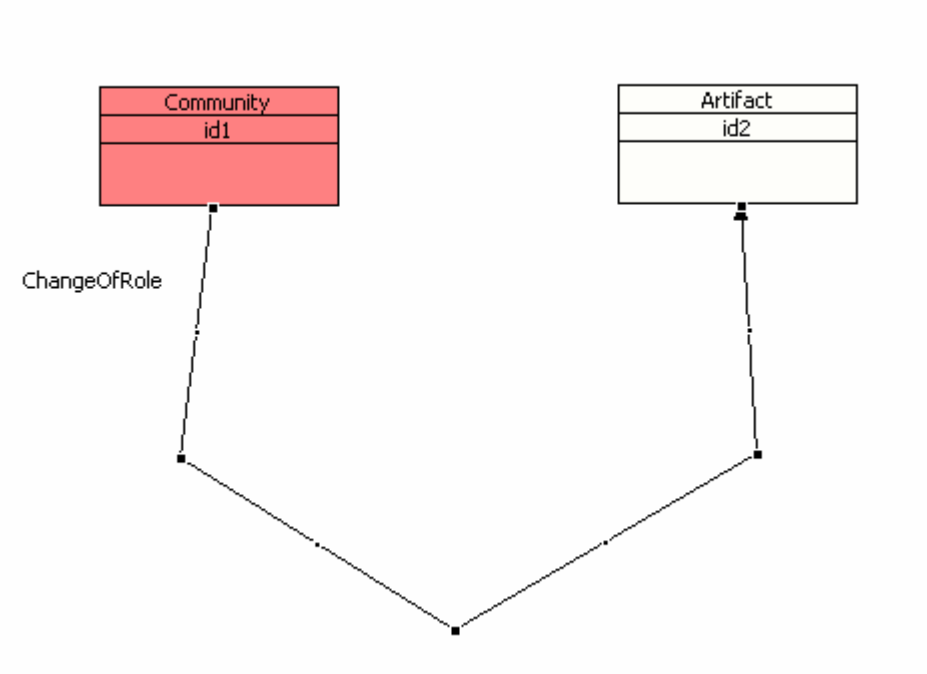

Fig. 4.27: Estado modificado de la representación de una relación

Se han generado tres rectas más, provocando también la aparición de tres puntos de apoyo por si se quieren hacer más modificaciones.

7. *Eliminar Entidad*: el usuario puede eliminar cualquier entidad que se encuentre en el modelo, para ello debe seleccionarla gráficamente en la ventana de edición y con el botón derecho pulsar la opción eliminar (*Delete*).

   Si una entidad tiene relaciones asociadas, todas estas relaciones serán eliminadas también del modelo.

   Así por ejemplo dado el siguiente modelo:

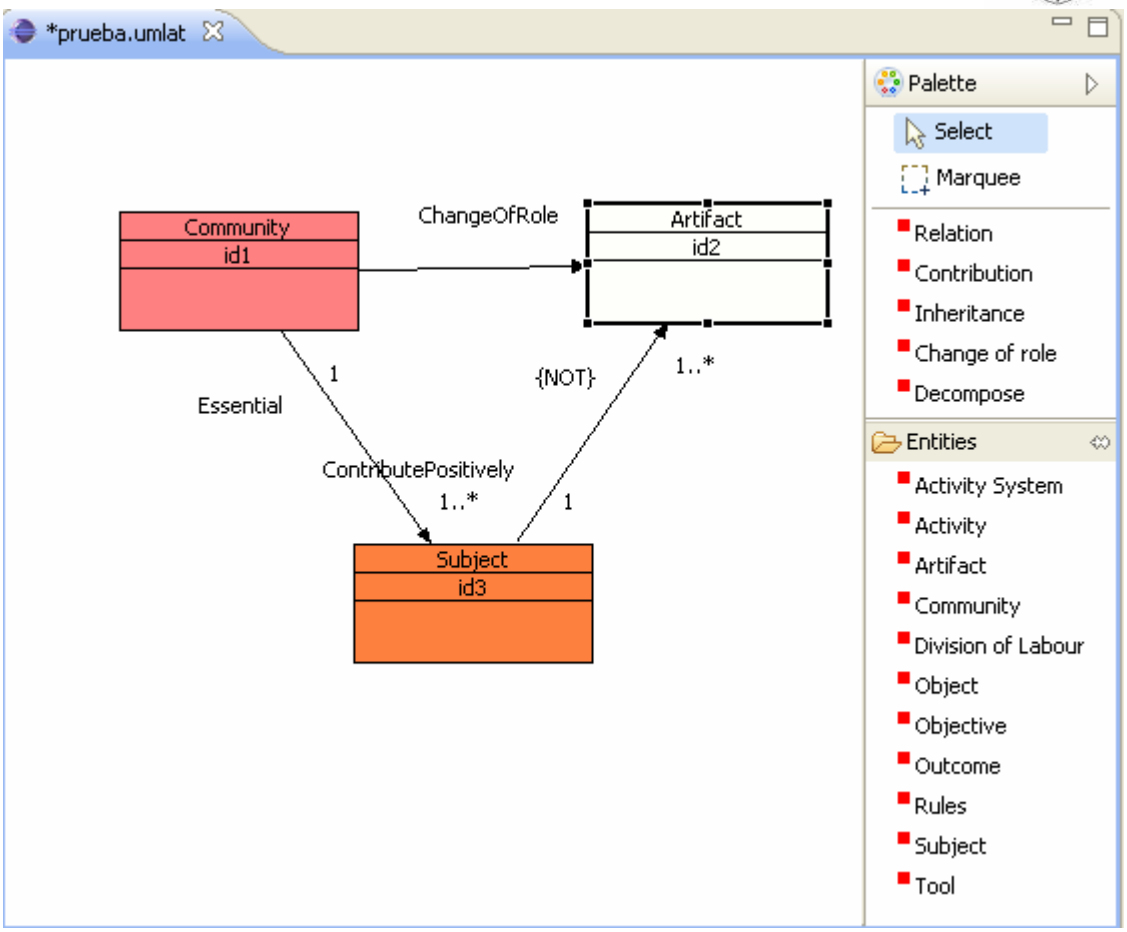

Fig. 4.28: Ejemplo de eliminar entidad

Si se elimina la entidad *Subject* con identificador *id3*, el nuevo modelo quedaría de la siguiente manera:

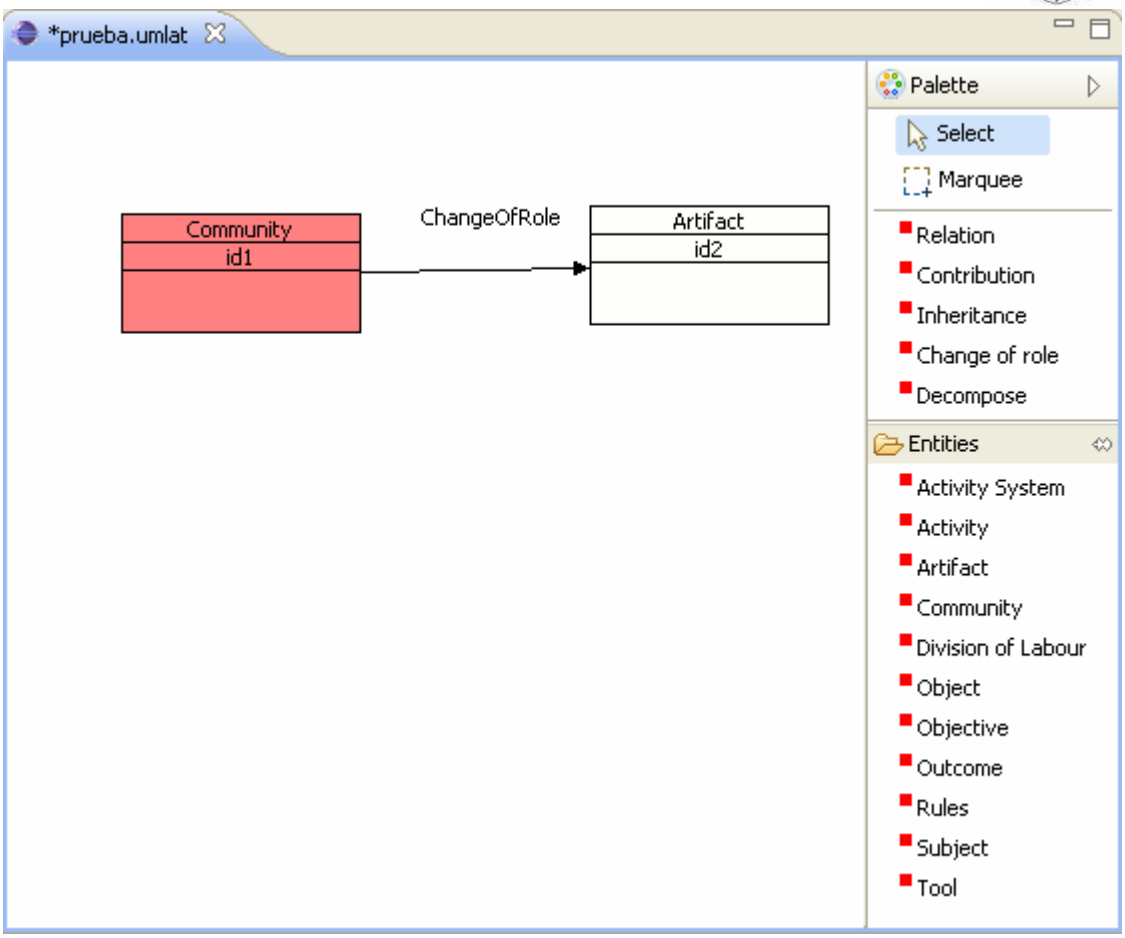

Fig. 4.29: Ejemplo anterior con la entidad *Subject* eliminada

8. *Eliminar Relación*: el usuario puede eliminar cualquier relación del modelo simplemente seleccionándola gráficamente y pulsando la opción eliminar (Delete) tras hacer clic con el botón derecho sobre ella. Es un caso de uso similar al anterior pero con relaciones.

9. *Guardar Modelo*: el usuario puede guardar el modelo mediante las opciones que vienen dadas por el herramienta Eclipse:
    a. *Guardar como*, que viene dada en la ruta *File->Save as*, donde deberá especificar el proyecto donde quiere ubicarlo y el nombre del fichero.
    b. *Guardar*, que viene dada en la ruta *File->Save* y que guarda el modelo automáticamente en el proyecto actual y con el nombre que ya tiene.

10. *Modificar Entidad*: para modificar una entidad se debe seleccionar primero la entidad a modificar y en la parte inferior de la ventana deben aparecer las propiedades de dicha entidad. Si no aparecieran sería necesario activar esa vista abriendo la ruta *Window->Show View->Other*:

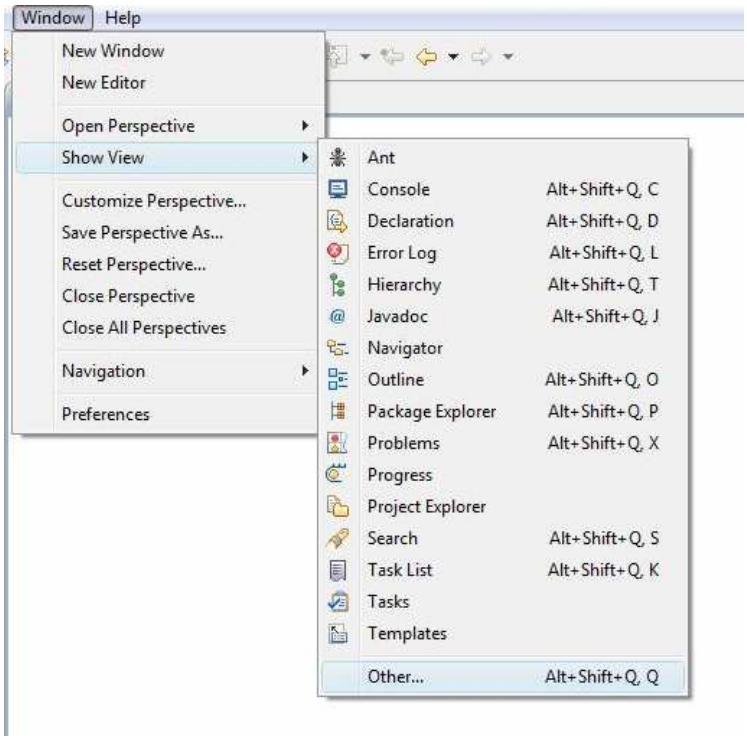

Fig. 4.30: Ruta de elección de la vista

Una vez aparezca la ventana de elección de la vista, se debe seleccionar la opción *Properties*:

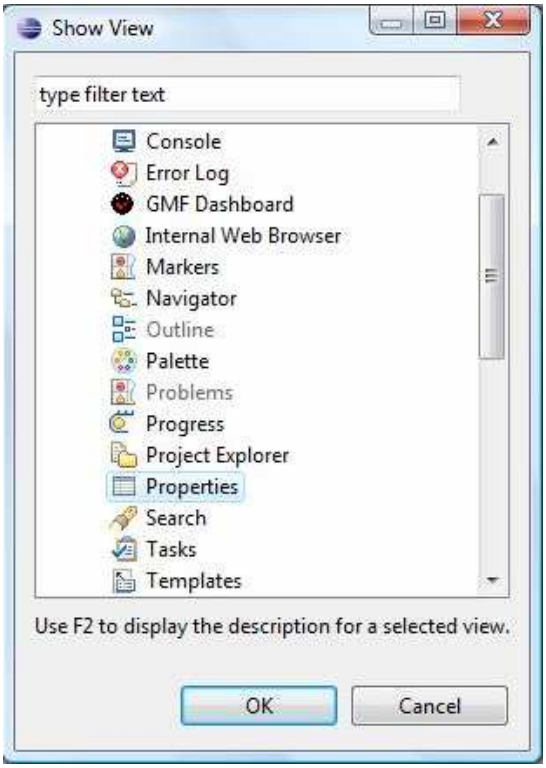

Fig. 4.31: Ventana de elección de la vista

Una vez realizado este paso aparecerá la siguiente vista en la ventana, donde el usuario puede modificar las propiedades a su gusto (destacar que no se puede modificar una entidad poniéndole el identificador de una entidad ya existente):

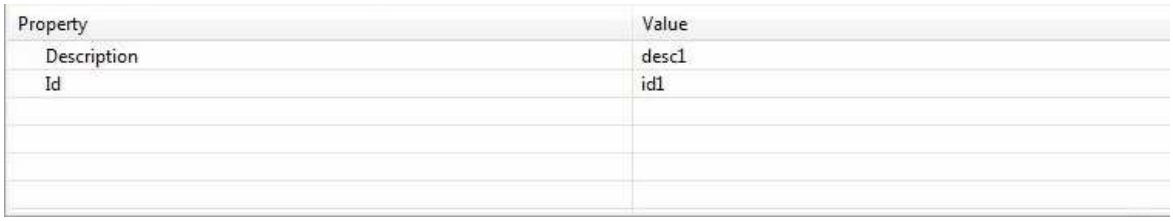

Fig. 4.32: Ventana de propiedades de una entidad

11. *Modificar Relación*: el usuario puede modificar dos tipos de las relaciones especificadas anteriormente, que son las de *Contribution* y *Relation*.

    Como en el caso de las entidades, si no aparece la ventana de propiedades se debe habilitar siguiendo los pasos anteriores, apareciendo el siguiente recuadro, en el cual se puede modificar el tipo de adorno y la multiplicidad:

Fig. 4.33: Ventana de propiedades de una relación

12. *Editar Modelo*: el usuario puede editar gráficamente el modelo, para ello puede realizar las siguientes acciones (para ello deberá tener seleccionada la opción *Select* de la paleta):
    a. *Desplazar entidades*, si existen relaciones entre las entidades que se desplacen éstas se amoldan automáticamente a los cambios producidos.
    b. *Establecer puntos intermedios en las relaciones*, se pueden establecer puntos intermedios en las relaciones para facilitar la visualización final del modelo.

4.4. Descripción del modelo

4.4.1. Diagrama de clases

El diagrama de clases correspondiente al modelo de la herramienta es el siguiente:

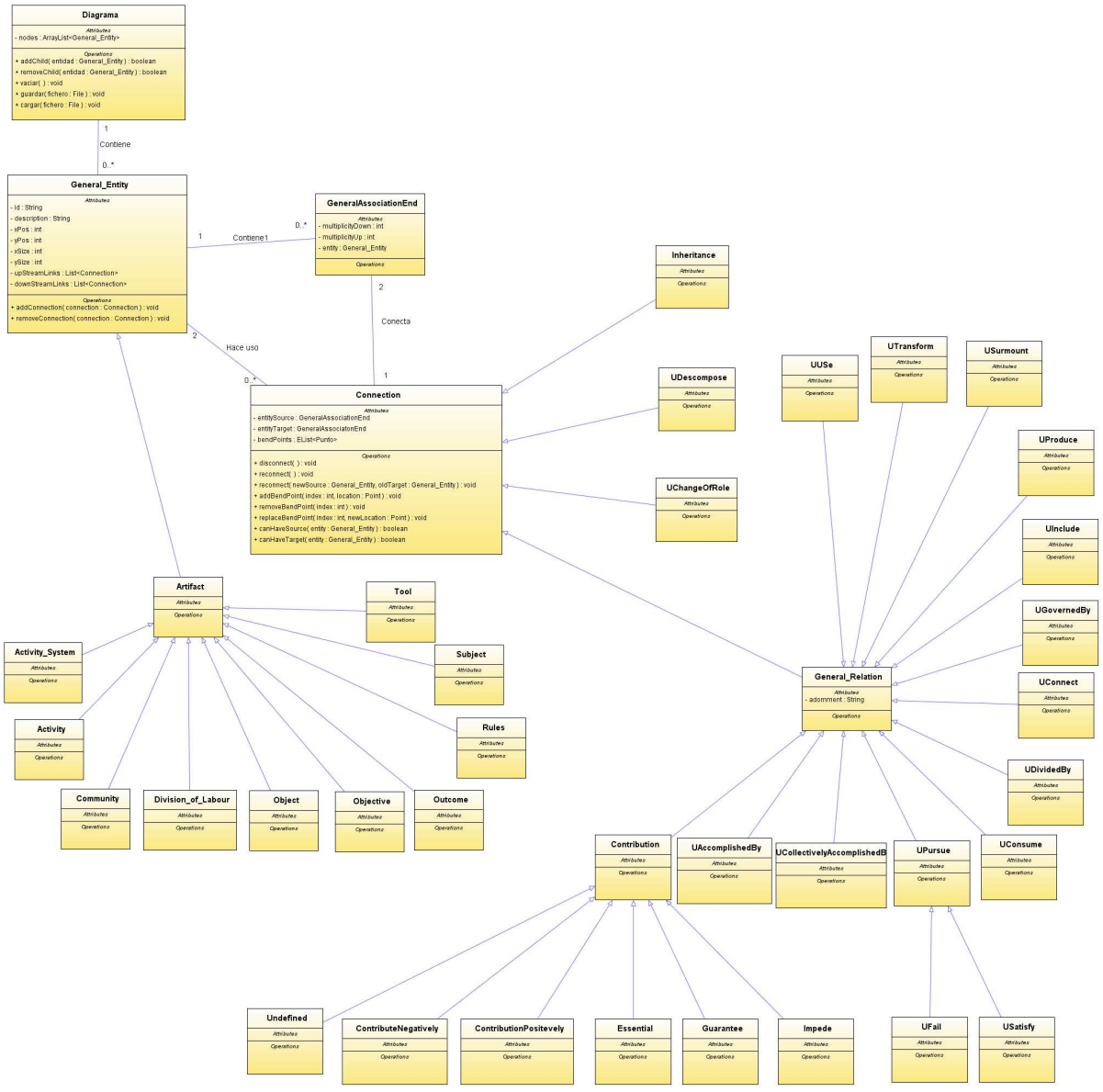

Fig. 4.34: Diagrama de clases

Se ha estructurado el diagrama en varios módulos para poder facilitar su visualización y su explicación:

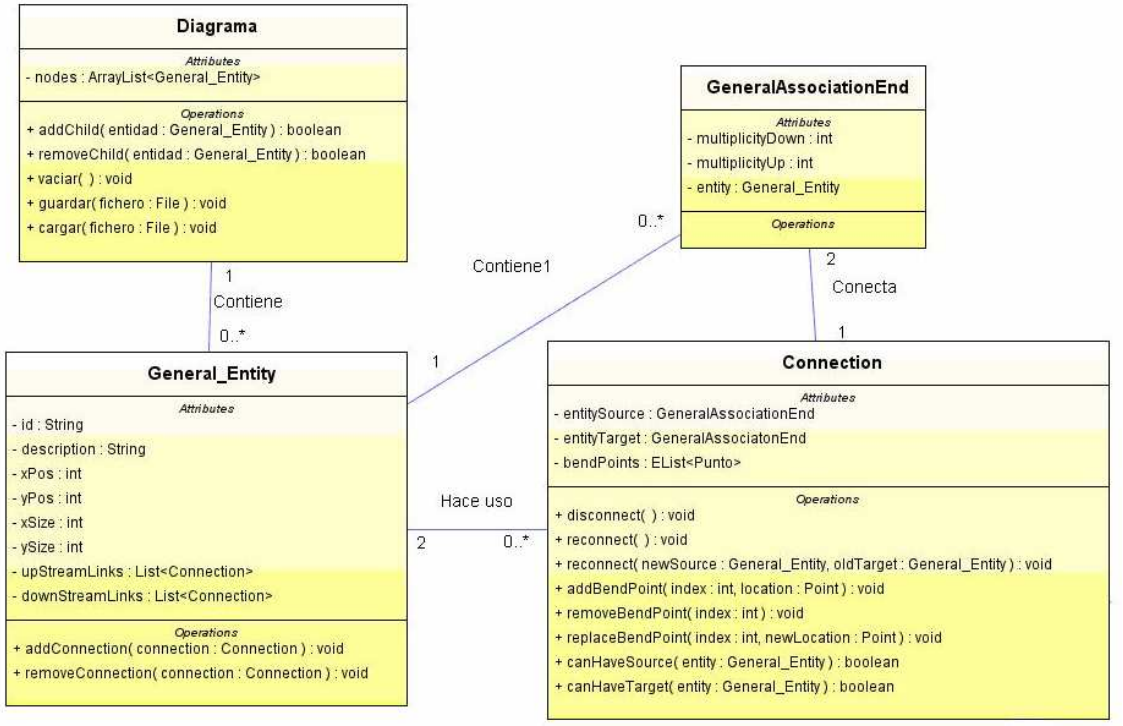

Fig. 4.35: Diagrama de clases (vista principal)

Esta primera vista es la principal y la que contiene las clases más relevantes en el funcionamiento del modelo.

La estructura es sencilla, la clase principal es *Diagrama* que es la que contiene todo el modelo; este modelo está representado por entidades a través de la clase *General_Entity*, que como se verá más adelante es la clase de la cual heredan todas las entidades.

Diagrama contiene un array de estas entidades, las cuales va añadiendo según se van creando. De ahí que entre los métodos principales que contiene estén *añadirChild* y *removeChild*, el primero se encarga de añadir una entidad en el array cuando ha sido creada por el usuario y el segundo la elimina. Aparte de un método destinado a vaciar por completo el array.
Los otros dos métodos importantes son los de *guardar* y *cargar*, que guarda un modelo en un fichero o genera un modelo respectivamente. El funcionamiento de cómo se guarda el modelo en formato XML viene explicado más adelante en un apartado.

La clase *General_Entity* es la única que se relaciona con el *Diagrama*, por tanto hace de enlace con otros elementos importantes como pueden ser las relaciones o los extremos de las relaciones. Esta clase, aparte de almacenar los parámetros que le asigna el usuario tales como el identificador y la descripción, también contiene un par de listas que almacenan las relaciones en las que participa, donde cada lista se corresponde a si la

relación entra o sale de la entidad. Además tiene unos atributos donde almacena su posición y tamaño en la ventana de edición. Estos parámetros se guardan en el fichero XML de forma que cuando se quiera abrir el modelo de nuevo simplemente se cogen esos datos para saber la ubicación y tamaño que han que tener.
Como métodos importantes se destacan los de añadir y eliminar relaciones. A estas dos funciones hace falta pasarles el tipo de relación que queremos añadir o eliminar.

Esta clase puede tener asociado varios extremos de relación ya que puede tener varias relaciones, esto se muestra en el diagrama a través de la relación que une *General_Entity* con *GeneralAssociationEnd*. También puede tener todas las relaciones que quiera, relación que une *General_Entity* con *Connection*, pero una relación únicamente debe tener asociados dos *General_Entity*.

La clase *GeneralAssociationEnd*, que representa el extremo de una relación, contiene la multiplicidad y la entidad a la que está asociado como elementos más relevantes. Como hemos dicho antes una entidad puede tener todos los extremos de relación como quiera el usuario, aunque este número estará relacionado con el número de relaciones que tenga la entidad, ya que tantas relaciones tenga tendrá tantos extremos de relación. De ahí también se ve que cada relación tiene que tener dos extremos de relación que irán conectadas con cada una de las entidades que una la relación.

Mientras que la clase *Connection* que representa la relación, contiene como atributos la entidad origen y destino además de los puntos de apoyo usados para la representación gráfica de la misma.
Entre los métodos importantes podemos destacar *disconnect* y *reconnect* para desconectar y conectar respectivamente entidades a una relación. Destacar también unos cuantos métodos para trabajar con los puntos de apoyo.
Los dos últimos métodos a mencionar son *canHaveSource* y *canHaveTarget* a los que se les pasa por parámetro una entidad y deciden si el tipo de entidad puede ir asociado a la relación dependiendo de las restricciones del lenguaje UML-AT.

Las entidades se relacionan con el modelo a partir de la siguiente relación:

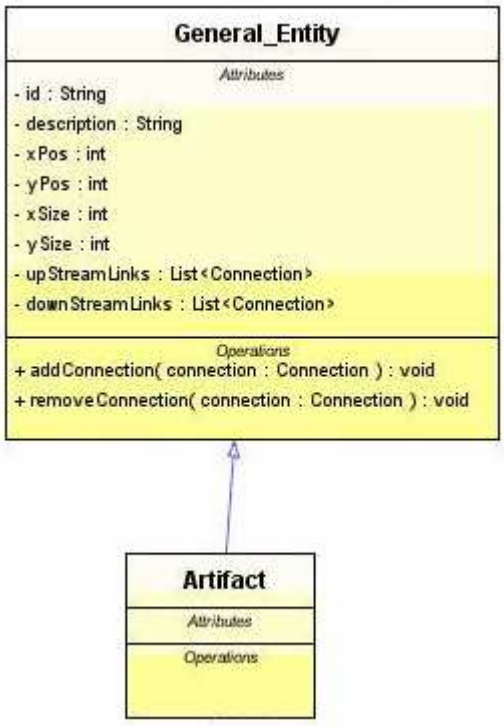

Fig. 4.36: Diagrama de clases (entidades)

Artifact hereda de *General_Entity*, no contiene ningún atributo ni método adicional ya que todo lo que necesita lo contiene la clase padre. Y como cualquier entidad del sistema puede substituirse por un *Artifact* hacemos que todas las demás entidades hereden de ésta. En la siguiente ilustración se puede observar esto:

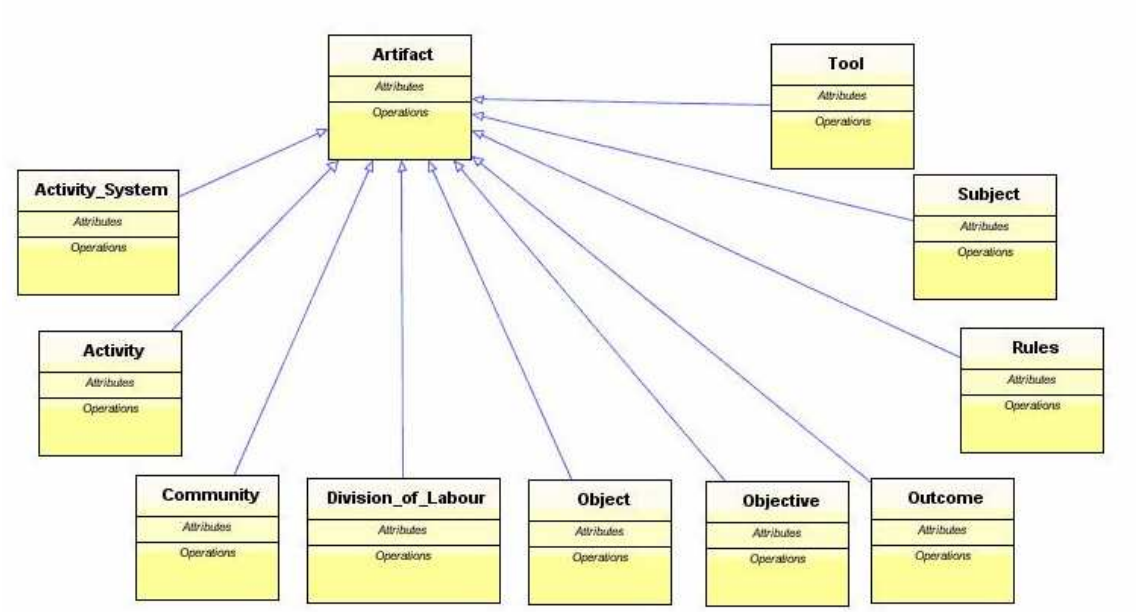

Fig. 4.37: Diagrama de clases (entidades II)

Se puede observar que ninguna de las entidades tiene ni atributos ni método adicionales, ya que todas funcionan igual y por tanto esos atributos y métodos que necesitan están en la clase más general, es decir, en *General_Entity*.

De la clase *Connection* heredan todos los tipos de relaciones existiendo alguna distinción entre ellas. Hemos establecido cuatro tipos de relaciones principales: *GeneralRelation, Inheritance,* D*escompose* y C*hangeOfRole*. En la siguiente figura se puede apreciar:

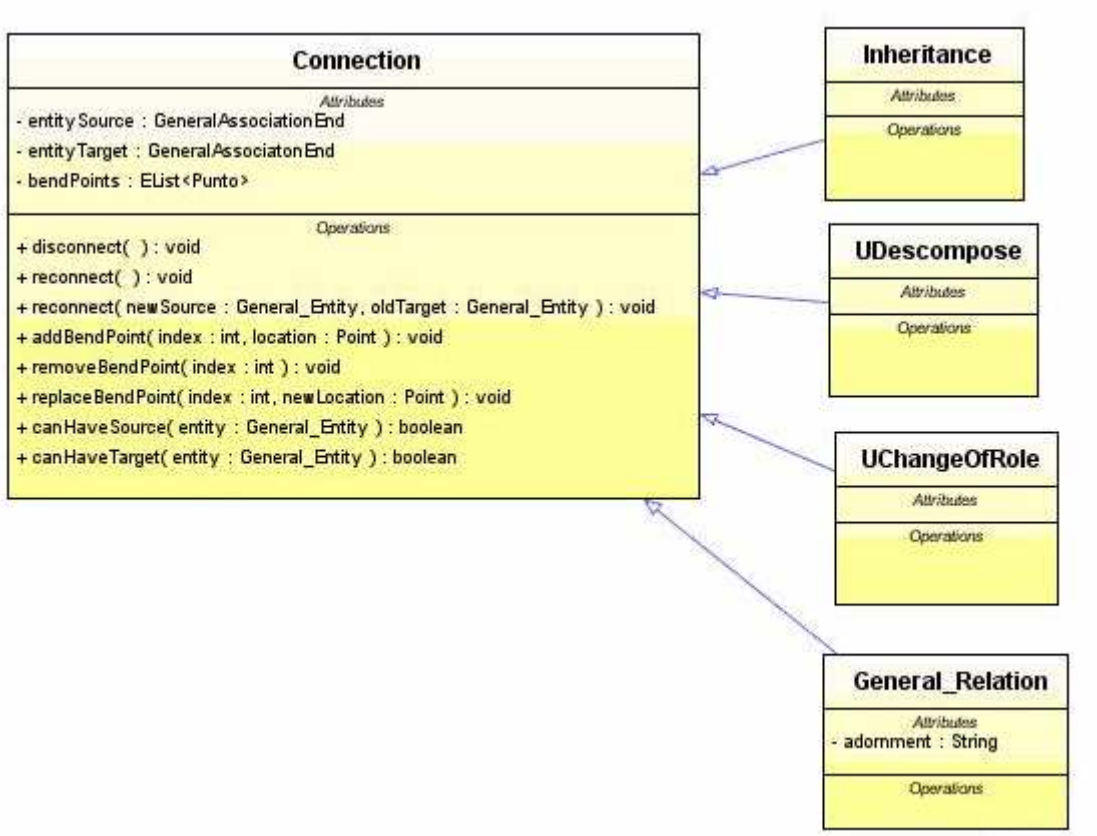

Fig. 4.38: Diagrama de clases (relaciones)

Se observa que todos los tipos de relaciones no contienen ni atributos ni métodos adicionales, ya que pasa lo mismo que con las entidades, salvo *General_Relation* que contiene el atributo *adornment*, atributo que el resto de relaciones que están en la figura no pueden contener.
Por ello de esta clase heredan el resto de relaciones que si lo pueden tener y que se muestra a continuación:

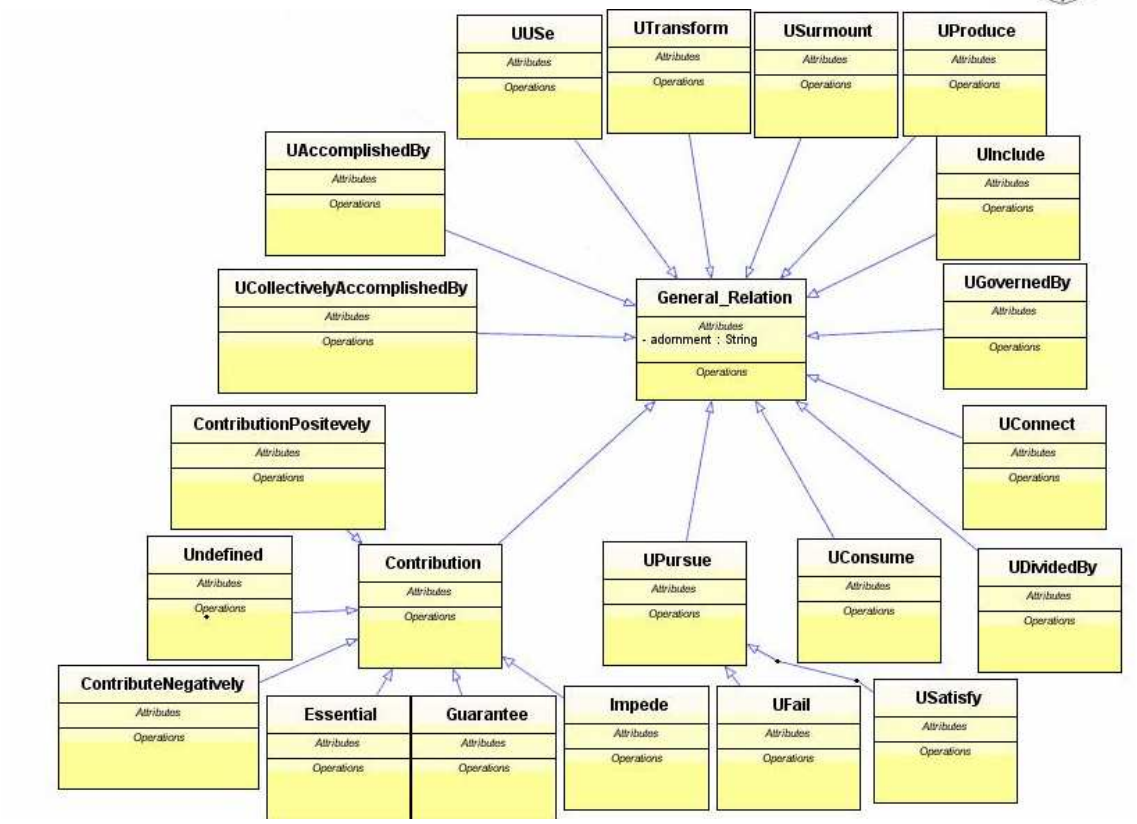

Fig. 4.39: Diagrama de clases (relaciones II)

En ésta figura puede observarse como todas las demás relaciones que no habían aparecido heredan de *General_Relation*. También se puede apreciar una generalización de relaciones en *Contribution*, ya que en sí no es una relación sino un tipo de relación, pero sí el resto de relaciones que heredan de ella, lo mismo pasa con *UPursue*.

4.4.2. Integración del EMF (Eclipse Modeling Framework)

La utilización de este plug-in ha sido de gran ayuda a la hora de poder desarrollar el modelo de la herramienta, sobre todo en lo que se refiere a organización. También nos ha facilitado la implementación creando una gran cantidad de clases con sus métodos y atributos básicos a las que luego hemos añadido los métodos y atributos que hemos considerado necesarios.

La funcionalidad del plug-in es la de generar una jerarquía de clases distribuidas en paquetes en función del metamodelo desarrollado con Ecore. Para ello tiene en cuenta todos los elementos y atributos especificados en el metamodelo.

El EMF genera, para cada uno de los paquetes que conforman el metamodelo (*association_end*, *entities*, *relations* y *specification*), tres paquetes en los que configura las distintas clases que estarán asociadas a los elementos que se encontraban dentro de cada paquete del metamodelo.

Los tres paquetes que se crean los siguientes (en el caso de nuestra herramienta):

1. *umlat.nombre_paquete*: contiene las interfaces de todos los elementos que se hayan definido dentro del paquete. Cada clase viene dada con los accedentes y mutadores para los atributos que se hayan definido en el metamodelo.
2. *umlat.nombre_paquete.impl*: contiene las clases que se encargan de la implementación de las interfaces definidas en el paquete anterior.
3. *umlat.nombre_paquete.util*: como en la configuración especificamos que se han de guardar los modelos en un formato XML, dentro de este paquete se encuentran las clases *nombre_paqueteResourceFactory* y *nombre_paqueteResourceImpl* que se encargan de proporcionar los recursos necesarios para poder guardar los modelos en dicho formato.

Tanto el paquete descrito en el punto 1 como el descrito en el punto 2 contienen un par de clases adicionales llamadas *nombre_paqueteFactory* y *nombrepaquetePackage*.
La clase *Factory* viene con las constructoras de todos los elementos definidos en el paquete correspondiente. En el primer paquete vendrá la interfaz y en el segundo la implementación de todos los métodos descritos en la interfaz aparte de aquellos privados que sean necesarios.
La clase *Package* nos define los métodos para obtener las meta-clases del paquete en el que nos encontremos. En nuestro caso la hemos utilizado para obtener la clase *Factory* y poder usarla para crear las instancias en el modelo.

4.4.3. Fichero XML generado

Como ya se ha mencionado anteriormente, Ecore guarda los modelos creados en formato XML [18].

El proceso iterativo de guardado es el siguiente:

```
for(int i=0;i<nodes.size();i++){

    General_Entity entidad=nodes.get(i);
    resource.getContents().add(entidad);

    for(Connection conn:entidad.getUpstreamLinks()){
        if(conn!=null){
            resource.getContents().add(conn);
            resource.getContents().add(conn.getEntitySource());
            resource.getContents().add(conn.getEntityTarget());
            for(Punto point:conn.getBendPoints())
                if(point!=null)
                    resource.getContents().add(point);
        }
    }
}
```

Fig. 4.40: Extracto del código fuente de guardar modelo

Como se puede observar, para cada entidad encontrada se realiza lo siguiente:

- Se guarda su propia información.
- Acto seguido se obtienen aquellas relaciones cuyo origen es la entidad en cuestión, guardando la información correspondiente de cada una de ellas. Cabe destacar que no se producen repeticiones al respecto al tratar sucesivas entidades.

A continuación se muestra un ejemplo de modelo junto con su XML correspondiente, llevándose a cabo un pequeño análisis para facilitar su comprensión:

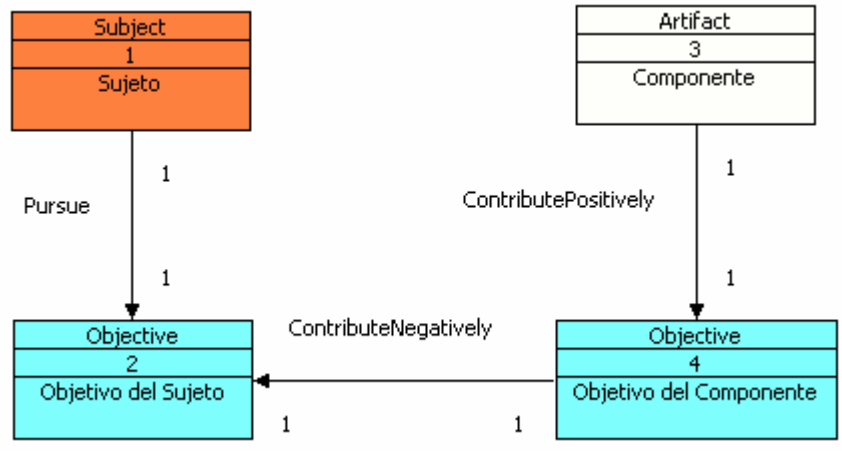

Fig. 4.41: Modelo ejemplo

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:association_end="bvc" xmlns:entities="rtyr"
xmlns:relations="asdf">
<entities:Subject id="1" description="Sujeto" xPos="172" yPos="139" xSize="120" ySize="60"/>
<relations:UPursue entitySource="/2" entityTarget="/3" adornment=""/>
<association_end:GeneralAssociationEnd multiplicityDown="1" multiplicityUp="1" entity="/0"/>
<association_end:GeneralAssociationEnd multiplicityDown="1" multiplicityUp="1" entity="/4"/>
<entities:Objective id="2" description="Objetivo del Sujeto" xPos="173" yPos="293" xSize="120" ySize="60"/>
<entities:Artifact id="3" description="Componente" xPos="454" yPos="136" xSize="120" ySize="60"/>
<relations:ContributePositively entitySource="/7" entityTarget="/8" adornment=""/>
<association_end:GeneralAssociationEnd multiplicityDown="1" multiplicityUp="1" entity="/5"/>
<association_end:GeneralAssociationEnd multiplicityDown="1" multiplicityUp="1" entity="/9"/>
<entities:Objective id="4" description="Objetivo del Componente" xPos="444" yPos="293" xSize="141" ySize="60"/>
<relations:ContributeNegatively entitySource="/11" entityTarget="/12" adornment=""/>
<association_end:GeneralAssociationEnd multiplicityDown="1" multiplicityUp="1" entity="/9"/>
<association_end:GeneralAssociationEnd multiplicityDown="1" multiplicityUp="1" entity="/4"/>
</xmi:XMI>
```

Fig. 4.42: Código XML del modelo ejemplo

Inicialmente y tras la cabecera XML comienza la información propia del modelo:

- *Entities*: aparece por cada entidad del modelo y contiene la siguiente información:
    - *Nombre* de la entidad.
    - *id*: identificador de la entidad.
    - *description*: descripción de la entidad.
    - *xPos*, *yPos*: posición que ocupa la entidad en el modelo (visualmente).
    - *xSize*, *ySize*: tamaño que ocupa la entidad en el modelo (visualmente).

- *Relations*: se corresponde con cada relación y su información es la siguiente:
    - *Nombre* de la relación.

- *entitySource*: indica la información referente al origen de la relación mediante un índice al número de línea correspondiente del archivo XML (comenzando desde cero y sin contar la información de cabecera).
- *entityTarget*: indica la información referente al destino de la relación de forma análoga al anterior.
- *adornment*: guarda el adorno de la relación en el caso de que exista.

- *Association_end*: hacen referencia a los extremos de la relación (por lo que siempre aparecen a pares) indicando la multiplicidad de la misma con respecto a la entidad origen y destino en ese orden:

  - *multiplicityDown*: expresa la multiplicidad inicial.
  - *multiplicityUp*: expresa la multiplicidad final.
  - *entity*: referencia a la entidad extremo de la relación (origen o destino), al igual que anteriormente mediante un índice al número de línea que ocupa dentro del archivo XML.

4.5. Descripción de la interfaz

4.5.1. Diagrama de clases

A continuación se muestra el diagrama de clases de la interfaz. Cabe destacar que hemos especificado con mayor claridad aquellas clases más importantes:

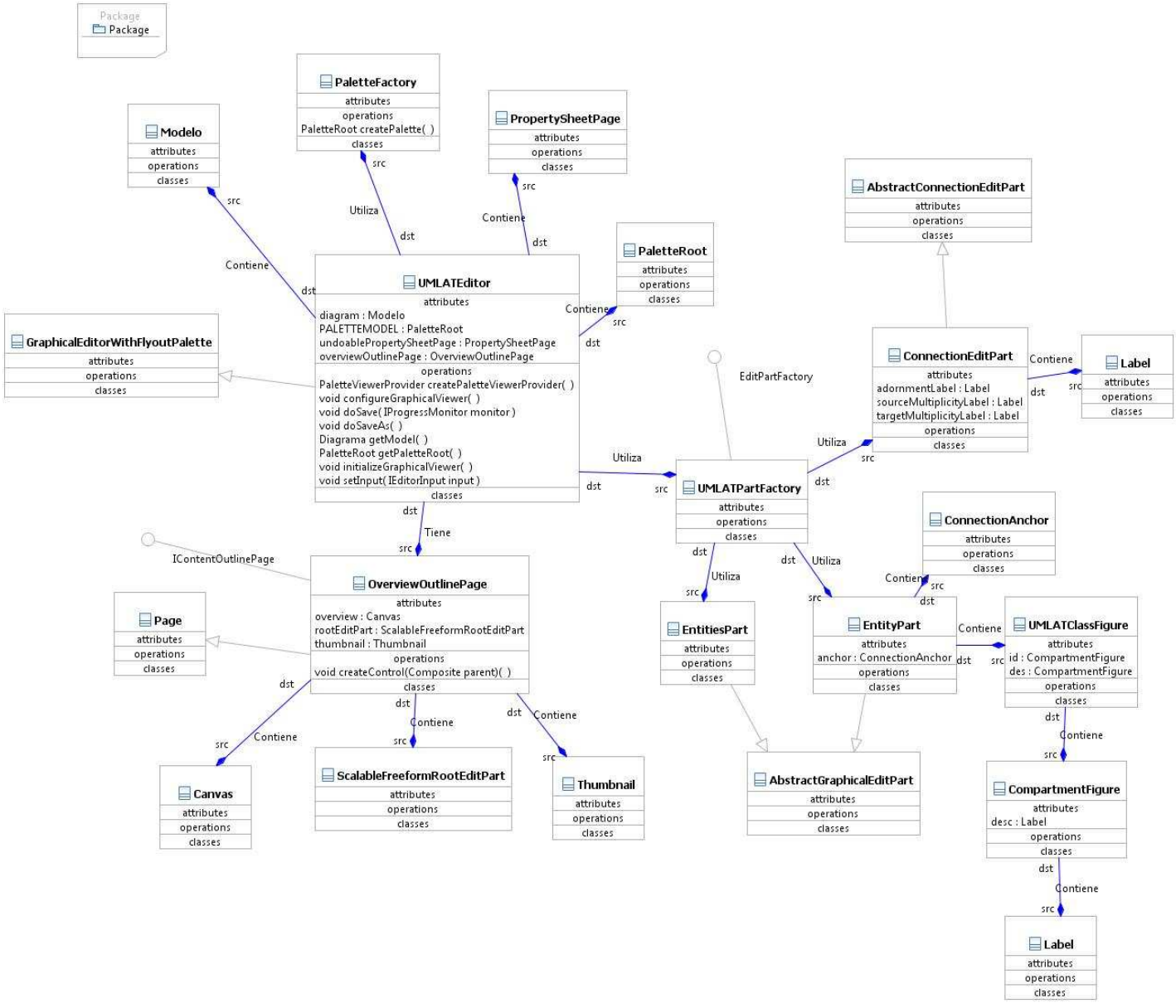

Fig. 4.43: Diagrama de clases de la interfaz

El diagrama dividido en varios módulos para facilitar su visualización es el siguiente:

- Clases principales del diagrama de clases de la interfaz:

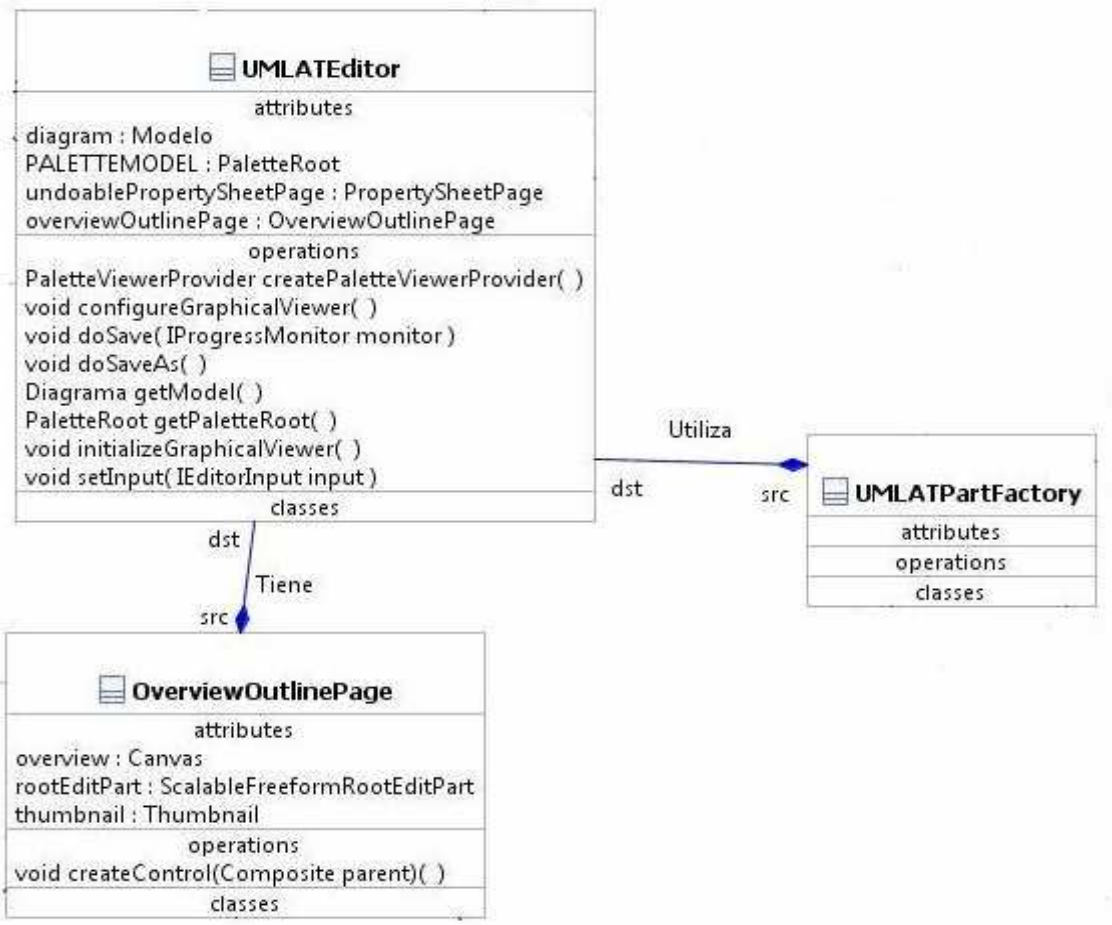

Fig. 4.44: Diagrama de clases de la interfaz (clases principales)

- Módulo de clases que se relacionan con *UMLATEditor*:

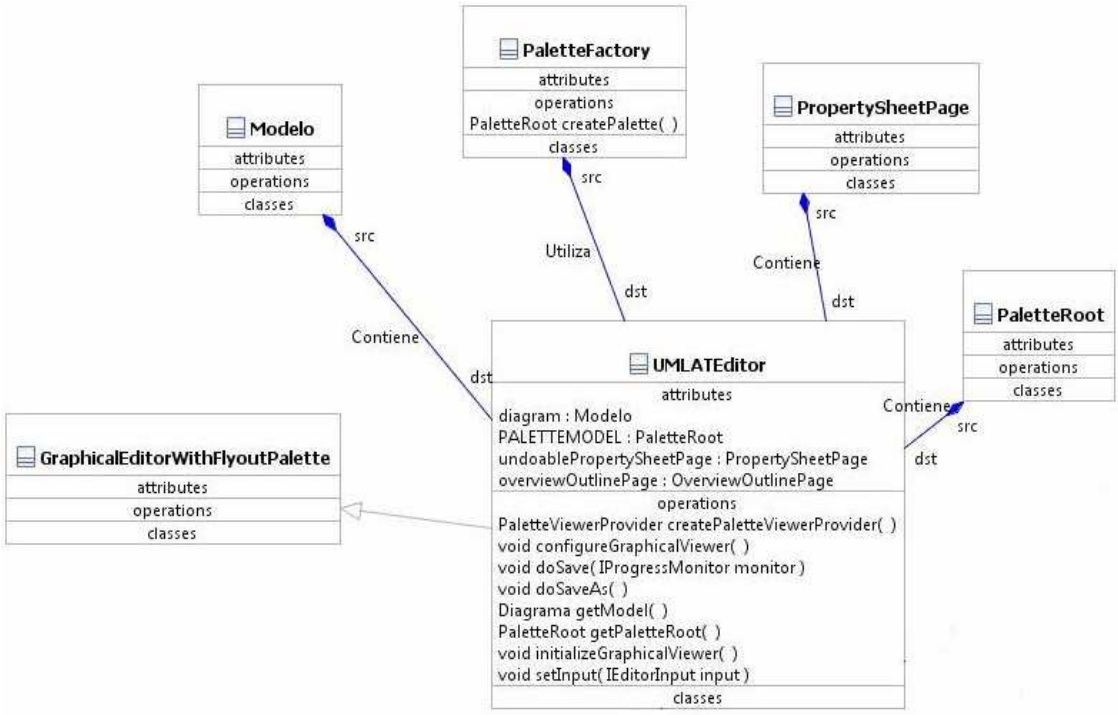

Fig. 4.45: Diagrama de clases de la interfaz (UMLATEditor)

- Módulo de clases que se relacionan con *OverviewOutlinePage*:

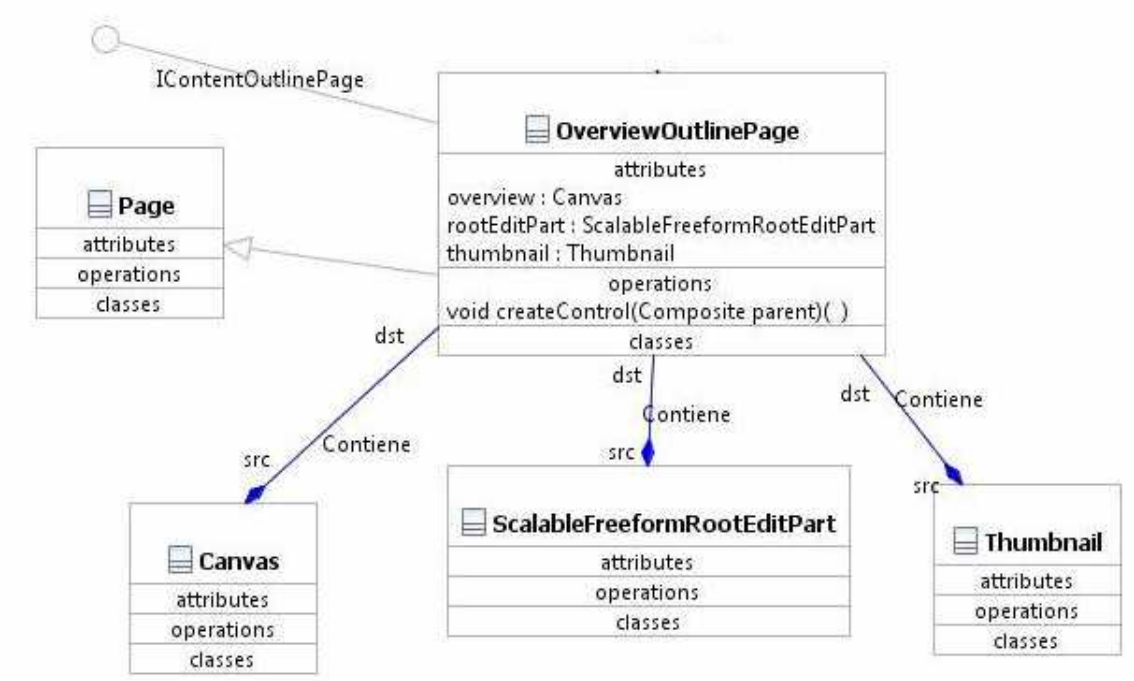

Fig. 4.46: Diagrama de clases de la interfaz (OverviewOutlinePage)

- Módulo I de clases que se relacionan con *UMLATPartFactory*:

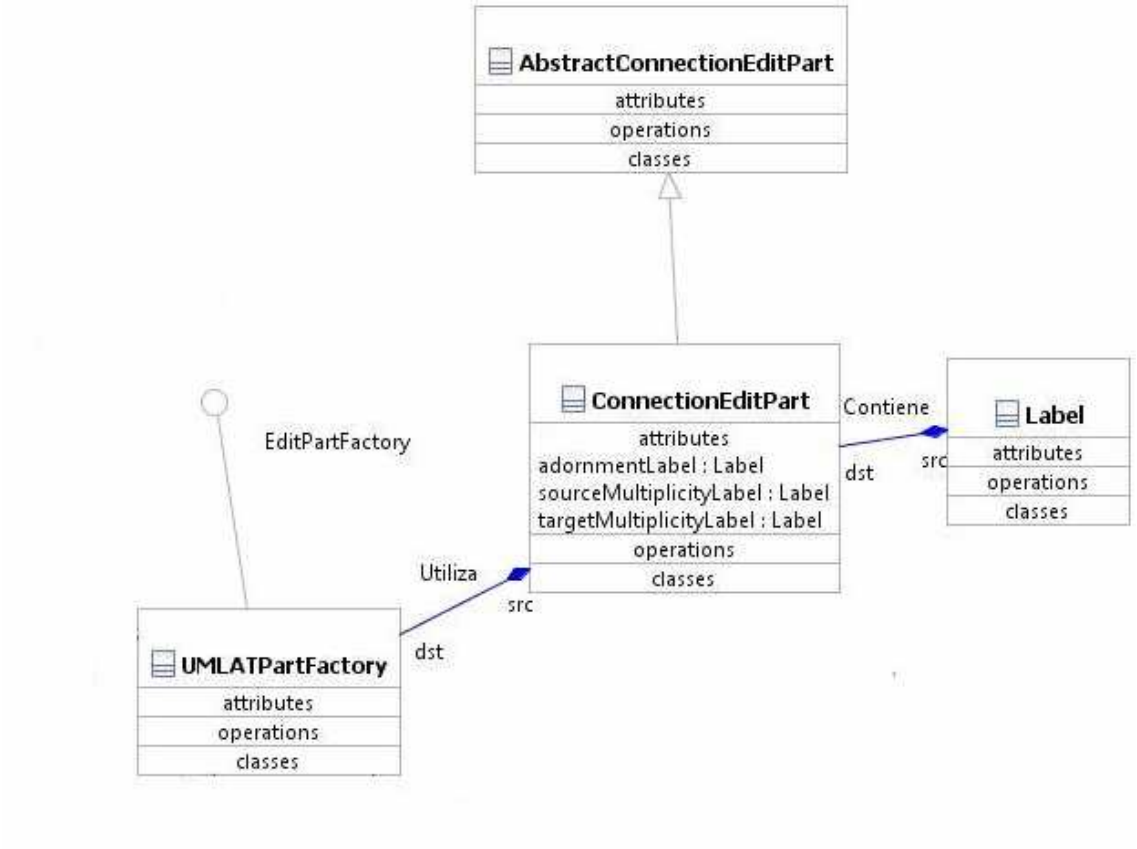

Fig. 4.47: Diagrama de clases de la interfaz (UMLATPartFactory)

- Módulo II de clases que se relacionan con *UMLATPartFactory*:

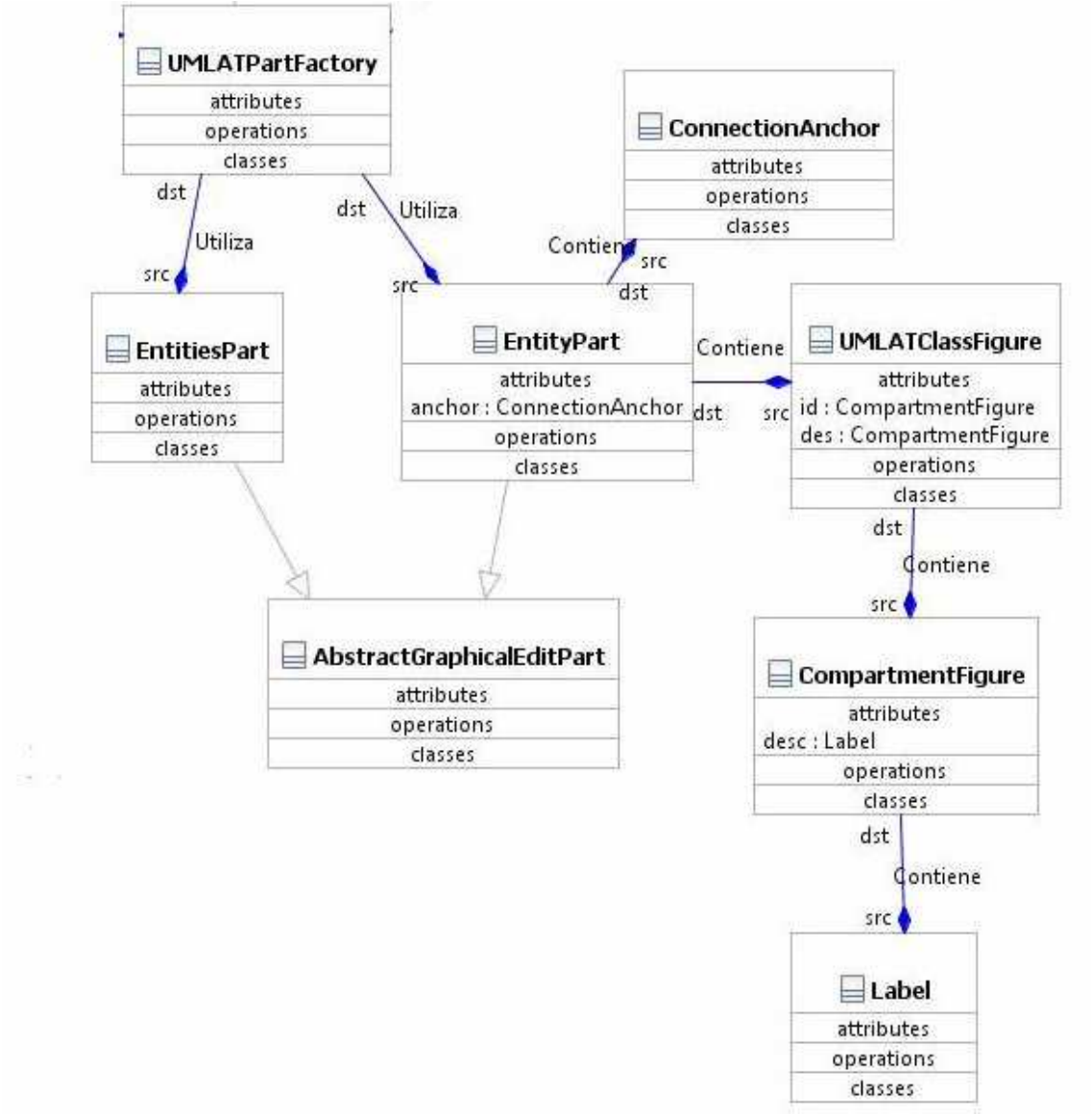

Fig. 4.48: Diagrama de clases de la interfaz (UMLATPartFactory II)

A continuación tiene lugar la explicación de cómo interacciona este diagrama de clases al completo.

La clase principal de la interfaz es la llamada *UMLATEditor*, que contiene los siguientes elementos principales:

- Una instancia del modelo, para poder crear, eliminar y modificar elementos. Toda la interacción entre el modelo y la interfaz se lleva a cabo a través del controlador *UMLATPartFactory* que se encarga de realizar los distintos cambios ante los eventos producidos.
- El controlador *UMLATPartFactory*, este elemento no está como atributo ya que se le asigna al *graphicalViewer* que es la parte visual del editor.

- La paleta (*PaletteRoot*), que contendrá los elementos que hayamos definido. En nuestro caso contendrán todas las entidades más los distintos tipos de relaciones.
- También podemos destacar el atributo *overviewOutlinePage* para poder contemplar en miniatura el modelo en una parte del editor.

Dentro de esta clase destacamos los siguientes métodos:

- Los métodos encargados de inicializar y configurar el *graphicalViewer* son de los más importantes ya que es donde establecemos toda la conexión entre el modelo, controlador y otras funcionalidades como puede ser el control por teclado o el menú para borrar u obtener la ruta de la paleta.
- El método *init* que es llamado cuando se crea el editor y se encarga de devolver la ruta donde se creara y también devuelve el editor. También se encargar de inicializar el *graphicarViewer* y la paleta.
- El método *setInput* que es donde se carga el modelo en el caso de que existiese, y si no crea uno nuevo.
- Los métodos encargados de crear la paleta y configurarla. Aquí establecemos los elementos que queremos mostrar en la paleta. Para ello hemos creado un fichero XML con los nombres de las entidades que queremos que se muestren así como los colores con los que aparecen en la ventana de edición. También establecemos las relaciones para mostrar.
- Los métodos de *guardar* y *guardar como* que se encargar de llamar a los respectivos métodos creados en el modelo.
- Entre otros métodos también destacamos el que nos devuelve la instancia del modelo o el comando *commandStackChanged* para detectar si ha habido cambios en el modelo sin haberlo guardado.

La siguiente clase a destacar es una ya citada como es la del controlador *UMLATPartFactory* que a partir del elemento que le pasamos nos devuelve el tipo específico del mismo, que puede ser una entidad, una relación o un modelo.
En el caso de que sea una entidad devuelve un objeto *EntityPart*; si es una relación un objeto *ConnectionEditPart*, y si es un modelo un *EntitiesPart*. Estos tres tipos de clases son muy importantes ya que contienen los elementos necesarios para la comunicación modelo-interfaz.
Este tipo se le pasa al editor que es quien se encarga de crear el elemento correspondiente en el modelo.

4.5.2. Integración del GEF (Graphical Editing Framework)

El plug-in GEF (Graphical Editing Framework) es una herramienta que ayuda en el desarrollo de editores gráficos, para ello nos valimos de un excelente documento [4] en el que vienen definidas todas las clases que se encuentran dentro de los paquetes:

- *umlat.parts*
- *umlat.presentation*

De dicha guía únicamente cogimos la implementación estándar que proporcionaba, para posteriormente retocarla y adaptarla a nuestra herramienta.

4.6. Diagramas de secuencia

Los diagramas de secuencia creados son los siguientes:

- Crear Modelo:

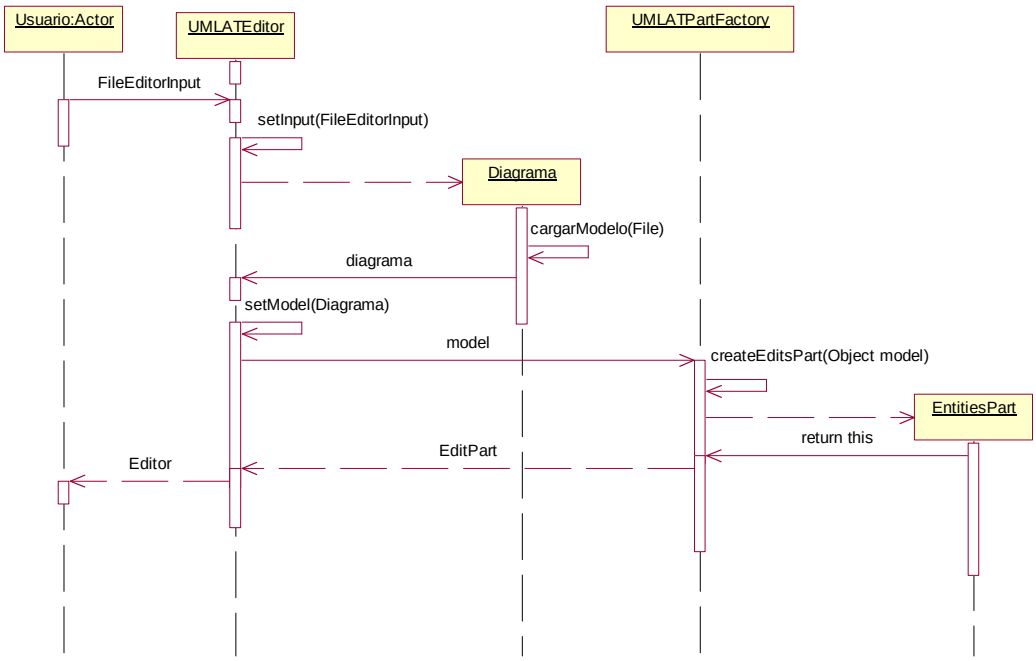

Fig. 4.49: Diagrama de secuencia “crear modelo”

Cuando el usuario abre un modelo que tenía previamente guardado, a la clase principal del editor se le pasa el fichero que contiene el diagrama.

Este fichero se carga en el editor mediante el método *setInput* cuyo fin es crear una instancia de *Diagrama*, que es la clase que se encarga de configurar todo lo relacionado

con el modelo; esta instancia crea las clases correspondientes al fichero que contiene el diagrama y devuelve el diagrama ya creado.
El editor recibe un cambio en su modelo, entonces se activa el controlador *UMLATPartFactory*, que recibe la parte del modelo que se ha actualizado, en este caso es la clase principal que contiene todos los elementos del modelo, ya que contiene la lista de todas las entidades y éstas contienen una lista de las relaciones que salen de ellas y otra lista de las relaciones que entran. *UMLATPartFactory* al recibir algo de tipo *Diagrama* se crea un *EditPart*(*EntitiesPart*) asociado al *Diagrama* y se lo devuelve al *UMLATEditor* que se lo muestra al usuario.

- Eliminar Entidad:

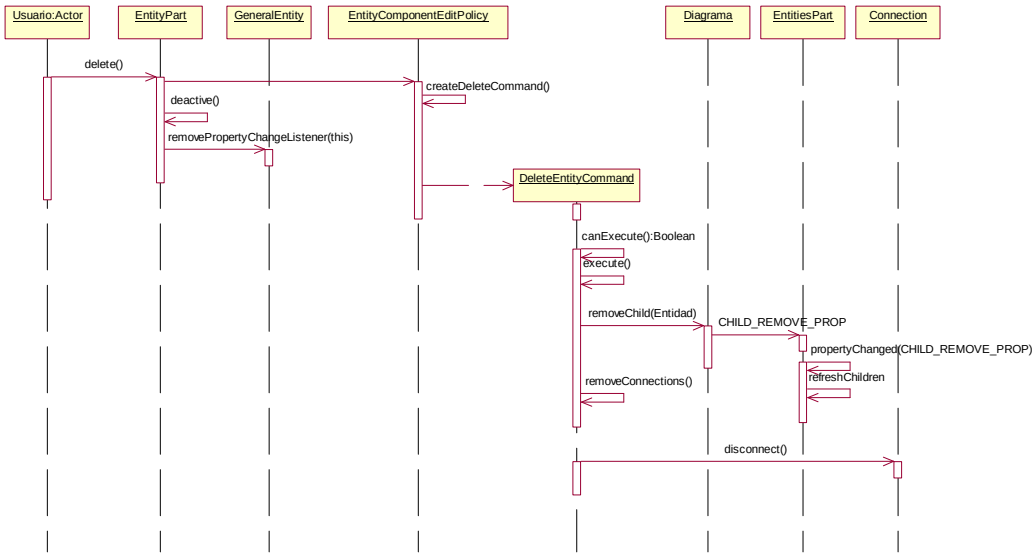

Fig. 4.50: Diagrama de secuencia “eliminar entidad”

Cuando un usuario elimina una entidad del diagrama esa llamada va a parar al *EntityPart*, el cual al recibir la llamada de eliminar activa el policy instalado *EditPolicy.COMPONENT_ROLE* el cual tiene asociada la clase *EntityComponentEditPolicy* que se encarga de crear el comando encargado de la eliminación de la entidad y de sus relaciones pertinentes en el modelo.

Todo esto lo maneja la clase *DeleteEntityCommand* que ha creado la clase anterior.
Primero comprueba si la acción de eliminar se puede ejecutar. Si la respuesta es afirmativa le dice al diagrama que elimine la entidad. El diagrama elimina la entidad del modelo y activa la propiedad *CHILD_REMOVE*, el *EditPart*(*EntitiesPart*) asociado al diagrama recibe la propiedad de *CHILD_REMOVE_PROP* y se activa el método *propertyChanged* que a su vez refresca y actualiza todas las entidades.

Tras eliminar la entidad, se deben eliminar todas las relaciones en las que participaba, para ello se llama al método *removeConnections*() que a su vez llama a al método

disconnect() de todas las relaciones pertinentes para que sean eliminadas. En el diagrama de secuencia de *eliminarRelación* se puede ver con más claridad como funciona todo el proceso.

- Guardar Modelo:

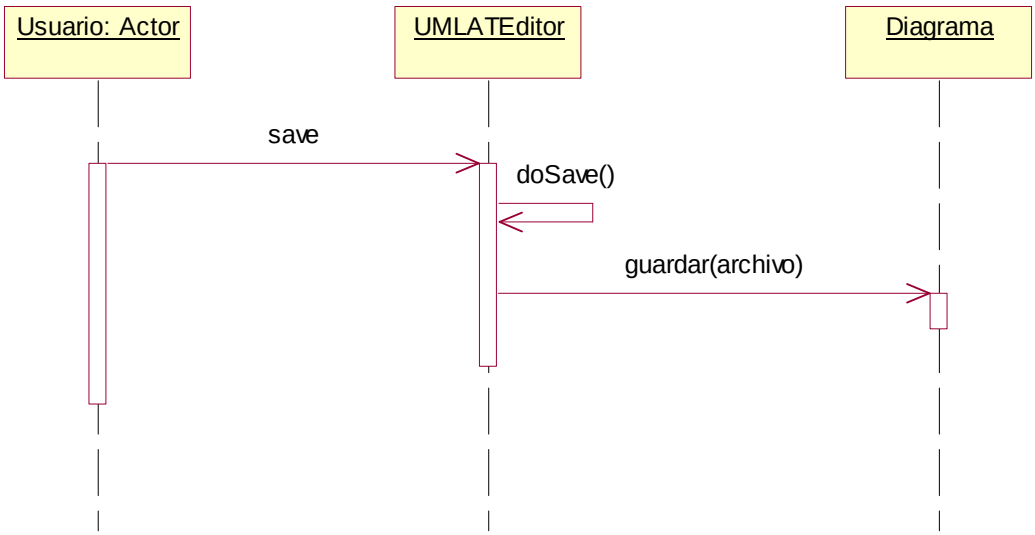

Fig. 4.51: Diagrama de secuencia “guardar modelo”

Cuando el usuario quiere guardar un modelo este paso se le comunica al editor. Como el editor ya tiene la ruta donde se guardó anteriormente el modelo, lo único que hace es pasarle la ruta del fichero donde se debe guardar el modelo al *Diagrama*.
El diagrama tiene un método interno llamado *save* que se encargar de realizar todo lo necesario para guardar el diagrama en formato XML.

- Borrar Relación:

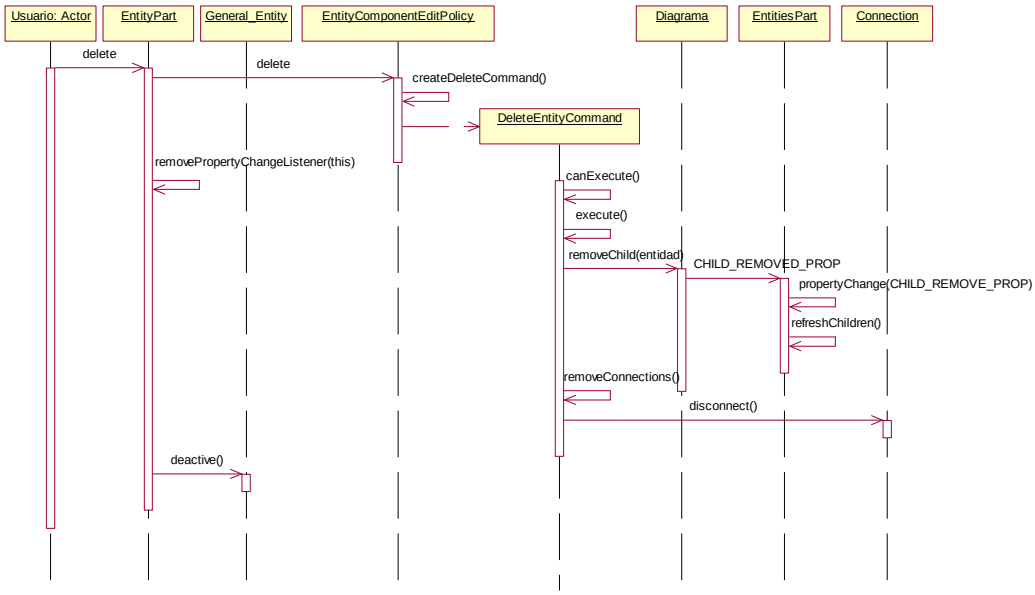

Fig. 4.52: Diagrama de secuencia “borrar relación”

Cuando un usuario elimina una relación del diagrama esa llamada va a parar al *ConnectionEditPart* asociado a la relación. Entonces se activa el policy *EditPolicy.CONNECTION_ROLE* que esta asociado a la clase *ConnectionEditPolicy*, que como en el caso de borrar entidades lo que hace es crear el comando que se encargará de la eliminación de la relación.

Para ello crea una instancia de la clase *ConnectionDeleteCommand*, en donde se realiza la desconexión con el método *disconnect*() de la relación, obligando a borrar la conexión del *source* y el *target* con el método *removeConnection*(*Connection*), que a su vez lanza la propiedad adecuada para que se refleje el cambio en la parte visual; al recibir dicha propiedad el *EditPart* del *source* y el *target*, ésta manda refrescar las relaciones de entrada y salida mediante refreshSourceConnections() o bien refreshTargetConnections().

Al final el *EditPart*(*ConnectionEditPart*) asociada a la relación, realiza el método *deactive*(), que manda a la relación que le borre como oyente.

- Añadir Entidad:

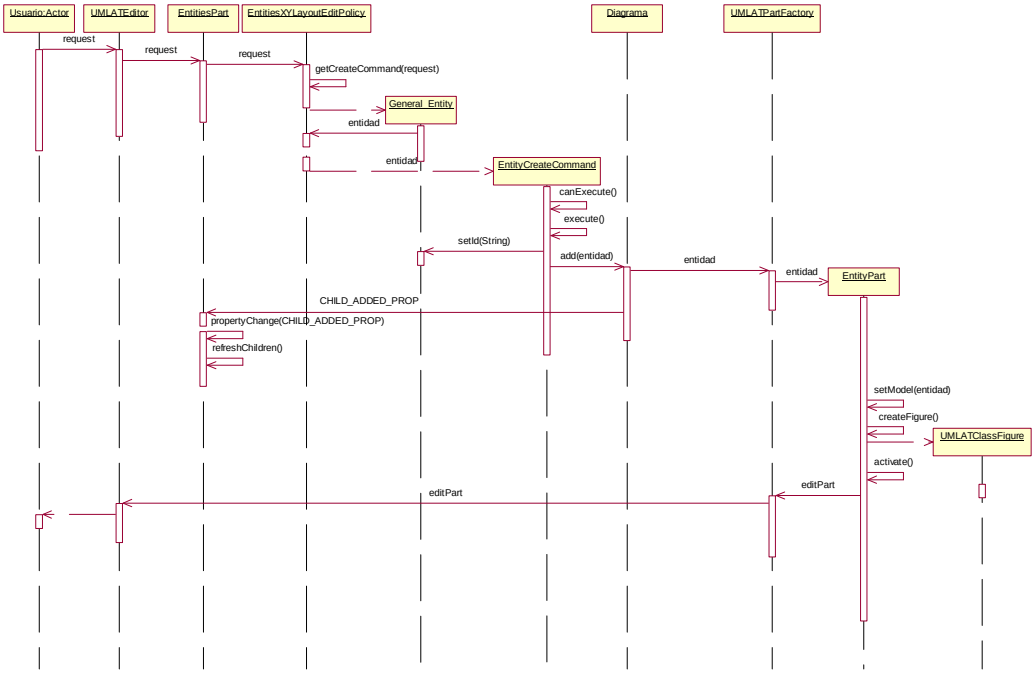

Fig. 4.53: Diagrama de secuencia “añadir entidad”

Cuando el usuario pincha en la paleta una entidad, la paleta le devuelve el tipo de entidad seleccionada, que va en el *request*; el editPolicy *EditPolicy.LAYOUT_ROLE* del *EntitiesPart* se activa al producirse un cambio en el diagrama y por tanto activa el método *getCreateCommand(CreateRequest request)* asociado a la policy *EntitiesXYLayoutEditPolicy*; el *request* viene de la paleta y ejecuta el comando *EntityCreateComand*(), en el que se pregunta si se puede ejecutar, y de ser así se comprueba que el identificador asignado no existe, en cuyo caso se añade al diagrama.
Al producirse un cambio en el modelo salta el *EditPartFactory*, al que le entra la nueva entidad y crea un nuevo *EntityPart(EditPart)* que es devuelto al *UMLATEditor* para ser mostrado al usuario.

Como el *Diagrama* ha añadido una entidad a su lista de entidades lanza *firePropertyChange(CHILD_ADDED_PROP)*, que lo recibe el método *propertyChange* de su *EditPart(EntitiesPart)*, y refresca todos las entidades con el método *refreshChildren*().

- Añadir Relación *Inheritance*:

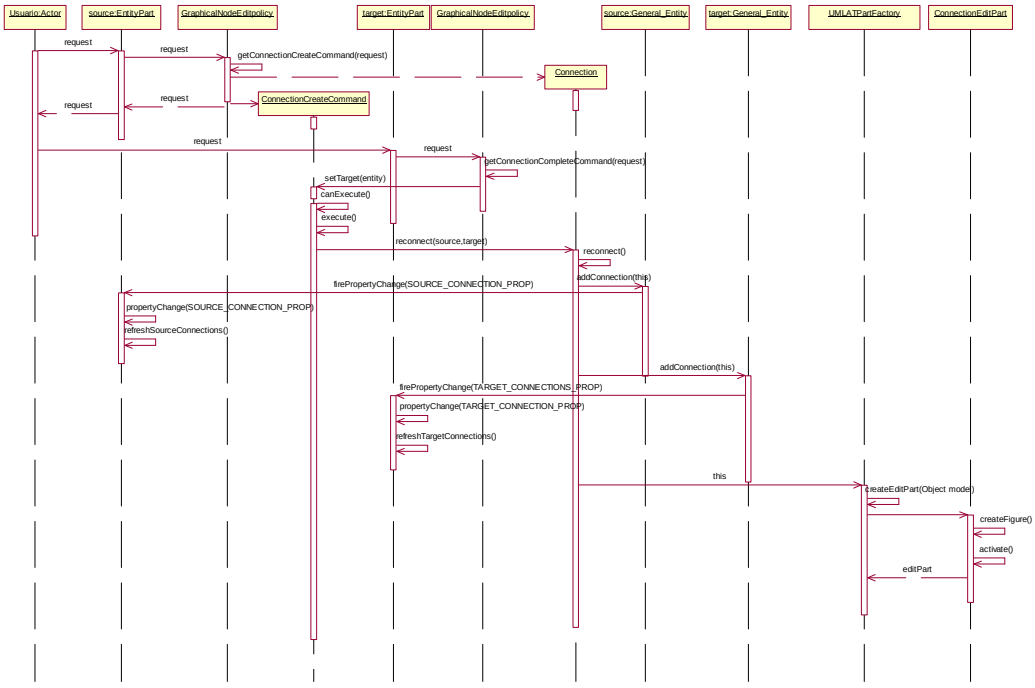

Fig. 4.54: Diagrama de secuencia “añadir relación *inheritance*”

Al igual que en *añadir Relación*, nos llega un *request* que viene de la paleta con el valor seleccionado de esta, en este caso se ha pinchado sobre un *EntityPart(EditPart)* que activa el policy *GraphicalNodeEditPolicy*, dentro se ejecuta el método *GetConnectionCreateCommand(CreateConnectioRequest request)*.

Posteriormente se crea una instancia de la clase *ConnectionCreateCommand* pasándole la entidad origen y el objeto que representa el tipo de relación seleccionada, en este caso la relación de herencia (*Inheritance*); se pregunta si se puede ejecutar el comando pero hasta que el usuario no haga clic en la entidad destino devuelve falso, es decir no se puede ejecutar. Una vez que el usuario ya ha hecho clic en la entidad destino, pasa igual que cuando selecciona en la primera *entidad(Source)*, aunque ahora se activa otro método del *GraphicalNodeEditPolicy*, ya que el comando ya lo ha empezado la entidad origen, y el método que se ejecuta es el *getConnectionCompleteCommand(CreateConnectionRequest request)*, aunque ahora el *request* es el comando creado por la entidad origen. A este comando se le asigna la entidad destino con el método *setTarget(GeneralEntity e)* y se devuelve el comando.

Como hemos asignado el *target*, el comando se puede desbloquear siempre y cuando la relación pueda tener como origen y destino las entidades seleccionadas, en este caso se puede siempre dado que se trata de una relación de herencia.

Como se puede ejecutar el comando pasamos a ejecución, donde llamamos al método *reconnect*() de la relación; este método lo que hace es asignarle como conexión de entrada y salida a cada una de las entidades, dependiendo de si son origen o destino, con el método *addConnection*(). Este método activa la propiedad *SOURCE_CONNECTION_PROP* o *TARGET_CONNECTION_PROP*, que activa el método *propertyChange* de *entityPart* y es donde se refrescan las conexiones de entrada o las conexiones de salida, dependiendo de si es origen o destino.

Como ha habido una modificación en el modelo ya que se les ha asignado una relación a dos entidades, se ejecuta el método *createEditPart(Object model)*, al cual le entra como parámetro la relación, por lo tanto el método *createEditPart* devuelve un *ConnectionEditPart* al que se le asigna como modelo la relación con *setModel(relación)*; finalmente este *ConnectionEditPart* lo recibe el *UMLATEditor* y lo muestra al usuario.

5. Diseño del segundo plug-in: Editor UML-AT con transformaciones

5.1. Introducción

En este segundo plug-in se ha creado un editor doble, basándonos en el editor creado en el primer plug-in. En esta herramienta aparecerán dos editores, uno fuente y otro destino, con las mismas funcionalidades que tenía el de la primera aplicación.

La principal novedad es que permite llevar a cabo transformaciones entre modelos, generando automáticamente las reglas de transformación necesarias en ATL. Para ello en el editor de la izquierda se diseñará el modelo fuente que se quiere utilizar como patrón, y en el de la derecha se diseñará el modelo destino en el que se desea transformar el diagrama fuente.

Las reglas de transformación se guardarán en un fichero ATL, de forma que pueda ser portable a otras aplicaciones.
Cabe destacar que se debe tener muy en cuenta los identificadores que se usan para las entidades en ambos editores, ya que si alguna entidad del modelo destino tiene un identificador del modelo origen, todas las características de esa entidad origen (en este caso la descripción) se copiarán automáticamente en los atributos de la entidad destino, por lo que es responsabilidad del usuario no cometer tal error.

5.2. Configuración del plug-in

Al igual que en la configuración del primer plug-in, seleccionamos el mismo tipo de plug-in y configuramos los mismos puntos.

De tal forma que la pestaña *Dependencies* queda así:

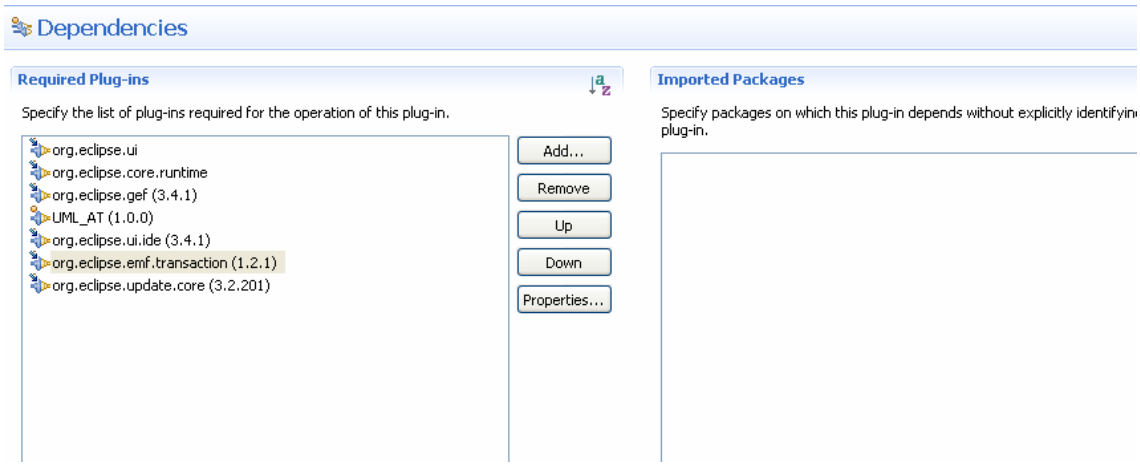

Fig. 5.1: Configuración *Dependencies*

Destacar que hemos añadido el primer plug-in en la lista de plug-ins requeridos debido a que constituye una parte indispensable para la realización de esta segunda herramienta.
Las demás dependencias que aparecen ya han sido explicadas en la configuración del primer plug-in.

Aparte de esto no hay que llevar a cabo ninguna configuración adicional, ya que al importar el primer plug-in, el segundo obtiene todo lo necesita para poder operar correctamente.

5.3. Casos de uso

Los casos de uso para esta segunda herramienta son similares a los del primer plug-in añadiéndole además las siguientes funcionalidades:

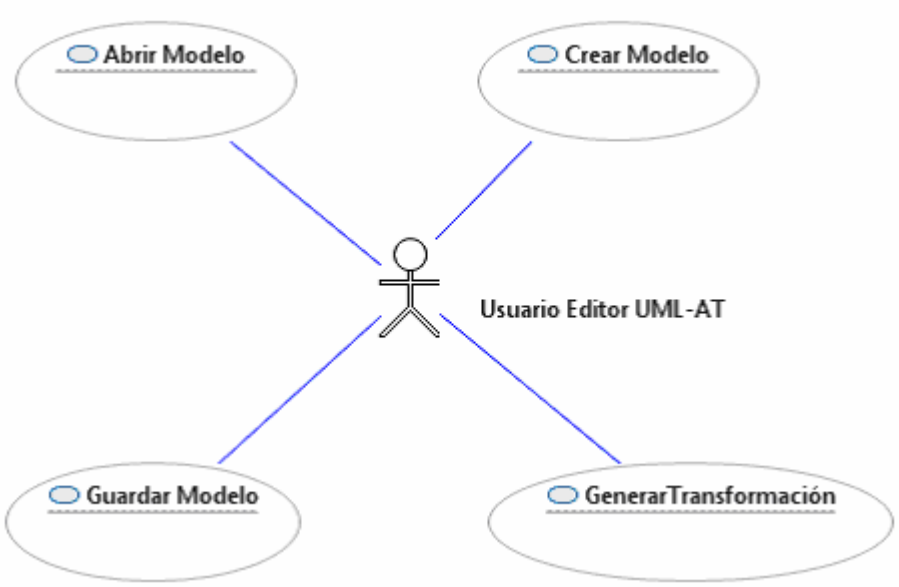

Fig. 5.2: Casos de uso

Algunos casos de uso como *Abrir Modelo* o *GuardarModelo* se vuelven a referenciar aquí puesto que incorporan nuevas funcionalidades sobre su uso:

1. *Abrir Modelo*: el usuario puede abrir un modelo de la misma forma ya comentada en el primer plug-in, salvo que ahora han de tener la extensión propia de este segundo plug-in.

   La novedad en esta funcionalidad es que ahora aparece un botón de “cargar modelo” para cada uno de los dos editores:

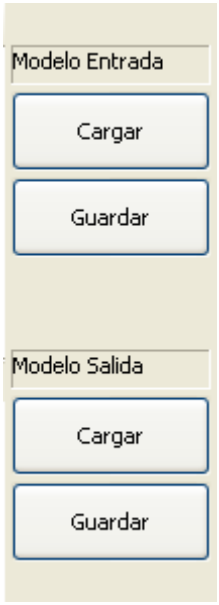

Fig. 5.3: Botones para cargar y guardar ambos modelos

Existen dos botones de “cargar”, el superior corresponde al modelo de entrada (visualizado en el editor de la izquierda) y el inferior corresponde al modelo de salida (visualizado en el editor de la derecha).
Al pulsarlos aparece una ventana en la que se debe indicar la ruta del fichero que contiene el modelo a cargar.

2. *Crear Modelo*: análogo al *crear modelo* del primer plug-in, salvo que en este caso la extensión del fichero debe ser “umlattrans”.
3. *Guardar Modelo*: análogo al guardar modelo del primer plug-in, salvo que cambia la extensión del fichero.
4. *Generar Transformación*: esta es la principal novedad de este segundo plug-in. El usuario puede generar un fichero de transformación ATL entre el modelo de origen y el de destino.

   Para ello primero debe tener los dos modelos correspondientes y una vez los haya diseñado a su gusto pulsar el botón “Generar”:

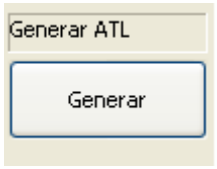

Fig. 5.4: Botón para generar la transformación entre modelos

Una vez se haya pulsado aparecerá una ventana en la que se indicará el proyecto donde se quiere guardar el fichero, así como el nombre del fichero (con extensión “atl”).

5.4. Descripción de la interfaz

5.4.1. Diagrama de clases

El diagrama de clases de la interfaz es el siguiente:

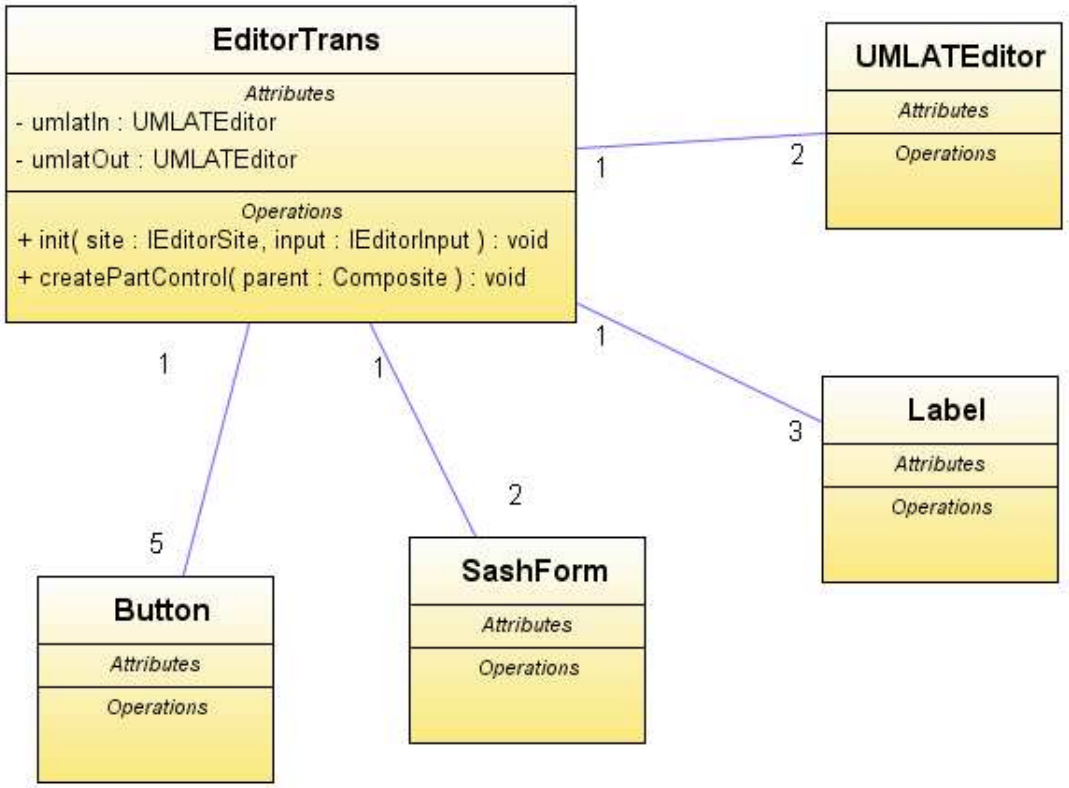

Fig. 5.5: Diagrama de clases de la interfaz

Como se puede observar en el diagrama de clases, el *EditorTrans* consta de dos *SashForm*, uno con formato en vertical donde insertamos los botones y las etiquetas, y otro con formato horizontal donde se insertan los dos *UMLATEditor* y el primer *SashForm*.

La disposición visual en la aplicación de los elementos que componen el segundo *SashForm* mencionado es la siguiente: a la izquierda se encuentra el editor *UMLATEditor* del modelo de entrada, en el medio de la aplicación el primer *SashForm* (con los botones de guardar y cargar para los modelos de entrada y salida) además del botón de generar la transformación ATL, y a la derecha el editor *UMLATEditor* del modelo de salida.

Las funcionalidades de los botones de la interfaz gráfica son las siguientes:

- Botón cargar: aparece un cuadro de diálogo donde se le pide al usuario la ruta del modelo UML-AT que desea cargar en el editor correspondiente (en el de la izquierda si es el de entrada o en el de la derecha si es el de salida).

- Botón guardar: permite guardar un modelo en una ruta especificada.

- Botón eliminar: permite eliminar el objeto seleccionado del editor correspondiente.

- Botón generar: genera las reglas de transformación ATL para transformar el modelo de entrada en el de salida, preguntando al usuario la ruta en la que desea guardar el fichero con dichas reglas.

5.5. Descripción del modelo

5.5.1. Diagrama de clases

El diagrama de clases del modelo es el siguiente:

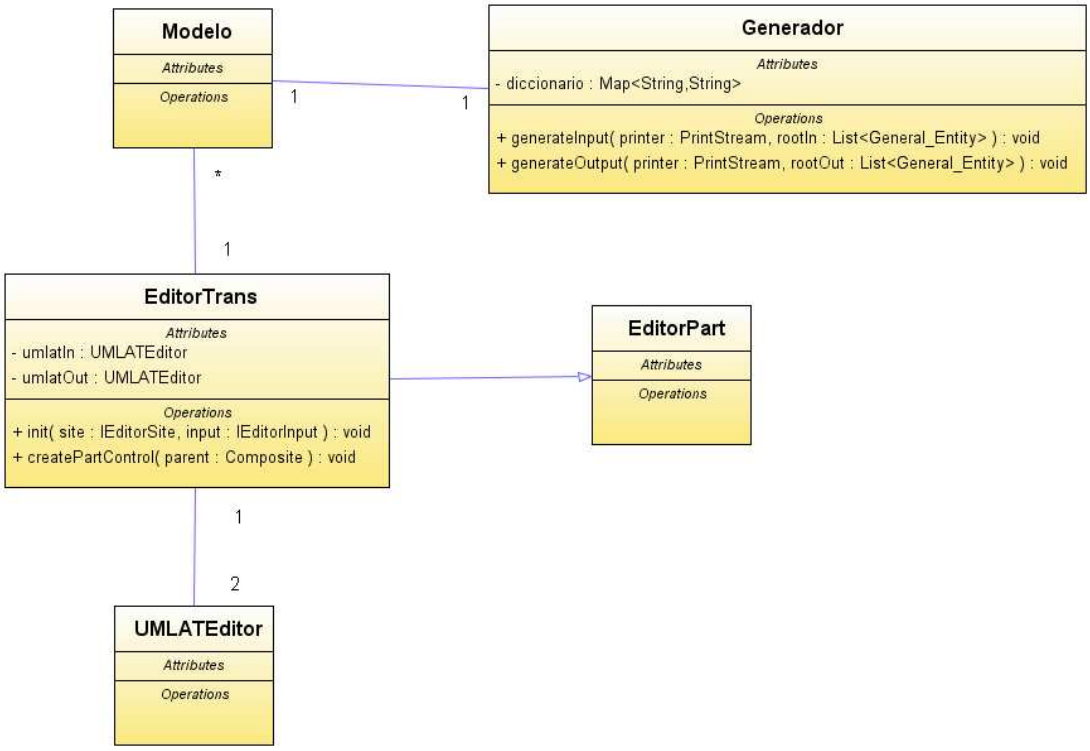

Fig. 5.6: Diagrama de clases del modelo

A continuación tiene lugar una pequeña explicación de cada clase:

- *EditorTrans*: es el editor para las transformaciones, hereda de *EditorPart*, por lo que hay que implementar los métodos no implementados ya que *EditorPart* es una clase abstracta; de esos métodos solo implementamos el *init*, que se ejecuta nada mas iniciar el editor y que recibe *IEditorSite* y *IEditorInput*, inicializando los dos editores UML-AT que contiene. Por otra parte en el método *createPartControl* es donde se va definiendo la interfaz del editor.

- *UMLATEditor*: es importado del primer plug-in.

- *Modelo*: se instancia cuando se generan las reglas de transformación y solo sirve para pasarle un *Printer* asignado a la ruta donde se desea generar las reglas ATL y dos listas de entidades, cada una correspondiente a cada modelo de los editores. Todo esto se pasa al *Generador*, que es donde se generan las reglas y se escriben en el *Printer* generado.

- *Generador*: se encarga de generar y escribir las reglas en el archivo ATL. Realmente solo genera una regla, que consiste en que a partir del diagrama de entrada devuelve el diagrama de salida; para ello se usan dos métodos, *generateInput*() que se encarga de escribir el modelo de entrada, y *generateOutput*() encargado de escribir el modelo de salida .

5.6. Diagramas de secuencia

A continuación se muestra el diagrama de secuencia de "generar las reglas de transformación ATL":

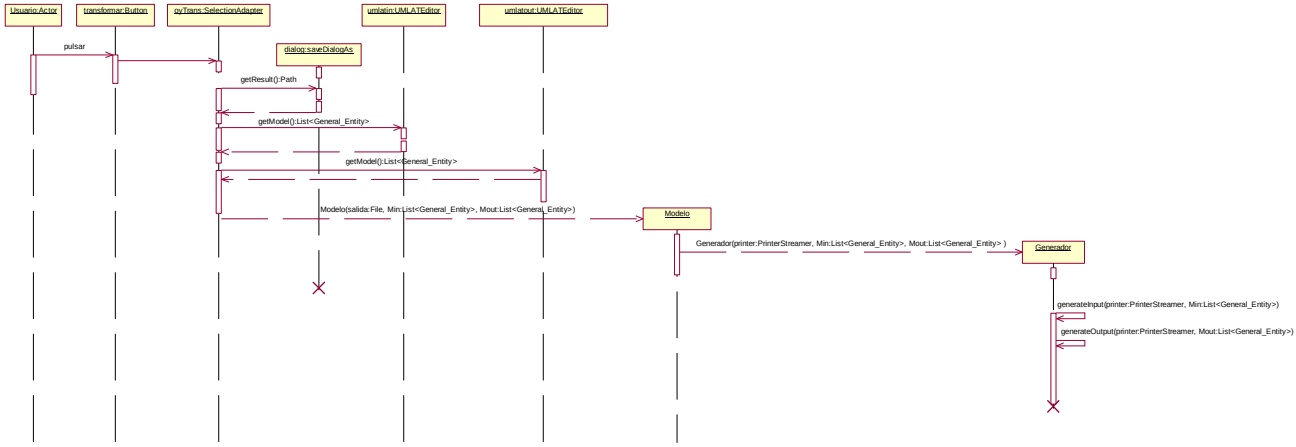

Fig. 5.7: Diagrama de secuencia de "generar las reglas de transformación ATL"

Cuando el usuario hace clic en el botón transformar, activa su oyente *oyTrans*, dando paso a una ventana *SaveAsDialog* donde se pide la ubicación donde generar la regla de transformación.

En dicha ubicación se crea un fichero para a continuación crear un *Modelo* al que se le pasa el susodicho fichero creado anteriormente y los diagramas de cada editor (origen y destino). En este nuevo *Modelo* se crea un *Generador* pasándole un *Printer* creado a partir del fichero que le entro al modelo, y los dos diagramas.

Dentro del editor se genera la correspondiente regla de transformación, en la cual queda definida que a partir del modelo de entrada (de esto se encarga el *generateInput*) se obtenga el modelo de salida (mediante *generateOutput*).

5.7. ATL

5.7.1. Introducción

ATL [6] [7] es un lenguaje de transformación entre modelos. Algunas de sus características más importantes desde el punto de vista de este proyecto son:

- Está soportado por Eclipse.
- Es el estándar de facto por su simplicidad.
- Existe bastante documentación al respecto.

Es por ello que resulta el candidato idóneo para llevar a cabo las transformaciones entre modelos en el plug-in que desarrollamos en este proyecto.

5.7.2. Fichero ATL generado

Dados dos modelos, uno de entrada y otro de salida, la aplicación genera como resultado un fichero ATL con las reglas necesarias para llevar a cabo la transformación entre ellos [15]. Considérese los siguientes modelos de ejemplo, siendo el modelo de entrada el situado a la izquierda y el de salida a la derecha:

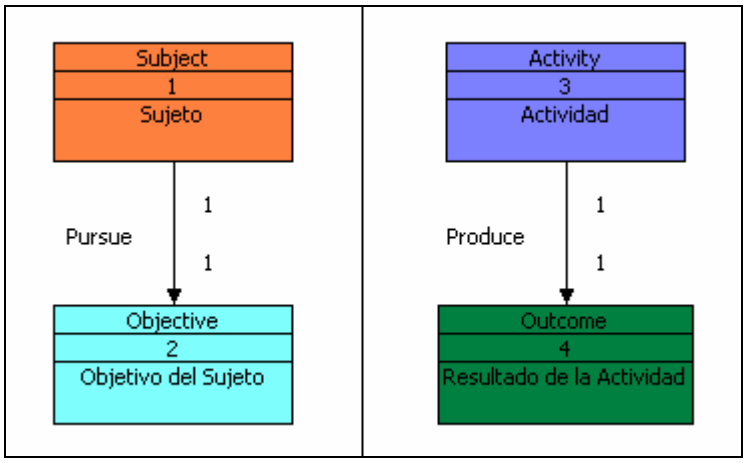

Fig. 5.8: Modelos ejemplo de entrada y salida respectivamente

El fichero ATL generado el siguiente:

```
module umlat2Umlat;
create OUT : MMB from IN : MMA;
rule In2out {
from
	cin:MMA!Subject(cin.id='1' and cin.description='Sujeto'
		and MMA!Objective.allInstances()->exists(e|e.id='2' and
			e.description='Objetivo del Sujeto')
		and MMA!UPursue.allInstances()->exists(r|r.entitySource.entity.id='1' and
			r.entityTarget.entity.id='2' and r.entitySource.multiplicityDown=1
			and r.entitySource.multiplicityUp=1 and
```

```
                    r.entityTarget.multiplicityDown=1 and
                    r.entityTarget.multiplicityUp=1 and r.adornment='')
 )
to

        oute1:MMB!Activity(
                id<-'3',
                description<-'Actividad',
                xPos<-82,
                yPos<-113,
                xSize<-120,
                ySize<-60),
        oute2:MMB!Outcome(
                id<-'4',
                description<-'Resultado',
                xPos<-76,
                yPos<-254,
                xSize<-120,
                ySize<-60),
        outa1:MMB!GeneralAssociationEnd(
                multiplicityDown<-1,
                multiplicityUp<-1,
                entity<-oute1),
        outa2:MMB!GeneralAssociationEnd(
                multiplicityDown<-1,
                multiplicityUp<-1,
                entity<-oute2),
        outr1:MMB!UProduce(
                entitySource<-outa1,
                entityTarget<-outa2,
                bendPoints<-Sequence{},
                adornment<-'')
}
```

Fig. 5.9: Fichero de transformación ATL generado

Como puede observarse se crea una regla de transformación cuyas características de entrada y salida son:

- Toma como *entrada* la entidad raíz del modelo de origen, a partir de la cual comienza la navegación del resto del modelo ayudándose de restricciones OCL para permitir la navegación a través de grafos inconexos cuando sea necesario. Para ello se recorren todas las entidades del modelo de entrada y se añade una restricción OCL en la que se comprueba si existe alguna instancia del tipo de la entidad con el mismo identificador y la misma descripción. A continuación se guarda la descripción en un diccionario asociándola a su identificador, para luego usarla en la salida. Tras esto, para cada relación de entrada (de cada entidad) se añade una restricción OCL donde se pregunta si existe una relación del mismo tipo, que tenga la misma entidad origen y la misma entidad destino (para ello se comprueba solo el identificador, ya que no puede haber identificadores repetidos, y si es de otro tipo no cumpliría la restricción de la entidad), comprobando también que tenga la misma multiplicidad de entrada y de salida.

- La *salida* se estructura como varios patrones de salida, uno por cada elemento del modelo destino, de forma que se copian las propiedades pertinentes a cada elemento del modelo a la vez que se establece las conexiones existentes entre ellos a través de variables locales (en el ejemplo *oute1* y *oute2* para las entidades; *outa1* y *outa2* para la relación *produce*). Además para las entidades se

comprueba si su identificador existe en el diccionario, en cuyo caso se copia la descripción de la entidad del modelo origen en la entidad del modelo de salida.

6. Conclusiones

Este proyecto nos ha ayudado enormemente a comprender la complejidad que conlleva el desarrollo de un proyecto a gran escala a la vez que nos ha aportado una gran satisfacción al concluirlo, debido a que por fin se cumple el eterno sueño del estudiante, por una parte finalizar su etapa académica, y por otra elaborar algo que realmente es de utilidad.

6.1. Aportaciones

Con la realización de este proyecto la principal aportación al mundo del desarrollo de software ha sido la creación de dos herramientas que ayudan y facilitan la elaboración de modelos orientados a aplicaciones cuyo contexto son las Ciencias Sociales, a la vez que sirve como primer paso hacia trabajos de ampliación futuros, tales como la generación automatizada de código a partir de los modelos creados.

Consideramos que los puntos fuertes de nuestra aplicación son por una parte su robustez y facilidad de uso, y por otra su gran versatilidad debido a que la integración de nuevos elementos en el lenguaje UML-AT se llevarían a cabo de forma sencilla. Desde el otro punto de vista su principal debilidad es la necesidad de una serie de componentes para su funcionamiento.

En lo que respecta a las mejoras que podrían llevarse a cabo, se podría mejorar el aspecto gráfico de la aplicación a la vez que implementar pequeñas funcionalidades tales como permitir cortar y pegar objetos del modelo, además de otras más útiles como permitir cambiar el tipo de una entidad o relación directamente.

6.2. Gestión del proyecto

Desde el comienzo, la gestión del proyecto se planteó como un proceso iterativo dividido en tres grandes fases:

- *Planificación y adquisición de conocimiento*: durante esta primera fase tuvo lugar toda la planificación temporal del proyecto así como la adquisición del conocimiento necesario para su desarrollo, tanto de la Teoría de Actividad como de los diferentes lenguajes y frameworks que se utilizan.

- *Desarrollo del proyecto*: posteriormente se procedió al desarrollo en sí del proyecto, es decir, la creación de ambas herramientas y su posterior evolución hacia su estado final.

- *Documentación y mantenimiento*: finalmente una tercera etapa en la que se trabajó en la documentación del proyecto así como también en la corrección de

pequeños errores encontrados tras someter las aplicaciones a una serie de pruebas finales.

6.3. Conclusiones académicas

Finalmente nos alegramos de que todas las asignaturas que hemos recibido durante la carrera hayan tenido su lugar de encuentro en este último trabajo, aunque hemos de destacar que las principales enseñanzas para el desarrollo del mismo han sido:

- Ingeniería del Software.
- Metodología y Tecnología de la Programación.
- Laboratorios de Programación.

También cabe destacar los conocimientos adquiridos durante la realización del proyecto, que son:

- Teoría de Actividad.
- Desarrollo de metamodelos con Ecore.
- Tecnologías EMF y GEF, junto con el lenguaje ATL.
- Ingeniería de Software dirigida por modelos.
- Desarrollo de software dirigido por modelos.
- Ingeniería de Software orientado a Agentes.
- Presentación de Trabajos Científicos.

7. Agradecimientos

Por último estos son nuestros agradecimientos particulares:

- **Francisco Domenech Marín**: a mis padres y profesores por la educación recibida, y a todos aquellos que me han apoyado.

- **Roberto Jiménez Domínguez**: a mis padres, por la excelente educación que he recibido mucho más allá de lo puramente académico.

- **Jorge Jiménez Rodríguez**: a todos aquellos compañeros de la facultad que me han ayudado en la elaboración del proyecto cuando he necesitado su ayuda.

Anexo: Equivalencias en UML-AT

Las equivalencias entre conceptos simplifican el procesamiento de los patrones. La idea es que un diagrama UML-AT puede ser sustituido por otro diferente sin cambiar nada de la especificación del sistema, manteniendo todo su significado por igual.

Un ejemplo de equivalencia sería que si tenemos actividades asociadas por la relación *connect*, todo puede ser sustituido por una sola actividad, ya que una secuencia de actividades puede interpretarse como una única actividad a un menor nivel de detalle.

La siguiente tabla describe las equivalencias existentes en el lenguaje UML-AT. Cada equivalencia se describe como una tupla con el concepto original, su sustituto y observaciones adicionales. En dicha tabla los roles aparecen en itálica y comenzando por mayúscula, las relaciones en itálica y en minúscula, y en los casos en los que varios elementos tienen el mismo rol se les asigna un nombre entre paréntesis.

Original	Sustituto	Observaciones
Artifact	Whatever concept	
Whatever concept	*Artifact*	
Artifact (origin)	*Artifact* (target)	*Artifact* (origin) → *inheritance* → *Artifact* (target)
Subjet (origin)	*Subject* (target)	*Subject* (origin) → *play* → *Subject* (target)
Activity → *satisfy* → *Objective*	*Activity* → *guarantee* → *Objective*	
Activity → *pursue* → *Objective*	*Activity* → *contribute positively* →*Objective*	
Activity → *fail* → *Objective*	*Activity* → *impede* → *Objective*	
Activity (origin) → *connect* → *Activity* (target)	*Activity*	
consume	*include* *use* *transform*	
Artifact (target)	*Artifact* (origin)	*Artifact* (origin) → *decompose* → *Artifact* (target)
Artifact (origin)	*Artifact* (target)	*Artifact* (origin) → *decompose* → *Artifact* (target)
Artifacto (target)	*Artifact* (origin)	*Artifact* (origin) → *decompose-OR* → *Artifact* (target)
Artifact (origin)	*Artifact* (target)	*Artifact* (origin) → *decompose-OR* → *Artifact* (target)
Artifact (target1) + *Artifact* (target2)	*Artifact* (origin)	*Artifact* (origin) → *descompone-AND* → *Artifact* (target1) + *Artifact* (target2)

Referencias bibliográficas

1. Eclipse Foundation. *Eclipse*. URL: http://www.eclipse.org

2. Eclipse Foundation. *Eclipse Modeling Framework Project (EMF).* URL: http://www.eclipse.org/modeling/emf/

3. R. Fuentes, J. J. Gómez-Sanz, and J. Pavón, *"Activity Theory for the Analysis and Design of Multi-Agent Systems"*, Agent-Oriented Software Engineering IV, 4th International Workshop on Agent-Oriented Software Engineering (AOSE-2003), Lecture Notes in Computer Science, vol. 2935, pp. 110-122, 2004, Springer-Verlag.

4. B. Moore, D. Dean, A. Gerber, G. Wagenknecht, P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, 2004.

5. D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., 2002.

6. ATLAS Group. *The Atlas Transformation Language (ATL).* URL: http://www.sciences.univ-nantes.fr/lina/atl

7. Eclipse Foundation. *ATL Project*. URL: http://www.eclipse.org/m2m/atl/

8. Eclipse Foundation. *Graphical Editing Framework (GEF).* URL: http://www.eclipse.org/gef/

9. Y. Engeström. *Learning by expanding: an activity-theoretical approach to developmental research*. Orienta-Konsultit, 1987.

10. A. N. Leontiev. *Activity, Consciousness, and Personality*. Prentice-Hall, 1978.

11. R. Fuentes, J. J. Gómez-Sanz, J. Pavón, *Captura del Entorno Social de Sistemas Multiagente*. Universidad Complutense Madrid, Dep. Sistemas Informáticos y Programación.

12. I. García, R. Fuentes, J. J. Gómez-Sanz, *Guideline for the defnition of EMF metamodels using an Entity-Relationship approach*, Department Software Engineering and Artifcial Intelligence. Facultad de Informática. Universidad Complutense de Madrid.

13. I. García, J. J. Gómez-Sanz, J. Pavón, *Representación de las relaciones en los Metamodelos con el lenguaje Ecore*, Departamento de Ingeniería del Software e Inteligencia Artificial. Facultad de Informática. Universidad Complutense de Madrid.

14. R. Fuentes, J. J. Gómez-Sanz, J. Pavón, *Integrating Agent-Oriented Methodologies with UML-AT*, In Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems *(AAMAS–2006)*, Hakodate, Japan, May 2006, pp. 1303-1310, ACM Press (2006).

15. *Model Transformation By-Example with generation of many-to-many rules*.

16. UCM. *Grasia Research Group*. URL: http://grasia.fdi.ucm.es/main/

17. Object Management Group. *Unified Modeling Languaje (UML)*. URL: http://www.uml.org/

18. W3C. *XML Schema*. URL: http://www.w3c.org/XML/Schema

Índice de figuras:

2. Teoría de Actividad

3. Metamodelo Ecore

4. Diseño del primer plug-in: Editor UML-AT

5. Diseño del segundo plug-in: Editor UML-AT con transformaciones