PROYECTO DE SISTEMAS INFORMÁTICOS

"EXTENSIÓN DE LA HERRAMIENTA DE MODELADO UML MOVA CON LA FUNCIONALIDAD DE EXPORTACIÓN E IMPORTACIÓN DE DIAGRAMAS EN EL ESTÁNDAR XMI"

AUTORES:

ANA MARIA GARCÍA SÁNCHEZ.
MIGUEL ÁNGEL GARCÍA DE DIOS.
CARLOS GALLEGO CARRICONDO.

DIRECTOR DEL PROYECTO:

Dr. MANUEL GARCÍA CLAVEL.

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

RESUMEN

La herramienta de modelado UML MOVA permite, además del diseño de diagramas UML, la evaluación de los mismos mediante restricciones OCL. Estas restricciones OCL garantizan que la información semántica, que el diseñador ha incluido en los diagramas, es la correcta. La herramienta MOVA es la única herramienta de modelado UML operativa que, actualmente, tiene una versión que incorpora este tipo de restricciones. MOVA almacena todos los diagramas mediante una representación propia en XML, lo que hace imposible el intercambio de diagramas con otras herramientas de modelado UML.

Nuestro proyecto consiste en dotar a la herramienta MOVA de la capacidad de exportar e importar diagramas UML mediante el estándar XMI 1.2 y el estándar "Diagram Interchange".

UML MOVA allows design UML diagrams and check them whit OCL restrictions. This OCL restrictions guarantee which the information contained in the UML diagrams correspond with the designer wants. MOVA is the unique running tool which actually has a version with OCL restrictions. MOVA store all the diagrams with a custom structure in XML. This circumstance makes impossible interchange diagrams with other UML modelling tools.

Our project consists of equipping the tool with the capacity to export and to import UML diagrams by XMI 1.2 standard and "Diagram Interchange" standard.

PALABRAS CLAVE

Diagramas UML

XMI

Diagram Interchange

DOM

JDOM

MOVA

OCL

BIBLIOGRAFÍA

- UML Resource Page: http://www.uml.org/

- Unified Modeling Language: Diagram Interchange.
http://lglpc35.epfl.ch/lgl/docs/Specifications/UML20/03-09-01%20DI.pdf

- UML 2.0 OCL Specification: http://www.omg.org/docs/ptc/03-10-14.pdf

- MOF 2.0 / XMI Mapping Specification, v2.1:
http://www.omg.org/technology/documents/formal/xmi.htm

- Mastering XMI: Java Programming with XMI, XML, and UML, Timothy J. Grose, Gary C. Doney , Stephen A. Brodsky .

- UML for Java Developers Model Constraints & The Object Constraint Language, Jason Gorman: http://www.parlezuml.com/tutorials/umlforjava/java_ocl.pdf

- Document Object Model (DOM): http://www.w3.org/DOM/

- JDOM : http://jdom.org/

Nosotros, Ana María García Sánchez, Miguel Ángel García de Dios y Carlos Gallego Carricondo, autores del presente proyecto, autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Ana María García Sánchez.
NIF: 03908944W

Miguel Ángel García de Dios
NIF: 30955188V

Carlos Gallego Carricondo.
NIF: 50892678A

INDICE

- **Introducción:**

  - **Proyecto**
  - **Tecnología XMI**
  - **Investigación:**
    - **Primera Parte**
    - **Segunda Parte**

- **Especificación del proceso**

- **Riesgos**

- **Librerías y Prototipos:**

  - **Escritura(DOM)**
  - **Lectura(JDOM)**

- **Implementación:**

  - **Escritura**
    - **Parte del modelo**
    - **Parte Gráfica**

  - **Lectura**
    - **Parte del modelo**
    - **Parte Gráfica**

- **Apéndice**

INTRODUCCION

PROYECTO

Nuestro proyecto ha consistido en desarrollar una aplicación que permita exportar e importar los diagramas UML que genera la herramienta de modelado MOVA Tool. Inicialmente, dicha herramienta exportaba e importaba los diagramas UML generados por el usuario utilizando la tecnología XML; con nuestro proyecto, dicha herramienta exporta e importa diagramas de clases utilizando la tecnología XMI.

Nuestro proyecto se ha basado tanto en la investigación como en el desarrollo. Inicialmente nos dedicamos por completo a la investigación hasta que encontramos una posible solución correcta a nuestro problema y así poder empezar la implementación. A partir de este momento, fuimos comprobando como nuestros primeros meses de investigación daban resultado, ya que las ideas y conclusiones extraídas se veían reflejadas en el correcto funcionamiento de nuestra aplicación. Una vez que nos aseguramos que nuestro desarrollo a partir de dicha investigación era el correcto, comenzamos una investigación y desarrollo en paralelo, con el objetivo de poder seguir implementando conforme íbamos descubriendo todas las soluciones de los problemas que iban apareciendo según se desarrollaba la aplicación.

Nuestra investigación podemos dividirla en dos partes, una primera parte basada en el conocimiento y familiarización con la tecnología XMI y una segunda parte basada en el funcionamiento de dicha tecnología en diferentes herramientas de modelado, como por ejemplo Poseidon o Rational Rose.

La primera parte de la investigación, comenzó por OMG (The Object Management Group), que fue quién desarrolló la tecnología XMI; Posteriormente fuimos descubriendo la relación existente entre OMG y UML, para finalmente asociar dicha información con la tecnología XML y por último XMI.

En la segunda parte, comparamos el funcionamiento de cada una de las herramientas que utilizan XMI e intercambiamos documentos comprobando cuál de ellas nos ofrece un código XMI más estandarizado. Adjuntamos un pequeño resumen del seguimiento de nuestra investigación en la sección “Investigación”.

TECNOLOGÍA XMI

XML Metadata Interchange (XMI) es un estándar de OMG que convierte MOF a XML. XMI define como deben ser usadas las etiquetas XML para representar modelos MOF serializados en XML. Los metamodelos MOF se convierten en DTD´s y los modelos en documentos XML que son consistentes con su DTD correspondiente. XMI resuelve muchos de los problemas encontrados cuando intentamos utilizar un lenguaje basado en etiquetas para representar objetos y sus asociaciones; el hecho de que XMI esté basado en XML significa que tanto los metamodelos como las instancias que describen pueden ser agrupados en el mismo documento, permitiendo a las aplicaciones entender rápidamente las instancias por medio de los metamodelos.

Desde el punto de vista de la evolución de XMI, The Object Management Group (OMG), adoptó XMI 1.0 en febrero de 1999. XMI 1.0 fue desarrollado ante la necesidad de estandarizar la representación de objetos. Se consiguió un estándar con una gran capacidad para representar información. XMI puede integrarse en aplicaciones JAVA, aplicaciones Web, aplicaciones XML y otros tipos de aplicaciones.

XMI ha evolucionado siguiendo los procesos y procedimientos abiertos de OMG; de hecho, cuatro versiones de XMI ya han sido presentadas hasta la fecha: además de XMI 1.0, apareció XMI 1.1 en febrero de 2000, que proporcionaba soporte para XML namespaces; posteriormente XMI 1.2 y XMI 2.0 en noviembre de 2001; XMI 1.2 fue una extensión de XMI 1.1 y XMI 2.0 que añade soporte para los esquemas XML e ingeniería inversa de los mismos, documentos y DTD´s. La última versión de XMI hasta la fecha, es XMI 2.1.

De todas las versiones existentes elegimos XMI 1.2, ya que es la versión utilizada por la mayoría de las herramientas de modelado en las que se ha basado nuestra investigación; y es la que encontramos en la mayoría de ejemplos y explicaciones de nuestras fuentes de información.

Las ventajas de este nuevo estándar para representar objetos son las siguientes:

XMI proporciona una representación estándar de objetos en XML, permitiendo un efectivo intercambio de datos a través de XML.

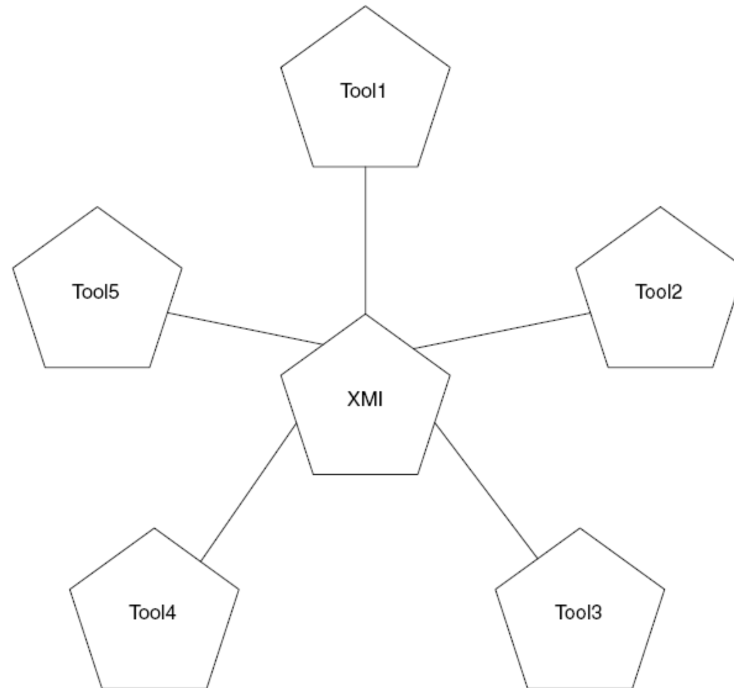
XMI especifica como crear esquemas XML desde diferentes modelos.

XMI permite crear un simple documento que fácilmente puede ir extendiéndose en función de las necesidades de nuestra aplicación.

XMI permite trabajar con XML sin necesidad de llegar a ser un experto en este lenguaje, además, permite fácilmente adaptar la representación de nuestros objetos en función de nuestras necesidades.

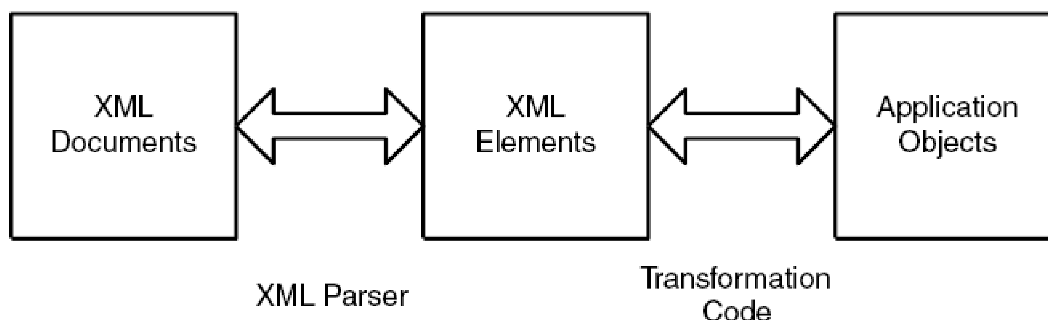
Los conceptos orientados a objetos que son relevantes para XMI son los conceptos que describen el estado de cada uno de ellos. XMI proporciona un mecanismo estándar para identificar cada objeto, por tanto no es necesario crear una nueva representación de los mismos, aunque XMI también lo permite.

XML no está orientado a objetos: define elementos y atributos, y no soporta características tales como herencia múltiple; por tanto, existen numerosos caminos para adaptar la representación de objetos en XML, lo cual es un inconveniente si queremos intercambiar documentos XML entre dos herramientas distintas. Si una herramienta obtiene la representación de objetos utilizando un determinado proceso y otra herramienta distinta utiliza otro proceso distinto para el mismo fin, es poco probable que ambas herramientas sean capaces de interpretar cada uno de los documentos generados por la otra herramienta. Sin embargo, si ambas utilizan XMI para tal propósito pueden intercambiar objetos fácilmente.

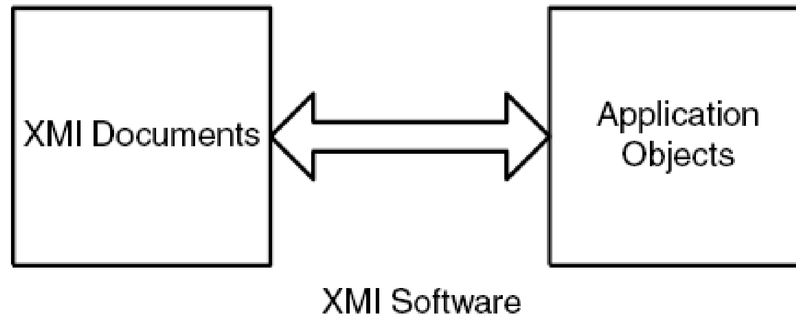


Uso de XMI para intercambiar objetos entre herramientas.

Podemos afirmar, que XMI es el puente establecido entre objetos y XML, ya que como hemos citado anteriormente, nos proporciona un estándar para representar objetos definidos mediante UML en XML: gracias a ello no es necesario definir nuestra propia representación en XML cada vez que necesitemos intercambiar objetos, y por consiguiente, no es necesario implementar nuestro propio código de transformación. Las siguientes dos imágenes muestran el camino a seguir a la hora de representar objetos utilizando directamente XML o utilizando XMI.

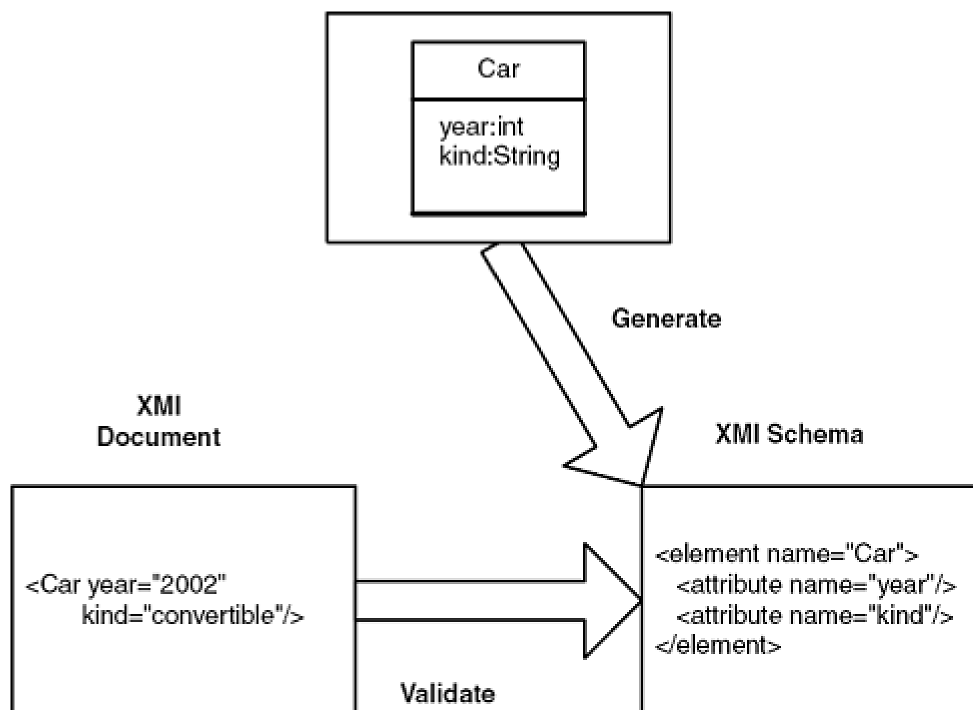


Transformación de objetos en elementos XML.



Transformación de objetos en elementos XML utilizando XMI.

Veamos un ejemplo de cómo, a partir de un sencillo diagrama UML, podemos representar toda su información utilizando XMI. Por ejemplo, a partir de un diagrama de clases formado por una clase llamada “Car” cuyos atributos son: “year” de tipo int y el atributo “kind” de tipo String, creamos el correspondiente esquema XMI que al mismo tiempo permite validar cualquier documento XMI que represente una determinada instancia de dicha clase “Car”:



Relación entre un modelo UML, un documento instancia XMI y un esquema XMI.

Tras este resumen de las características básicas de XMI, haremos una explicación de cómo crear un documento XMI que almacene toda la información necesaria. Puesto que un documento XMI es realmente un documento XML, debemos especificar en la primera instrucción la versión de XML y la codificación utilizada:

<?xml versión= “1.0” enconding = “UTF-8”?>

Posteriormente abrimos el documento, indicando que se trata de un documento XMI que utiliza una versión determinada y uno o varios espacios de nombres (namespaces):

<xmi:XMI xmi:version= “1.2”

xmlns:xmi= “*http://www.omg.org/XMI/*”

xmlns:UML = “org.omg.xmi.namespace.UML”

xmlns:UML2 = “org.omg.xmi.namespace.UML2”>

Para representar cada una de las clases y demás elementos que forman parte de nuestro diagrama, podemos utilizar elementos XML, cada uno de ellos con una etiqueta determinada que vendrá especificada por el namespace utilizado en cada caso, por ejemplo, para las clases:

<UML:Class xmi.id = “id1” name = “Clase_1”>

Para los atributos:

<UML:Attribute nam= “color” type= “id_tipo_String” xmi.id= “id_2”/>

En el caso de XMI 1.2 nuestro esquema XMI correspondiente al ejemplo anterior es el siguiente:

```
<UML:Class name= “Car”>
        <UML:Attribute name= “year” type= “int” />
        <UML:Attribute name= “kind” type= “String” />
</UML:Class>
```

Una de las principales características de XMI es su flexibilidad, XMI nos permite construir nuestra propia representación siempre y cuando la manera de representación elegida cumpla el estándar. Vamos a utilizar el ejemplo anterior para expresar gráficamente dicha característica de XMI: en el caso del tipo de los atributos, el

estándar de XMI 1.2 nos permite representar el tipo de los atributos como atributo XML tal y como hemos mostrado anteriormente, o bien como un elemento XML, con la consiguiente capacidad expresiva adicional. En este caso, nuestra representación quedaría de la siguiente manera:

```
<UML:Class name= “Car”>
        <UML:Attribute name= “year” >
            <UML2:TypedElement.type>
                <UML:DataType xmi.type= “int”/>
            </UML2:TypedElement.type>
        </UML:Attribute>
        <UML:Attribute name= “kind” />
            <UML2:TypedElement.type>
                <UML:DataType xmi.type= “String”/>
            </UML2:TypedElement.type>
        </UML:Attribute>
</UML:Class>
```

Ambas representaciones quedan perfectamente definidas desde el punto de vista de XMI 1.2. Siendo las etiquetas “UML:”, “UML2:” y “xmi:” los namespaces que incluímos en nuestro documento para poder utilizar los elementos especificados en los distintos metamodelos de UML y XMI respectivamente; como por ejemplo los elementos “:Class” o “:Attribute” del metamodelo de UML.

Una vez familiarizados con la tecnología XMI, comenzamos a desarrollar la parte de nuestra aplicación dedicada a exportar un diagrama de clases en un documento XMI. Para ello planteamos la siguiente solución: para poder exportar toda la información necesaria de un diagrama UML, concretamente de un diagrama de clases, necesitamos incluir en un mismo documento XMI la representación de la información del diagrama desde el punto de vista del modelo y por otro lado, la información que almacena todo lo relacionado con la parte gráfica del diagrama. La primera parte, incluye las clases,

nombre de las clases, atributos de cada una de ellas, tipos de dichos atributos, herencia, asociaciones entre clases, etc. Y la segunda parte, almacena el ancho, alto de las clases, puntos donde están situadas, tamaño de las letras, colores, y demás información.

INVESTIGACION

- ***Primera Parte***

Nuestra investigación de la tecnología XMI comenzó por el grupo OMG, (The Object Management Group), el cual ha definido el estándar MDA (Model Driven Architecture), que tiene como objetivo proporcionar un entorno para el desarrollo capaz de evolucionar con un esfuerzo mínimo. Para ello, este estándar se apoya en múltiples tecnologías, algunas creadas junto con MDA. Las principales tecnologías con las que nos encontramos son el tan conocido lenguaje de modelado UML, un framework para la definición de metamodelos, MOF, y un formato de intercambio de modelos basado en XML y denominado XMI.

El estándar Model Driven Architecture (MDA) es el nombre que la OMG ha dado a su nueva arquitectura. MDA, se basa en el lenguaje de modelado unificado UML, el lenguaje de modelado de la OMG. De hecho, la nueva arquitectura de la OMG surge en un nivel de abstracción superior en un esfuerzo por encontrar un mecanismo universal para integrar aplicaciones. La base del estándar MDA de la OMG es UML y Meta Object Facility, que hacen posible modelar no sólo UML sino también otros metamodelos,

Al mismo tiempo, OMG ha estandarizado el XML Metadata Interchange (XMI), un conjunto de "tags" basados en el lenguaje XML. XMI puede ser usado para enviar modelos UML entre herramientas, bases de datos y aplicaciones. Usando XMI, una herramienta visual de modelado puede enviar diagramas UML a otra herramienta de diseño, a otra aplicación o a una base de datos.

UML es el lenguaje gráfico para el modelado de sistemas y nos permite modelar la arquitectura, los objetos, las interacciones entre objetos, datos y aspectos del ciclo de vida de una aplicación, así como otros aspectos más relacionados con el diseño de componentes incluyendo su construcción y despliegue. Además pueden ser intercambiados entre herramientas utilizando XMI.

El lenguaje UML está formalmente definido por un metamodelo, que está a su vez definido de forma recursiva en UML. Este mecanismo circular permite definir UML en un número reducido de elementos.

La OMG ha creado el estándar MOF, Meta Object Facility que extiende UML para que este sea aplicado en el modelado de diferentes sistemas de información. El estándar MOF define diversos metamodelos, esencialmente abstrayendo la forma y la estructura que describe los metamodelos. MOF es un ejemplo de un modelo del metamodelo orientado a objetos. Define los elementos esenciales, sintaxis y estructuras de metamodelos que son utilizados para construir modelos orientados a objetos de sistemas. Además, proporciona un modelo común para los metamodelos CWM y UML.

Muchas de las herramientas CASE como Rational Rose, Together, Omondo, etc.., soportan XMI y el estándar para importar y exportar el formato XMI no solo puede ser usado como un formato de intercambio UML, sino que puede ser utilizado para cualquier formato descrito por medio de un metamodelo MOF. De hecho, la información del modelo podría ser intercambiada con otra herramienta compatible con MOF por medio de XMI.

Como conclusión de esta primera parte de la investigación, deducimos que UML, MOF y XMI son tres tecnologías clave para el desarrollo de software bajo el enfoque de MDA. Usadas de forma conjunta nos proporcionan grandes ventajas que hacen que nuestros modelados sean más claros y fáciles de mantener. Realmente, lo que definen estas tecnologías es una forma estándar de almacenar e intercambiar modelos, bien sean de negocio o de diseño. Esto permite a los constructores de herramientas CASE establecer un lenguaje común que se transformará en grandes beneficios para el desarrollador.

Actualmente la mayoría de las herramientas CASE ya implementan estas tecnologías por lo que solo tenemos que hacer uso de ellas para obtener sus beneficios, y poder así crear modelos, almacenarlos e intercambiarlos.

- ***SEGUNDA PARTE***

La segunda parte de nuestra investigación, consistió en estudiar la utilidad de XMI en distintas herramientas de modelado. Según íbamos descubriendo nuevas herramientas que hacen uso de esta tecnología, estudiábamos su comportamiento e intercambiábamos proyectos exportados en formato xmi con el objetivo de encontrar la representación más estandarizada, es decir, nuestro criterio se basa en intentar que el máximo número de herramientas interpreten correctamente una determinada representación.

A su vez, podemos dividir esta segunda parte de nuestra investigación en dos estudios: inicialmente, comenzamos con la investigación de todas las posibles representaciones para los diagramas de clases en cada una de las herramientas de modelado; y posteriormente, la investigación de las posibles representaciones para los diagramas de objetos; esta última, finalmente, no implementada.

Junto a estas dos investigaciones, que fueron llevadas a cabo de manera secuencial, llevamos en paralelo el estudio de la representación en XMI de la parte gráfica; ya que no todas las herramientas que exportan el modelo de un diagrama de clases o de un diagrama de objetos, exporta además su correspondiente información gráfica, e incluso en algunos casos, ésta representación no es estándar, sino que es una representación propia de la herramienta. Estos casos, no son interesentes para nuestra investigación, ya que recordemos que nuestro fichero .xmi que exporte nuestros diagramas, debe contener tanto el modelo con la información gráfica de nuestro diagrama correspondiente.

  - ***CLASES***

    La mayoría de las herramientas que utilizan XMI, te permiten exportar diagramas de clases; pero no todas ellas almacenan la información gráfica. A continuación, haremos un pequeño resumen de algunas de las herramientas que han formado parte de nuestra investigación.

Nuestra investigación comenzó con MagicDraw.UML, un programa que utiliza la versión 2.1 de XMI para almacenar la parte del modelo, y una representación interna, propia de MagicDraw.UML, para representar la información gráfica; de tal forma, que cualquier otra herramienta que sea capaza de interpretar XMI 2.1, pueda importar la información del modelo pero no la información gráfica del diagrama correspondiente.

Poseidon For UML fue nuestra segunda herramienta en nuestra investigación, cabe destacar que Poseidon es la herramienta que finalmente consigue interpretar todos nuestros diagramas exportados en xmi, asi como nuestra aplicación es capaz de interpretar todos los diagramas exportados a partir de Poseidon. En este caso, la versión utilizada es 1.2, y tanto la representación de la información necesaria del modelo como de la información gráfica están especificadas en el estándar de XMI. Por tanto, ya tenemos una primera herramienta con la que poder intercambiar archivos .xmi procedentes de otras herramientas y así comprobar las diferentes representaciones de cada uno de los elementos que forman parte del diagrama de clases.

El primer intercambio de archivos tuvo lugar entre Rational Rose y Poseidon, Rational Rose utiliza la versión 1.1, pero apenas existen diferencias entre la versión 1.1 y 1.2 de XMI. Lo más destacado en este punto, fue el estudio de la representación de los distintos tipos de los atributos de una clase; nuestra representación inicial, estaba basada en la representación que Poseidon utiliza, sin embargo al añadir Rational Rose a nuestra investigación, nos dimos cuenta que Poseidon interpreta perfectamente la representación que Rational utiliza para los tipos, y sin embargo, Rational no reconocía los tipos tal y como los representa Poseidon, ya que esta ultima representación formaba parte del estándar 1.2 y Rational interpreta 1.1, sin embargo, la representación que utiliza Rational también es una posible representación especificada en el estándar 1.2 y por tanto, consideramos la posibilidad de representar los tipos tal y como especifica Rational y ambas versiones de XMI.

Las siguientes dos imágenes muestran la diferencia en dicha representación:

```
<UML:Attribute xmi.id = 'S.001.1812.27.3'
  name = 'marca' visibility = 'private' isSpecification = 'false'
  ownerScope = 'instance'
  changeability = 'changeable' targetScope = 'instance'
  type = 'G.4'>
      .
      .
      .
<UML:DataType xmi.id = 'G.4'
  name = 'String' visibility = 'public' isSpecification = 'false'
  isRoot = 'false' isLeaf = 'false' isAbstract = 'false' />
```

Representación Tipos – Rational Rose

```
<UML:Attribute xmi.id = 'I126b7f29m10f78cdf4a7mm7ca7' name = 'atributo_1'
  visibility = 'private' isSpecification = 'false' ownerScope = 'instance'
  changeability = 'changeable'>
  <UML2:TypedElement.type>
    <UML:DataType xmi.idref = 'I126b7f29m10f78cdf4a7mm7cab'/>
  </UML2:TypedElement.type>
</UML:Attribute>
      .
      .
      .
<UML:Namespace.ownedElement>
  <UML:DataType xmi.id = 'I126b7f29m10f78cdf4a7mm7cab' name = 'int' isSpecification = 'false'
    isRoot = 'false' isLeaf = 'false' isAbstract = 'false'/>
  <UML:DataType xmi.id = 'I126b7f29m10f78cdf4a7mm7ca8' name = 'void' isSpecification = 'false'
    isRoot = 'false' isLeaf = 'false' isAbstract = 'false'/>
</UML:Namespace.ownedElement>
```

Representación Tipos – Poseidon

Otra herramienta de modelado que nos ha servido de gran ayuda es Enterprise Architect, herramienta que exporta e importa diagramas en cualquier versión de XMI, con representación estándar o representación propia de la aplicación, todo ello en función de la decisión del usuario. Por tanto, esta herramienta nos ha servido de gran ayuda en cuanto al estudio de intercambio de proyectos desde dicha herramienta a Poseidon y a nuestra aplicación y viceversa.

Además de estas tres herramientas que nos han servido de gran utilidad, cabe destacar otras como: VisualStudio, VisualUML, Metamil, D.OM, Objecteering,… Estas aplicaciones no nos han servido de gran ayuda en nuestra investigación debido a que, o bien no utilizan el estándar de XMI para representar la información gráfica o tan sólo exportan proyectos en xmi pero no los importan, lo cual dificulta el

hecho de poder comprobar su interpretación de nuestros diagramas almacenados en xmi.

Una vez completado el proceso de investigación para los diagramas de clases, y puesta en marcha su correspondiente implementación, abrimos la investigación de los diagramas de objetos que detallaremos a continuación.

- ***OBJETOS***

Para los diagramas de objetos, tuvimos que buscar nuestras herramientas de modelado que trabajasen con diagrama de objetos, ya que no todas las herramientas de modelado soportan este tipo de diagramas; por ejemplo, Poseidon es una de las herramientas que no soportan diagrama de objetos. Por tanto, las herramientas que nos han servido de utilidad en esta segunda parte de la investigación han sido: Enterprise Architect y Metamil, ya que al igual que nos ocurría con los diagramas de clases, el resto de herramientas encontradas que trabajan con diagrama de objetos, no representan la información necesaria respetando el estándar o directamente no representan la información gráfica.

Con Enterprise Architect, las dos posibilidades de representación que más se acercan a nuestras necesidades, son: la representación estándar sin incluir "Tagged Values" y la representación utilizando "Tagged Values". Si consideramos el siguiente diagrama de objetos:

Diagrama de objetos

Los resultados obtenidos al exportar dicho diagrama en xmi, empleando para ello la primera opción, son los siguientes:


```
<XMI.metamodel xmi.name="UML" xmi.version="1.4"/>
</XMI.header>
<XMI.content>
    <UML:Model name="Model" xmi.id="MX_EAID_F134AD53_73C3_43ae_BE82_B1DD2F5660A3" isRoot="true" isLeaf="false" isAbstract="false" isSpecification="f
        <UML:Namespace.ownedElement>
            <UML:Package name="objetos" xmi.id="EAPK_C6CF2E97_1B60_4df6_888D_55E7C50ECF88" isRoot="true" isLeaf="false" isAbstract="false" visibilit
                <UML:Namespace.ownedElement>
                    <UML:Collaboration xmi.id="EAID_C6CF2E97_1B60_4df6_888D_55E7C50ECF88_Collaboration" name="Collaborations">
                        <UML:Namespace.ownedElement>
                            <UML:ClassifierRole name="ana" xmi.id="EAID_245376CC_448B_4a5a_B658_7ADED392D921" visibility="public" isSpecification="f
                        </UML:Namespace.ownedElement>
                        <UML:Collaboration.interaction/>
                    </UML:Collaboration>
                    <UML:Class name="Persona" xmi.id="EAID_6CB104B1_41C0_494a_B8D7_C6A141734491" visibility="public" isSpecification="false" isRoot=
                        <UML:Classifier.feature>
                            <UML:Attribute name="edad" xmi.id="EAID_ECACFFD0_5569_4994_997F_B30A12A37298" visibility="private" ownerScope="instance"
                                <UML:StructuralFeature.type>
                                    <UML:DataType xmi.idref="eaxmiid0"/>
                                </UML:StructuralFeature.type>
                                <UML:StructuralFeature.multiplicity>
                                    <UML:Multiplicity xmi.id="EAID_19078582_D216_4312_B849_51A94880CCC3">
                                        <UML:Multiplicity.range>
                                            <UML:MultiplicityRange xmi.id="EAID_5D3E9083_F571_429e_B40F_29E32E757B83" lower="1" upper="1"/>
                                        </UML:Multiplicity.range>
                                    </UML:Multiplicity>
                                </UML:StructuralFeature.multiplicity>
                                <UML:Attribute.initialValue>
                                    <UML:Expression xmi.id="EAID_25390DBB_B9D1_4cc7_BEA8_1802E9E2D50B"/>
                                </UML:Attribute.initialValue>
                                <UML:ModelElement.taggedValue/>
                            </UML:Attribute>
                        </UML:Classifier.feature>
                    </UML:Class>
                </UML:Namespace.ownedElement>
            </UML:Package>
            <UML:DataType xmi.id="eaxmiid0" name="int" visibility="private" isRoot="false" isLeaf="false" isAbstract="false"/>
        </UML:Namespace.ownedElement>
    </UML:Model>
    <UML:Diagram name="clases" xmi.id="EAID_C9C565BB_81E8_4022_8BA3_8E163B92E0BD" diagramType="ClassDiagram" owner="EAPK_C6CF2E97_1B60_4df6_888D_55E
        <UML:ModelElement.taggedValue/>
        <UML:Diagram.element>
```


Representación Objetos – Enterprise Architect (Sin Tagged Values)

Como podemos observar en la sección recuadrada, el objeto de nombre "ana" no está identificado como instancia de la clase "Persona" que aparece a continuación, ni sus correspondientes atributos, ni por supuesto, los valores correspondientes que estos contienen. Es por tanto, una representación de un diagrama de colaboración, tal y como indica la propia etiqueta que aparece en el recuadro.

La otra posibilidad, exportar el diagrama utilizando en su representación "Tagged Values", nos ofrece el siguiente resultado:

```
<UML:TaggedValue.dataValue>Public</UML:TaggedValue.dataValue>
<UML:TaggedValue.type>
  <UML:TagDefinition xmi.idref="EAID_E5EDE1AC_1E5C_406e_95AE_1A1865B9FD49"/>
</UML:TaggedValue.type>
</UML:TaggedValue>
</UML:ModelElement.taggedValue>
<UML:Namespace.ownedElement>
<UML:Collaboration xmi.id="EAID_C6CF2E97_1B60_4df6_888D_55E7C50ECF88_Collaboration" name="Collaborations">
  <UML:Namespace.ownedElement>
    <UML:ClassifierRole name="ana" xmi.id="EAID_245376CC_448B_4a5a_B658_7ADED392D921" visibility="public" isSpecification="
      <UML:ModelElement.taggedValue>
        <UML:TaggedValue xmi.id="EAID_0705D860_27E5_4459_A6A9_169B26AD6210" isSpecification="false">
          <UML:TaggedValue.dataValue>1.0</UML:TaggedValue.dataValue>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref="EAID_CECD46F5_B2B6_455c_92CF_C87071953F9F"/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
        <UML:TaggedValue xmi.id="EAID_A94F4C4D_BE41_4042_AC10_4487B9E7EE88" isSpecification="false">
          <UML:TaggedValue.dataValue>@VAR;Variable=edad;Value=22;Op==;@ENDVAR;</UML:TaggedValue.dataValue>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref="EAID_1F0D9F6F_F3F3_4cd0_A665_6FAC5FC2A7EF"/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
        <UML:TaggedValue xmi.id="EAID_F2CDCECC_5821_4721_93E2_76CD35E54C83" isSpecification="false">
          <UML:TaggedValue.dataValue>1</UML:TaggedValue.dataValue>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref="EAID_D83E41E4_9708_4F34_815B_EF136EE6F635"/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
        <UML:TaggedValue xmi.id="EAID_8DD3D81A_7C79_45a9_8DE9_D3B0B4173F69" isSpecification="false">
          <UML:TaggedValue.dataValue>2007-03-05 23:17:09</UML:TaggedValue.dataValue>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref="EAID_7285EA04_351F_4560_91E2_4F261E233EBE"/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
        <UML:TaggedValue xmi.id="EAID_9317AD29_0183_4d24_8A23_59D12CA06530" isSpecification="false">
          <UML:TaggedValue.dataValue>2007-03-05 23:17:17</UML:TaggedValue.dataValue>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref="EAID_8F02CED7_C28B_4e02_9B0D_5B97841A4CCE"/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
```

Representación Objetos – Enterprise Architect (Con Tagged Values)

En este caso, el objeto “ana” identificado con la etiqueta “UM:ClassifierRole”, si especifica sus correspondientes atributos con sus respectivos valores, que en este caso, es tan sólo el atributo de tipo entero “edad” con su valor “22” como podemos observar en la segunda sección recuadrada. Sin embargo, no especifica tampoco, que dicho objeto es instancia de “Persona”. Por tanto, volvemos a no tener constancia de que realmente este especificando un diagrama de objetos, sino más bien un diagrama de colaboración.

En el caso de nuestra segunda herramienta “Metamil” que tan sólo utiliza la versión 1.2 con un solo tipo de representación, obtenemos el siguiente resultado:

```
<XMI.content>
  <UML:Model name="new_model" xmi.id="mm:f03aa4ae-cb67-11db-8b52-8c6911324037">
    <UML:ModelElement.taggedValue>
      <UML:TaggedValue tag="mpacksdir" value="${BASEMPACKSDIR}"/>
      <UML:TaggedValue tag="sourcedir" value="${BASESOURCEDIR}"/>
      <UML:TaggedValue tag="documsdir" value="${BASEDOCUMSDIR}"/>
      <UML:TaggedValue tag="scriptdir" value="${BASESCRIPTDIR}"/>
      <UML:TaggedValue tag="exportdir" value="${BASEEXPORTDIR}"/>
      <UML:TaggedValue tag="imprules"/>
      <UML:TaggedValue tag="cf_codelang" value="1"/>
      <UML:TaggedValue tag="cf_genemark" value="1"/>
      <UML:TaggedValue tag="cf_codeconf" value="suppcimp=true;hppext=.h;cppext=.cpp;visualc=false;attrpfix=m_;getpfix=get_;setpfix=set_;listcpp=list&lt
      <UML:TaggedValue tag="cf_macros"/>
      <UML:TaggedValue tag="cf_integralcode" value="0"/>
    </UML:ModelElement.taggedValue>
    <UML:Namespace.ownedElement>
      <UML:Class xmi.id="mm:f92b67c5-cb67-11db-8b52-8c6911324037" name="Persona" visibility="public" isRoot="false" isLeaf="false" isAbstract="false" i
        <UML:Classifier.feature>
          <UML:Attribute xmi.id="mm:00527c53-cb68-11db-8b52-8c6911324037" name="edad" visibility="public" ownerScope="instance" changeable="none" targe
            <UML:ModelElement.taggedValue>
              <UML:TaggedValue tag="__M_type" value="int"/>
            </UML:ModelElement.taggedValue>
          </UML:Attribute>
        </UML:Classifier.feature>
      </UML:Class>
      <UML:ClassifierRole xmi.id="mm:0b0d1385-cb68-11db-8b52-8c6911324037" name="ana" visibility="public" base="mm:f92b67c5-cb67-11db-8b52-8c6911324037
        <UML:ModelElement.taggedValue>
          <UML:TaggedValue tag="edad" value="22"/>
        </UML:ModelElement.taggedValue>
      </UML:ClassifierRole>
    </UML:Namespace.ownedElement>
  </UML:Model>
</XMI.content>
```

Representación Objetos – Metamil

En este caso, podemos observar como especifica tanto la clase a la que pertenece el objeto mediante el atributo “base”, y los atributos con sus correspondientes valores en el segundo recuadro, “edad” y “22”. Por primera vez, descubrimos la existencia de este nuevo atributo “base” en XMI, y al comprobar sus posibles utilidades en el estándar 1.2, observamos como está claramente especificado para representar la instancia de un determinado objetos a su correspondiente clase. El problema está, en que Metamil no exporta la información gráfica del diagrama, es decir, aunque supuestamente tendríamos solucionada la representación del modelo, no tendríamos ninguna posible representación para nuestra información gráfica.

La decisión que tomamos fue no implementar los diagramas de objetos, y empezar a trabajar sobre la importación de diagramas, es decir, el proceso inverso del proceso hasta ahora desarrollado. Para está última parte de nuestro proyecto, prácticamente no necesitamos investigación, tan sólo descubrir el parseador que podríamos utilizar para leer fácilmente un archivo .xmi, dicho parseador es JDOM que interpreta toda la información almacenada en forma de árbol y nosotros debemos ir accediendo a cada uno de sus nodos para obtener nombres, atributos, valores, … de cada uno de los elementos de nuestro diagrama.

ESPECIFICACIÓN DEL PROCESO

El proceso consiste en añadir a la herramienta preexistente MOVA una funcionalidad adicional: Exportar e importar diagramas UML mediante el estándar XMI 1.2.

- **Problemas:**

  MOVA podía realizar el guardado y carga de diagramas UML almacenándolos en XML, pero sin el uso de ningún estándar, lo cual impedía la comunicación de la herramienta con otras herramientas de modelado UML. En todo momento MOVA mantiene una representación interna de los diagramas que se muestran en ese instante en la aplicación. Cuando el usuario desea guardarlos para su posterior uso, MOVA dispone de los métodos necesarios para convertir esa representación interna del diagrama UML en un documento en formato XML, el cual solo ella es capaz de comprender, siendo MOVA, por lo tanto, la única herramienta de modelado UML que puede realizar su posterior lectura, para la cual también dispone de los métodos adecuados.

- **Solución:**

  Para permitir a MOVA exportar e importar diagramas UML en formato XMI 1.2, ha sido necesario crear nuevos métodos de guardado, que pasen de la representación interna del diagrama UML a un documento en XMI, y de lectura, que pasen de un documento XMI a la representación interna del diagrama UML. No obstante, los métodos de guardado y de lectura que se han añadido no son completamente análogos a los preexistentes, ya que no pasan directamente de la representación interna del diagrama al documento XMI y viceversa, sino que pasan por una representación intermedia llamada "Almacén". Esta representación intermedia antes carecía de sentido, ya que el paso de la representación interna de MOVA al XML era completamente trivial, porque, al no ser necesario respetar ningún estándar, podía almacenarse el diagrama en el

documento XML de la forma más cómoda. No obstante, XMI es un estándar en el cual están predefinidas las formas de guardar la información, lo cual hace la labor de guardado y carga de archivos más compleja, siendo recomendable el uso de herramientas auxiliares tanto para el guardado (DOM), como para la lectura (JDOM). Dichas herramientas, como se explica en los apartados siguientes, manejan árboles para llevar a cabo su labor, árboles que sería necesario construir a partir de la representación interna de MOVA, en el caso del guardado, y árboles que sería necesario leer y transformar directamente en la representación interna de MOVA, en el caso de la lectura. Esta labor sería tremendamente compleja si se hace de forma directa, es por ello por lo que se emplea el denominado Almacén, el cual almacena la misma información pero colocada de forma que sea fácilmente accesible para construir el árbol, en el caso del guardado, y la representación interna de MOVA, en el caso de la lectura.

RIESGOS

A lo largo del proyecto han existido numerosos riesgos que, sin embargo, han sido superados. No obstante, eventualmente, algunos de ellos podrían reaparecer, siendo necesario en ese caso aplicar una nueva solución. Es por ello que consideramos oportuno reseñar los más significativos:

- Licencias DOM y JDOM: Ambas librerías se encuentran, a fecha de hoy, clasificadas como código libre, la librería DOM incluida en el paquete de librerías XERCES y la librería JDOM bajo la librería JDOM. A pesar de ser código libre puede que eventualmente adquieran copyright, siendo necesario pagar por ellas o buscar otras librerías alternativas.
- Carencias del XMI 1.2: No es posible en esta versión almacenar diagramas de objetos. Esto puede indicar que, en el caso de querer añadir una nueva funcionalidad a la herramienta de cara a la representación de diagramas, puede que no sea posible su almacenamiento en XMI,no obstante, puede que futuras versiones del mismo si lo permitan.
- Aceptación y seguimiento del estándar de representación de la parte gráfica por el grueso de las herramientas de modelado UML: Existen herramientas que respetan el estándar XMI 1.2 pero solo en la parte del modelo, almacenando la parte gráfica con una representación interna propia. Parte de las herramientas que si respetan el estándar en la parte gráfica, la inmensa mayoría en este momento, puede que dejen de hacerlo en un futuro, momento en el cual puede que no tenga sentido seguir actualizando el estándar de la parte gráfica con las novedades que vayan surgiendo, ya que sería únicamente para consumo interno de nuestra propia herramienta de modelado UML.
- Comunicación con otras herramientas: Pese a que XMI es un estándar, no existe una representación única para la misma información, es decir, que es posible que dos códigos XMI estructuralmente distintos representen el mismo diagrama UML. Esto ha sido una fuente de incongruencias con otras herramientas, las cuales, respetando el estándar, guardaban la información en

XMI 1.2 de forma diferente a nosotros, siéndonos en principio imposible leer los archivos que estas generaban. Para resolver ese problema optamos por modificar nuestros métodos de lectura para poder admitir todas las formas de representación de la información, de forma que, independientemente de la forma en que este almacenado, y siempre y cuando se respete el estándar, MOVA va a leer de forma univoca el diagrama UML almacenado en el documento XMI 1.2.

- Namespaces: Los namespaces, que acompañan a las etiquetas a lo largo del documento XMI, han de ser definidos al principio de este, asignándoles las direcciones donde se encuentran los metamodelos correspondientes. Estos metamodelos pueden cambiar conforme vayan saliendo nuevas versiones de los mismos, siendo necesario en ese caso modificar la dirección del namespace para que pase a apuntar al nuevo metamodelo.

LIBRERIAS Y PROTOTIPOS

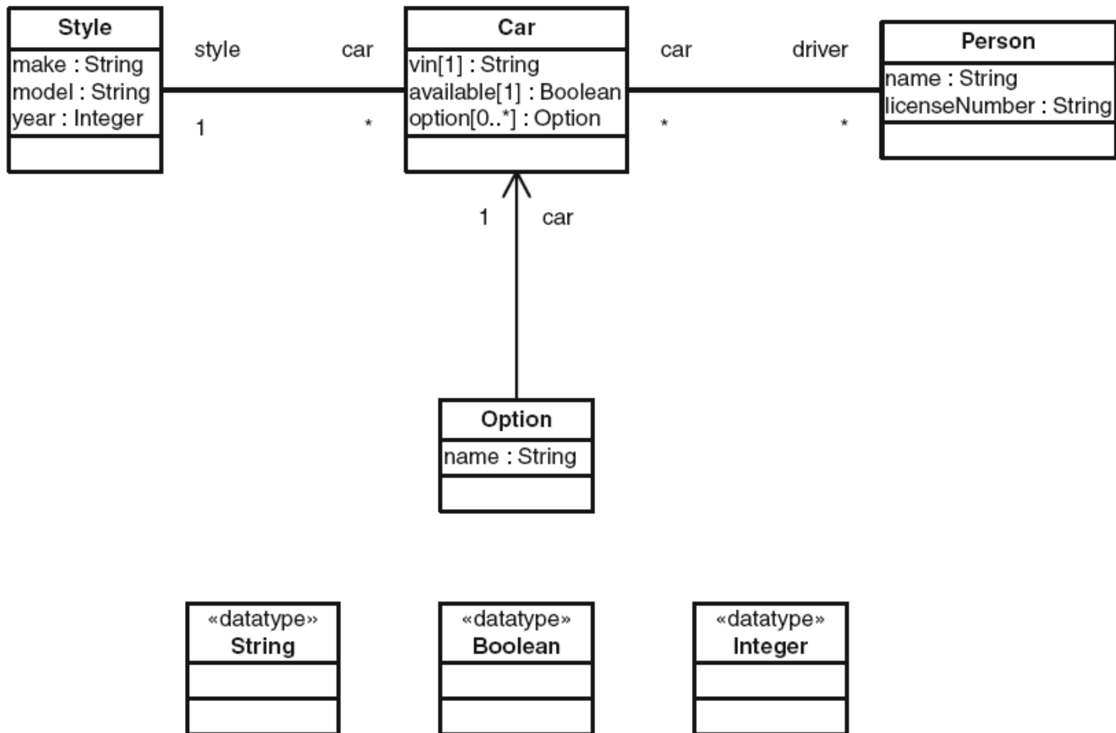
En el desarrollo del proyecto ha sido necesario, como hemos indicado anteriormente, el uso de distintas librerías, las cuales se han erigido como herramientas auxiliares de gran valor que han simplificado enormemente, y en algunos caso hecho posible, la implementación de la herramienta. No obstante, el uso de los recursos y métodos que estas librerías han proporcionado ha requerido de un periodo de aprendizaje y adaptación a las mismas, durante el cual se han ido desarrollando diversos prototipos que iban incorporando nuevas funcionalidades de forma progresiva. Las librerías a las que nos referimos son:

1. **DOM:** El ***Document Object Model*** es una forma de representar los elementos de un documento estructurado (tal como una página web HTML o un documento XML) como objetos que tienen sus propios métodos y propiedades. El responsable del DOM es el *World Wide Web Consortium* (W3C).En efecto, el DOM es una API para acceder, añadir y cambiar dinámicamente contenido estructurado en documentos con lenguajes como ECMAScript (Javascript).

   En nuestro proyecto ha resultado determinante, ya que ha proporcionado todas las estructuras de datos y los métodos necesarios para poder, a partir del Almacén (Previamente construido tomando como origen la representación interna de MOVA), construir una estructura arbórea que contuviese toda la información de diagrama, tanto del modelo como de la parte gráfica, necesaria para su posterior recuperación por parte de nuestra herramienta o de cualquier otra que implementase importación en XMI 1.2

   A partir de esta estructura arbórea, y mediante el uso de métodos proporcionados por la propia librería se genera automáticamente, en un fichero indicado previamente, el documento parseado a XMI.

   Uno de los prototipos generados para poder probar esta librería y que resulta de gran utilidad didáctica es el siguiente:



El diagrama modela la estructura de una empresa de alquiler de coches, donde una persona puede alquilar coches, cada uno de los cuales con un estilo (que indica la marca, modelo y el año del vehículo) y una o más opciones.

A continuación vamos a mostrar el código necesario para construir un documento XMI 1.2 que almacene un diagrama de objetos instancia del anterior diagrama de clases :

```
// DOMWrite.java
import org.apache.xerces.dom.*;
import org.apache.xml.serialize.*;
import org.w3c.dom.*;
import java.util.*;
import java.io.*;
// This program writes the Car, Option, Style, and Person objects
// described in the Car Rental Agency Application section to an XMI
// 2.0 document using the steps described in the Using DOM section
// in this chapter.
public class DOMWrite {
public static void main(String[] args) throws Exception {
```

```
// Overview step 1.
Document doc = new DocumentImpl();
// Overview step 2. Create the XMI Element node.
Element xmi = doc.createElement("xmi:XMI");
xmi.setAttribute("xmi:version", "2.0");
xmi.setAttribute("xmlns:xmi", "http://www.omg.org/XMI");
doc.appendChild(xmi);
// Overview step 3. Create the XMI Documentation element, specifying
// the program that produced the file (the exporter) and its version.
Element documentation = doc.createElement("xmi:Documentation");
Element exporter = doc.createElement("exporter");
Text exporterText = doc.createTextNode("XMI DOM Serialize Example");
exporter.appendChild(exporterText);
Element version = doc.createElement("exporterVersion");
Text versionText = doc.createTextNode("0.5");
version.appendChild(versionText);
documentation.appendChild(exporter);
documentation.appendChild(version);
xmi.appendChild(documentation);
// Overview step 4. Write the objects.
// Object step 1 for the Car object.
Element car = doc.createElement("Car");
// Object step 2.
car.setAttribute("xmi:id", "_1");
// Object step 3. Since available and vin are data attributes
// with one value, put the values in XML attributes.
car.setAttribute("available", "false");
car.setAttribute("vin", "v1");
// Object step 4. Because style and driver are references,
// put the xmi:id of the referenced object in an XML attribute.
car.setAttribute("style", "_2");
car.setAttribute("driver", "_3");
// Object step 6. Since option is an object attribute
// of the Car object indicating a contained Option object, make
```

```
// an Element node for the contained object as a descendant of the
// car Element node.
Element option = doc.createElement("option");
// Object step 2 for the Option object.
option.setAttribute("xmi:id", "_1.1");
// Object step 3.
option.setAttribute("name", "air conditioning");
// Object step 4.
option.setAttribute("car", "_1");
// Last part of Object step 6.
car.appendChild(option);
// Object step 7 for the Car object.
xmi.appendChild(car);
// Object step 1 for the Style object.
Element style = doc.createElement("Style");
// Object step 2.
style.setAttribute("xmi:id", "_2");
// Object step 3.
style.setAttribute("make", "Jalopy");
style.setAttribute("model", "Deluxe");
style.setAttribute("year", "2002");
// Object step 4.
style.setAttribute("car", "_1");
// Object step 7.
xmi.appendChild(style);
// Object steps 1-4 and 7 for the Person object.
Element person = doc.createElement("Person");
person.setAttribute("xmi:id", "_3");
person.setAttribute("name", "Anita Karr");
person.setAttribute("licenseNumber", "ln1");
person.setAttribute("car", "_1");
xmi.appendChild(person);
// Create the XMI document from the parse tree. See the XML4J
// parser documentation for more details.
```

```
OutputFormat format = new OutputFormat(doc, "UTF-8", true);
FileWriter file = new FileWriter("DOMWrite.xmi");
XMLSerializer serial = new XMLSerializer(file, format);
serial.asDOMSerializer();
serial.serialize(doc.getDocumentElement());
file.close();
}
}
```

El programa produce como salida el siguiente documento XMI 1.2:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI">
	<xmi:Documentation>
		<exporter>XMI DOM Serialize Example</exporter>
		<exporterVersion>0.5</exporterVersion>
	</xmi:Documentation>
	<Car available="false" driver="_3" style="_2" vin="v1" xmi:id="_1">
		<option car="_1" name="air conditioning" xmi:id="_1.1"/>
	</Car>
	<Style car="_1" make="Jalopy" model="Deluxe" xmi:id="_2"
		year="2002"/>
	<Person car="_1" licenseNumber="ln1" name="Anita Karr"
		xmi:id="_3"/>
</xmi:XMI>
```

2. **JDOM:** Es una librería de código fuente para manipulaciones de datos XML optimizados para Java. A pesar de su similitud con DOM del consorcio World Wide Web (W3C), es una alternativa como documento para modelado de objetos que no está incluido en DOM. La principal diferencia es que mientras que DOM fue creado para ser un lenguaje neutral e inicialmente usado para manipulación de páginas HTML con JavaScript, JDOM se creó específicamente para usarse con Java y por lo tanto beneficiarse de las características de Java, incluyendo sobrecarga de métodos, colecciones, etc.

En nuestro proyecto, al igual que DOM, JDOM ha resultado determinante, ya que ha proporcionado todas las estructuras de datos y los métodos necesarios para poder, a partir de un documento en XMI 1.2, construir una estructura arbórea que contuviese toda la información del diagrama UML almacenada en el documento, tanto del modelo como de la parte gráfica, y, a partir de esta estructura arbórea, y mediante los métodos que el mismo nos proporciona, acceder a toda la información necesaria para construir el Almacén. A continuación, y tomando como origen la información contenida en el Almacén, se crean las estructuras de datos que constituyen la representación interna de MOVA para ese diagrama UML concreto.

Uno de los prototipos generados para poder probar esta librería y que resulta de gran utilidad didáctica es el siguiente:

Dado el siguiente documento XML:

```
<?xml version="1.0"?>
<liga tipo="Champions League">
 <equipo>
   <club valoracion="10" ciudad="Bilbao">Athletic Club Bilbao</club>
   <presidente>Uria</presidente>
   <plantilla>
     <nombre>Julen Guerrero</nombre>
     <nombre>Joseba Etxeberria</nombre>
     <nombre>Ismael Urzaiz</nombre>
   </plantilla>
 </equipo>
 <equipo>
   <club valoracion="5" ciudad="Madrid">Real Madrid</club>
   <presidente>Mandamas</presidente>
   <plantilla>
     <nombre>Bota de oro</nombre>
     <nombre>Milloneti</nombre>
     <nombre>Canterano quemado</nombre>
```

```
    </plantilla>
    <conextranjeros/>
  </equipo>
  <arbitros>
    <nombre>No doy una</nombre>
    <nombre>Rafa</nombre>
  </arbitros>
</liga>
```

Vamos a hacer un programa que lo lea y nos muestra cierta información. Este sería el código:

```
import java.io.*;
import java.util.*;
import org.jdom.*;
import org.jdom.input.*;
import org.jdom.output.*;

public class Ejemplo {

  public static void main(String[] args) {
     try {
        SAXBuilder builder=new SAXBuilder(false);
        //usar el parser Xerces y no queremos
        //que valide el documento
        Document doc=builder.build("liga.xml");
        //construyo el arbol en memoria desde el fichero
        // que se lo pasaré por parametro.
        Element raiz=doc.getRootElement();
        //cojo el elemento raiz
        System.out.println("La liga es de tipo:"+
                raiz.getAttributeValue("tipo"));
        //todos los hijos que tengan como nombre plantilla
        List equipos=raiz.getChildren("equipo");
```

```
        System.out.println("Formada por:"+equipos.size()+" equipos");
        Iterator i = equipos.iterator();
        while (i.hasNext()){
           Element e= (Element)i.next();
           //primer hijo que tenga como nombre club
           Element club =e.getChild("club");
           List plantilla=e.getChildren("plantilla");
           System.out.println
                       (club.getText()+":"+"valoracion="+
                        club.getAttributeValue("valoracion")+","+
                        "ciudad="+club.getAttributeValue("ciudad")+","+
                        "formada por:"+plantilla.size()+"jugadores");
            if (e.getChildren("conextranjeros").size()==0)
             System.out.println("No tiene extranjeros");
            else  System.out.println("Tiene extranjeros");


        }
     }catch (Exception e){
        e.printStackTrace();
     }
  }
}
```

IMPLEMENTACIÓN

Una vez que entendimos perfectamente el estándar XMI 1.2, incluyendo el estándar de la parte gráfica [Poner aquí el nombre], y aprendimos a manejar correctamente todas las funcionalidades que nos ofrecían las bibliotecas DOM y JDOM, comenzamos a implementar, como tal, el código de esta nueva funcionalidad de MOVA.

El primer paso que dimos fue implementar la escritura de diagramas UML, desde su representación mediante la estructura interna de MOVA, a un documento en formato XMI 1.2. En un primer lugar se hizo únicamente para la parte del modelo, obviando la información de la parte grafica. La razón de esta decisión fue lógica, ya que era preferible tener bien implementada la funcionalidad para la parte del modelo, que era, en principio, más sencilla, para a continuación, incorporar la escritura de la información grafica, que era complementaria a la anterior.

Una vez que la funcionalidad de escritura en XMI se tuvo operativa, y en perfecto funcionamiento, comenzamos a implementar la funcionalidad de lectura desde un archivo en formato XMI 1.2 hasta la representación interna de MOVA.

La estructura general de esta aplicación es la siguiente

A continuación vamos a explicar como está implementada la aplicación. En primer lugar vamos a tratar el proceso de escritura y, en segundo lugar, el proceso de lectura. A su vez, en cada caso, vamos a explicar en primer lugar el proceso en el caso del modelo, obviando la información de la parte grafica, y a continuación trataremos la inclusión de esta información en el correspondiente proceso.

- **PROCESO DE ESCRITURA**

  Como hemos comentado, en primer lugar vamos a tratar el proceso de escritura, únicamente, de la información del modelo, para, a continuación, incluir en el mismo la información de la parte grafica.

  - **Escritura de la información del modelo**

    Nuestro punto de partida es el diagrama UML que la aplicación MOVA esta manejando en el momento de la escritura. Como hemos comentado, MOVA representa este diagrama mediante una estructura de datos propia, la cual resulta muy "incómoda" para poder tratarla directamente y convertirla en un documento XMI 1.2. Para simplificar el proceso, el primer paso es construir, a partir de la representación interna de MOVA, un Almacén que contenga la información necesaria, y ordenada de la forma más eficiente, de cara a la creación del documento XMI 1.2. La creación del Almacén se realiza mediante la constructora "Almacen(ClassDiagram d,String filename)". Es la propia constructora la que, mediante métodos de acceso al contenido de la representación interna de MOVA, extrae toda la información necesaria para poder crear el documento XMI 1.2. Estos métodos de acceso a la representación interna de MOVA son proveídos por la misma herramienta, y no ha sido necesaria su implementación. La constructora, conforme extrae la información de la representación interna de MOVA, va incorporándola a la propia estructura de datos del Almacen, mediante los métodos de inclusión que hemos implementado:

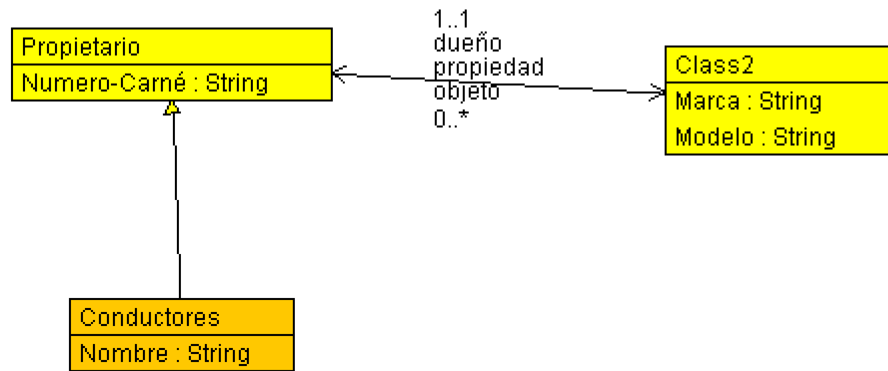
      - incluirClase( )
      - incluirAtributo( )
      - incluirClaseEnumerada( )
      - incluirAtributoEnum( )
      - incluirTipo( )
      - incluirGeneralizacion( )

- incluirAsociacion( )
- incluirClaseAsociacion( )

Una vez que el Almacen está creado se llama al método parseador( ). Este método es el encargado de construir, a partir del Almacen, y, empleando los métodos proporcionados por la librería DOM, la estructura arbórea antes mencionada. En el arbol se incluye, además de toda la información necesaria del diagrama UML, información adicional acerca de la versión XMI a la que se exporta, la versión del metamodelo UML empleada, la aplicación que lo realiza, etc. Esta estructura arbórea es la que, mediante los siguientes métodos de DOM, se transformas en el documento XMI 1.2 que contiene toda la información del diagrama UML:

```
OutputFormat format = new OutputFormat(doc, "UTF-8",true);

OutputFormat(doc, "UTF-8",true);

FileWriter file = new FileWriter(filename);

XMLSerializer serial = new XMLSerializer(file,format);

serial.asDOMSerializer();

serial.serialize(doc.getDocumentElement());

file.close();
```

A continuación vamos a poner un ejemplo de este proceso. Vamos a generar el documento XMI 1.2 que almacena la información correspondiente al siguiente diagrama:



El documento XMI 1.2 generado es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.2" xmlns:UML="org.omg.xmi.namespace.UML"
xmlns:UML2="org.omg.xmi.namespace.UML2">
    <XMI.header>
        <XMI.documentation>
            <XMI.exporter>Netbeans XMI Writer</XMI.exporter>
            <XMI.exporterVersion>1.0</XMI.exporterVersion>
            <XMI.metaModelVersion>1.4.5</XMI.metaModelVersion>
        </XMI.documentation>
    </XMI.header>
    <XMI.content>
        <UML:Model name="modeloPrueba" xmi.id="id_modeloPrueba">
            <UML:Namespace.ownedElement>
                <UML:Class name="Propietario" visibility="public"
                  xmi.id="id_clase_Propietario">
                    <UML:Classifier.feature>
                        <UML:Attribute name="Numero-Carné"
                            type="id_tipo_String" visibility="private"
                        xmi.id="id_clase_Propietario_atributo_Numero-
                        Carné"/>
                    </UML:Classifier.feature>
                </UML:Class>
                <UML:Class name="Class2" visibility="public"
                  xmi.id="id_clase_Class2">
                    <UML:Classifier.feature>
                        <UML:Attribute name="Marca"
                            type="id_tipo_String" visibility="private"
                             xmi.id="id_clase_Class2_atributo_Marca"/>
                        <UML:Attribute name="Modelo"
                            type="id_tipo_String" visibility="private"
                                          xmi.id=
                             "id_clase_Class2_atributo_Modelo"/>
                    </UML:Classifier.feature>
                </UML:Class>
                <UML:Class name="Conductores" visibility="public"
                  xmi.id="id_clase_Conductores">
                    <UML:GeneralizableElement.generalization>
                        <UML:Generalization          xmi.idref=
                  "id_Generalizacion_Conductores_Propietario"/>
                    </UML:GeneralizableElement.generalization>
                    <UML:Classifier.feature>
                        <UML:Attribute name="Nombre"
```

```
                        type="id_tipo_String" visibility="private"
                    xmi.id="id_clase_Conductores_atributo_Nombre"/>
                </UML:Classifier.feature>
            </UML:Class>
            <UML:DataType name="String" xmi.id="id_tipo_String"/>
            <UML:Association name="propiedad"    xmi.id=
              "id_ClaseAsociacion_Propietario_Class2propiedad">
                <UML:Association.connection>
                    <UML:AssociationEnd name="dueño"
                        type="id_clase_Propietario"
                        visibility="public"
xmi.id="id_ClaseAsociacion_Propietario_Class2propiedad_asociationEnd_1
">
                        <UML:AssociationEnd.multiplicity>
                            <UML:Multiplicity
xmi.id="id_ClaseAsociacion_Propietario_Class2propiedad_asociationEnd_1
_multiplicidad">
                                <UML:Multiplicity.range>
                                <UML:MultiplicityRange lower="1"
                                upper="1"
xmi.id="id_ClaseAsociacion_Propietario_Class2propiedad_asociationEnd_1
_rangoMultiplicidad"/>
                                </UML:Multiplicity.range>
                            </UML:Multiplicity>
                        </UML:AssociationEnd.multiplicity>
                        <UML:AssociationEnd.participant>
                            <UML:Class
                          xmi.idref="id_clase_Propietario"/>
                        </UML:AssociationEnd.participant>
                    </UML:AssociationEnd>
                    <UML:AssociationEnd name="objeto"
                        type="id_clase_Class2" visibility="public"
xmi.id="id_ClaseAsociacion_Propietario_Class2propiedad_asociationEnd_2
">
                        <UML:AssociationEnd.multiplicity>
                            <UML:Multiplicity
xmi.id="id_ClaseAsociacion_Propietario_Class2propiedad_asociationEnd_2
_multiplicidad">
                                <UML:Multiplicity.range>
                                <UML:MultiplicityRange lower="0"
                                upper="-1"
xmi.id="id_ClaseAsociacion_Propietario_Class2propiedad_asociationEnd_2
_rangoMultiplicidad"/>
                                </UML:Multiplicity.range>
                            </UML:Multiplicity>
                        </UML:AssociationEnd.multiplicity>
                        <UML:AssociationEnd.participant>
                            <UML:Class
                          xmi.idref="id_clase_Class2"/>
                        </UML:AssociationEnd.participant>
                    </UML:AssociationEnd>
                </UML:Association.connection>
            </UML:Association>
            <UML:Generalization
              xmi.id="id_Generalizacion_Conductores_Propietario">
                <UML:Generalization.child>
                    <UML:Class xmi.idref="id_clase_Conductores"/>
                </UML:Generalization.child>
                <UML:Generalization.parent>
                    <UML:Class xmi.idref="id_clase_Propietario"/>
                </UML:Generalization.parent>
```

```
                </UML:Generalization>
            </UML:Namespace.ownedElement>
        </UML:Model>
</XMI.content>
</XMI>
```

- **Escritura de la información gráfica**

Una vez conseguido el correcto funcionamiento de almacenar en XMI toda la información correspondiente al modelo, nos vimos en condiciones de ampliar el documento XMI con la información de la parte gráfica del diagrama.
Para ello hubo que seguir varios pasos:

1.-Puesto que la intención es la de representar la parte gráfica a través del estándar "Diagram Interchange" en el cual se basan otras herramientas comerciales como Poseidon o Racional, el primer objetivo era recopilar toda la información que era necesaria para la parte gráfica. Esa información necesaria se detalla a continuación para cada posible elemento que forme parte de un diagrama de clases sin contar otra información que todos incluyen que es la relacionada con referencias necesarias hacia otras partes del diagrama a través del atributo idref, o información sobre el color, tamaño o fuente de letra usada:

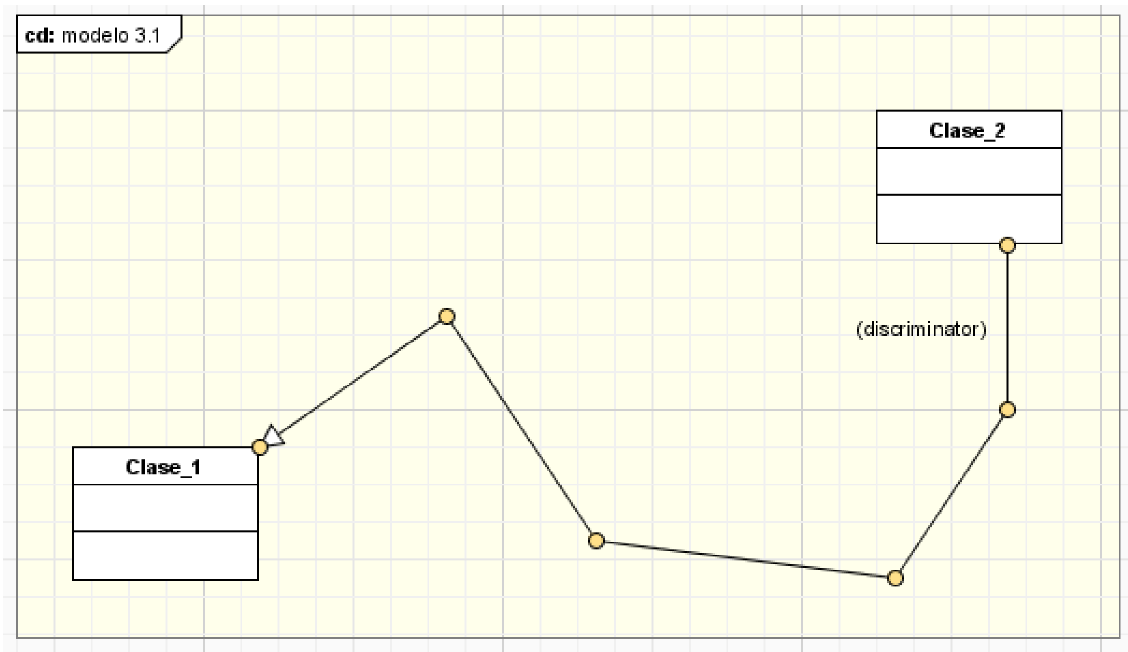
-Para las clases y clases enumeradas:

-Posición x e y del vértice superior derecho de la clase.
-Ancho y alto.
-Alto de los huecos correspondientes al nombre y a los atributos.

-Para las generalizaciones:

-Principalmente el listado de los puntos que forman la unión de la clase hijo con la clase padre, ya que en una

relación de un diagrama UML, la unión entre los dos elementos de la relación no tiene por qué ser una línea recta sino que puede ser un conjunto de segmentos todos rectos pero unidos cada uno en una dirección distinta, como muestra la siguiente ilustración.



Por eso en necesario mantener en conjunto de puntos que son los circulitos naranja de la figura, para poder representar relaciones de este tipo.

-Para las asociaciones:

-De igual forma que las generalizaciones, las asociaciones también necesitan almacenar la información sobre los puntos que forman la unión de los elementos asociados.

-Además las asociaciones necesitan información sobre la posición del nombre de la asociación en el diagrama.

-La posición en el diagrama de los roles que forman la asociación.

-Y también la posición de las multiplicidades correspondientes a esos roles.

-Para las clases de asociación:

-Se necesita la información gráfica referente a la clase que forma la clase de asociación.

-Además se necesita la información necesaria sobre la asociación que a su vez asocia la clase con la asociación.

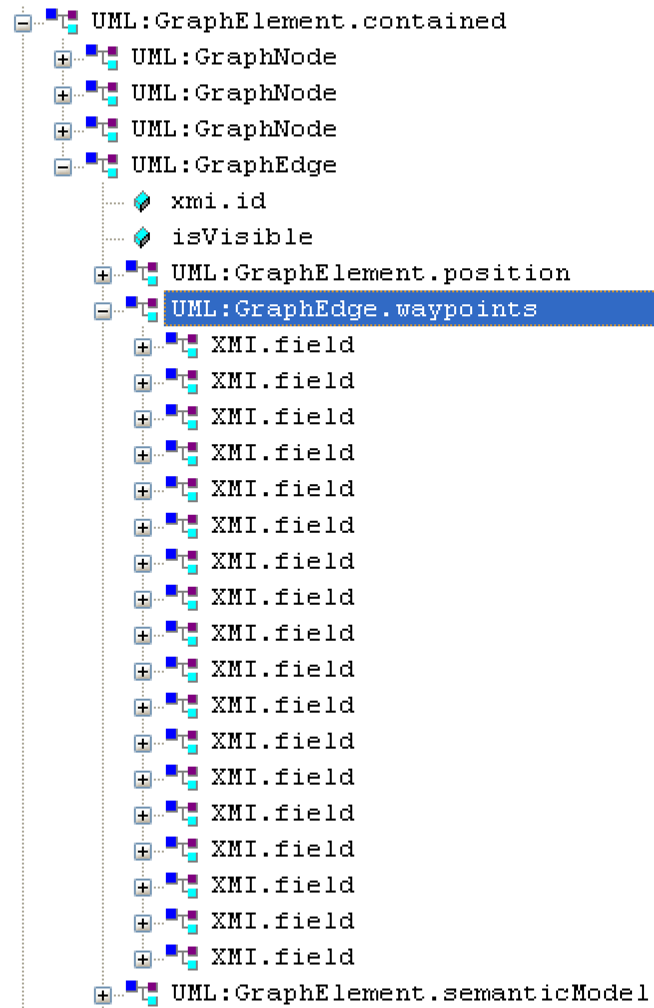
2.-Tras tener claro qué información era la necesaria, pasamos a comparar esa información con la información que la representación interna de MOVA nos proporcionaba, llegando a la conclusión de que la información gráfica que la representación interna de MOVA era distinta a la necesaria para crear la parte gráfica del documento XMI, por lo que nos vimos obligados a crear código para adaptar la información necesaria que MOVA nos daba.

-Para las clases:

-Se necesitaba establecer valores por defecto sobre la posición relativa de los atributos y del nombre de la clase con respecto al hueco al que pertenecen.

-Para los distintos tipos de relaciones:

-Tuvimos que almacenar el conjunto de puntos que forma la relación, tal y como se hace en el estándar que es almacenar cada punto seguido de dos puntos con la posición (0,0) tal y como muestra la siguiente figura. Donde cada campo XMI.field que contiene la información de un punto en concreto, a continuación aparecen dos campos XMI.field con puntos de posición (0,0).



-Además había que incluir la información sobre las posiciones del nombre del diagrama, los roles y las multiplicidades ya que en la representación interna de MOVA siempre se colocan en la misma posición.

-Por otro lado, en la parte gráfica de MOVA, cuando una relación se une por un extremo a una clase, siempre está orientado de forma que se una justo en el punto centro de la clase, pintando primero la relación y encima la clase. Esto era un problema ya que en el estándar, la información sobre la unión de la relación con la clase puede estar en cualquier punto de la clase y esa información necesaria no

es almacenada por la representación interna de MOVA, por lo que tuvimos la necesidad de a través del método obtenerListaPuntos(Vector segmento, datosClase clase1,datosClase clase2). Este método recibe la lista de puntos que forma la relación y las clases que forman los extremos de la asociación (información que nos proporciona la representación interna de MOVA) y nos devuelve la lista de puntos lista para ser representada a partir del estándar de la parte gráfica.

-A grandes rasgos estos son las principales modificaciones que tuvimos que hacer sobre la información gráfica que nos proporciona la representación interna de MOVA para poder crear correctamente el código XMI correspondiente a la parte gráfica.

3.-Una vez conseguida toda la información gráfica para cada posible elemento del diagrama de clases, el siguiente paso fue estudiar bien cómo representar a partir del estándar "Diagram Interchange" toda esa información en el formato XMI.

4.-Ya por último sólo quedaba ampliar el código del método parseador de la clase Almacén, de la misma forma que para la implementación del código del modelo a partir de los métodos proporcionados por la librería JDOM.
Puesto que la cantidad de código para llevar a cabo la implementación de la parte gráfica era muy elevada, se llegó a la conclusión que lo mejor sería trabajar de forma modular. De esta forma pudimos poco a poco ir implementando y probando su correcto funcionamiento para luego seguir la implementación de una parte más compleja.
Los pasos que se tomaron para llevar a cabo una implementación modular de la parte gráfica fueron los siguientes:

1.-Conseguir representar la información de la parte gráfica de una clase vacía y probar su correcto funcionamiento.

2.-Conseguir representar la información de la parte gráfica de una clase sólo con su nombre y probar su correcto funcionamiento.

3.-Ampliar lo anterior incluyendo los atributos y su tipo.

4.-En este punto ya teníamos implementado el código necesario para la representación de la parte gráfica de cualquier clase, por lo que ampliar la implementación para las clases enumeradas fue nuestro siguiente paso.

5.-Ahora nos encontramos con el problema de implementar la parte gráfica relacionada con las asociaciones y generalizaciones, por lo que elegimos empezar primero por la implementación de las generalizaciones ya que éstas son más simples que las asociaciones puesto que carecen de roles y multiplicidades.

6.-Tras conseguir que las generalizaciones funcionaran correctamente, nos pasamos a implementar las asociaciones.

7.-Por último tras tener implementado todo lo dicho anteriormente y más en concreto las clases y las asociaciones, estábamos en disposición de implementar el código referente a las clases de asociación, ya que éstas están formadas por una clase y una asociación.

8.-Una vez conseguida la implementación de la parte gráfica de cualquier diagrama de clases, se hicieron

numerosas pruebas para depurar posibles fallos. Estas pruebas consistieron en utilizar nuestro código generado en XMI por MOVA y probarlo en otras herramientas comerciales como Poseidon para así, comprobar que la implementación era correcta.

- **PROCESO DE LECTURA**

Una vez completado el proceso de escritura, el siguiente paso fue implementar la lectura de diagramas desde documentos XMI 1.2. Este proceso es el inverso al anterior, ya que se recorren los mismos pasos pero a la inversa. Igual que antes, en primer lugar vamos a tratar el proceso de lectura, únicamente, de la información del modelo, para, a continuación, incluir en el mismo la información de la parte grafica.

  - **Lectura de la información del modelo**

    Nuestro punto de partida es el documento XMI 1.2 que contiene toda la información del modelo UML. En esta ocasión hemos de construir el Almacen tomando como partida, no la representación interna de MOVA, sino el documento XMI 1.2. Para poder extraer la información del documento XMI 1.2 y crear el Almacén, el primer paso es crear un árbol que represente toda la información del diagrama guardada en el documento, para, a continuación, empleando los métodos proporcionados por la librería JDOM, extraer la información del árbol e ir insertándola en el Almacén. La creación del Almacén se realiza mediante la constructora "Almacen ( )", la cual crea un Almacén vacío. La construcción del árbol y la extracción de la información del mismo la realiza el método parseadorInv(String filename). La extracción de información del árbol y su inserción en el Almacen se realiza en el siguiente orden:

      - Tipos de Datos
      - Estereotipos (de clases enumeradas)

- Clases y Clases enumeradas junto con sus atributos
- Generalizaciones
- Asociaciones
- Clases Asociación

Para algunos de estos elementos existen distintas formas aleatorias, y reconocidas por el estándar, para representarlos. Nuestra herramienta reconoce estos elementos independientemente de la forma legítima en la que se encuentren.

La herramienta esta diseñada para no dar error si no existe alguno de estos elementos o si hay elementos adicionales. Esto es, sin duda, una ventaja, ya que, en algunos casos, determinadas herramientas pueden generar elementos para consumo propio no pertenecientes al estándar, en cuyo caso nuestra herramienta seguiría funcionando perfectamente.

Una vez que el Almacén se encuentra creado se procede a la creación del diagrama UML en la representación interna de MOVA. El método encargado de realizar esta tarea es loadDataFromXMI( ).

MOVA representa los diagramas de forma gráfica, sin emplear una representación en forma de explorador. Esto hizo necesario durante el proceso de depuración implementar un descriptor del Almacen, el cual muestra en forma de texto toda la información del diagrama. De esta forma se podían observar claramente los errores y proceder a su correción.

- **Lectura de la información de la parte gráfica**

  Una vez implementada y probada la lectura de la información de la parte del modelo, ahora sólo nos quedaba implementar el código necesario para recoger la información gráfica que la representación interna de MOVA necesita.

Ya que la información gráfica almacenada en el código XMI se encuentra de forma dispersa, la mejor solución para afrontar este problema fue la siguiente:

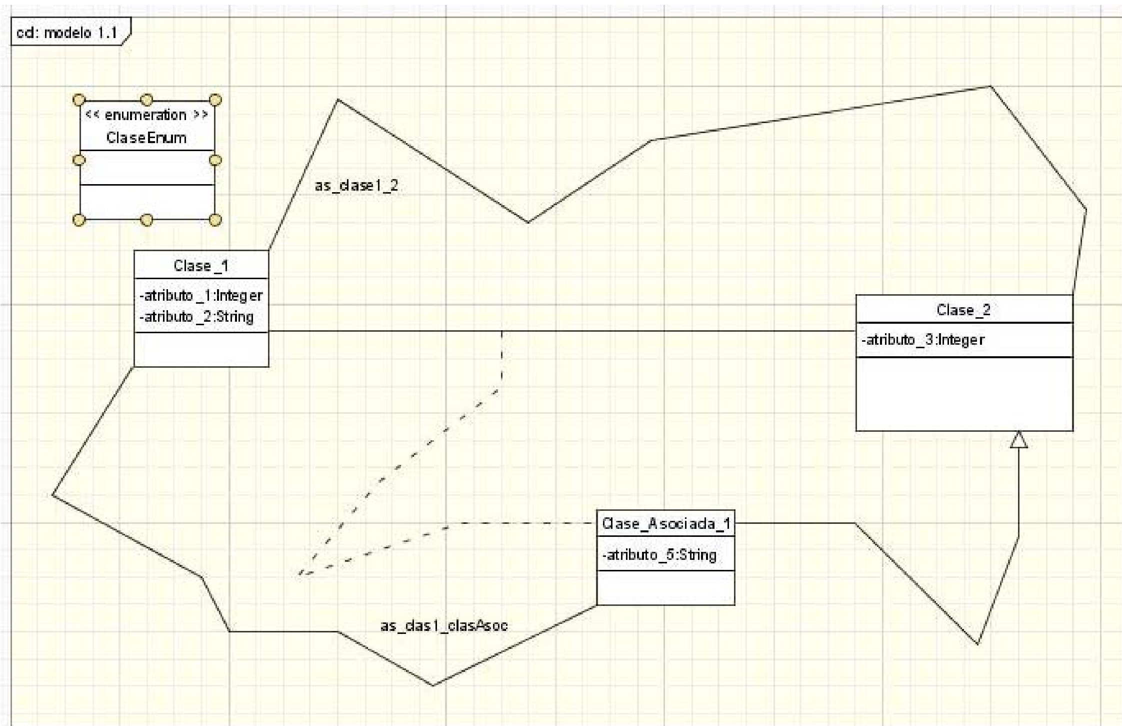
1.-Elaborar un documento en donde se detallara de forma clara qué información es la que se necesita y dónde se encuentra esta información para cada uno de los posibles elementos que pueden formar parte del diagrama de clases.
Así a partir de este documento, se tendría claro cómo conseguir la información gráfica a partir del archivo XMI.
El documento elaborado está creado a partir de un ejemplo completo. Este documento puede revisarse en el apéndice.

2.-Implementar el código necesario para obtener toda la información necesaria a partir del archivo XMI añadiendo este código en el método llamado método parseadorInv(String filename) de igual forma que se realizó en la lectura de la parte del modelo.

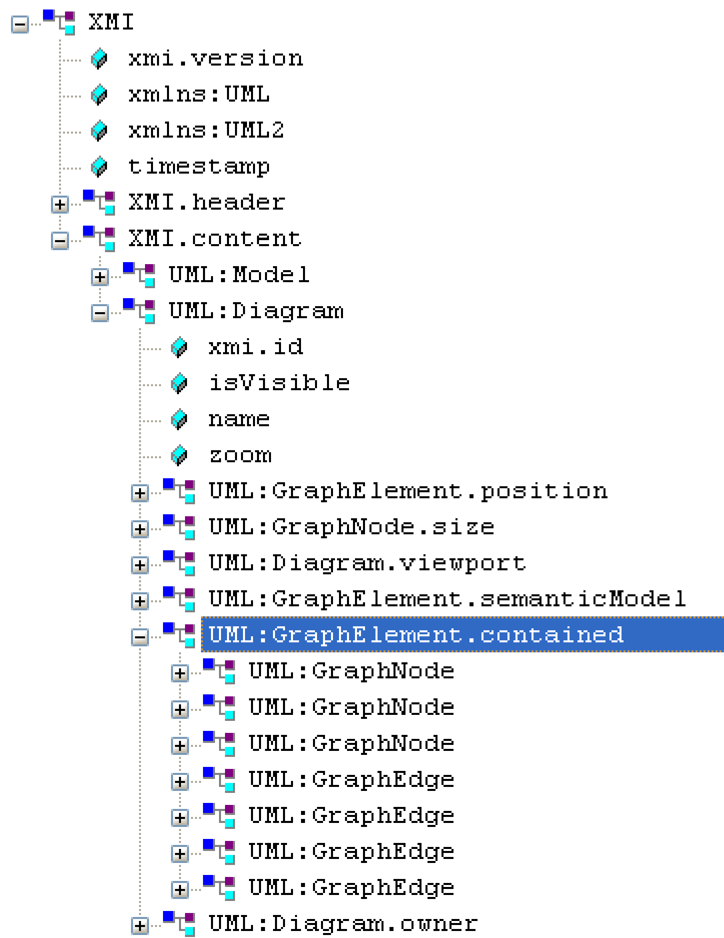
3.-Realizar las pruebas pertinentes para probar el correcto funcionamiento del código implementado.

APÉNDICE

Información de la parte grafica a partir de un ejemplo



La información de los elementos de la parte grafica se encuentran aquí:

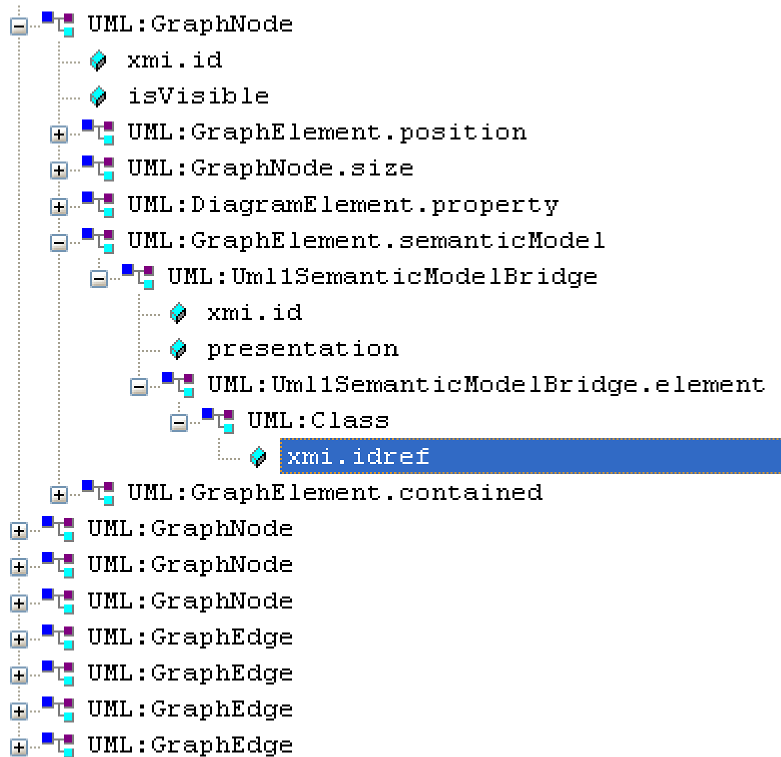


-Las clases y las clases enumeradas son los **GraphNode**.
-Las asociaciones, generalizaciones y clases de asociación son los **GraphEdge**.

-Distinguidos ya los distintos elementos de un posible diagrama en la parte grafica, procedo a explicar que información necesitamos de cada elemento, donde esta esa información y como saber a que elemento del modelo hace referencia.

CLASES

Primero de todo queremos saber un GraphNode en concreto, a que clase del modelo hace referencia para saber donde guardar su información. El **idref** que hace referencia a la clase en el modelo se encuentra aquí:

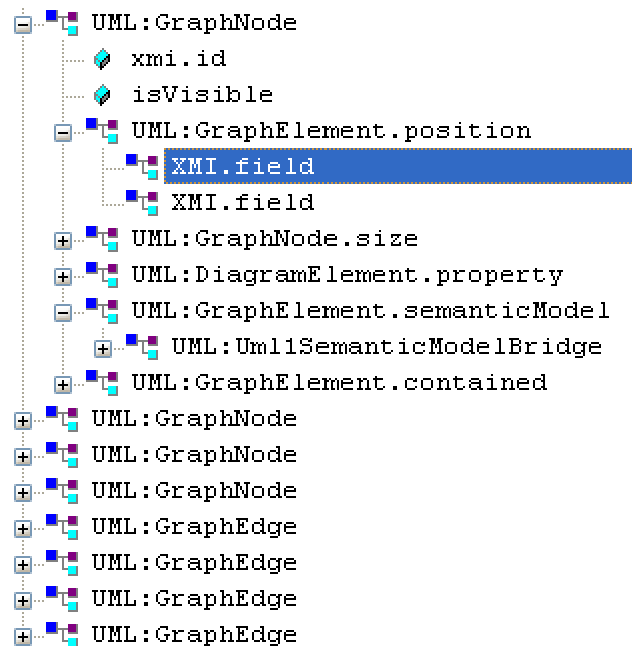


Notar que los elementos del árbol mas a la izquierda hace referencia a cada elemento de nuestro modelo, mas en concreto he desplegado el primero de ellos, el GraphNode que hace referencia a la clase con id el que viene como valor en donde esta marcado con azul. Así ya sabemos a que clase pertenece esta información grafica.

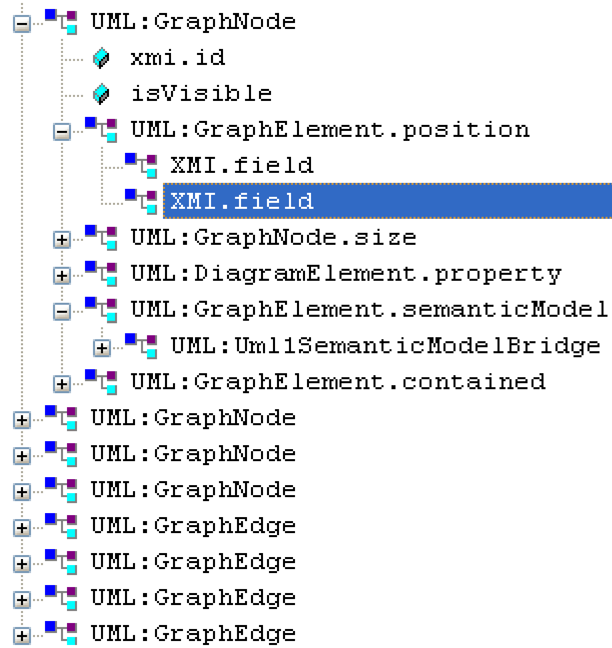
La información grafica que necesitamos de las clases es la siguiente:

-**Posición superior izquierda** de la clase:

-Posición X siempre la primera componente XMI.field:

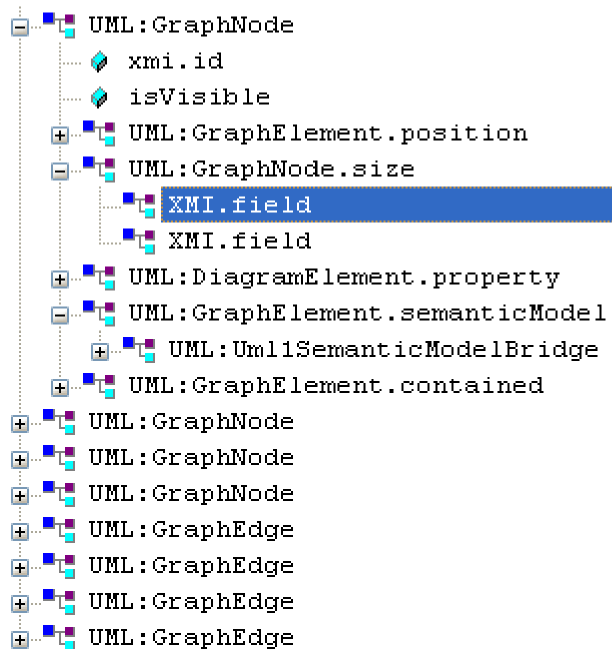


-Posición Y siempre la segunda componente XMI.field:

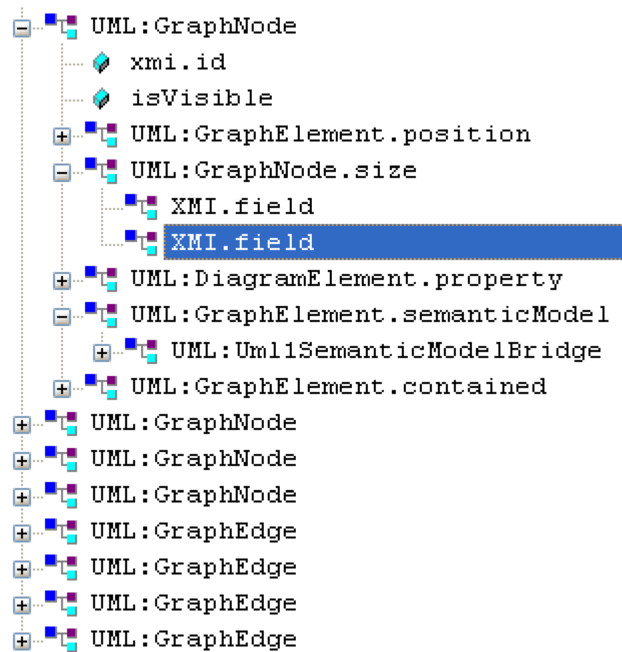


-La otra información que necesitamos es el **ancho y el alto**:

-El ancho siempre es la primera componente XMI.field:



-El alto siempre es la segunda componente XMI.field:



Distinguimos si GraphNode es clase enumerada o no directamente con el idref mencionado anteriormente, ya que con el, sabrás a que clase pertenece y por lo tanto sabrás distinguir si es enumerada o no.

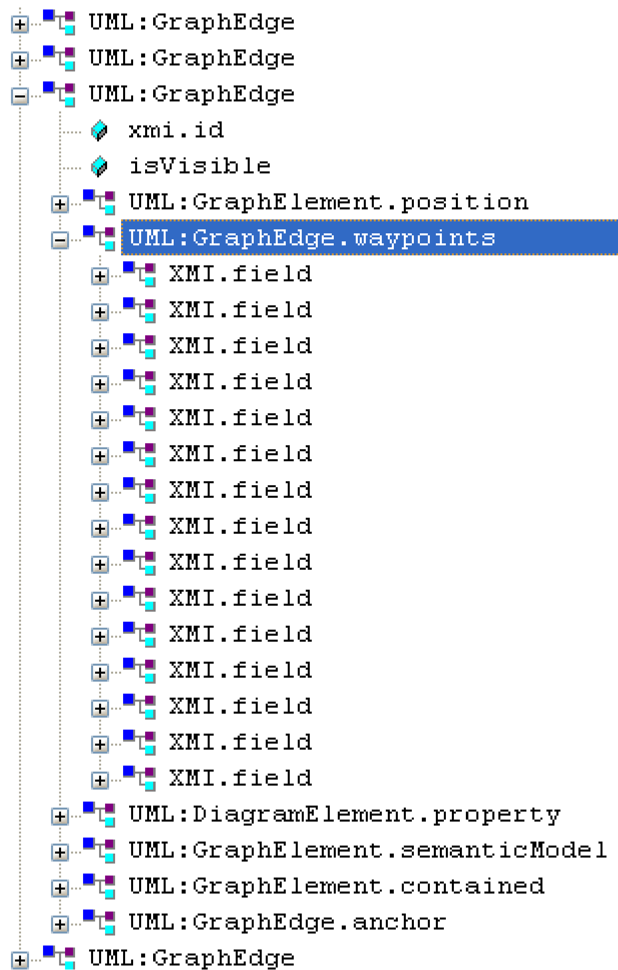
GENERALIZACIONES Y ASOCIACIONES

Primero sabemos que de entre todos los elementos de la parte grafica, una generalización o una asociación, es un GraphEdge. Ahora, para saber si ese GraphEdge es una generalización o una asociación, basta con comprobar que tiene un nodo *UML:Generalization* o *UML:Association* qu pertenece a las asociaciones y no otro cualquiera como por ejemplo *UML:AssociationClass* que pertenece a las clases de asociación
Una vez detectado que es una generalización o una asociación, sabremos cual es por su idref que reverenciará al id de la generalización o asociación a la que pertenece.
Esta información se encuentra aquí:

```
UML:GraphNode
UML:GraphNode
UML:GraphNode
UML:GraphNode
UML:GraphEdge
UML:GraphEdge
UML:GraphEdge
    xmi.id
    isVisible
    UML:GraphElement.position
    UML:GraphEdge.waypoints
    UML:DiagramElement.property
    UML:GraphElement.semanticModel
        UML:Uml1SemanticModelBridge
            xmi.id
            presentation
            UML:Uml1SemanticModelBridge.element
                UML:Generalization
                    xmi.idref
    UML:GraphElement.contained
    UML:GraphEdge.anchor
UML:GraphEdge
```

```
UML:GraphNode
UML:GraphNode
UML:GraphNode
UML:GraphNode
UML:GraphEdge
    xmi.id
    isVisible
    UML:GraphElement.position
    UML:GraphEdge.waypoints
    UML:DiagramElement.property
    UML:GraphElement.semanticModel
        UML:Uml1SemanticModelBridge
            xmi.id
            presentation
            UML:Uml1SemanticModelBridge.element
                UML:Association
    UML:GraphElement.contained
    UML:GraphEdge.anchor
UML:GraphEdge
UML:GraphEdge
UML:GraphEdge
```

-Ahora la única información que necesitamos es el **conjunto de puntos que forman la unión de un extremo con otro de la generalización o asociación**. Para ahorrar trabajo, Ana solo necesita los puntos intermedios, los puntos de los extremos no nos hacen falta, pero creo que es más fácil de implementar el coger todos los puntos sin miramientos y Ana ya se encarga de desechar los extremos.
Bien, los puntos que forman la generalización o la asociación, se encuentran aquí:



-No nos asustemos por la cantidad de campos XMI.field que hay porque es fácil de entender: si nos fijamos en el dibujo del ejemplo en el principio del documento, nuestra generalización esta formada por cinco puntos y tenemos quince campos XMI.field, esto se debe a que para cada punto se utilizan tres XMI.field de los cuales:

-El primero tiene la información que necesitamos.

-Los otros dos restantes están a cero y no nos proporciona nada.

-Por lo tanto, nosotros deberemos leer los campos XMI.field uno si, dos no, uno si, dos no y asi hasta que no haya ninguno.

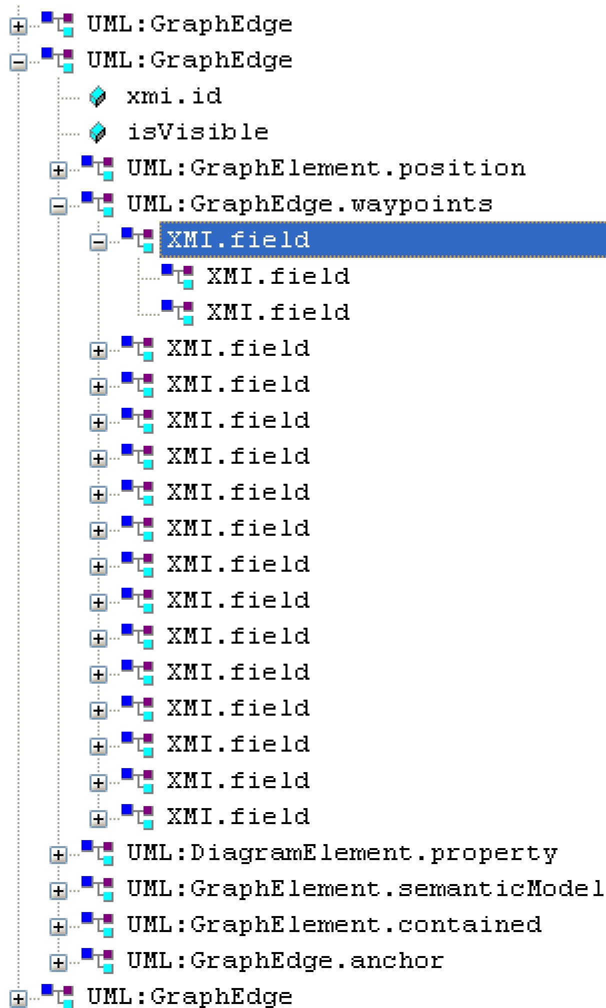
Entendido esto, nos encontramos con quince XMI.field de los cuales ya sabemos cuales son los que nos interesan. Dentro de cada campo XMI.field hay dos campos XMI.field:

-El primero es la posición x del punto.

-El segundo es la posición y del punto.

-Notar que los campos XMI.field están en orden. Esto quiere decir que el primer campo corresponde con el primer punto (si es generalización, el extremo que es la clase hijo) y el último con el último punto (si es generalización, el extremo que es la clase padre).

-Para verlo con más claridad, muestro los dos campos que componen cada XMI.field:

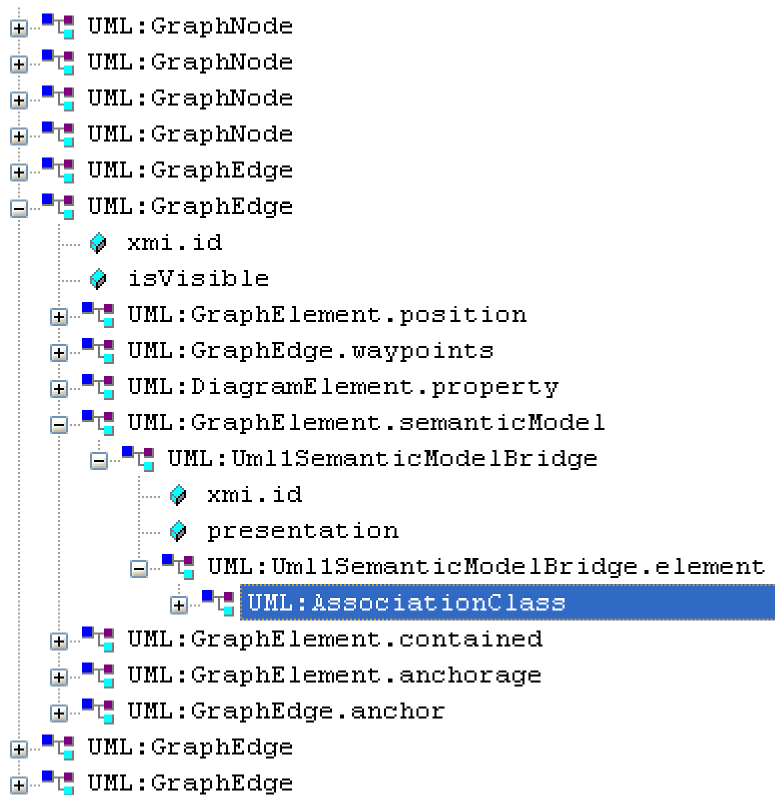


CLASES DE ASOCIACION

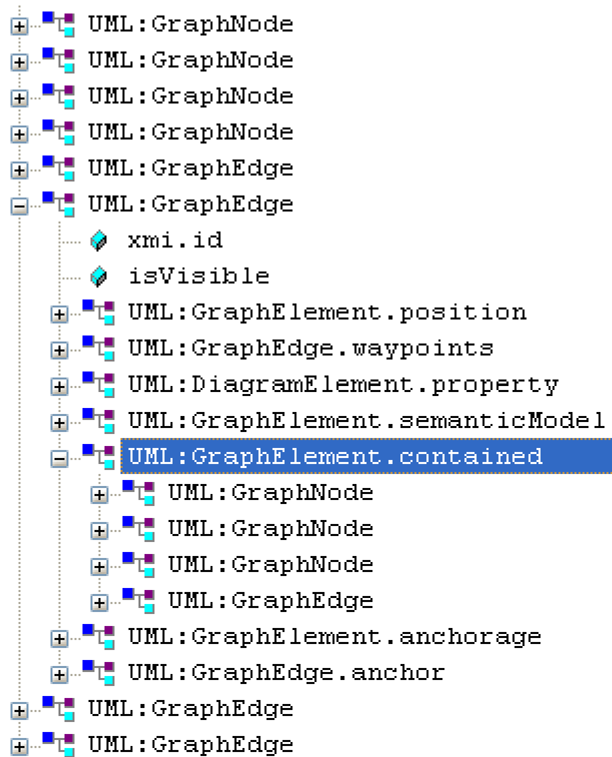
-Las clases de asociación se podrían descomponer (y así lo hace Mova) en una clase y en una asociación. Por lo tanto la información que necesitamos es la de:

- -La clase que la forma:
    - -Punto superior izquierdo.
    - -Ancho.
    - -Alto.
- -La asociación que la forma:
    - - El conjunto de puntos que forma la linea punteada.

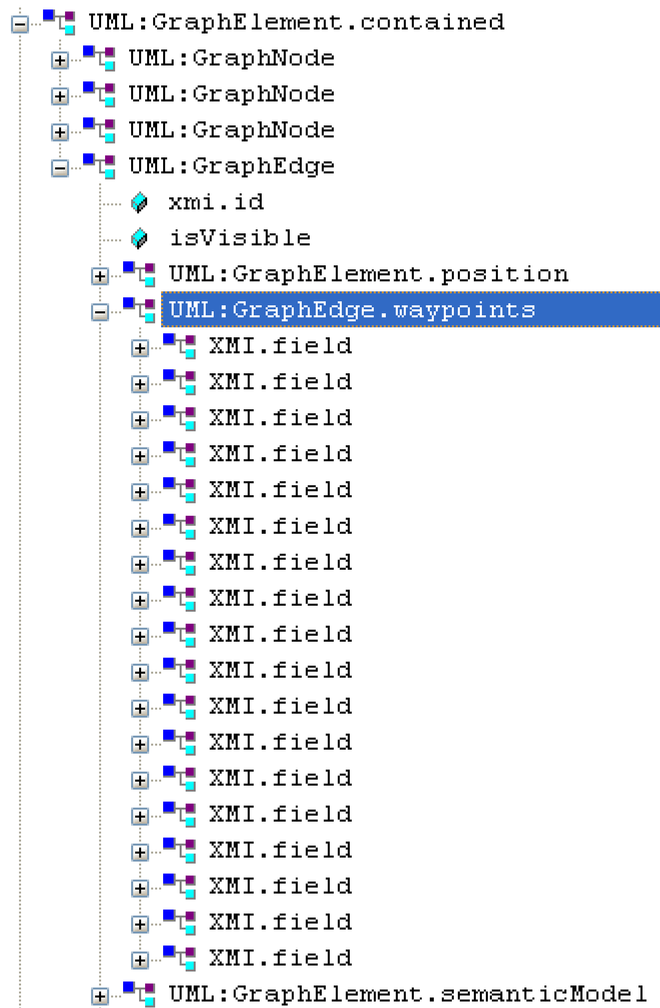
Primero tenemos que saber donde encontrar las clases de asociación, y como ya explique para las generalizaciones y las asociaciones, miramos el mismo campo pero ahora tendrá que aparecer *UML:AssociationClass* tal y como detallo en la imagen:



-Ya detectadas las clases de asociación, explico donde encontrar la información que necesitamos.
-El contenido grafico de una clase de asociación es:
    -Tres GraphNode:
        -Dos son los extremos de la asociación (información irrelevante).
        -El otro es el que contiene la información grafica de la clase.
    -Un GraphEdge que contiene la información que necesitamos de la asociación (la línea punteada).
-Veamos la imagen para comprobarlo:

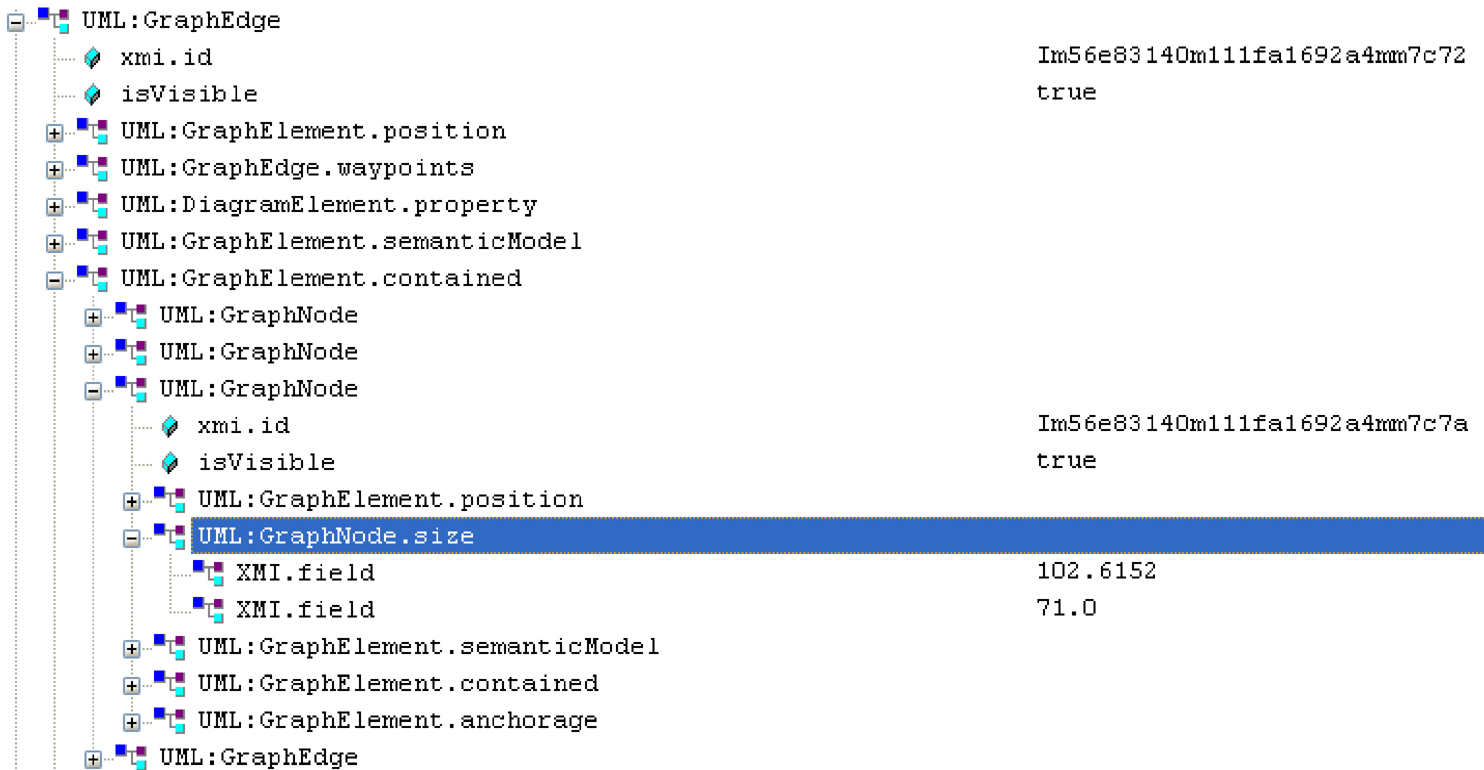


Vamos primero donde esta la información de la asociación que forma la clase de asociación (la línea punteada). Esta información se encuentra en el campo *UML:GraphEdge.waypoints* que esta contenido dentro del campo *UML:GraphEdge* mencionado justo antes. Los puntos son interpretados de la misma forma que para las asociaciones y generalizaciones.
Para ver exactamente donde están estos puntos, veamos el dibujo:



Para situarnos, los tres GraphNode y el GraphEdge que aparecen como elementos contenidos más a la izquierda, hacen referencia a los explicados anteriormente.

-Ahora solo nos falta localizar la información de la clase que forma la clase de asociación. Esta información se encuentra en uno de los tres GraphNode que vemos en el dibujo. En este caso es el último pero nadie nos asegura que pudiera ser uno de los otros tres. La forma de saber cual de los tres es, es viendo su contenido. Hay varias formas de identificarlo pero se que la mas fácil es ver que el campo *UML:GraphNode.size* tiene sus dos *XMI.field* con el valor distinto de cero ya que los GraphNode que tienen sus *XMI.field* con valor a cero, son lo extremos de la asociación.
-Veamos en el dibujo cual es el GraphNode que nos interesa porque tiene sus XMI.field con valores distinto de cero:



Para situarnos, saber que el primer campo *UML:GraphEdge* es el que pertenece a la clase de asociación, y dentro, en su campo *UML:GraphElement.contained* estan los tres GraphNode's y el GraphEdge explicados anteriormente.

Una vez detectado donde esta el GraphNode que contiene la información grafica necesaria de la clase que forma la clase de asociación, nos servimos del dibujo anterior para decir que el campo marcado (el de azul) contiene (de igual forma que para las clases, explicado anteriormente) el ancho y el alto, y el campo *UML:GraphElement.position* contiene el punto superior izquierdo de la clase.