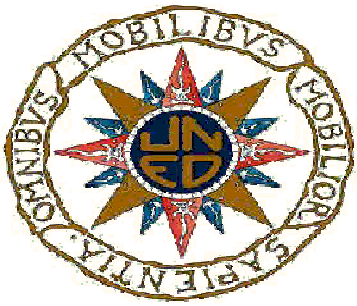
UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA



INGENIERÍA DIRIGIDA POR MODELOS APLICADA AL INTERCAMBIO ELECTRÓNICO DE DATOS

Realizado por:
Daniel Pérez Berenguer

Directora:
Dra. Elena Ruiz Larrocha

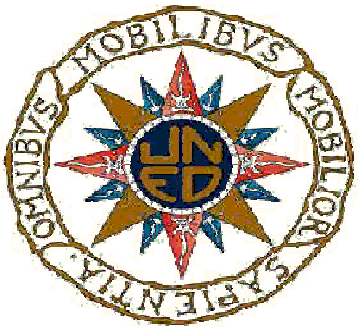
Itinerario de Ingeniería de Software

Trabajo Fin de Máster. Código 105128

MÁSTER EN INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS

Junio 2012

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA



INGENIERÍA DIRIGIDA POR MODELOS APLICADA AL INTERCAMBIO ELECTRÓNICO DE DATOS

Realizado por:
Daniel Pérez Berenguer

Directora:
Dra. Elena Ruiz Larrocha

Ingeniería Dirigida por Modelos

Itinerario de Ingeniería de Software

Trabajo Fin de Máster. Código 105128

MÁSTER EN INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS

Autorización

Autorizo a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del Autor

Resumen

El intercambio electrónico de datos (EDI) es una técnica muy usada por las empresas desde su aparición a mediados de los noventa, como forma de transferir documentos e información entre las empresas. EDI, por tanto, ha contribuido a automatizar la transferencia de documentos como pedidos o facturas, que puede ser realizada sin intervención humana.

La Ingeniería de Modelos (MDE) surge como el área de la Ingeniería del Software centrada en el uso sistemático de los modelos para la construcción de software. Los modelos han mostrado su potencial para aumentar el nivel de abstracción y automatización, y de este modo mejorar diferentes aspectos de la calidad del software.

En el contexto de aplicación del EDI, este proyecto plantea el uso de las técnicas MDE para el manejo de los datos o ficheros electrónicos en formato EDI. Para ello se ha creado una infraestructura que permite la generación automática de los ficheros electrónicos de facturas a partir de los datos existentes en la base de datos.

Palabras clave: Ingeniería de Modelos (MDE), Desarrollo de Software Dirigido por Modelos (MDD), Electronic Data Interchange For Administration, Commerce and Transport (EDIFACT).

Agradecimientos

A mí directora de proyecto, Dra. Dña. Elena Ruiz Larrocha, por la confianza depositada, por su apoyo y por guiarme en este proyecto.

Al Dr. D. Jesús García Molina, por su apoyo y ayuda prestada. Persona que ha confiado siempre en mí y que ha sido un referente a seguir a lo largo de toda mi carrera profesional.

A D. Javier Luis Cánovas Izquierdo, por su ayuda prestada con el entorno Schemol.

A mi mujer y mi hija, un apoyo fundamental en mi vida. Sufridoras de la dedicación en exclusiva a este proyecto.

En general a toda mi familia.

Muchas gracias.

El agradecimiento es la memoria del corazón
Lao-tsé

Índice

Índice de Figuras

1 Introducción

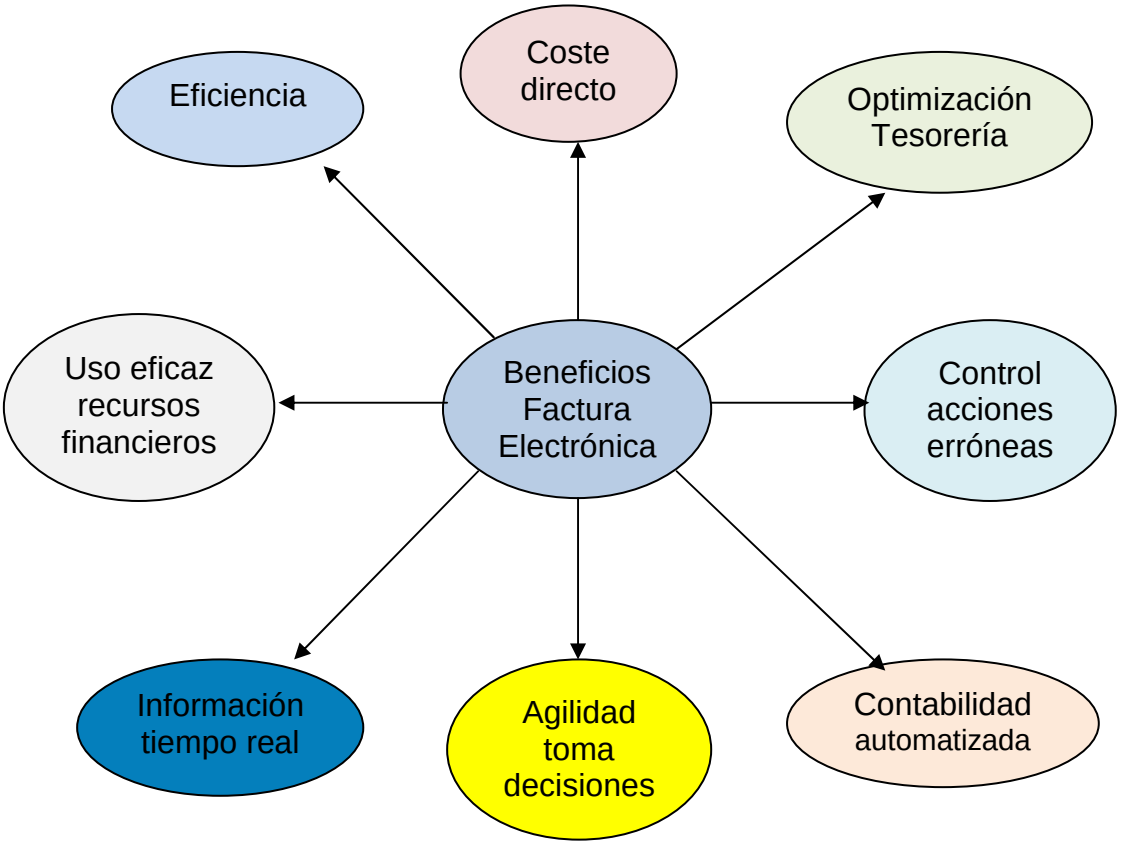
En el contexto económico en el que nos encontramos, las empresas tratan de ser lo más competitivas posibles para poder mantenerse en los mercados en los que están consensuadas pero también para abrirse a nuevos mercados que permitan suplir las carencias derivadas por la disminución de ventas generadas por la crisis. Una de las formas más utilizadas por una empresa para ser más competitiva es el abaratamiento de costes directos e indirectos. Si tienes un buen producto y eres capaz de lanzarlo al mercado soportando menores costes que tu competencia, serás capaz de conseguir nuevos clientes. Esto ha llevado a las empresas a estudiar sus procesos productivos para intentar hacerlos más eficientes.

El intercambio electrónico de datos (EDI) es una técnica muy usada por las empresas desde su aparición a mediados de los noventa, como forma de transferir documentos e información entre las empresas. EDI, por tanto, ha contribuido a automatizar la transferencia de documentos como pedidos o facturas, que puede ser realizada sin intervención humana.

Dentro de los datos electrónicos destaca la factura electrónica, cuya transmisión entre medios electrónicos con firma digital le da la misma validez que la factura en papel pero suponiendo un ahorro sustancial para el emisor y receptor de la misma. Un proceso normal de facturación le supone al emisor la impresión de la factura mediante un medio electrónico, sobre y correo, mientras que al receptor le supone la conciliación, contabilización, pago y archivo de la factura. Un estudio de la Asociación Española de Codificación Comercial (AECOC) determina que el ahorro de coste que le supone al emisor de una factura esta en torno a 0,70€/factura mientras que el ahorro de coste que le supone al receptor esta en torno a 2,78€/factura. En la Figura 1 podemos ver algunos de los beneficios que le supone a una empresa la implantación de la factura electrónica.

Existen otros datos que permiten incrementar la eficiencia de las operaciones realizadas en el seno de una empresa como *Pedidos* (gestión de pedidos), *Albaranes* (gestión de albaranes) o una confirmación de entrega, todos ellos,

una vez representados en un formato electrónico, pueden ser integrados dentro de una aplicación de *workflow* de ventas y compras de la empresa.



(Fuente: propia)

Figura 1. Beneficios derivados de la factura electrónica.

En el intercambio de datos electrónicos, surge la necesidad de definir formatos estructurados estándar y han aparecido, por tanto, algunos como Facturae del Centro de Cooperación Interbancaria [44], UN/Edifact (United Nations/Electronic Data Interchange For Administration, Commerce and Transport) de las Naciones Unidas [45] o UBLXML de la organización OASIS [46].

En los últimos años, el desarrollo de software dirigido por modelos (Model-Driven Development, MDD) se ha convertido en una de las principales líneas de investigación y desarrollo en la comunidad del software, tanto en el mundo académico como en la industria, habiendo surgido la Ingeniería de Modelos (MDE) como el área de la Ingeniería del Software centrada en el uso

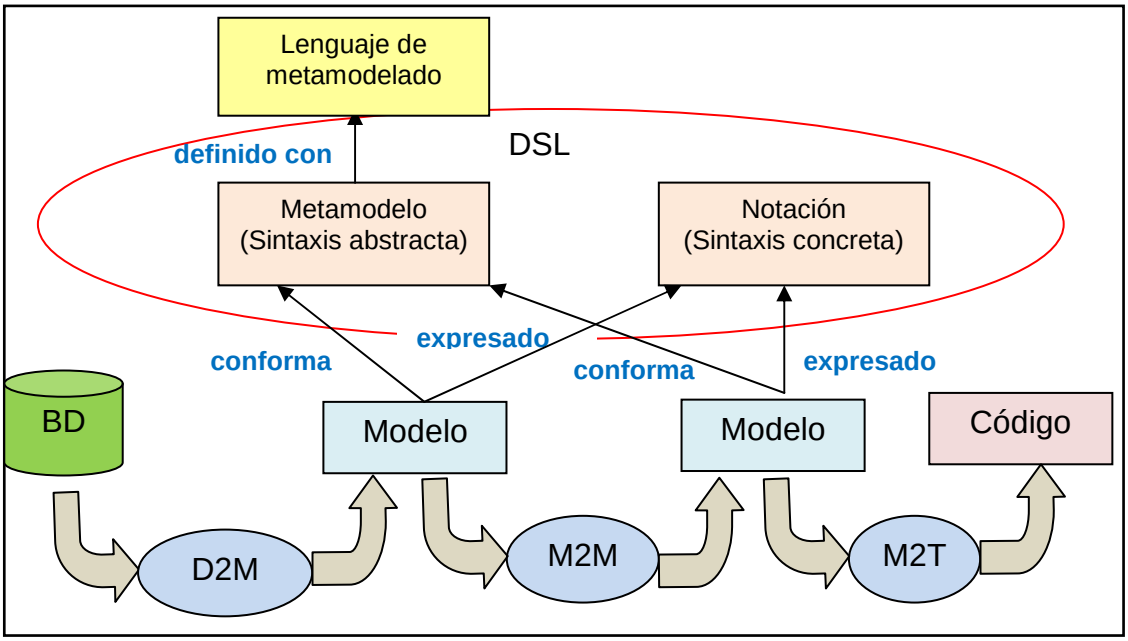
sistemático de los modelos para la construcción de software. Los modelos han mostrado su potencial para aumentar el nivel de abstracción y automatización, y de este modo mejorar diferentes aspectos de la calidad del software. Detrás de MDE hay varios paradigmas de desarrollo que comparten unos mismos principios: los modelos son expresados con lenguajes específicos del dominio (DSL); los DSLs son formalmente definidos con metamodelos, y la automatización se obtiene mediante transformaciones de modelos. Para integrar la tecnología MDE con otras tecnologías y de este modo aprovechar las técnicas y herramientas MDE destinadas a automatizar tareas del desarrollo de software, se están construyendo herramientas que permiten transformar artefactos existentes (por ejemplo, código fuente, código XML, datos relaciones o datos electrónicos) en modelos.

En el contexto de aplicación del EDI, este proyecto plantea el uso de las técnicas MDE para el manejo de los datos o ficheros electrónicos en formato EDI. Para ello se ha creado una infraestructura que permite la generación automática de los ficheros electrónicos de facturas a partir de los datos existentes en la base de datos.

Una solución MDE consiste en una cadena de transformaciones de modelos en las que se parte de un modelo inicial y a través de una o más transformaciones de modelos (modelo-a-modelo y modelo-a-texto) se genera total o parcialmente una aplicación software [47]. En este caso se parte de la información sobre las facturas existente en la base de datos y se generan las facturas electrónicas en un formato estándar. Puesto que se parte de información no representada mediante modelos, es necesario un proceso de inyección o extracción de modelos, para lo que se ha utilizado la herramienta Schemol [48], creada para extraer modelos a partir de datos en bases de datos relacionales.

Dado que en MDD, los modelos conforman a metamodelos que describen formalmente su estructura [47], en este proyecto se ha creado un metamodelo para representar facturas al que conforman los modelos extraídos con Schemol, y otro metamodelo para representar modelos de facturas electrónicas en el estándar EDIFACT (Electronic Data Interchange For Administration,

Commerce and Transport) [45]. Estos dos metamodelos junto con la cadena de transformaciones creada constituyen la aportación original de este trabajo.



(Fuente: propia)

Figura 2.Estructura MDE aplicada a ficheros electrónicos.

En la Figura 2 se puede ver la estructura general planteada para el manejo de ficheros electrónicos. Los pasos son:

- Se define un metamodelo cercano a los conceptos del tipo de fichero que se quiere modelar: pedidos, facturas, albaranes, etc.

- Se define una notación para generar modelos conformes al metamodelo anterior a través de la herramienta EMFText, que permite definir la sintaxis concreta de forma textual acorde al metamodelo Invoic, o se configura Schemol para extraer dichos modelos directamente de la base de datos.

- Se realiza una transformación modelo-modelo que permita acercar estos conceptos generales al formato estructurado particular que se quiere implementar.

- Por último, se aplica una transformación modelo-texto que automatice la generación del fichero electrónico.

Schemol es un lenguaje específico del dominio utilizado para extraer modelos de bases de datos relaciones que surge del trabajo conjunto de Grupo ModelUM de la Universidad de Murcia y del Grupo ONEKIN de la Universidad del País Vasco. Tras contactar con el grupo de trabajo de la Universidad de Murcia, permitieron el uso de la herramienta Schemol en el ámbito de este proyecto.

La aplicación de MDD requiere de una infraestructura que soporte tres operaciones básicas que permitan la definición de: metamodelos, Lenguajes Específicos del Dominio (*Domain Specific Language*, DSL) y transformaciones de modelos. La arquitectura de metamodelado de Eclipse se ha convertido en la principal plataforma para aplicar la Ingeniería Dirigida por Modelos (*Model-Driven Engineering*, MDE). *Eclipse Modeling Framework* (EMF) implementa lo que se denomina una arquitectura de metamodelado. Este proyecto se basa en el uso de Eclipse en general y en particular de EMF.

Este documento se ha organizado del siguiente modo. Después de esta presentación de los objetivos, en el segundo Capítulo 2 se presentan los conceptos que definen la factura electrónica, ventajas que se obtienen así como proceso de implantación en la empresa. En el Capítulo 3 se definen los conceptos del Desarrollo Dirigido por Modelos y se resume la tecnología MDE utilizada en este proyecto. En el Capítulo 4 se presenta la aplicación de MDE sobre la generación de la factura electrónica. En el Capítulo 5 se recogen las conclusiones y los posibles trabajos futuros. Además, este documento va acompañado de dos anexos. En el Anexo A se encuentra la estructura del mensaje Edifact, en el Anexo B se muestra un ejemplo de definición de un modelo que conforma con el metamodelo Edifact, el Anexo C contiene la configuración de Schemol, el Anexo D es un manual de configuración de RubyTL, el Anexo E refleja las reglas de transformación de modelos Invoic a modelos Edifact y en el Anexo F se presenta la configuración de MOFScript.

2 La Factura Electrónica

2.1 Introducción

El proceso tradicional de facturación en cualquier empresa consiste en la impresión de la factura a través de un medio informático, impresión del sobre y envío a través del correo. Por otro lado, el destinatario recibe la factura, la concilia, contabiliza, paga y por último, la archiva quedando a disposición de auditorías o inspecciones fiscales que se basan en el valor documental del soporte en papel.

La **facturación electrónica** consiste en la transmisión de las facturas o documentos análogos entre emisor y receptor por medios electrónicos (ficheros informáticos) y telemáticos (de un ordenador a otro), firmados digitalmente con certificados cualificados, con la misma validez legal que las facturas emitidas en papel.

Se puede definir “Factura Electrónica” como el documento tributario generado por medios informáticos en formato electrónico, que reemplaza al documento físico en papel, pero que conserva su mismo valor legal con unas condiciones de seguridad no observadas en la factura en papel [49].

Las claves de la factura electrónica son [49]:

- Conservar los datos de las facturas. No es necesario conservar las facturas emitidas sino la “matriz” o base de datos que permite generarlas.
- Asegurar la legibilidad en formato original.
- Garantizar acceso completo a las facturas: visualización, búsqueda selectiva, copia o descarga en línea e impresión.
- En caso de emisión de factura electrónica: firmar electrónicamente la factura o delegar esta acción en un tercero o en el Receptor.

Las facturas electrónicas se pueden emitir en diferentes formatos (EDIFACT [45], XML, PDF, HTML, DOC, XLS, GIF, JPEG o TXT, entre otros) siempre que se respete el contenido legal exigible a cualquier factura y que se cumplan con ciertos requisitos para la incorporación de la firma electrónica reconocida.

En el intercambio de facturas electrónicas se debe conseguir **autenticidad** e **integridad.** El empleo de la firma electrónica y el uso de sistemas EDI permiten conseguir estas cualidades.

La factura electrónica queda incluida dentro del *workflow* de ventas y compras de la empresa y por tanto, la gestión electrónica en cada uno de los pasos anteriores permitirá una implantación eficaz de la factura electrónica. En la Figura 3 podemos ver un ejemplo de *workflow* de compras:

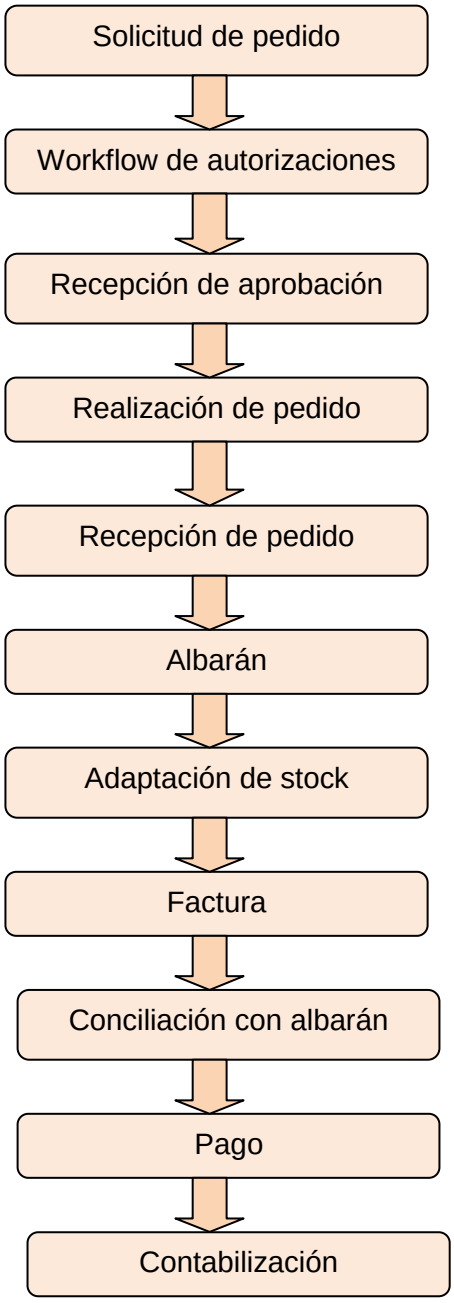


Figura 3. Workflow del proceso de compras (Fuente: propia)

Existen numerosas ventajas relacionadas con el uso de la factura electrónica como:

- **Ahorro de costes:** tanto del lado del emisor como del receptor. Derivado de la supresión del papel, el abaratamiento de los medios de comunicación electrónicos, eliminación de los gastos de franqueo, gastos derivados de la introducción manual de datos, etc. La Asociación Española de Codificación Comercial (AECOC) [50], cifra en aproximadamente 0,70€ de ahorro por factura para el emisor y hasta 2,78€ por factura para el receptor en el empleo de tecnología EDI. En la Figura 4 podemos ver un resumen del desglose de costes.

- **Mejora de la eficiencia:** la liberación de tareas administrativas, permite destinar los recursos humanos a aspectos productivos en las compañías.

- **Optimización de la tesorería:** la automatización permite cuadrar los apuntes contables y comparar documento (albarán / factura), minimizando a la vez el margen de error humano.

- **Obtención de información en tiempo real:** permite verificar el estado en el que se encuentra toda factura y toda su información asociada de forma exacta y actual.

- **Agilidad en la toma de decisiones:** la inmediatez de las comunicaciones permite adoptar decisiones, como la necesidad de financiación, en un espacio más corto de tiempo.

- **Administración y contabilidad automatizadas:** la integración en los sistemas de la empresa permite toda la inserción de datos y que las operaciones contables requieran mucho menos participación humana.

- **Control de acciones erróneas:** a través de sistemas de alertas que detectan discrepancias entre operaciones de contabilidad y facturación.

- **Uso eficaz de recursos financieros:** la adopción de la factura electrónica favorece el acceso a medios de financiación como el *factoring* o el *confirming*.

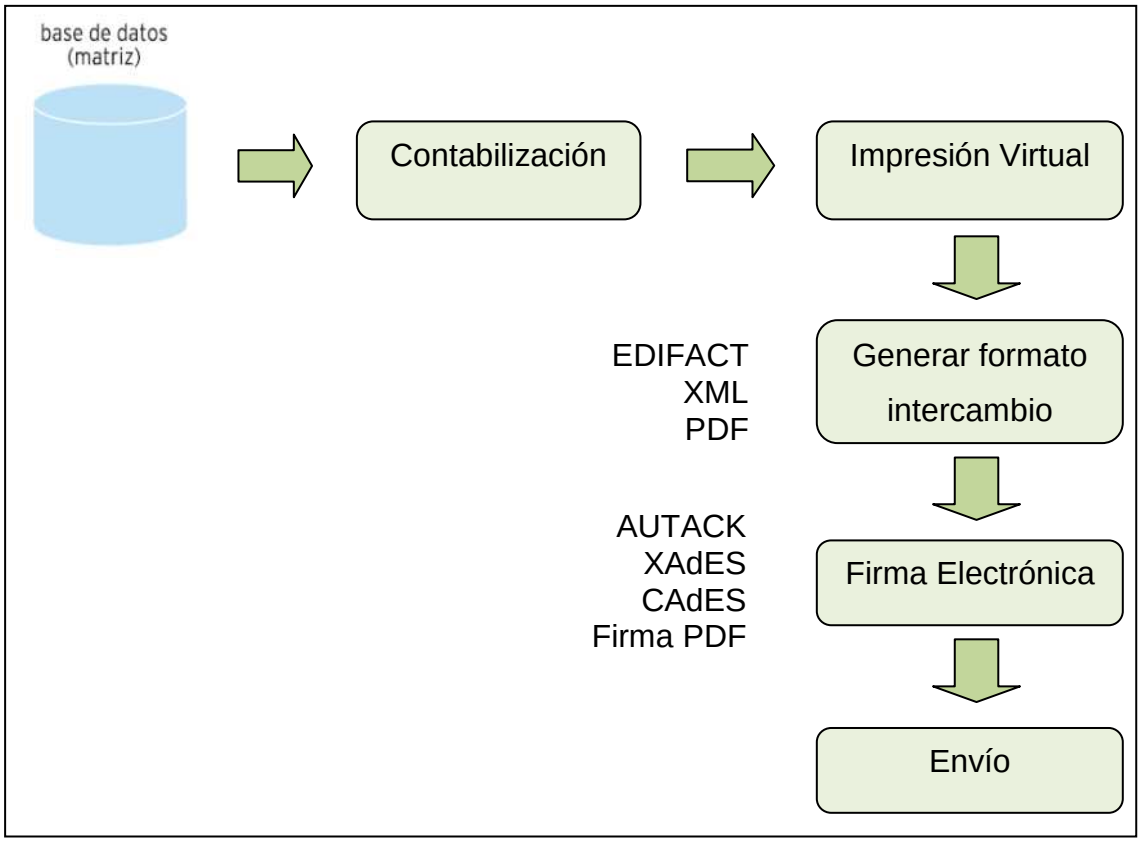
(Fuente: AECOC)

Figura 4. Beneficios de la facturación electrónica. AECOC.

A la hora de abordar un proyecto de factura electrónica, existen diferentes opciones en base a la complejidad del mismo que pueden ir desde un envío por mail de una factura firmada electrónicamente, pasando por un nivel intermedio que lo formarían las plataformas de facturación de terceros, hasta llegar al nivel de la plena integración con el resto de elementos informáticos que permitan la automatización de múltiples tareas.

Si el proyecto se analiza desde el punto de vista del **emisor**, se pueden encontrar proyectos más sencillos pero que tienen menos repercusión en cuanto a costes. El proceso se puede observar en la Figura 5.

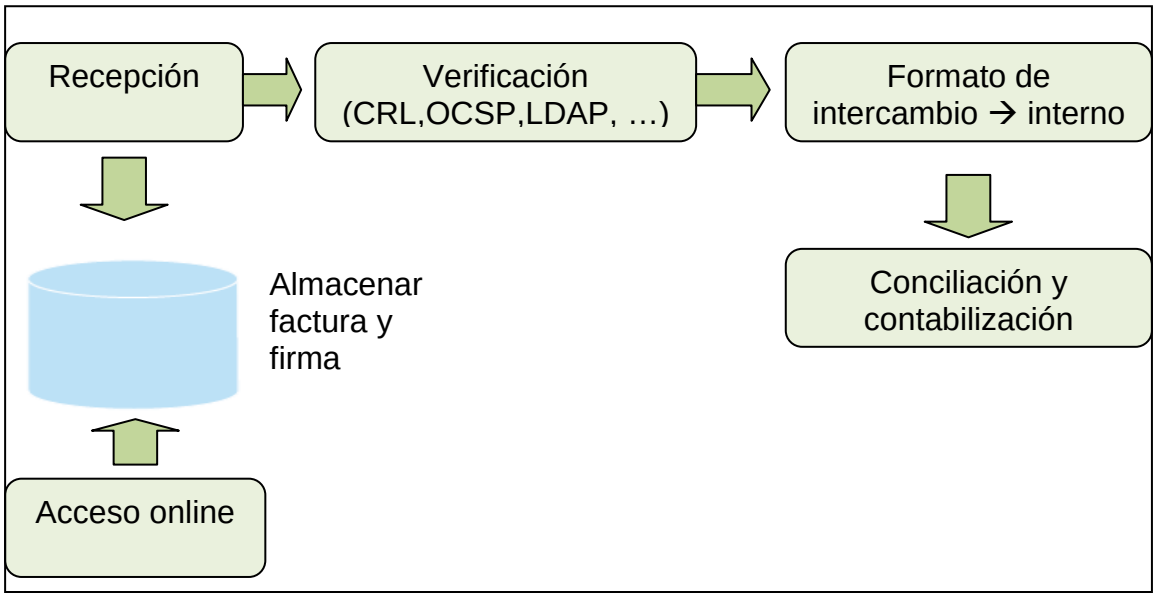
En este tipo de proyectos la mayor complejidad suele residir en el proceso de firma electrónica. Podemos hacer uso de Prestadores de Servicios de Sellado de Tiempo (TSA) y Prestadores de Servicios de Validación (VA).



(Fuente: propia)

Figura 5. Proceso de envío de factura electrónica.

Por otro lado, desde el punto de vista de la **recepción** la complejidad radica en la recepción de formatos diferentes y de los prestadores de los certificados electrónicos. En la Figura 6 se puede observar el proceso para la recepción.



(Fuente: propia)

Figura 6. Proceso de recepción de factura electrónica.

2.2 Formatos de Factura Electrónica

El formato de la factura es todo aquel fichero donde se almacenan los datos que conforman la factura antes de aplicarle la firma electrónica. Podría tratarse de un PDF, un RTF, EXCEL, HTML, XML, etc.

De esta forma, un fichero XML permite definir nombres de campos y dotarles de significado, información que es compartida por el emisor y receptor de dicha información. En este sentido se encuentra OASIS (Organization for the Advancement of Structured Information Standards) con sus propuestas de EBXML(Electronic Business XML) y UBL (Universal Business Language) [46] como se verá a continuación.

Para facilitar el conocimiento de la colocación así como para que el formato de la información sea conocida tanto por el emisor como por el receptor del fichero, se crean formatos estructurados. A continuación, se detallan los formatos que se están utilizando actualmente:

FACTURAE (antiguo XML-CCI y AEAT-CCI) [44]

El Centro de Cooperación Interbancaria (CCI) es una asociación profesional sin ánimo de lucro que se constituyo como consecuencia de la acción conjunta del colectivo de entidades de depósito (Bancos, Cajas de ahorros y Cooperativas de crédito). Entre las acciones promovidas, está la estandarización de la mensajería de datos entre las entidades y de estas con sus proveedores y cliente y la estandarización de un modelo de Factura Electrónica, entre las entidades de depósito y otros agentes relacionados con las mismas.

Desde el 16 de octubre de 2007, el formato AEAT-CCI ha sido adoptado por la Administración General del Estado, pasando a llamarse “Facturae”. A partir del esquema AEAT-CCI versión 2.0 se ha obtenido el esquema Facturae versión 3.0.

UN/EDIFACT [45]

El origen surge en el seno de Naciones Unidas (UNECE, United Nations Economic Commission for Europe). El fin de UNECE es proveer un forum para la comunicación entre estados; aportar instrumentos jurídicos internacionales

sobre comercio, transporte y el medioambiente; y suministrar fuentes estadísticas para el análisis económico y del entorno.

Se pueden observar las adaptaciones AECOC o GS1 Spain, Odette/ANFAC, ANSI (X12 Principalmente en Estados Unidos).

Existen diferentes versiones por cada documento y utilización según el país. Por ejemplo, la factura en España se usa el INVOIC D93A y, en otros países como Francia y Portugal, se utiliza INVOIC D96A. Se está trabajando para el uso generalizado para toda Europa con versiones como INVOIC D01B.

El uso del formato de factura EDIFACT está íntimamente relacionado con la lógica en la cadena de suministro del sector donde se aplica: Pedido (Orders), Albarán (Desadv), Confirmación Recepción (Recadv) y Factura (Invoic). En España su uso es masivo, más de 5000 empresas lo usan para su actividad continua en el sector de la distribución.

Se han desarrollado algunas acciones para el uso del XML con el EDIFACT (GS1 XML). De hecho las Naciones Unidas han desarrollado UN/CEFACT que permite el uso del XML para la gestión de documentos como la factura (equipara el uso del INVOIC EDI al XML).

UBLXML [46]

OASIS es un consorcio internacional sin ánimo de lucro que dirige el desarrollo, convergencia y adopción de estándares para el e-business. OASIS fue fundada en 1993 bajo el nombre SGML Open como consorcio de vendedores y usuarios relacionados con el desarrollo de guía para la interoperabilidad de productos que soportan el Estándar Generalized Markup Language (SGML). Como referencia el HTML está basado en SGML y es propulsor del uso de SML.

Existen usos en la administración general (proyecto CODICE- componentes y documentos interoperables para la contratación Electrónica) y administración autonómica (Factura solidaria en Baleares).

2.3 Sistema EDI

El EDI implica la transferencia electrónica de ordenador a ordenador de datos comerciales o administrativos que se estructuran en un mensaje normalizado que permite su procesado automático.

En sentido estricto, los sistemas EDI tradicionales utilizan el formato **EDIFACT [45]** para codificar los mensajes y redes de valor añadido (VAN: Value Added Network) para transmitirlos y distribuirlos.

En un sentido más amplio, son sistemas de intercambio de datos que permiten al destinatario de la transmisión que pueda utilizar un medio de recepción informatizado que le evite el proceso de reintroducir los datos de tales mensajes en los programas informáticos que los tratan, como sucedería por ejemplo, en el caso de las facturas transmitidas en papel o en formatos no directamente interpretables por el ordenador destino.

Los sistemas EDI garantizan la autenticidad e integridad del mensaje de dos formas:

- Mediante una firma electrónica avanzada de acuerdo con lo dispuesto en el artículo 2.2 de la Directiva 1999/93/CE del Parlamento Europeo y del Consejo, de 13 de diciembre de 1999, por la que se establece un marco comunitario para la firma electrónica, basada en un certificado reconocido y creado mediante un dispositivo seguro de creación de firmas, de acuerdo con lo dispuesto en los apartados 6 y 10 del artículo 2 de la mencionada Directiva.

- Mediante un intercambio electrónico de datos (EDI), tal como se define en el artículo 2 de la Recomendación 1994/820/CE de la Comisión, de 19 de octubre de 1994, relativa a los aspectos jurídicos del intercambio electrónico de datos, cuando el acuerdo relativo a este intercambio prevea la utilización de procedimiento que garanticen la autenticidad del origen y la integridad de los datos.

Cuando se emplea la firma electrónica en EDI, se utilizan las posibilidades del estándar a través de dos variantes:

- Cabeceras y pies de seguridad (USH “security header” – UST “security tráiler”), que permiten firmar cada mensaje del intercambio.

- Mensaje AUTACK que permite firmar todos los mensajes de un intercambio.

Aunque cuando se utiliza EDI en el intercambio electrónico de facturas no es necesario el uso de la firma electrónica, éste está previsto y en España se está adoptando de forma generalizada.

2.4 Gramática UN/EDIFACT

La gramática UN/EDIFACT es un estándar mundial aprobado por las Naciones Unidas con las normas relativas al Intercambio Electrónico de Datos para la Administración, Comercio y Transporte.

Las reglas que permiten estructurar un mensaje EDI, constan de una gramática o sintaxis y de un vocabulario definidos en la norma ISO 9735.

Los mensajes EDIFACT están estructurados jerárquicamente, por una serie de segmentos de información que siguen unas reglas de sintaxis definidas. Estos segmentos a su vez están compuestos por elementos de datos simples y complejos.

Los principales elementos estructurales que componen un mensaje EDIFACT son los siguientes:

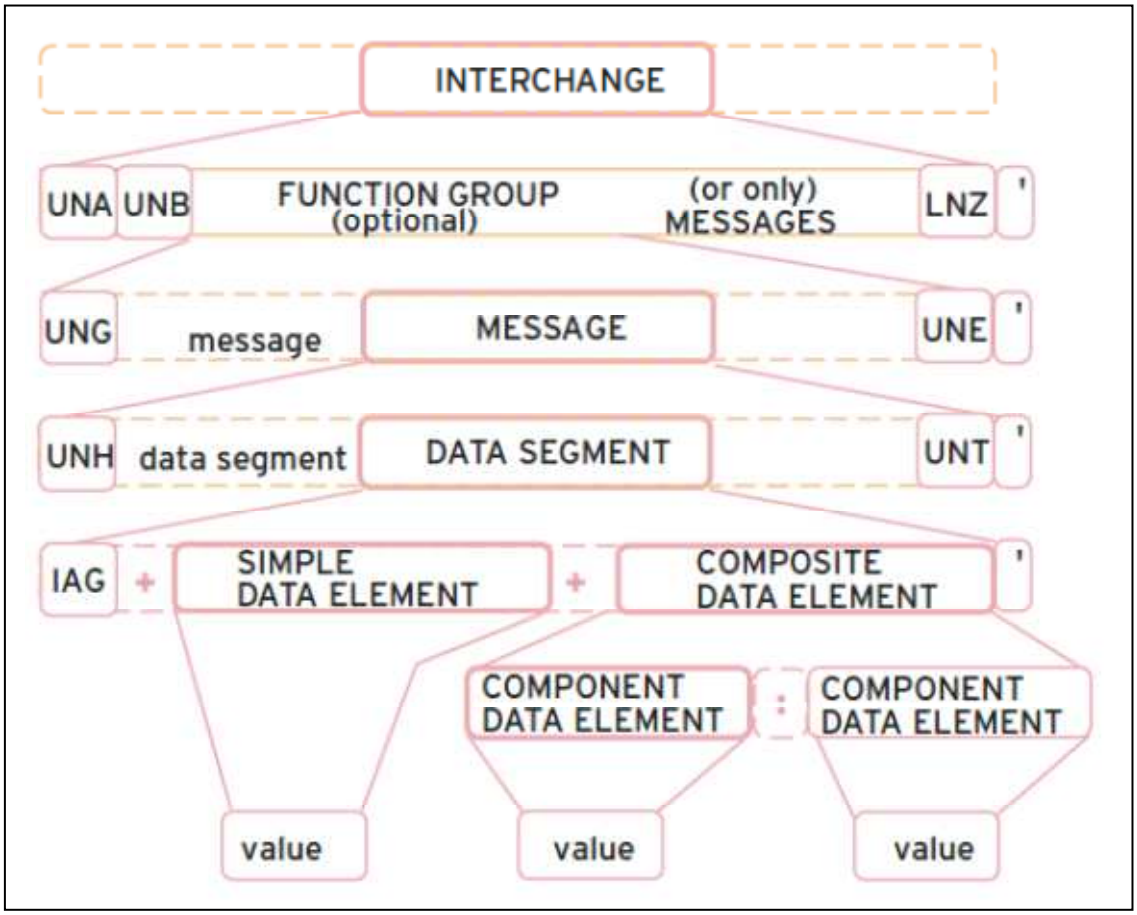
- Intercambio.
- Mensaje
- Segmento
- Elemento de datos

El *intercambio* es el elemento estructural de nivel más alto en un mensaje EDI. En él se definen elementos como la dirección del emisor y el receptor del mensaje. Las partes de un intercambio se delimitan por encabezados y finalizadores de control, los segmentos de sobre.

Existen encabezados y finalizadores a lo largo de un mensaje EDI que pueden ser opcionales como UNA o UNG.

Los *mensajes* (delimitados por el encabezado UNH y finalizador UNT) contienen segmentos que forman los datos del mismo. A su vez, cada *segmento* constan de uno o más *elementos de datos*, comenzando por un identificador de segmento de datos de tres caracteres y finaliza con un terminador de segmentos, de forma predeterminada el apóstrofo (‘).

En la Figura 7 podemos ver una estructura de intercambio EDI.



(Fuente: ISO9735)

Figura 7. Estructura de Intercambio en EDI.

Como se puede ver en el Anexo A, los mensajes EDIFACT están estructurados jerárquicamente por una serie de segmentos de información siguiendo reglas de sintaxis definidas. A continuación, presentamos un ejemplo de traducción de una factura al formato EDIFACT.

En la Figura 8 se muestra un ejemplo de factura y en la Figura 9 se indica el marcado de segmentos del formato EDIFACT sobre la factura de la figura anterior.

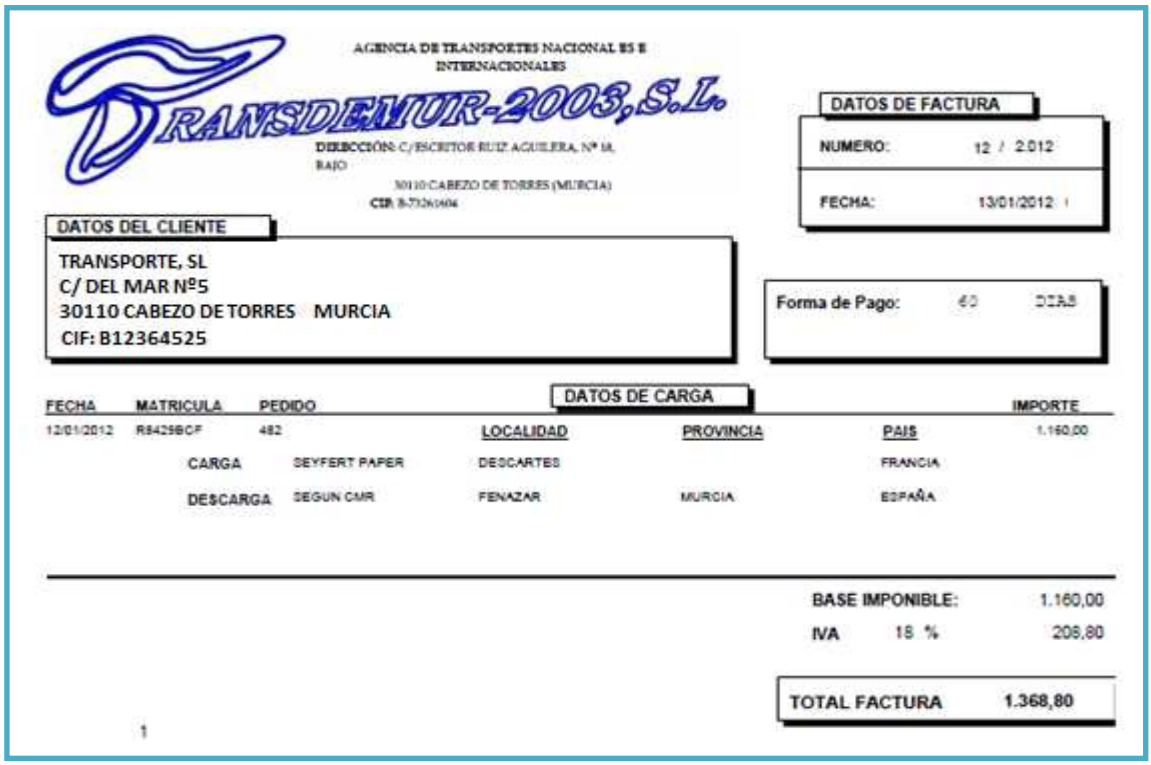
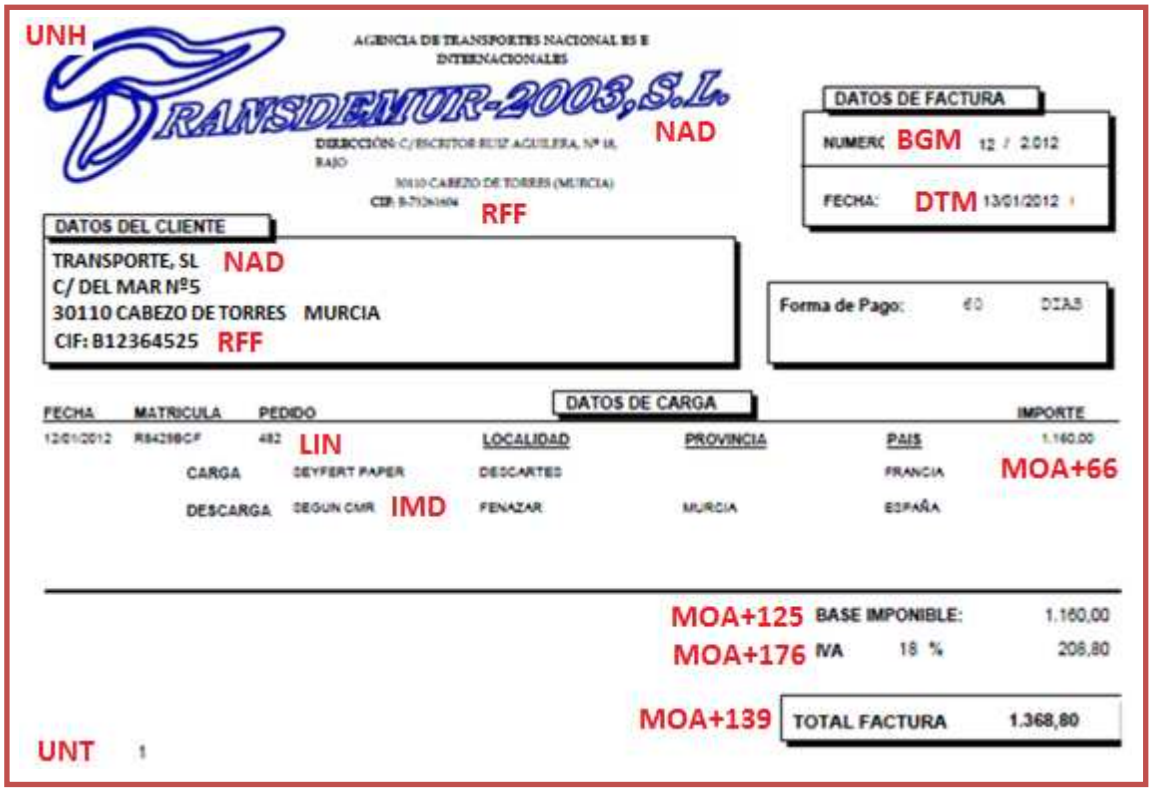


Figura 8. Ejemplo de factura. (Fuente: propia)



(Fuente: propia)

Figura 9. Marcado segmentos EDIFACT.

Los segmentos que hemos marcado son los siguientes:

- UNA: Inicio del intercambio.
- UNH: Cabecera de mensaje.
- BGM: Identificación del tipo y del número de factura.
- DTM: Fecha de generación de la factura.
- NAD: Utilizado para identificar al emisor y receptor de la factura.
- RFF: Utilizado para indicar la identificación fiscal del emisor y receptor de la factura.
- LIN: Línea del artículo.
- IMD: Descripción del artículo.
- MOA+ID: Importe de la línea del artículo, de la base imponible, IVA y total de la factura.
- UNS: Separador de secciones.
- UNT: Final del mensaje.
- UNZ: Final del intercambio.

Por último, mostramos el fichero EDIFACT resultado de generar el fichero electrónico a partir de la factura anterior.

```
UNA:+,? '
UNB+UNOA:2+EMISOR+DESTINATARIO+990802:1557+9908021557'
UNH+INVOIC001+INVOIC:D:93A:UN'
BGM+380+12/2012'
DTM+137:120113:101'
NAD+EM++Agencia Transdemur-2003 SL++C/Escritor Ruiz Aguilera, 18+Murcia++30110'
RFF+VA:ESB73261604'
NAD+RE++TRANSPORTE SL++C/DEL MAR Nº5+MURCIA++30110'
RFF+VA:ESB12364525'
LIN+1+482'
IMD+F+M+:::Viaje Descartes/Francia a Murcia/España'
QTY+47:1'
MOA+66:1160'
UNS+S'
MOA+125:1160'
MOA+176:280,80'
MOA+139:1440,80'
UNT+16+INVOIC001'
UNZ+1+9908021557'
```

(Fuente: propia)

Figura 10. Segmentos EDIFACT.

2.5 Mensajes EDI

Como se indicó anteriormente existen otro tipo de ficheros que pueden incrementar la eficiencia de las operaciones realizadas en el seno de una empresa. Los ficheros más utilizados son los siguientes:

- **ORDER (Pedido):** Documento a través del cual se solicita un producto/mercancía o la prestación de un servicio.

- **DESADV (Albarán):** Documento que acredita la entrega de un producto o la prestación de un servicio. También conocido como aviso de expedición. Este documento sirve al comprador para validar que la entrega se corresponde con lo solicitado en el pedido. Adicionalmente al vendedor le puede valer como constancia de que la entrega se ha realizado.

- **RECADV** (Confirmación de entrega): Confirmación de recepción. A través de este documento el comprador además de confirmar al vendedor la recepción del producto solicitado, indica posibles anomalías en la entrega.

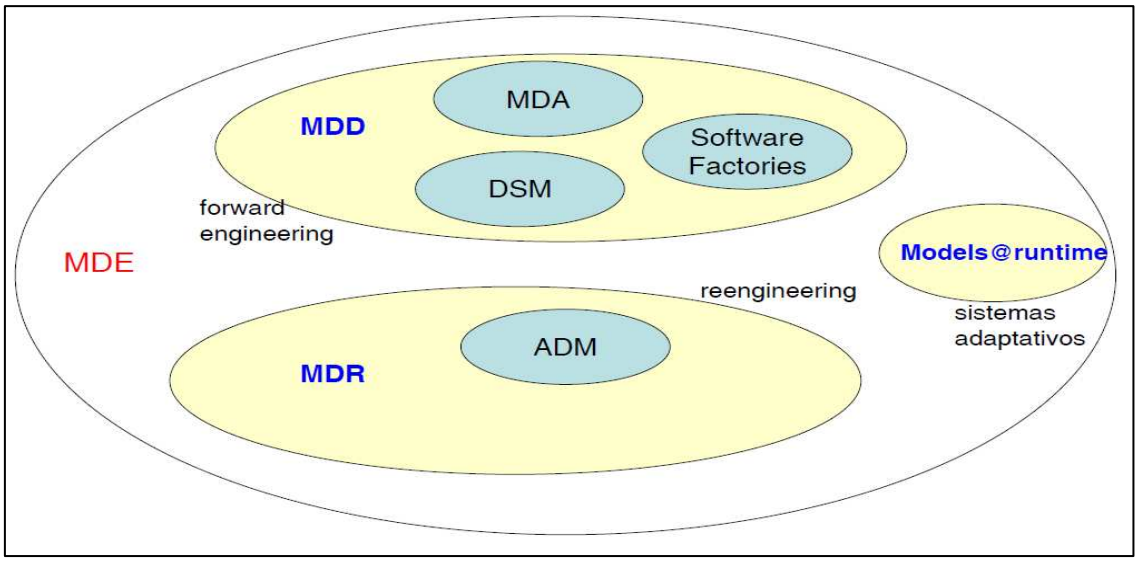
- **INVOIC** (Factura) Es el documento generado en formato electrónico, que refleja finalmente la realización de la transacción comercial entre comprador y vendedor, así como sus obligaciones de pago y de liquidación de impuestos.

3 Fundamentos de la Ingeniería Dirigida por Modelos

En esta sección se presentan los fundamentos de la Ingeniería Dirigida por Modelos (MDE) a partir de los capítulos 1 y 3 del libro "Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas" [47].

La *Ingeniería Dirigida por Modelos* (*Model-Driven Engineering*, MDE) es una nueva disciplina dentro de la ingeniería del software que se ocupa de la utilización sistemática de modelos del software para mejorar la productividad y algunos aspectos de la calidad del software como el mantenimiento y la interoperabilidad entre sistemas. La visión de la construcción del software que propugna esta nueva disciplina ha mostrado su potencial para dominar la complejidad arbitraria del software al proporcionar un mayor nivel de abstracción y elevar también el nivel de automatización [1]. Estos beneficios no sólo son aplicables en la construcción de nuevas aplicaciones sino que son también útiles en la reingeniería de aplicaciones *legacy* y en la configuración dinámica de sistemas en tiempo de ejecución [2].

En realidad, detrás del término MDE no hay un solo paradigma de desarrollo de software, como sucedía con la tecnología de objetos, sino que se pueden distinguir diferentes paradigmas entre los que destaca Model-Driven Architecture (MDA) [3], *Desarrollo Específico del Dominio* [4], *Modernización basada en modelos* y Models@runtime [5]. Para englobar al conjunto de paradigmas enfocados a la construcción de aplicaciones partiendo de cero (ingeniería directa) se utiliza en español el término *Desarrollo de Software Dirigido por Modelos* (en español se utiliza el acrónimo DSDM y en inglés está muy extendido MDD como acrónimo de *Model-Driven Development*). La Figura 11 muestra cómo MDE engloba diferentes paradigmas.



(Fuente: [47])

Figura 11. Paradigmas MDE y terminología

3.1 Principios básicos del MDE

Todos los paradigmas MDE comparten unos mismos principios básicos que podemos considerar como los elementos fundamentales de esta disciplina [6, 7]: i) un modelo representa total o parcialmente un aspecto de un sistema software; ii) estos modelos son representados con lenguajes de modelado o lenguajes específicos del dominio (DSL); iii) un metamodelo es empleado para representar formalmente un DSL; iv) la automatización es normalmente conseguida a través de la traducción de los modelos a código mediante transformaciones de modelos.

Como es bien conocido, un *modelo* es una simplificación de la realidad, como resultado de un proceso de abstracción, y ayuda a comprender y razonar sobre esa realidad. Un modelo oculta ciertos detalles para mostrar aquellos relevantes para cierto propósito. Los modelos son expresados mediante alguna notación que depende de su propósito y destinatarios. En el caso del software, un modelo es una descripción de un aspecto de un sistema software escrita en un lenguaje bien definido. Los modelos software permiten especificar aspectos tales como los requisitos, la estructura, el comportamiento o el despliegue de un sistema. Como es lógico, si un modelo debe ser interpretado por una máquina para generar código o controlar un sistema, deberá estar expresado con una notación descrita formalmente, por ejemplo mediante un metamodelo.

El término *lenguaje específico de dominio* (DSL, *Domain Specific Language*) se está imponiendo al de *lenguaje de modelado* para referirse a las notaciones usadas para expresar modelos, aunque éste último se usa con frecuencia cuando se trata de lenguajes gráficos o visuales. Un DSL consta de tres elementos principales: la *sintaxis abstracta* que define los conceptos del lenguaje y las relaciones entre ellos, así como las reglas que establecen cuando un modelo está bien formado; la *sintaxis concreta* que establece la notación, y la *semántica* que normalmente es definida a través de la traducción a conceptos de otro lenguaje destino cuya semántica está bien definida (por ejemplo, un lenguaje de programación como Java). La separación entre sintaxis concreta y sintaxis abstracta es un elemento central de MDE que posibilita definir diferentes notaciones para una misma sintaxis abstracta.

En MDE, la sintaxis abstracta de un DSL se define mediante un *metamodelo*, esto es, un modelo conceptual del DSL, el cual es expresado con un lenguaje de metamodelado, junto con un conjunto de reglas que definen restricciones adicionales para que un modelo se considere bien formado. Estas reglas se suelen expresan con el lenguaje OCL (o alguna variante) [8]. El lenguaje de metamodelado más utilizado es Ecore, el cual es el elemento central del *Eclipse Modeling Framework* (EMF) que proporciona la infraestructura básica del proyecto de Eclipse para crear herramientas y soluciones MDE [9]. Realmente, Ecore es un subconjunto de MOF [10] que fue el lenguaje de metamodelado creado por OMG (Object Management Group) para definir formalmente UML (Unified Modeling Language). Ecore proporciona al creador de un DSL los conceptos propios del modelado orientado a objetos para que describa el lenguaje: clases para expresar los conceptos, atributos para expresar las propiedades, agregación y referencias entre clases para expresar las relaciones entre conceptos y generalización para expresar que un concepto es una especialización de otro.

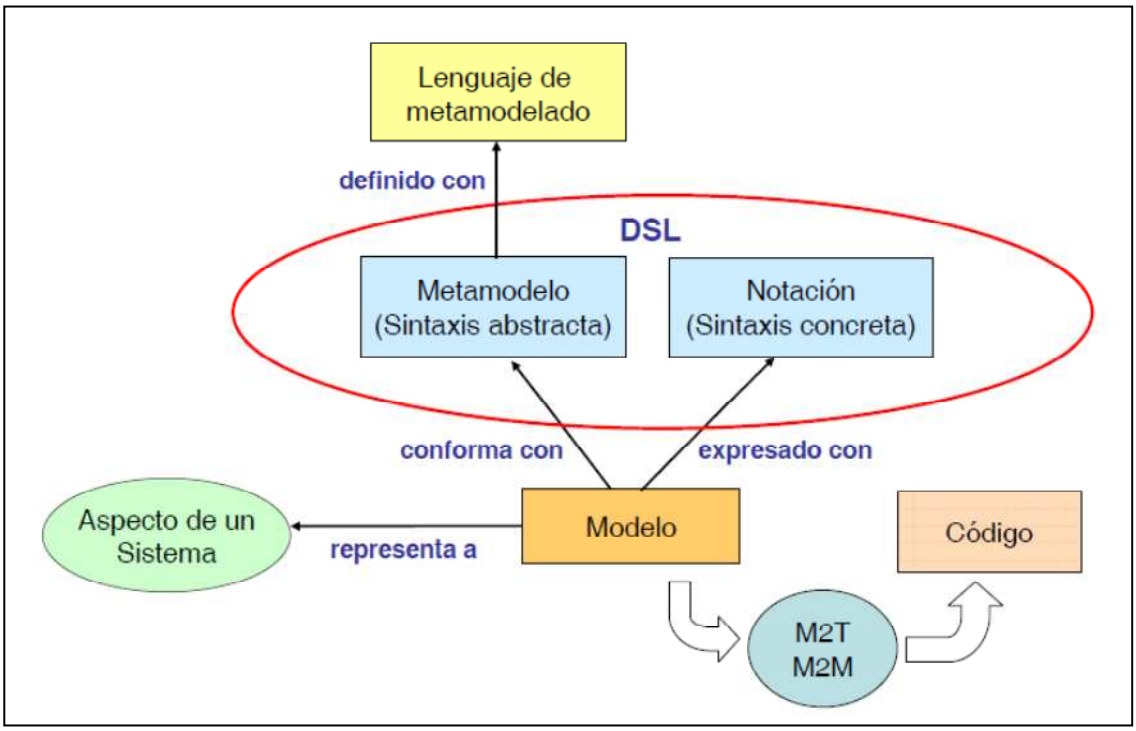
La notación o sintaxis concreta puede ser textual o gráfica y en el contexto de MDE han aparecido un buen número de herramientas basadas en el metamodelado que permiten crear tanto DSLs textuales [11,12] como gráficos [13,14] y la tendencia es a disponer de herramientas que permitan crear DSLs con una naturaleza híbrida que combine texto y gráficos. Hay que subrayar que

en ocasiones no es necesario asociar una notación a un metamodelo, dado que los modelos serán generados como parte de una etapa intermedia hacia la obtención de los artefactos software finales deseados.

Con DSLs es posible expresar las soluciones software a un nivel de abstracción más alto que con la utilización de los lenguajes software normalmente utilizados: lenguajes de programación de propósito general, XML, HTML, WDSL, etcétera. Aunque los modelos han sido usados desde hace décadas como forma de documentación o de razonar sobre el código, en MDE se han convertido en un artefacto clave en el desarrollo de software, de modo que son parte de la solución, de la misma forma que código Java o unos ficheros de configuración XML [15].

Un aumento en el nivel de abstracción debe ir acompañado de un aumento del nivel de automatización para que realmente sea efectivo. En el caso del MDE esto se consigue normalmente mediante la generación automática de código a partir de los modelos creados, ya sea directamente a través de *transformaciones modelo a texto* (m2t) o de forma indirecta mediante la definición de modelos intermedios generados con *transformaciones modelo a modelo* (m2m) que facilitan la conversión de modelos de alto nivel de abstracción en el código final. ATL [16] y QVT [17] (tanto QVT relational como QVT operational) son los lenguajes de transformación m2m más extendidos, mientras que entre los lenguajes de transformación m2t podemos destacar Acceleo [18] que implementa el estándar Mof2Text de OMG [19], MofScript [20] y XPand [21].

La Figura 12 muestra un esquema en el que se relacionan todos los conceptos introducidos.



(Fuente: [47])

Figura 12. Relaciones entre modelo, metamodelo y meta-metamodelo.

3.2 Arquitectura de cuatro niveles

Según se ha explicado, un modelo es expresado con un DSL cuya sintaxis abstracta es representada mediante otro modelo (metamodelo) que representa sus construcciones de forma independiente a su notación. A su vez, dado que un metamodelo es también un modelo, un metamodelo es expresado también con un lenguaje que se denomina lenguaje de metamodelado (por ejemplo Ecore). En este punto se podría plantear con qué lenguaje se define el lenguaje de metamodelado y así sucesivamente. Dado que el lenguaje de metamodelado también tiene su sintaxis abstracta y concreta, la cuestión es cómo definir el metamodelo de esta sintaxis abstracta, que realmente sería un meta-metamodelo (un modelo de un metamodelo, o lo que es lo mismo, un modelo de un modelo de un modelo).

La noción de *arquitectura de cuatro niveles* es usada tradicionalmente para establecer la relación entre modelos y metamodelos [22, 23, 24] y responder a la cuestión planteada. En ella se establecen los siguientes niveles:

M0. Nivel de datos del usuario o del mundo real.

Caracteriza los del mundo real que son manipulados por el software. Utilizamos el término “objeto” en un sentido amplio para significar también “procesos”, “conceptos”, “estados”, etcétera.

M1. Nivel de modelo.

Caracteriza a los modelos que representan los datos del nivel M0.

M2. Nivel de metamodelo.

Caracteriza a los modelos (metamodelos) que describen los modelos del nivel M1.

M3. Nivel de meta-metamodelo.

Caracteriza a los modelos (meta-metamodelos) que describen los metamodelos del nivel M2.

De acuerdo a esta arquitectura, un meta-metamodelo se describe con los mismos conceptos que se desea describir (definición circular). De este modo, la sintaxis abstracta de Ecore se define con el propio Ecore, es decir, los conceptos del propio Ecore son descritos en el meta-metamodelo usando clases, atributos, referencias y agregación.

Cuando se llega a un nivel “meta-meta” las cosas se complican, pero un paralelismo con las gramáticas y los lenguajes de programación ayuda a comprender las relaciones entre modelos, metamodelos y meta-metamodelos. La ejecución de una aplicación, por ejemplo un programa Java (sería un proceso del mundo real), es determinada por un programa que es conforme a la gramática del lenguaje, la cual a su vez es definida por un meta-lenguaje como EBNF. En este caso, M0 correspondería a la ejecución de programas, M1 a los programas, M2 a las gramáticas de los lenguajes y M3 a los metalenguajes como EBNF. Como ilustra la Figura 13, un paralelismo similar se podría establecer con otros dominios como XML, las ontologías, las bases de datos o la propia programación orientada a objetos en el caso de lenguajes que soportan meta-programación o reflexión, como es el caso de Ruby o Java.

Este paralelismo se ha utilizado para definir lenguajes de extracción de modelos a partir de diversos artefactos: código fuente [25], tablas de una base de datos relacional [26] y APIs [27].

Metamodelado	XML	Ontologías	Bases de datos	Lenguajes OO con reflexión
Meta-metamodelo (Ecore)	XML Schema	OWL	E/R	Metaclases
Metamodelo	Definición de un esquema	Ontología	Esquema relacional	Clases
Modelo	Documento XML	Base de conocimiento	Datos en BDR	Objetos
Datos mundo real	Datos mundo real	Conocimiento mundo real	Datos mundo real	Datos mundo real

(Fuente: [47])

Figura 13. Arquitectura de cuatro niveles en diferentes dominios.

La Figura 14 visualiza la arquitectura de cuatro niveles en el caso de un metamodelo de máquinas de estados, creado con Ecore, con el que se ha creado un modelo que representa una máquina expendedora de snacks.

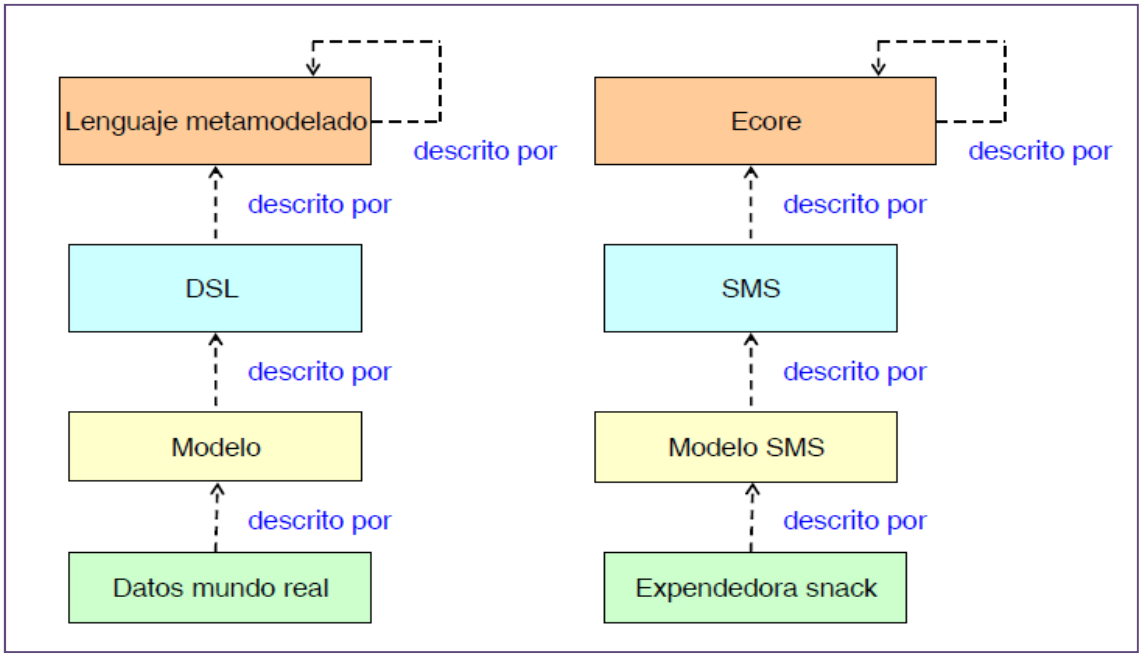
Nivel	Descripción	Elementos
M3	**Meta-metamodelo de Ecore**	EClass, EAttribute, EReference, EDataType, ..
M2	**Metamodelo** "Máquina de Estados"	MaquinaEstado, Estado, Transicion,..
M1	**Modelos** del sistema real conformes al metamodelo "Máquinas de estado"	Inicio, Seleccion, DevolucionDinero, ..
M0	el **sistema real**, una máquina expendedora de snacks	Sus componentes y comportamiento real

(Fuente: [47])

Figura 14. Arquitectura para el ejemplo de modelo de máquina de estados.

En la arquitectura de cuatro niveles es posible establecer dos relaciones entre los elementos de un nivel y del nivel superior: “instancia-de” o “descrito-por”. En primer caso, los elementos de M1 son instancias de los elementos de M2 (su metamodelo). Por ejemplo, el estado “Seleccion” sería una instancia de la metaclase Estado y una transición del estado Selección al estado DevolucionDinero sería una instancia de la metaclase Transicion. A su vez, los elementos del metamodelo son instancias de los elementos de M3 (su meta-metamodelo). Por ejemplo, Estado sería una instancia de EClass (metaclase de Ecore), y las dos referencias entre Estado y Transicion serían instancias de EReference. Por último, los elementos de M3 son instancias de sí mismos. Por ejemplo, EClass es una instancia de EClass. Los elementos de M0, lógicamente no son instancias de M1, sino que se entiende que los elementos de M1 son “representaciones” de las “cosas” del mundo real.

En cuanto a la relación “descrito por”, los objetos del mundo real (nivel M0) son descritos por modelos del nivel M1; éstos, a su vez, son descritos con el lenguaje proporcionado por el nivel M2, que a su vez es descrito por el lenguaje proporcionado en el nivel M3. Ahora se utiliza la idea de lenguaje y se entiende que se trata de una relación ontológica. La Figura 15 ilustra este tipo de relación con el ejemplo de la máquina de estados simple supuesto que llamamos SMS al metamodelo de máquinas de estado.



(Fuente: [47])

Figura 15. Arquitectura de cuatro niveles: relación “descrito por”

3.3 Arquitectura de metamodelado

De acuerdo a los fundamentos de MDE expuestos en este capítulo, la construcción de una solución MDE para la creación o evolución de software requiere de una infraestructura que soporte tres operaciones básicas que permitan la definición de: metamodelos, DSLs y transformaciones de modelos. Podemos encontrar herramientas que soportan algunas de estas operaciones y herramientas que las soportan todas. Todas esas herramientas, sea cual sea su tipo, se basan en una arquitectura de metamodelado que es una implementación del concepto teórico de arquitectura de cuatro niveles explicado anteriormente, junto con algunos mecanismos básicos para facilitar el manejo de modelos y metamodelos. Por tanto, una arquitectura de metamodelado consta de:

- Un lenguaje de metamodelado.
- Un lenguaje para escribir reglas tipo OCL para la validación.
- Soporte para la serialización de modelos y metamodelos en formato XML según el estándar XMI [32].
- Uno o más *mappings* a lenguajes de programación.
- Un repositorio para almacenar modelos y metamodelos.

La arquitectura de metamodelado de Eclipse se ha convertido en la principal plataforma para aplicar MDE. *Modeling Project* es el nombre del proyecto Eclipse destinado a ofrecer soporte a la construcción de herramientas y soluciones MDE. Todos estos proyectos se articulan en torno al *Eclipse Modeling Framework* (EMF) que proporciona los mecanismos básicos para manejar (meta)modelos, esto es, implementa lo que hemos denominado una arquitectura de metamodelado. EMF proporciona los siguientes elementos:

- Ecore como lenguaje de metamodelado.
- Editores para crear metamodelos en forma de diagrama de clases, árbol de contención y como texto Emfatic [33].
- Un framework para validación de modelos que proporciona soporte para manejo de reglas de validación y que soporta reglas expresadas en OCL

y Java (aunque ahora la forma más cómoda de asociar reglas a un metamodelo es el DSL textual OCLinEcore [34]).

- Una implementación de XMI para la serialización de (meta)modelos.
- Un *mapping* a Java que establece cómo representar (meta)modelos en código Java.
- El repositorio de modelos CDO (*Connected Data Objects*).
- Otras utilidades como EMF Compare para comparar (meta)modelos y Model Query que facilita la consulta de los elementos de un metamodelo a través de un API o consultas tipo SQL.

3.4 Tecnología MDE en el proyecto

El entorno utilizado para construir la solución MDE para el problema que se plantea en este proyecto sobre el intercambio de ficheros electrónicos es Eclipse y en particular el subproyecto EMF que proporciona los mecanismos básicos para implementar la arquitectura de metamodelo.

3.4.1 Eclipse

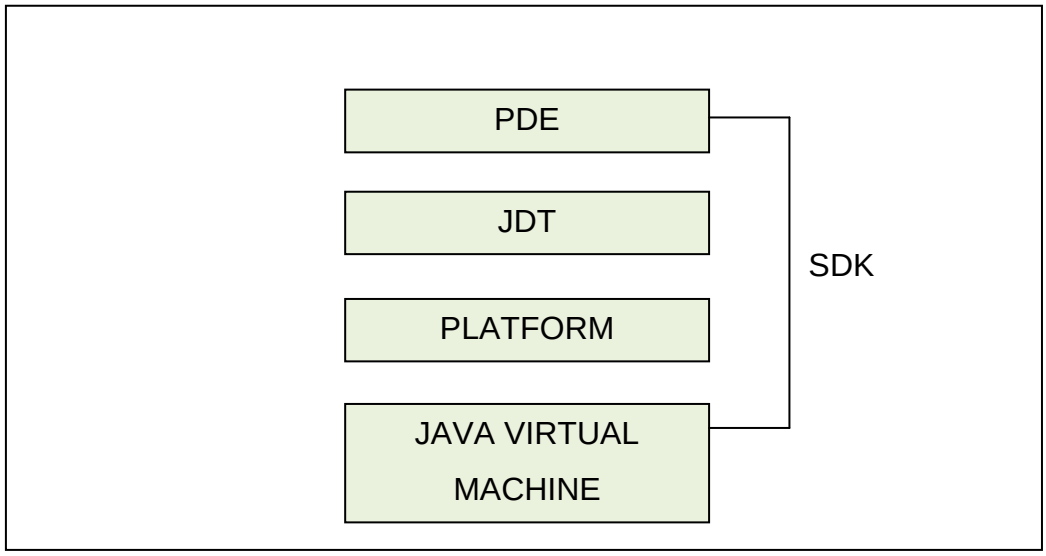
Eclipse es un entorno de desarrollo integrado de código abierto multiplaforma compuesto por un núcleo que incluye un framework genérico para la integración de herramientas y un entorno de desarrollo Java. Los proyectos en Eclipse son implementados en Java y soporta múltiples sistemas operativos como Windows, Mac, OSX y Linux.

El proyecto Eclipse se divide en numerosos proyectos entre los que se pueden destacar:

- *Eclipse Project:* contiene los componentes necesarios para el desarrollo mediante Eclipse. Estos componentes son fijos y pueden ser descargados mediante *Eclipse Software Development Kit* (SDK). Proporcionan una plataforma de desarrollo de herramientas integrada, robusta y completa. El framework Eclipse esta implementado en Java pero puede ser usado para implementar herramientas de desarrollo para otros lenguajes como C++, XML, etc. Este proyecto se subdivide en tres subproyectos:

- Equinox y la Plataforma: es el núcleo básico o el *kernel* de Eclipse. Emplea una estructura abierta de *plug-ins* que permite expandir las capacidades de la plataforma base. Provee a las capas superiores de servicios tales como editor de código fuente, infraestructura para depuración independiente del lenguaje de programación, soporte de versiones, búsqueda, compilación, asistentes para creación, etc.
- JDT *(Java Development Toolkit):* añade a la plataforma un IDE de Java completamente equipado, incluyendo: editor, compilador, depurador y ejecución de código Java.
- PDE *(Plug-in Development Environment):* es un conjunto de herramientas diseñadas para ayudar al desarrollador de Eclipse en las tareas de desarrollo, prueba, depuración, construcción y distribución de *plug-ins.*

La arquitectura de la plataforma Eclipse se puede ver en la Figura 16. Los tres subproyectos proporcionan todo lo necesario para extender el framework y desarrollar herramientas basadas en Eclipse.



(Fuente: propia)

Figura 16. Estructura plataforma Eclipse.

- *Eclipse Tools Project:* formado por amplio conjunto de herramientas basadas en Eclipse como pueden ser herramientas que permiten trabajar con otros lenguajes como C/C++, COBOL y PHP. EMF comenzó

como parte de *Tool Project* antes de que se formara un proyecto de modelado. Por tanto, los *Tool Project* proporcionan un punto común entre los desarrolladores para minimizar la duplicidad y promover la interoperabilidad entre los diversos tipos de herramientas.

- *Modeling Project:* es el punto central que promueve el desarrollo dirigido por modelos en Eclipse. Su núcleo es EMF, el framework utilizado para modelar. Otros subproyectos de modelado pueden ser construidos sobre EMF permitiendo la transformación de modelos, integración de base de datos, un editor gráfico, etc.

- *Eclipse Technology Project:* su misión es proporcionar nuevos canales de comunicación a desarrolladores, profesores y educadores para participar en la evolución de Eclipse. Está organizado en tres proyectos relacionados:
    - *Research:* investigación en dominios relacionados con Eclipse, tales como lenguajes de programación, herramientas y entornos de desarrollo.
    - *Incubators:* son pequeños proyectos que añaden capacidades al software base de Eclipse.
    - *Education:* estos proyectos se centran en el desarrollo de material de ayuda.

Por tanto, Eclipse es un framework para la construcción de *IDEs.* Eclipse define la estructura básica de un IDE permitiendo ir agregando *plug-in* que vayan constituyendo el IDE final.

Un *plug-in* es la unidad mínima de funcionalidad de Eclipse que puede ser distribuida de manera separada. Dependiendo de la complejidad de la herramienta, estará constituida por un *plug-in* o por varios. Excepto el *Platform* Runtime, el resto de la funcionalidad de la plataforma Eclipse está implementada como *plug-ins.* Estos *plug-ins* suponen una de las principales ventajas de esta plataforma ya que al ser agregados permiten ir componiendo la funcionalidad final.

Cada *plug-in* tiene un fichero denominado manifiesto (*manifiest)* en el cual se declaran sus interconexiones con otros *plug-ins.* En este fichero un *plug-in* declara sus puntos de extensión y las extensiones para los puntos de extensión de otros *plug-ins.* Este modelo permite una integración sin dificultad ni conflictos.

Al iniciar la Plataforma se descubren de manera dinámica el conjunto de *plug-ins* disponibles, se leen sus ficheros *manifiest*, y se construye en memoria un registro de *plug-ins.*

Al trabajar con el entorno Eclipse se hace uso del *Workspace*, el espacio de trabajo. Se trata del directorio donde se almacenan los archivos de trabajo de un usuario y sobre los que actúan las diferentes herramientas instaladas en la Plataforma. En un espacio de trabajo es posible almacenar uno o más proyectos en los diferentes directorios que el usuario haya seleccionado.

El conjunto de proyectos, archivos y carpetas que son generados constituyen los recursos. Estos recursos se organizan en el entorno Eclipse mediante una estructura de árbol.

Por otro lado, el *Workbench* implementa el aspecto visual que permite al usuario navegar por los recursos y utilizar las herramientas integradas. El *Workbench* se puede dividir en dos categorías: editores y vistas.

Los editores permiten al usuario abrir, editar y guardar objetos mientras que las vistas proporcionan información sobre aquellos objetos con los que el usuario está trabajando en el *Workbench.*

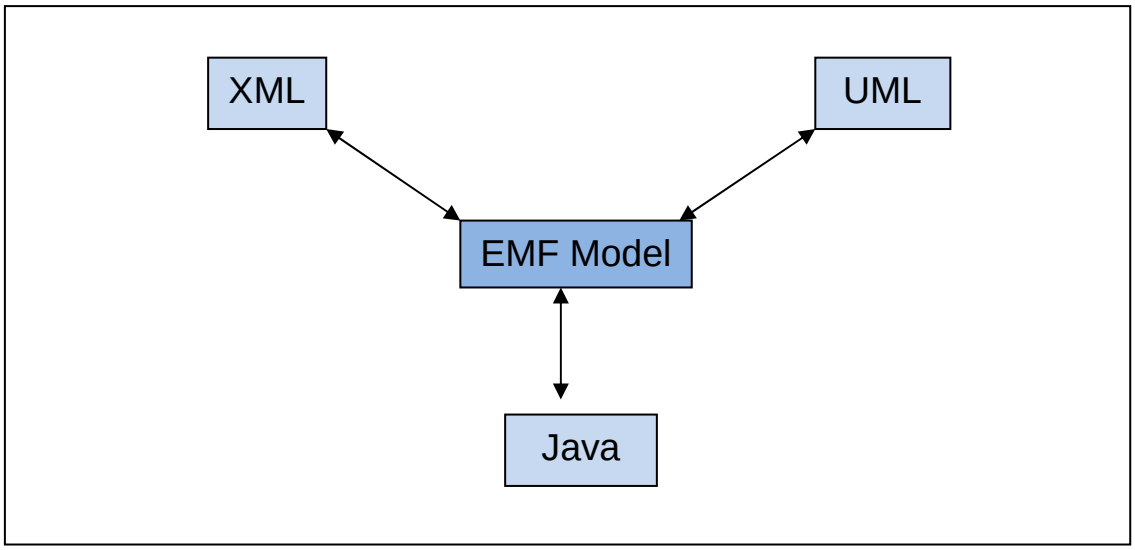
3.4.2 Eclipse Modeling Framework (EMF)

El proyecto EMF es una herramienta creada para facilitar el modelado de sistemas y la generación automática de código Java mediante patrones y buenas prácticas de diseño.

EMF es un framework que nos permite definir un modelo a partir de un entorno de ejecución y de un conjunto de herramientas. EMF unifica tres tecnologías (Figura 17): Java, XML y UML. EMF nos permite tener un modelo de representación de alto nivel que permite combinar estos tres lenguajes. Se

puede definir un modelo en una de estas alternativas y mediante EMF conseguir generarlo en las otras dos. EMF suministra las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.

Por ejemplo, se podría definir un esquema XML y desde él, generar el modelo EMF a partir del cual se podría obtener el resto de representaciones.



(Fuente: [43])

Figura 17. Unificación de XML, UML y Java mediante EMF.

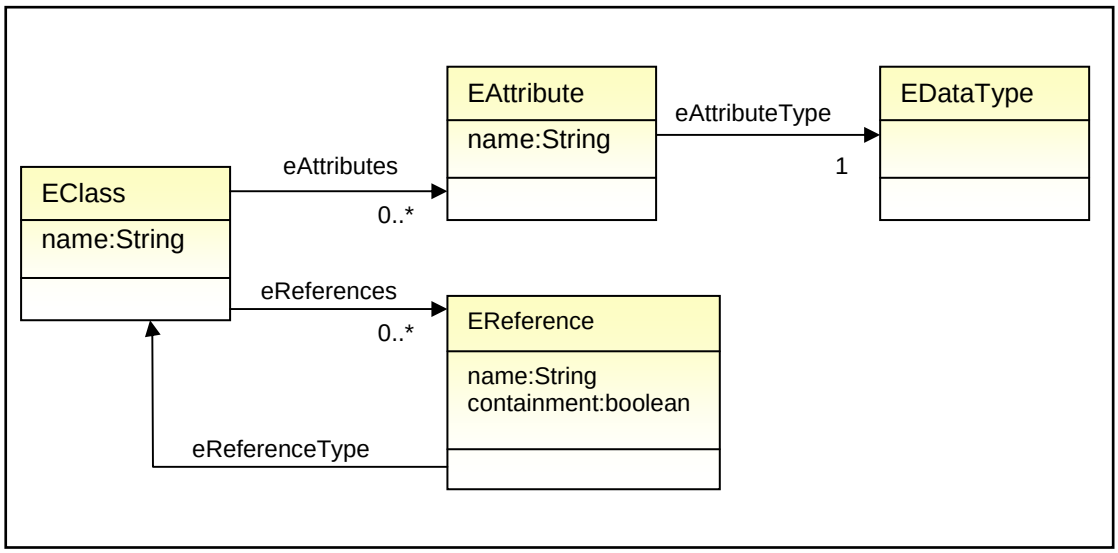
Empezar el desarrollo de una aplicación mediante un modelo EMF nos permite tener una representación de alto nivel en donde se muestren todos los conceptos de la aplicación y nos permita generar de forma eficiente y fácilmente parametrizable sino todo el código, parte de él. Una vez especificado el modelo en EMF, el generador EMF puede crear el conjunto de clases correspondientes.

EMF se divide en dos subproyectos: *EMF Core* y *EMF.Edit. EMF Core* es el núcleo de EMF que proporciona el metamodelo ECore utilizado para describir modelos y un motor de ejecución para los modelos que soporta la creación de las clases Java de un modelo. *EMF.Edit* proporciona clases genéricas reutilizables para construir editores para modelos EMF.

Ecore es el metamodelo a partir del cual se pueden definir modelos en EMF. Ecore es en sí mismo un modelo EMF. A partir de Ecore se pueden definir

metamodelos, así como funcionalidades para definir los cambios en el modelo, gestionar su persistencia mediante la exportación del formato XMI y una API eficiente para manipular los objetos EMF de forma genérica.

En la Figura 18 se muestra un subconjunto del modelo Ecore. En el modelo Ecore podremos observar clases como EClass, EAttribute, EDataType o ENamedElement utilizada para representar el nombre de un atributo. Es decir, conceptos UML que hacen de Ecore un subconjunto de UML. UML soporta modelados más complejos de los que soporta EMF como el modelado del comportamiento de una aplicación así como su estructura de clases.



(Fuente: [43])

Figura 18. Subconjunto del metamodelo Ecore.

A partir del metamodelo anterior, utilizaremos *EClass* para representar clases que pueden contener *EAttributes* y *EReferences*. *EAttribute* se utiliza para representar los atributos que serán de un tipo determinado modelado a través de *EDataType*. *EReference* representa las relaciones entre *EClass*.

Como se indicó anteriormente, se puede obtener el modelo Ecore partiendo de una definición en los lenguajes XML, Java o UML. En caso de utilizar UML, se dispone de las siguientes opciones:

- Utilizar el editor Ecore suministrado por EMF ya sea mediante el editor con estructura de árbol que incluye EMF, utilizando alguna herramienta

gráfica que nos permita definir modelos UML dentro de EMF o utilizar Emfatic para definir el modelo de manera textual.

- Se puede utilizar la exportación e importación de modelos UML de la siguiente manera:
    - Exportar un modelo UML al formato XMI desde la herramienta que se esté utilizando e importarlo desde EMF.
    - Importar desde EMF conectando directamente con *Rational Rose.*
    - Exportar al formato de EMF desde aquellas herramientas que lo permitan.

Construir aplicaciones utilizando EMF proporciona beneficios variados tales como notificaciones de modificaciones de modelos, serialización basada en *XML Metadata Interchange* (XMI), un marco de trabajo para validación de modelos y una interfaz para la manipulación de objetos EMF eficiente, pero uno de los beneficios más importante es la generación automática de código desde los modelos.

EMF Core realiza la generación del código Java de manera distinta en función de los elementos del modelo. Para las entidades, EMF Core crea un interfaz que hereda del interfaz EObject del metamodelo de EMF Core (ECore) que contiene todas las funciones y crea una clase que implementa dicho interfaz y a su vez hereda de la clase EObjectImpl de Ecore (que también implementa el interfaz EObject).

Para las relaciones entre las entidades se genera el código de las funciones que generan dicha relación teniendo en cuenta las cardinalidades de cada uno de los sentidos de estas relaciones para elegir el patrón de diseño más apropiado en cada caso.

EMF Core también trata de manera especial los atributos enumerados siendo convertidos en entidades, si bien es cierto que tienen funcionalidad limitada, pero forman parte del sistema de herencia del nuevo modelo y como tales pueden acceder a las factorías y paquetes que EMF Core genera automáticamente para el modelo.

En cuanto a la herencia, debido a que las clases generadas son clases Java, como tal no pueden tener herencia múltiple por eso en EMF Core se tiene que decidir qué clase base va a ser de la que se va a heredar y para el resto de las clases padre del modelo la nueva clase simplemente se limitará a implementar cada uno de sus interfaces.

El código generado puede ser personalizado sin perder los cambios cada vez que se regenere el código a partir del modelo. Esto se puede hacer de dos maneras:

- Eliminando la etiqueta "@generated" que incluye EMF al generar el código. Cada vez que encuentre un conflicto entre un método del modelo y del código, mirará si existe la etiqueta. Si la etiqueta no está respetara el código y no lo regenerará.

- Modificando el nombre de los métodos añadiéndole el sufijo Gen. El generador no generará el código para aquellos métodos que con el mismo nombre no presente tal sufijo.

3.4.3 Emfatic

Emfatic es un plugin de Eclipse que dota de sintaxis concreta al metamodelo Ecore de EMF. De esta forma, los metamodelos Ecore pueden ser definidos textualmente. El plugin incorpora la infraestructura necesaria para la creación de metamodelos Ecore textualmente así como la conversión automática de Emfatic a un modelo Ecore y viceversa. En el menú contextual para los ficheros .emf y .ecore se encuentra la opción *Generate Ecore Model* y *Generate Emfatic Source* respectivamente.

El primer componente que debe ser declarado en un fichero Emfatic es la declaración de *package.* Esta declaración es imprescindible y contiene el resto de los componentes del fichero Ecore generado. Esta declaración corresponde al *EPackage* de un fichero Ecore y se define de la siguiente manera:

```
package p;
```

Mediante la siguiente declaración definimos los atributos **nsURI** y **nsPrefix**:

```
@namespace(uri="http://www.eclipse.org/emf/2002/Ecore", prefix="ecore")
package ecore;
```

La sintaxis de Emfatic para declarar una clase es muy similar a Java aunque presenta algunas peculiaridades que permite todas las posibilidades de Ecore. A continuación, podemos ver un ejemplo donde se hace uso de clases, interfaces, herencia y uso de una declaración abstracta.

```
abstract class className extends className { }
class C1 { }
class C extends A, B { }
abstract class C2 { }
interface I1 { }
abstract interface I2 { }
```

Mediante las palabras reservadas **ref** y **val** se definen las asociaciones y agregaciones y mediante **attr** se definen los atributos de las clases. Mediante la declaracion "#opposite" definimos referencias bidireccionales.

```
ref className[cardinality]#opposite refName;
val className[cardinality]#opposite refName;
attr dataTypeName[cardinality] attrName;
```

Las cardinalidades soportadas por Emfatic son: **[0..1], [1], [*], [+]** y los tipos predefinidos son: **String, Integer, Boolean, Double.**

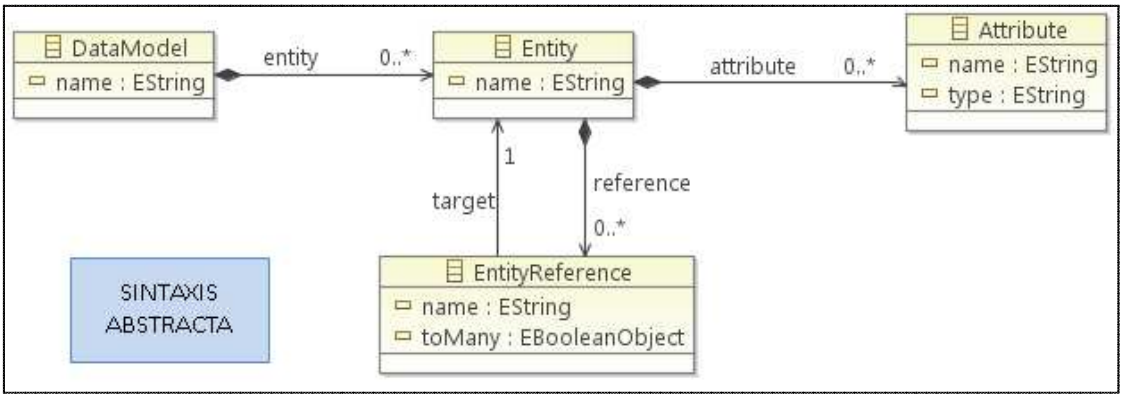
A continuación, podemos ver el uso de los tipos enumerados. El primer literal de una declaración de tipos enumerados es cero, en ausencia de la asignación de un valor a un literal se considera uno más del valor anterior.

```
enum E {   A;  // = 0   B = 3;   C;  // = 4   D;  // = 5 }
```

La sintaxis utilizada para definir anotaciones fue inspirada de la sintaxis Java introducida en Java 1.5. Primero se especifica el símbolo @, a continuación el nombre del atributo origen y después las parejas clave/valor para anotar los detalles.

```
@annName(Key1="value1",…)
```

En la Figura 19 se muestra la sintaxis abstracta de un metamodelo de ejemplo y en la Figura 20 la sintaxis concreta que le corresponde. Con estas dos figuras se pretende ilustrar el uso de los elementos básicos de Emfatic.



(Fuente: [52])

Figura 19. Ejemplo de metamodelo.

```
class DataModel{
   attr String name;
   val Entity[*] entity;
}
class Entity {
   attr String name;
   val Attribute[*] attribute;
   val EntityReference[*] reference;
}
```

```
class Attribute {
  attr String name;
  attr String type;
}

class EntityReference {
  attr String name;
  attr Boolean toMany;
  ref Entity[1] target;
}
```

(Fuente: Propia)

Figura 20. Sintaxis concreta del metamodelo de la Figura 19.

3.4.4 EMFText y XText

Existen diferentes herramientas para la definición de DSLs, estas herramientas se encuentran reflejadas en Figura 21. En este proyecto se ha creado un DSL textual y por tanto, se ha centrado en las herramientas de definición de DSLs textuales. En concreto, se ha estudiado Xtext [12] y EMFText [11]. Estas herramientas crean un inyector de modelos a partir de una gramática del DSL

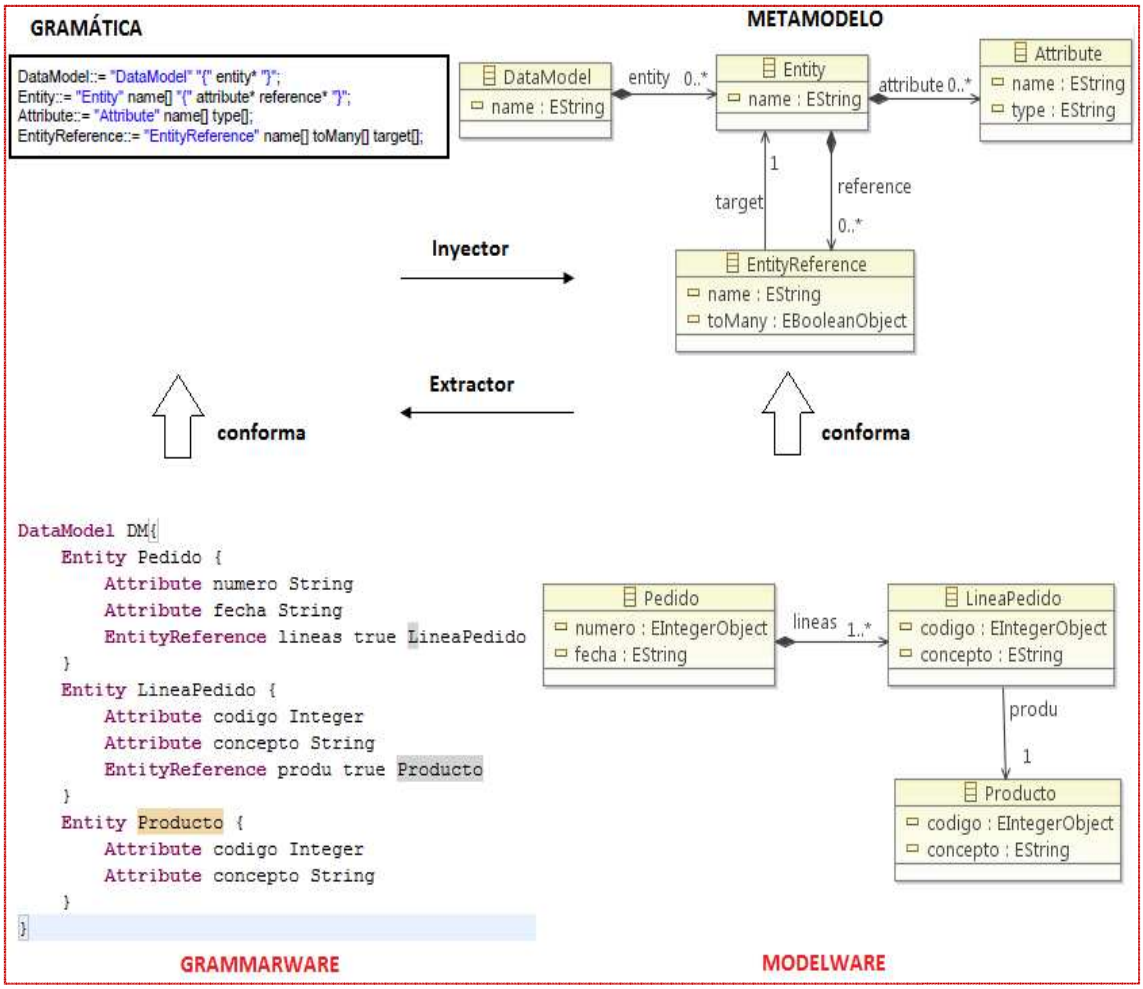
(caso de Xtext) o de un metamodelo del DSL (caso de EMFText). En este proyecto, se ha decidido utilizar esta última dado que se optó por crear el metamodelo del DSL y será la que se describa con más detalle.

DSLs Textuales	DSL Gráficos	DSL Hibridos
Xtext (oaw) EMFText TCS (AtlanMod)	MetaEdit (Metacase) DSL Tools (Microsoft) GMF (Eclipse/EMF)	MPS (Metaprogramming Systems)

(Fuente: [42])

Figura 21. Herramientas para definir DSLs.

Como se puede ver en la Figura 22, a partir de una gramática se puede utilizar un **inyector** para generar modelos conformes a un metamodelo a partir de texto. Por otro lado, a partir de un modelo conforme a un metamodelo se puede utilizar un **extractor** para crear su representación textual.



(Fuente: propia)

Figura 22. Comparación de Inyector y un Extractor.

XText es un lenguaje que pertenece al *framework* openArchitectureWare y permite construir DSLs textuales en la plataforma Eclipse. En este lenguaje, la sintaxis concreta textual del DSL se especifica por medio de un lenguaje de definición de gramáticas similar a EBNF. A partir de la gramática especificada se genera automáticamente el metamodelo del DSL, un analizador sintáctico para reconocer la sintaxis del DSL y para instanciar el metamodelo, y un editor específico del DSL. La estructura de la herramienta XText se puede ver en la Figura 23.

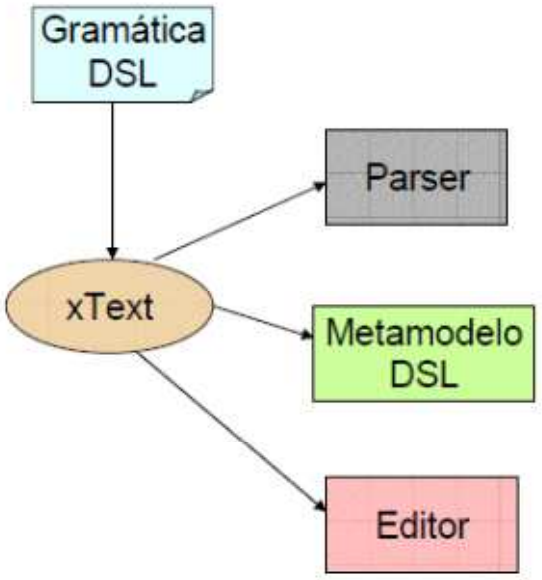


Figura 23. Estructura de xText (Fuente: [42])

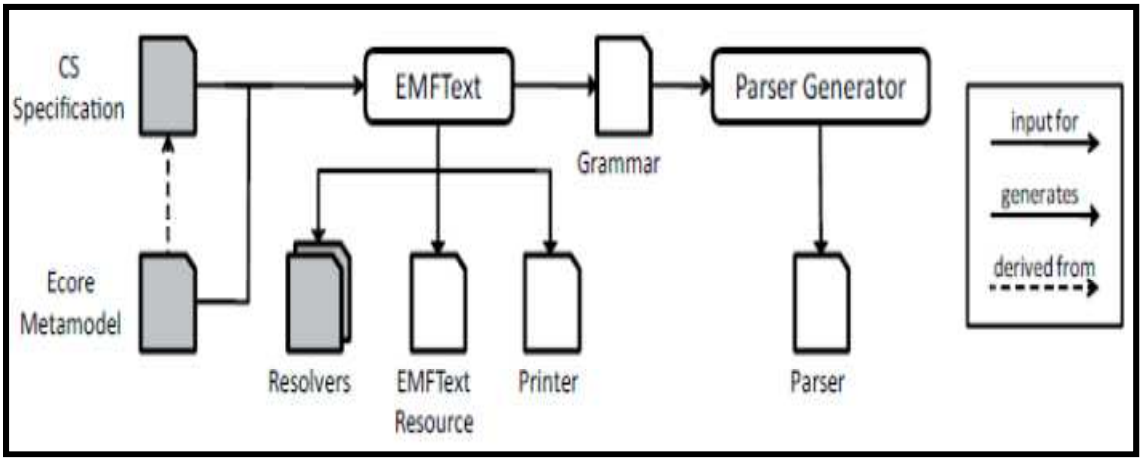
EMFText permite la definición de sintaxis textuales para lenguajes descritos por un metamodelo Ecore, ofreciendo un lenguaje de especificación de sintaxis textuales. A partir de un fichero descrito por dicho lenguaje, EMFText genera:

- Un inyector para crear modelos conformes al metamodelo de sintaxis abstracta a partir de texto.

- Un extractor para crear la representación textual a partir de un modelo de la sintaxis abstracta.

- Un editor de Eclipse con resaltado de sintaxis para la sintaxis textual, el cual incorpora una vista outline y detección de errores.

En la Figura 24 se puede ver el funcionamiento de EMFText. Primero se define la sintaxis concreta para la sintaxis abstracta, dicha sintaxis concreta está representada por el elemento CS Specification. A partir de la sintaxis concreta y la abstracta, EMFText genera un conjunto de artefactos software. El elemento *printer* representa el extractor que genera la representación textual a partir de un modelo conforme a la sintaxis abstracta. El elemento *grammar* denota la gramática que es la entrada al generador del parser, esto es el inyector que reconocerá la sintaxis concreta y generará los modelos conformes a la sintaxis abstracta. El elemento EMFText Resource es utilizado por la plataforma Eclipse para almacenar y cargar los modelos Ecore de la sintaxis abstracta. Finalmente, los Resolvers se generan para ofrecer puntos de extensión al comportamiento predeterminado de EMFText para resolver las referencias entre elementos.



(Fuente: [11])

Figura 24. Estructura de EMFText

A continuación, se va a detallar la forma de definir las sintaxis textuales con EMFText. Estas sintaxis se almacena en un fichero con extensión .cs y se compone de tres bloques:

- Bloque de configuración. Este bloque contiene el nombre, el metamodelo base y la metaclase que actúa como raíz. Opcionalmente puede importar otras sintaxis y metamodelos, así como también pueden establecerse atributos para parametrizar la generación.

- Bloque opcional de tokens léxicos. Estos tokens son utilizados por el analizador léxico. También puede incluirse, opcionalmente, un bloque que indique el estilo de los tokens.

- Bloque de reglas. En este bloque contiene las reglas que definen la sintaxis concreta para cada metaclase.

Bloque de configuración

Este bloque está compuesto por los siguientes elementos:

- Nombre de la definición de la sintaxis concreta indicando de esta manera la extensión de los ficheros de este DSL.

```
SYNTAXDEF fileExtension
```

- Se debe indicar la URI del metamodelo para el cual se quiere definir la sintaxis textual. De manera opcional se puede indicar la ruta para localizar el metamodelo. Si no se indica dicha ruta, primero se busca en el registro de EMF y, si no lo encuentra busca en el directorio donde se encuentre el fichero .cs.

```
FOR <genModelURI> <locationOfYourGenmodel>
```

- A continuación el elemento raíz del metamodelo. Existe la posibilidad de definir varios puntos de comienzo siempre separados por coma.

```
START nameOfAMetaClass
```

- De forma opcional se puede importar de otros modelos. Para ello cada entrada de importación indica el prefijo a utilizar para referirse a los elementos que se importan, la URI del metamodelo y opcionalmente el nombre de la sintaxis concreta definida para dicho metamodelo.

```
IMPORTS {
        prefix: <genModelURI><locationOfTheGenmodel> WITH SYNTAX
        syntaURI <locationOfTheSyntax>; }
```

- De forma opcional, se puede parametrizar la generación.

```
OPTIONS { optionName = “optionValue”; }
```

Bloque de tokens léxicos

Durante la fase de reconocimiento de la sintaxis textual, los caracteres de entrada son convertidos a tokens. Por defecto, EMFText proporciona los siguientes tokens predefinidos:

```
TEXT: (‘A’ .. ‘Z’ | ’a’ .. ‘z’ | ’0’ .. ‘9’ | ’_’ | ’-‘)+

LINEBREAK: (‘\r\n’ | ’\r’ | ‘\n’)

WHITESPACE: (‘ ‘ | ‘\t’ | ‘\f’)
```

El lenguaje permite la definición de tokens personalizados mediante el bloque de tokens léxicos. Cada token está compuesto por un nombre y una expresión regular.

Mediante el bloque *Tokenstyles* se puede configurar el estilo de los token (color y formato) en el editor textual de Eclipse.

Bloque de reglas

Las reglas sintácticas se definen para cada metaclase concreta de la sintaxis abstracta. La parte izquierda de la regla indica el nombre de la metaclase a la cual se va a indicar la sintaxis concreta. La parte de la derecha define los elementos sintácticos que conforman la representación textual de la metaclase.

Es posible el uso del símbolo ‘?’ para indicar opcionalidad así como los simbolos ‘+’ y ‘*’ para indicar la cardinalidad.

Los atributos de las metaclases se representan añadiendo los corchetes ‘[]’ al final del nombre del atributo. Entre los corchetes se puede escribir el token

utilizado para reconocer el atributo. En caso de no indicar ningún token, se utiliza el token por defecto TOKEN.

Una regla básica sería:

```
MyMetaClass ::= “someKeyword”;
```

Mediante esta regla indica que siempre que se encuentre el texto someKeyword se creará una instancia de MyMetaClass.

La diferencia en la forma de definir las referencias contenedores y las referencias no contenedores (solo apuntan a los elementos) es añadiendo los corchetes tras el nombre de la referencia. A continuación podemos ver un ejemplo:

```
MyContainerMetaClass ::= “CONTAINER” myContainmentReference*;

MyPointerMetaClass ::= “POINTER” myNonContainmentReference[]*;
```

3.4.5 RubyTL

RubyTL [41] es un lenguaje de transformación modelo-modelo basado en reglas, que ha sido construido como un lenguaje embebido dentro de Ruby. Esto significa que no se ha utilizado la técnica habitual de implementar un parser y un intérprete o un compilador, sino que la sintaxis de RubyTL está basada en las construcciones ofrecidas por el propio Ruby, en este caso, llamadas a métodos. La implicación fundamental de esta técnica de implementación es que es posible utilizar código Ruby imperativo en cualquier parte de una definición de transformación.

Es un lenguaje de transformación híbrido, cuya parte declarativa está basada en reglas y *bindings*, mientras que la parte imperativa viene dada por los constructores proporcionados por el propio Ruby. En RubyTL una definición de transformación está compuesta de reglas de transformación.

Una transformación de modelos es un proceso por el cual un modelo origen es transformado en un modelo destino. Para realizar la transformación deben definirse las correspondencias (*mappings*) entre los elementos del metamodelo origen y del metamodelo destino. Estas correspondencias se expresan a nivel

de los metamodelos origen y destino. Los lenguajes de transformación de modelos permiten especificar un mapping utilizando reglas de transformación. Estas reglas normalmente especifican cómo se relacionan las instancias de una determinada metaclase (del modelo origen) con instancias de otra metaclase (del modelo destino).

Una definición de transformación en RubyTL está compuesta de cuatro partes *from, to, filter* and *mapping*:

- *from.* Especifica una metaclase origen, que pertenece a un metamodelo origen. La regla se aplicará sobre elementos del modelo origen que sean de ese tipo o de alguna de sus subclases.

- *to.* Especifica una o más metaclases destino, que pertenecen a un metamodelo destino. La regla creará elementos de ese tipo (excepto si es una regla de refinamiento).

- *filter.* Una expresión sobre el elemento origen. La regla solo se ejecuta si esta expresión (un bloque de código, en realidad) se evalúa a *true.*

- *mapping.* Un bloque de código de Ruby, que se ejecuta cuando se aplica la regla. El bloque recibe un elemento origen (que conforma a la metaclase especificada en la parte *from*) y el elemento destino (que conforma con las metaclases destino especificadas en la parte *to*) creado por la regla. Dentro del bloque es posible escribir cualquier código Ruby, aunque se recomienda un estilo declarativo basado en *bindings.*

Un *binding* especifica una correspondencia entre elementos del modelo origen y elementos del modelo destino. Un *binding* tiene la estructura expr_izq.propiedad = expr_der. Si el tipo de la expresión de la parte derecha no es compatible con el tipo de la propiedad especificada, el motor RubyTL automáticamente busca una regla capaz de transformar la parte de la derecha en la parte de la izquierda. La estructura sigue:

- expr_der. Es una expresión cuyo resultado es un elemento o una colección de elementos, que deben pertenecer al modelo origen. El tipo de la parte derecha viene dado por el tipo de la expresión. Si el resultado de la expresión es una colección, RubyTL iterará automáticamente sobre la misma al asignarla a la parte izquierda.

- expr_izq. Es un parámetro del bloque de código del mapping que representa un elemento destino. Su tipo viene dado en la parte *to* de la regla.

- propiedad. Debe ser una propiedad de un elemento destino. El tipo de la parte izquierda de la asignación viene dado por el tipo de tal propiedad. Si la propiedad destino es multievaluada (es una colección), entonces la semántica es "transformar y añadir".

En la Figura 25 podemos observar la sintaxis concreta de RubyTL.

```
module <module-name> do

  rule <rule-name> do
    from <source-metaclass>
    to   {target-metaclass}

    filter do |source_element|
      <expression>
    end

    mapping do |<source_element>, {target_element}|
      {bindings}
      # bindings has the form:
      #     target_element.property = source_element.property
    end
  end

  # one or more rules
end
```

(Fuente: [41])

Figura 25. Sintaxis concreta RubyTL.

En RubyTL hay varios tipos de reglas, y cada una proporciona una funcionalidad distinta para tratar ciertos problemas de transformación. Los tipos básicos de reglas son:

- Regla “top”. Sirve como “regla main” para iniciar la ejecución. Se aplica sobre todos los elementos del modelo origen del tipo que indique su parte *from*.

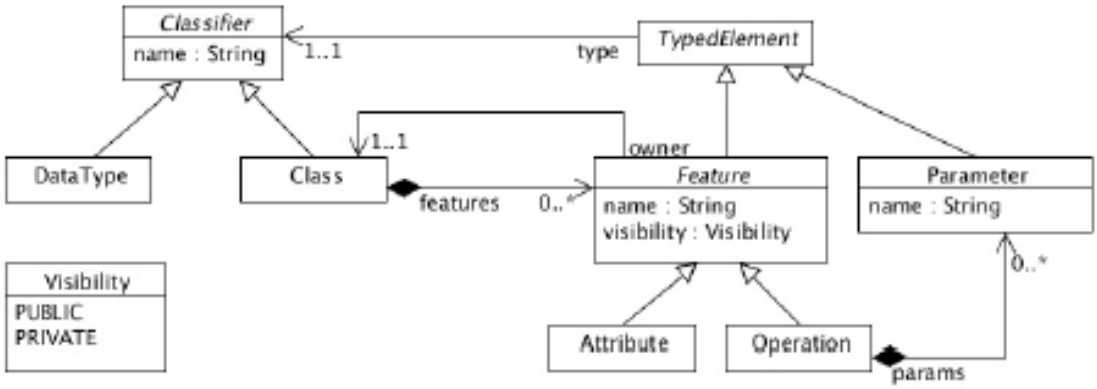
- Regla “normal”. Se nombra simplemente con *rule* y se ejecuta solo como resultado de evaluar un *binding o* cuando es la primera regla de una transformación y la transformación no tiene ninguna otra regla *top.* Una característica peculiar (que comparte con las reglas “top”) es que nunca transforma dos veces el mismo elemento. Si esto sucede, se devuelve el elemento destino creado previamente. Éste tipo de regla resuelve problemas de recursividad no transformando dos veces el mismo elemento fuente.

- Regla “copy”. Se ejecuta solo como resultado de evaluar un *binding,* pero en este caso un elemento puede ser transformado más de una vez.

La ejecución de una transformación empieza con la ejecución de las reglas “top”. Para cada regla top (normalmente habrá una por cada transformación) se obtienen todos los elementos del modelo origen cuyo tipo es la clase especificada en la parte *from* o una de sus subclases. Para cada elemento origen se crea el correspondiente elemento destino. Si no existen reglas top se considera la primera regla de la transformación como tal.

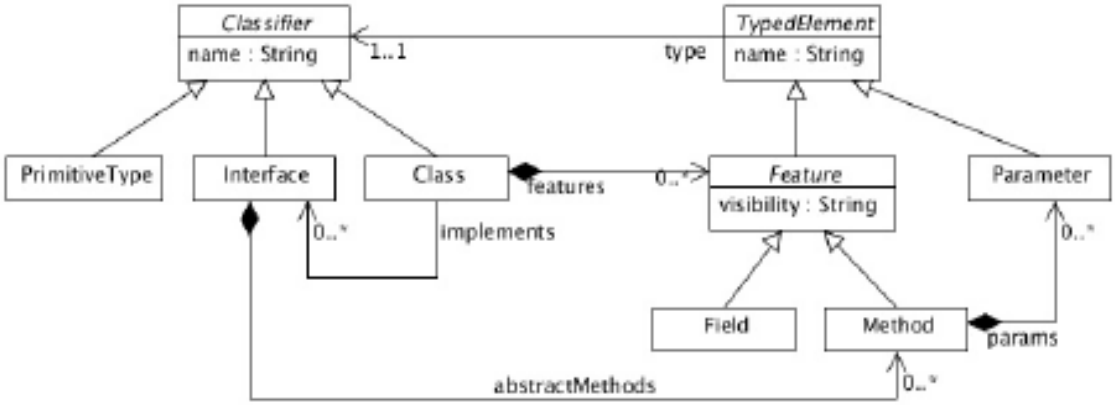
A continuación se ejecuta la parte *mapping,* ejecutando los *bindings* que se hayan especificado. La ejecución de los *bindings* implicará la ejecución de ciertas reglas, para las cuales se ejecutará también los *bindings* correspondientes.

En las Figuras 26 y 27 se pueden ver los metamodelos de clases y de Java correspondientemente. A partir de estos dos metamodelos, y estableciendo las correspondencias: Clase UML - Clase Java, Attribute UML - Field Java y Attribute UML - Method Java, se definen las reglas de transformación mediante RubyTL.



(Fuente: [41])

Figura 26. Metamodelo de clases



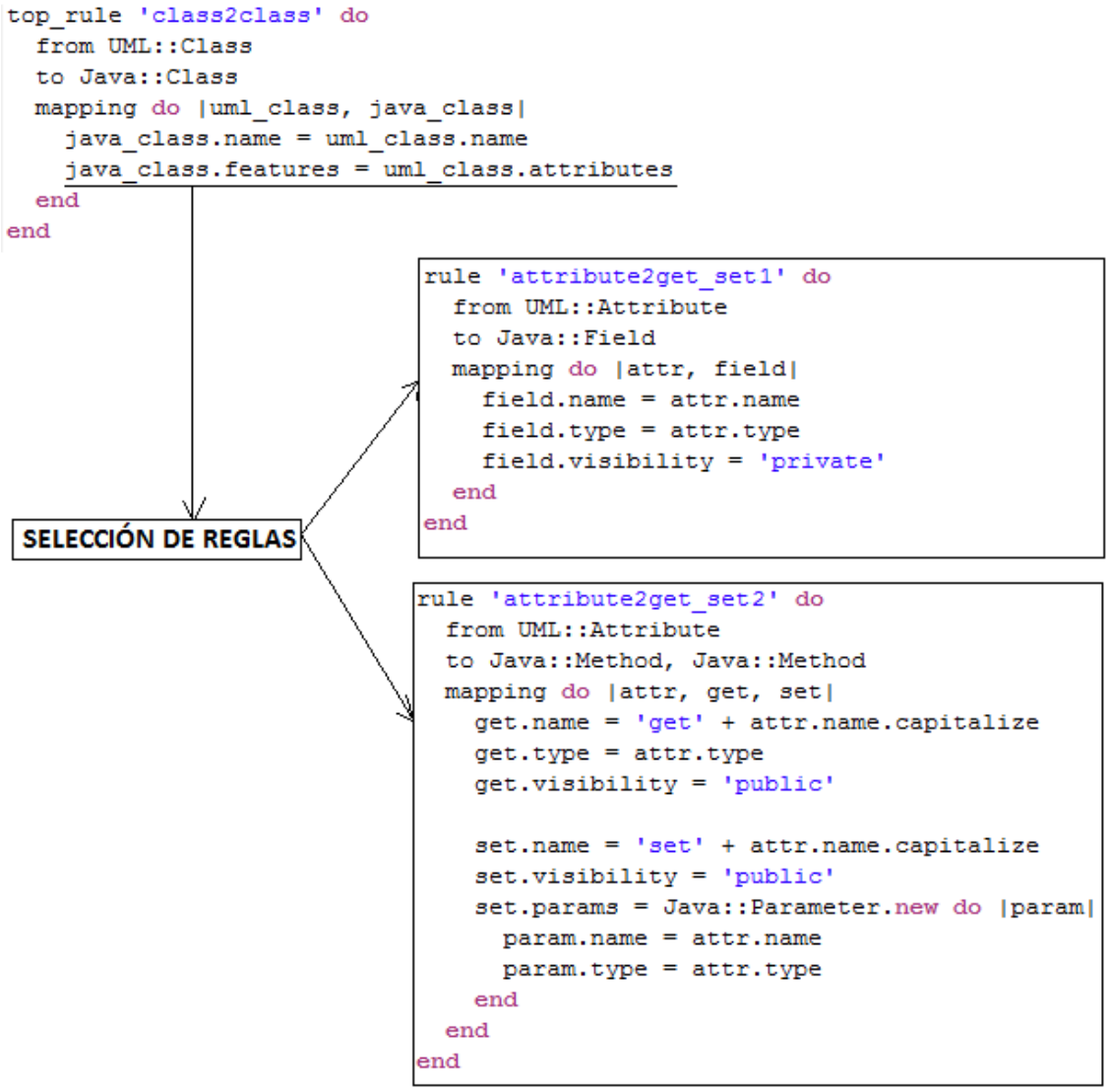
(Fuente: [41])

Figura 27. Metamodelo de Java

En la Figura 28 se puede ver un ejemplo de reglas de transformación con RubyTL. En primer lugar, se ejecuta la regla “top” “class2class” obteniendo todos los elementos del modelo origen de tipo “Class”. Para cada elemento encontrado, se crea el correspondiente elemento destino del tipo “Class”. Al ejecutar el mapping de esta regla, asignamos al nombre de la clase Java el nombre de la clase UML. Al llegar a la segundo *binding*, el motor RubyTL comprueba que el tipo de la derecha de la regla no es compatible con el tipo de la izquierda de la regla y comienza la búsqueda de una regla que sea capaz de transformar la parte de la derecha en la parte de la izquierda. En esa búsqueda, encuentra dos reglas que puede aplicar, “attribute2get_set1” y “attribute2get_set2”. Mediante la primera regla crea un elemento “Field” del

metamodelo Java asignando valor a sus propiedades y mediante la segunda regla, crea los métodos “get” y “set” por cada atributo del modelo origen.

En la regla “attribute2get_set2” se puede observar un ejemplo de creación de objetos con Ruby al crear el objeto “Parameter” del metamodelo Java.



(Fuente: Propia)

Figura 28. Ejemplo de regla de transformación con RubyTL

Otra característica presente en RubyTL son los decoradores. Mediante los decoradores se pueden hacer las transformaciones más legibles, normalmente factorizando el código de navegación en el decorador.

Un decorador está compuesto de uno o más métodos Ruby (especificados con '*def methodname – end').* Los métodos podrán ser invocados por cualquier instancia de la metaclase o de cualquiera de sus subclases. Se pueden utilizar variables de instancia dentro de los decoradores pero no es una práctica recomendada.

Si es una práctica recomendada escribir los decoradores antes de las reglas de transformación. También, es posible factorizar los decoradores en librerías

A continuación, se muestra un ejemplo de decorador.

```
decorator UML::Class do
        def attributes
                self.features.select { |f| f.kind_of?(UML::Attribute) }
        end

        def operations
                self.features.select { |f| f.kind_of?(UML::Operations) }
        end
end
```

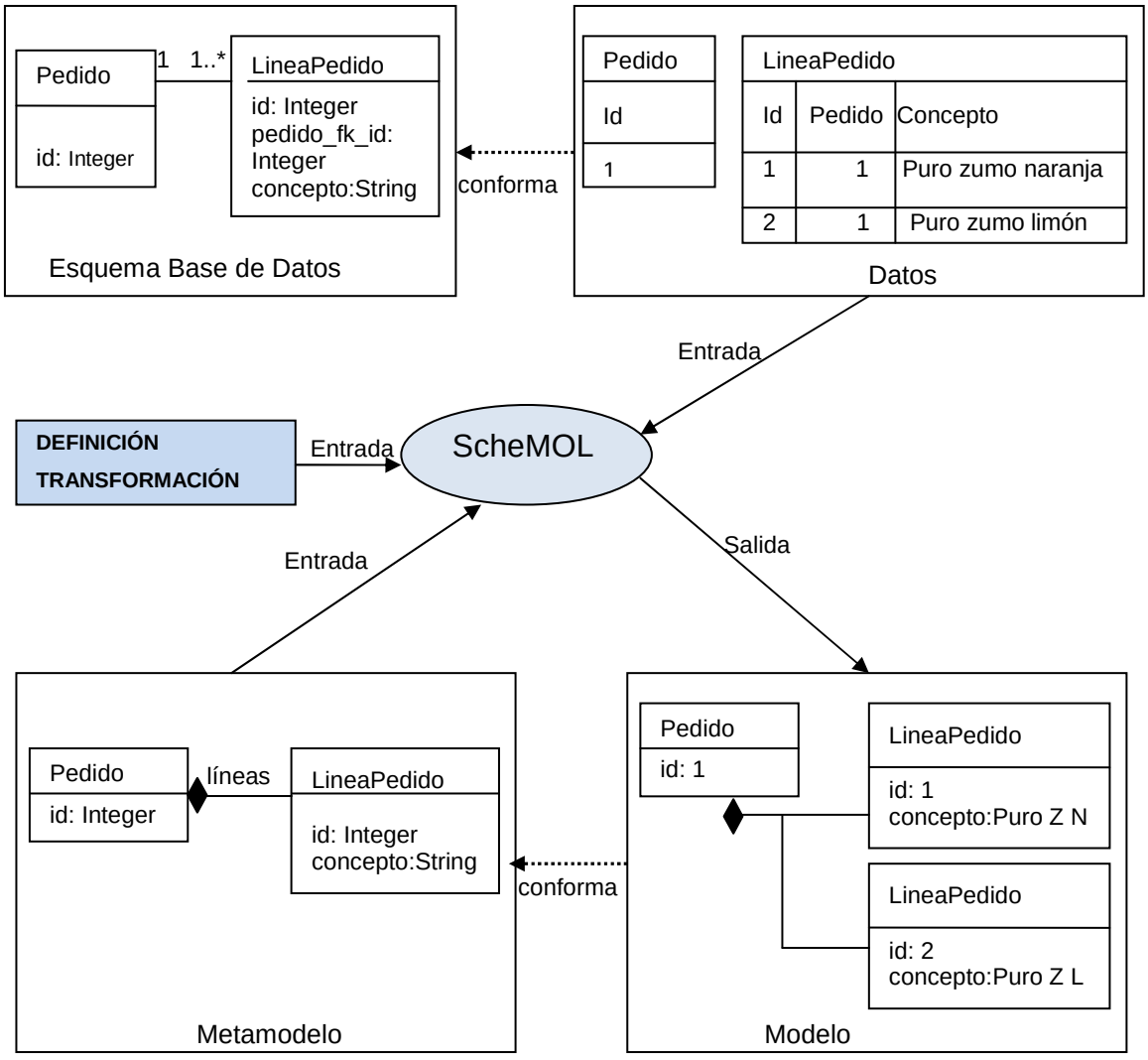
3.4.6 Schemol

Un proceso MDD típico comienza con la definición de modelos que representan algún aspecto del sistema software y prosigue con las transformaciones modelo-modelo (m2m) y modelo-texto (m2t) a través de las que se crean las diferentes partes del sistema. Ahora bien, el problema surge cuando se parte de un sistema existente y se quiere trasladar al ámbito de MDE, esto es convertir en modelos, alguna información existente, por ejemplo código fuente o documentos XML. En [53] se ha estudiado la extracción de modelos a partir de código de lenguajes de programación, de datos de bases de datos relacionales y de APIs, para lo cual se han definido, respectivamente, los DSLs Gra2MoL [51], Schemol [49] y Api2MoL [54].

Schemol es un DSL utilizado para extraer (inyectar) modelos a partir de datos en bases de datos relacionales.

Este lenguaje permite expresar transformaciones datos-a-modelo (d2m) mediante un conjunto de reglas que establecen las correspondencias entre los elementos del esquema de la base de datos (tuplas) y los elementos del

metamodelo destino. En la ejecución de una transformación Schemol se tiene como entrada los datos que conforman al esquema, el metamodelo destino, y la definición de la transformación, y a partir de estas entradas se genera el modelo conforme al metamodelo especificado. En la Figura 29 se puede ver un ejemplo de transformación datos a modelo aplicado a un sistema de pedidos.



(Fuente: propia)

Figura 29. Ejemplo de transformación con ScheMOL

Una definición de transformación en Schemol está compuesta por un conjunto de reglas y, opcionalmente, un preámbulo.

Cada regla Schemol define las correspondencias entre una tabla del esquema de la base de datos y una metaclase del metamodelo y tiene cuatro secciones principales:

- La sección *from,* que especifica la tabla origen junto con una variable que se asociará a la tupla cuando la regla sea ejecutada.

- La sección *to*, que especifica la metaclase destino junto con una variable que se asociará a la instancia de dicha metaclase cuando la regla sea ejecutada.

- La sección *filter* es opcional e incluye una expresión condicional que se aplica a la tupla origen de forma que solamente aquellas tuplas que satisfagan dicha condición ejecutarán la regla.

- La sección *mapping*, que contiene un conjunto de *bindings* para establecer las propiedades del elemento del metamodelo destino.

Existen dos tipos de reglas en Schemol:

- Top rules. Marcan el punto de comienzo de la transformación y son lanzadas directamente sin ser necesario que sean llamadas por otras reglas. Se definen con “toprule ‘ruleName’”

- Rules. Son reglas normales que pueden ser llamadas por una regla *top* o por otra regla normal. Se definen “rule ‘ruleName’”.

Es posible la definición de **filtros** que permiten la comprobación de condiciones. Los filtros son opcionales en la definición de una regla. Tenemos dos tipos de filtros:

- Filtros en reglas. Si la condición es satisfecha, la regla es lanzada. Se define:

```
rule 'ruleName'
  from schemaName::tableName fromId
```

```
  to packageName::className toId
  filter
    expression
  mapping |fromId, toId|
    toId.attributeName = expression
end_rule
```

- Filtros en consultas. Las condiciones son comprobadas contra las consultas filtrando los valores devueltos.

```
mapping
 toId.attributeName = fromId.@query{nameColumn = value};
```

La ejecución de una transformación en Schemol está guiada por los *bindings*, que tienen una sintaxis y semántica muy similares a las utilizadas en Gra2MoL y en el lenguaje de transformación modelo-a-modelo ATL [16]. En Schemol, la parte izquierda del binding debe ser una propiedad de la metaclase destino de la regla pero la parte derecha puede ser un valor literal, una expresión o una consulta, donde las dos últimas pueden tratar con tuplas. Las consultas se expresan utilizando un lenguaje especialmente adaptado que permite obtener información de la base de datos sin necesidad de escribir consultas complejas en SQL (*structured query language*).

A continuación se muestran dos ejemplos de reglas de transformación. En el primer ejemplo la regla “mappedido” establece la correspondencia entre la tabla “Pedido” de la base de datos y la metaclase “Pedido” del metamodelo destino.

```
rule 'mappedido'
  from test::pedido pedb
  to order::Pedido pedi
  mapping
      pedi.id     =  pedb.numero;
      pedi.lineas =  pedb.@lineapedido;
end_rule
```

En el segundo ejemplo la regla “maplineapedido” establece la correspondencia entre la tabla “LineaPedido” y la metaclase “LineaPedido” del metamodelo destino. Esta regla es ejecutada a partir del *binding* “pedi.lineas = pedb.@lineapedido”.

```
rule 'maplineapedido'
  from test::lineapedido linb
  to order::LineaPedido lini
  mapping
      lini.id        = linb.id;
      lini.concepto  = linb.concepto;
  end_rule
```

3.4.7 Mofscript

Para llevar a cabo la generación de código, se utilizará el lenguaje de transformación modelo-a-texto MOFScript, el cual es un lenguaje basado en reglas integrado en la plataforma Eclipse. Una transformación MOFScript está compuesta por un conjunto de reglas que definen las secciones de código a generar a partir de los elementos del modelo origen. Estas secciones a generar actúan como plantillas donde se inserta determinada información para completarlas, por este motivo, MOFScript también recibe el nombre de lenguaje de plantillas.

El lenguaje MOFScript surge tras la petición (RFP-Request for Proposal) del OMG de desarrollo de un lenguaje de transformación modelo-texto. Fue desarrollado por SINTEF ICT como parte del proyecto MODELWARE y del proceso de estandarización del OMG.

MOFScript fue diseñado como un DSL cuya sintaxis abstracta fue definida con un metamodelo MOF.

A continuación, se presentarán algunas de las construcciones más importantes de MOFScript.

Mediante un ***Texttransformation*** se define el nombre del módulo y a través de un parámetro de entrada se indica el metamodelo de entrada:

```
texttransformation tf (in edifact:"http://tbfinmaster.es/finmaster/edifact")
```

Un *Texttransformation* puede tener varios parámetros de entrada siempre separados por comas.

En un módulo de transformación se pueden **importar** otras transformaciones de la siguiente manera:

```
import aSimpleName (“std/stdLib2.m2t”)
```

La **regla principal “main”** define dónde comienza la ejecución de la transformación (similar a un main de Java). Puede definirse en base a un elemento del metamodelo de entrada, indicando de esta manera el elemento del metamodelo de entrada utilizado como punto de partida.

```
edifact.Estandar::main(){

}
```

En el caso de que el elemento del metamodelo de entrada referido tenga varias instancias, el cuerpo de la regla principal será ejecutado una vez por cada instancia.

Las regla principal puede ser especificada sin hacer referencia a ningún elemento del metamodelo de entrada utilizando la palabra reservada *module* o sin ninguna palabra reservada. En este caso, se ejecutará el cuerpo de la regla una sola vez y para recuperar el modelo de entrada es necesario utilizar el parámetro de entrada junto a la operación *objectsOfType* para recuperar los elementos del modelo de un tipo determinado.

```
uml.objectsOfType(uml.Package)
```

Las **reglas** pueden hacer referencia a un elemento del metamodelo de entrada y pueden tener un tipo de retorno. El cuerpo de la regla contiene el conjunto de sentencias que ejecutan la transformación.

Una regla puede devolver un valor que sea utilizado en otra regla. Esta devolución se realiza a través de la sentencia *result*.

```
result += self.name.toLower().replace(" ", "_");
```

Una regla puede tener cualquier número de parámetros. Un parámetro puede ser de uno de los tipos definidos por MOFScript o por puede ser de tipo metamodelo.

```
uml.Model::testParam3(s1:String, r2:Real, b1:Boolean, package:uml.Package)
```

Las **propiedades** y las **variables** pueden ser definidas en forma global o local dentro de una regla o un bloque. Una propiedad es una constante a la cual se le asigna un valor. El tipo de una propiedad puede ser de los tipos definidos por MOFScript, un modelo o puede carecer de tipo, en cuyo caso el tipo de la propiedad será determinado por el valor asignado a la propiedad. Por el contrario, una variable puede cambiar de valor durante la ejecución.

```
property packageName:String = “org.mypackage”

var myInteger = 7
```

Los **tipos primitivos** ofrecidos por MOFScript son los siguientes: String, Integer, Real, Boolean, Hashtable, List y Object. El tipo Object permite representar cualquier tipo.

En cuanto al **trabajo con ficheros,** la sentencia *file* permite establecer el fichero a crear donde se generará el código indicado en la transformación. El nombre del fichero se indica como parámetro y puede incluir una ruta relativa o absoluta.

```
file (“myfile.java”)

file f1 (“myfile2.java”)
```

Las instrucciones de impresión proveen el dispositivo de salida, éste puede ser un archivo o la salida estándar. Si no se declara ningún archivo, se utiliza la salida estándar. Si la salida estándar debe ser forzada, la sentencia de impresión debe tener el prefijo stdout. Las sentencias principales son print y printl.

```
stdout.println(‘’mensaje’’)
file f1 (“myfile2.java”)
f1.print (‘’codigo’’)
```

Los **iteradores** se utilizan para iterar sobre colecciones de elementos del modelo de origen. La instrucción forEach define un iterador sobre una colección de elementos, tales como elementos del modelo, list, hashtable, integer o string. Una instrucción forEach puede ser restringida por una restricción de tipo (colección -> foreach (c:SomeType)), donde el tipo puede ser un tipo del metamodelo o un tipo integrado. Si no se utiliza una restricción de tipo, la instrucción se aplica a todos los elementos de la colección. Una instrucción foreach también puede tener alguna restricción adicional que es descrita mediante el símbolo '|' (colección ->foreach (a:String | a.size()=2).

MOFScript ofrece sentencias condicionales del tipo if y while.

```
If (c.states > 10) {
}else if (c.states > 5) {
}else  {
}
While (c.states > 1) {
}
```

La sentencia **select** permite realizar una consulta sobre una colección de elementos y devuelve una lista con los elementos encontrados. Actualmente solamente puedes ser utilizada en asignaciones y su sintaxis es muy similar a la utilizada en la sentencia *forEach*.

```
var result :List = self.states ->select(st:stateMM.State | st.name.startswith("ST"))
```

MOFScript da soporte a las expresiones lógicas, expresiones que se evalúan como verdaderas o falsas y se utilizan en las instrucciones de iteración y condicionales.

En la Figura 30 se muestra un metamodelo muy simple utilizado para definir una transformación mediante MOFScript. En dicho metamodelo se definen las metaclases "Clase" y "Atributo" que representan una clase y los atributos que contiene la misma.

Figura 30. Metamodelo de ejemplo para MOFScript

A continuación, se define mediante MOFScript una transformación que recorre un modelo conforme al metamodelo de la Figura 30 generando un fichero Java que contiene el nombre de la clase y sus atributos.

```
edifact.Clase::main(){
    file(outputFile)

    'public class ' + self.nombre + '{'
     newline(1)
     self.atributos->forEach(c){
      c.mapAtributo()
      newline(1)
     }
    '}'
  }

  edifact.Atributo::mapAtributo() {
    self.visibilidad + ' ' + self.tipo + ' ' + self.nombre ';'
  }
```

Como se puede ver en la transformación anterior, se comienza por la regla “main” encargada de recorrer todas las clases del modelo y se utiliza el iterador “forEach” para recorrer todos los atributos contenidos en cada clase llamando a la regla “mapAtributo” por cada atributo encontrado.

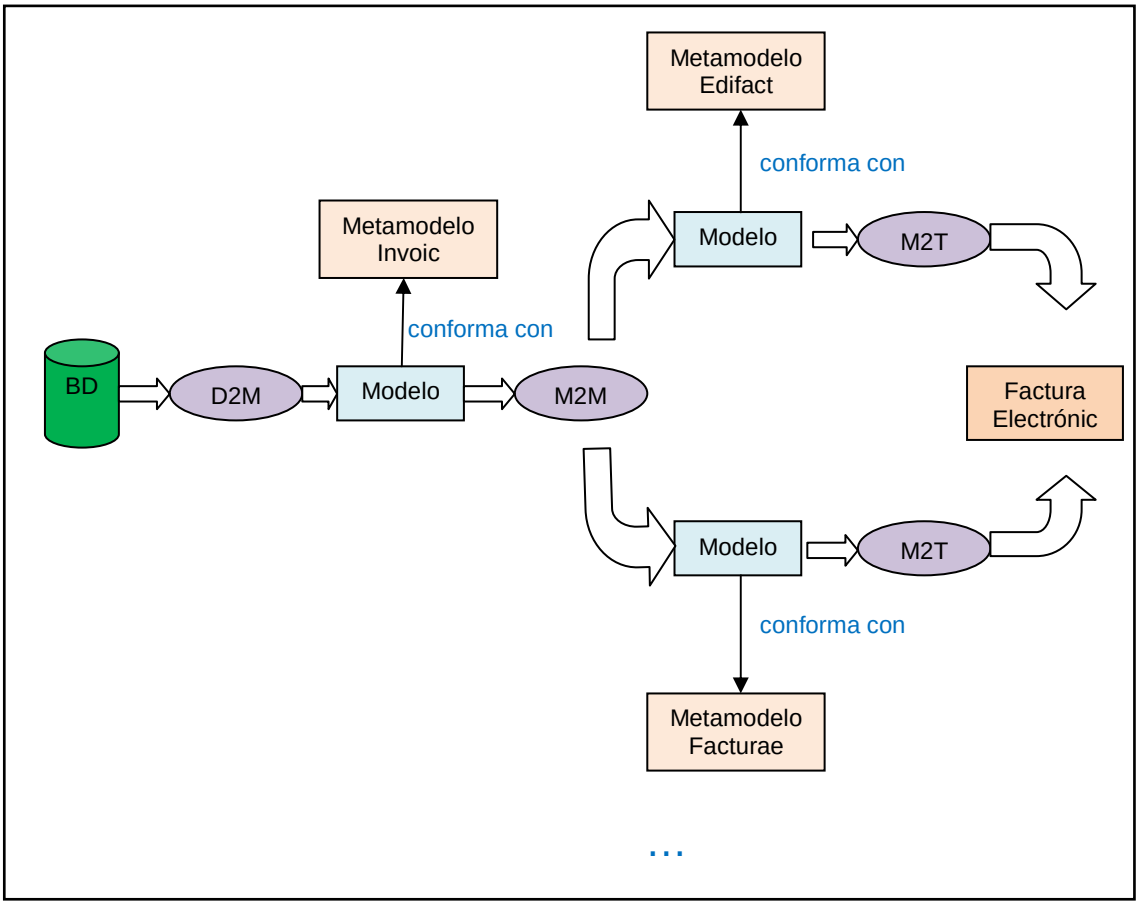
El resultado de la transformación es el siguiente:

```
public class Persona{
      private String nombre;
      private int edad;
      private String direccion;
}
```

4 MDE aplicada al formato EDIFACT

4.1 Introducción

En la Figura 31 se puede ver la estructura generada a través de la aplicación de las técnicas de la Ingeniería Dirigida por Modelos para el caso particular de la factura electrónica en el formato estructurado EDIFACT.



(Fuente: propia)

Figura 31. Estructura MDE aplicada a Ficheros Electrónicos.

En primer lugar, se va a definir un metamodelo que recoja todos los conceptos próximos a la factura, el metamodelo Invoic. Para definir dicho metamodelo, se va a utilizar la herramienta Emfatic. A continuación, se va a utilizar un doble mecanismo para inyectar modelos conformes al metamodelo Invoic. Por un lado, se utiliza la herramienta EMFText para definir una sintaxis concreta para el metamodelo Invoic y por otro lado, se va a utilizar la herramienta Schemol

que permite inyectar modelos directamente desde la base de datos conformes al metamodelo Invoic.

En una segunda etapa, se define el metamodelo Edifact que recoge las características del formato estructurado Edifact. Mediante EMFText se ha definido su sintaxis concreta para una etapa de pruebas que permitiera inyectar modelos y de esta manera posibilitar la depuración.

A continuación, se utiliza RubyTL para realizar transformaciones modelo a modelo que nos permita pasar de modelos conformes al metamodelo Invoic a modelos conformes al metamodelo Edifact.

Por último, la automatización se consigue a través de MOFScript mediante las transformaciones modelo a texto que permiten generar los ficheros de facturas electrónicas para el formato estructurado particular para el que se definen.

De igual manera, se podrían añadir otros formatos estructurados e incluso un formato personalizado simplemente definiendo el metamodelo que recoja los conceptos del formato y las transformaciones modelo a modelo y modelo a texto.

4.2 Solución MDE propuesta

En esta sección se va a explicar con detalle cada una de las etapas de la solución MDE propuesta y que ha sido presentada en la sección anterior. Primero se presenta el metamodelo que representa a las facturas, a continuación la definición de la sintaxis concreta para el metamodelo Invoic mediante EMFText, se sigue con la definición d2m realizada a través de Schemol, posteriormente se presenta el metamodelo que representa al formato estructurado EDIFACT, a continuación su sintaxis concreta definida mediante EMFText, después la definición modelo a modelo definida mediante RubyTL y por último, la transformación modelo a texto definida mediante MOFScript.

4.2.1 Metamodelo Invoic

Se ha definido un metamodelo que representa los conceptos y relaciones básicos de una factura. Este metamodelo se ha denominado, Invoic, y se ha creado con la herramienta Emfatic. La Figura 32 muestra el metamodelo como un diagrama de clases y la Figura 33 muestra la definición textual con Emfatic.

El metamodelo Invoic es representado mediante un diagrama de clases UML donde cada concepto es mostrado como una clase y las relaciones entre los conceptos representan las agregaciones y las asociaciones.

Este metamodelo comienza con la metaclase “Envio”. Esta metaclase representa los envíos EDI que se realizan en el seno de la empresa. Cada envío se encuentra identificado por un número y se corresponde a un cliente y emisor particular. Por simplificación, no se le han agregado más atributos a esta clase “Envío”, pero podría ampliarse con la fecha de creación, la fecha de envío, etc. En un envío EDI se pueden mandar un conjunto de facturas. Cada factura almacena si ha sido enviada mediante EDI y el número de envío en el que ha sido transmitida, permitiendo de esta manera tener trazabilidad de los envíos EDI. Cada factura contiene una referencia al cliente y emisor correspondiente y además, se compone a su vez de un conjunto de líneas de factura. Cada línea de factura identifica las unidades servidas de un pedido, su precio así como el total de cada línea.

Se utiliza la metaclase “Emisor” por la opción multiempresa que puede tener cualquier aplicación.

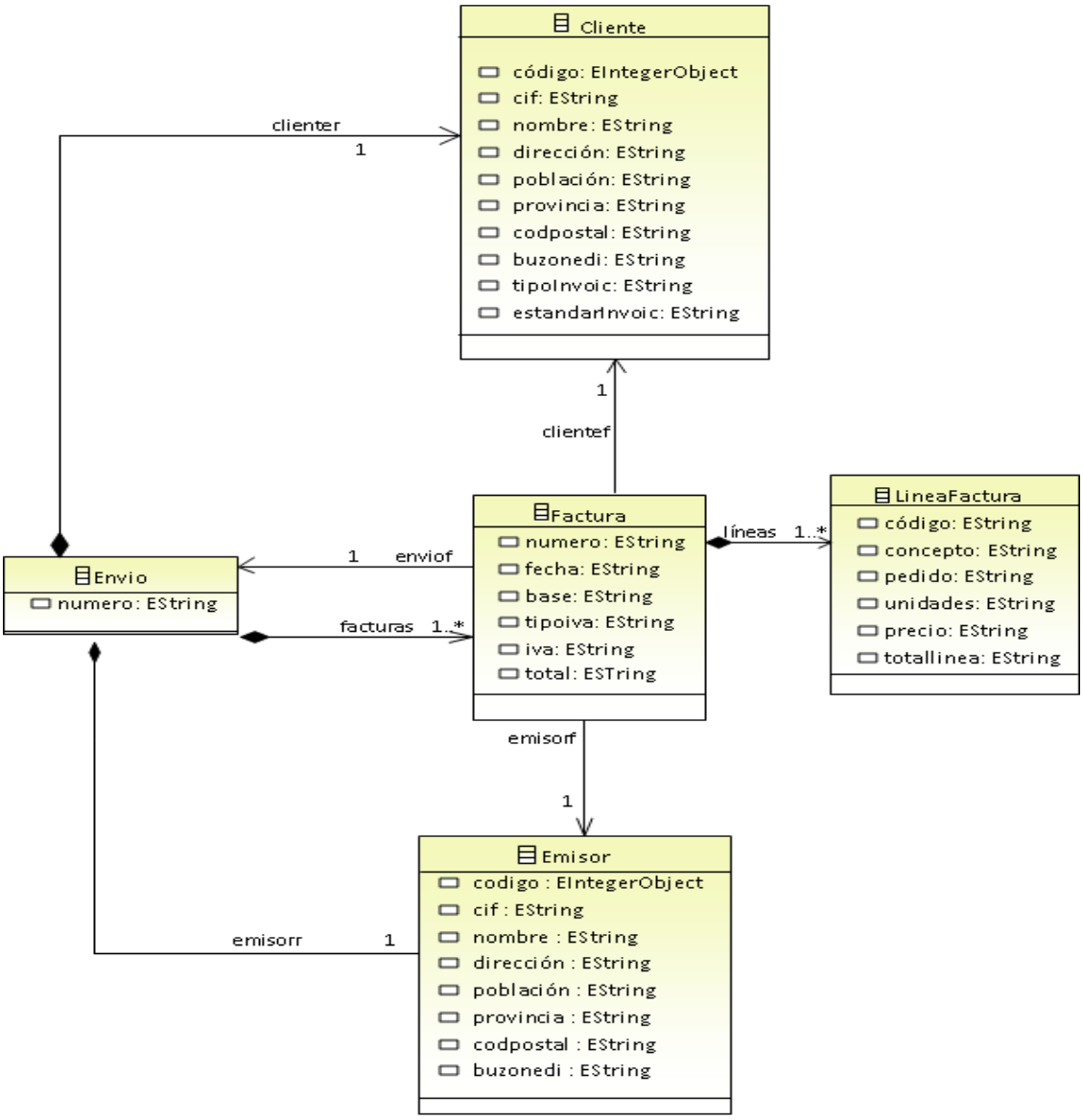
La relación de la metaclase “Envio” con las metaclases “Cliente” y “Emisor” se representa mediante una agregación. Un envío contiene los datos de los clientes y emisores que posteriormente serán referenciados desde las facturas. En la Figura 33 se puede ver que para representar estas relaciones se utiliza la palabra reservada de Emfatic “val”.

La relación de la factura con el cliente y emisor se define mediante una asociación. Una factura es emitida por un emisor particular y es dirigida hacia un cliente determinado. Esta asociación se define en Emfatic mediante la palabra reservada “ref”.

La relación de una factura con sus líneas de factura es claramente una agregación ya que una factura contiene una o muchas líneas de factura que representan el detalle de la misma.

La relación de un envío EDI con las facturas que contiene se define mediante una agregación, puesto que un envío contiene una o muchas facturas enviadas

mediante EDI. Por otro lado, cada factura es enviada en un envío EDI particular, lo cual se representa mediante una asociación con cardinalidad uno.



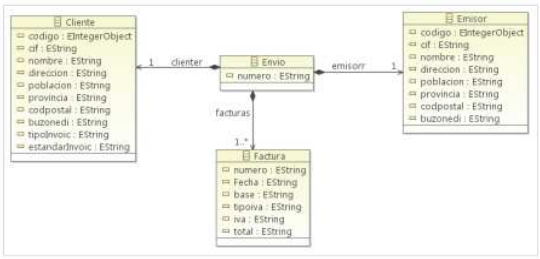
(Fuente: propia)

Figura 32. Metamodelo Invoic.

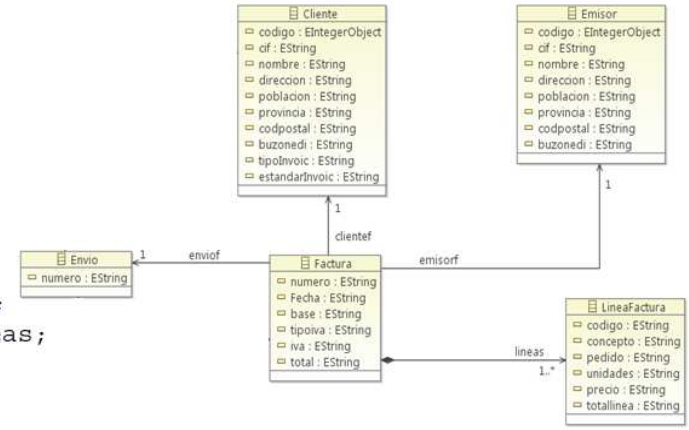
En la figura 32 mostramos la correspondencia entre la definición realizada en Emfatic para el metamodelo Invoic y las clases y relaciones que se generan en un diagrama de clases UML que representa estos conceptos y relaciones.

Una vez realizada la definición mediante Emfatic, el entorno permite la creación del metamodelo Ecore y del diagrama Ecore (diagrama de clases de la Figura 32) a través de dos opciones directas.

```
@namespace(uri="http://fmaster.es/finmaster/invoic", prefix="invoic")
package invoic;

class Envio {
  attr String numero;
  val Cliente[1] clienter;
  val Emisor[1] emisorr;
  val Factura[+] facturas;
}
```

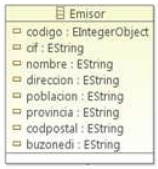



```
class Factura {
  attr String numero;
  attr String Fecha;
  attr String base;
  attr String tipoiva;
  attr String iva;
  attr String total;
  ref Envio[1] enviof;
  ref Emisor[1] emisorf;
  ref Cliente[1] clientef;
  val LineaFactura[+] lineas;
}
```




```
class LineaFactura {
  attr String codigo;
  attr String concepto;
  attr String pedido;
  attr String unidades;
  attr String precio;
  attr String totallinea;
}
```


```
class Emisor {
  attr Integer codigo;
  attr String cif;
  attr String nombre;
  attr String direccion;
  attr String poblacion;
  attr String provincia;
  attr String codpostal;
  attr String buzonedi;
}
```




```
class Cliente {
  attr Integer codigo;
  attr String cif;
  attr String nombre;
  attr String direccion;
  attr String poblacion;
  attr String provincia;
  attr String codpostal;
  attr String buzonedi;
  attr String tipoInvoic;
  attr String estandarInvoic;
}
```

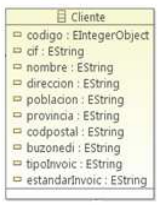


Figura 33. EMFatic para el Metamodelo Invoic. (Fuente: propia)

4.2.2 Sintaxis Concreta del metamodelo Invoic con EMFText

Al comienzo de este proyecto se utilizó la herramienta EMFText para definir la sintaxis concreta o notación como forma de inyectar modelos conformes al metamodelo Invoic. En el transcurso de la investigación se descubre la herramienta Schemol y se decide (previa autorización de los autores) incorporarla a la solución del proyecto puesto que permite trabajar con modelos a lo largo de todo el proyecto. A partir de Schemol, se añade la transformación d2m que inyecta modelos conformes al metamodelo Invoic directamente desde la base de datos. Aun así, resulta interesante el uso de la herramienta EMFText como alternativa al uso de Schemol y por tanto, se decidió seguir en paralelo el estudio y aplicación de esta herramienta como forma de inyectar modelos conformes a un metamodelo.

Partiendo del metamodelo Invoic, se pasa a definir su sintaxis concreta o notación. Para ello, se utiliza la herramienta EMFText que nos permite definir una sintaxis concreta textual para la sintaxis abstracta de un DSL.

Partimos de la gramática BNF que podemos ver a continuación para definir las reglas de EMFText.

Envio ::= EnvioNumero (Cliente)$^{+}$ (Emisor)$^{+}$ (Factura)$^{+}$

Cliente::= ClienteCodigo ClienteCIF ClienteNombre ClienteDireccion
ClientePoblacion ClienteProvincia ClienteCodPostal ClienteBuzonEdi
ClienteTipoInvoic ClienteEstandarInvoic

Emisor::= EmisorCodigo EmisorCIF EmisorNombre EmisorDireccion
EmisorPoblacion EmisorProvincia EmisorCodPostal EmisorBuzonEdi

Factura::= FacturaNumero FacturaFecha EmisorRef ClienteRef (LineaFactura)$^{+}$
FacturaBase FacturaTipoIva FacturaIva FacturaTotal

EmisorRef ::= Identifier

ClienteRef ::= Identifier

LineaFactura ::= LineaFacturaConcepto LineaFacturaPedido
LineaFacturaUnidades LineaFacturaPrecio LineaFacturaTotalLinea

A partir de la gramática anterior definimos las reglas con EMFText. A continuación se incluye el fichero de definición de sintaxis textual (fichero .cs).

```
SYNTAXDEF Invoic
FOR <http://fmaster.es/finmaster/invoic>
START Envio

OPTIONS {
	overridePluginXML = "false";
	overrideManifest = "false";
	reloadGeneratorModel = "true";
	generateCodeFromGeneratorModel = "true";
	generateTestAction = "true";

}

TOKENS {
	DEFINE STRING_LITERAL $'\''('\\ '('n'|'r'|'t'|'b'|'f'|'"'|'\''|'\\
'|'>'|'u'('0' .. '9'|'a'.. 'f'|'A'.. 'F')('0' .. '9'|'a'.. 'f'|'A'.. 'F')('0'
.. '9'|'a'.. 'f'|'A'.. 'F')('0' .. '9'|'a'.. 'f'|'A'.. 'F')|.) |~( '\''|'\\
'))('\\ '('n'|'r'|'t'|'b'|'f'|'"'|'\''|'\\ '|'>'|'u'('0' .. '9'|'a'..
'f'|'A'.. 'F')('0' .. '9'|'a'.. 'f'|'A'.. 'F')('0' .. '9'|'a'.. 'f'|'A'..
'F')('0' .. '9'|'a'.. 'f'|'A'.. 'F') |.) |~( '\''|'\\ '))*'\''$;

}

RULES {

Envio::= "Envio" numero[] clienter emisorr "Facturas" "{" facturas+ "}";

Factura::= "Factura" numero[STRING_LITERAL] Fecha[STRING_LITERAL] "{"
"Emisor" emisorf[] "Cliente" clientef[] "LineaFactura" "{" lineas+ "}" "Base"
base[STRING_LITERAL] "TipoIva" tipoiva[] "IVA" iva[STRING_LITERAL] "Total"
total[STRING_LITERAL] "}";

LineaFactura::= "Linea" concepto[STRING_LITERAL] pedido[STRING_LITERAL]
unidades[] precio[STRING_LITERAL] totallinea[STRING_LITERAL];

Emisor::= "Emisor" "{" codigo[] cif[] nombre[STRING_LITERAL]
direccion[STRING_LITERAL] poblacion[STRING_LITERAL] provincia[STRING_LITERAL]
codpostal[] buzonedi[] "}";

Cliente::= "Cliente" "{" codigo[] cif[] nombre[STRING_LITERAL]
direccion[STRING_LITERAL] poblacion[STRING_LITERAL] provincia[STRING_LITERAL]
codpostal[] buzonedi[] tipoInvoic[] estandarInvoic[] "}";
```

En bloque de token léxicos nos definimos un token personalizado que permita identificar una cadena de caracteres con los caracteres identificados sobre la propia definición durante la fase de reconocimiento.

En el bloque de reglas se definen las reglas EMFText correspondientes a la gramática BNF identificada anteriormente.

Por último, se muestra un ejemplo de modelo textual definido a partir de las reglas EMFText y que conforma con el metamodelo Invoic. Este modelo será

almacenado en formato XMI para su uso posterior en las transformaciones modelo a modelo.

```
Envio 1

      Cliente { 1 ESB12265645 'Transporte SL' 'C/Del Mar Nº5' 'Murcia'
      'Murcia' 30110 8052602236 Edifact D93A }

      Emisor { 1 ESB73261604 'Transdemur 2003' 'C/Escritor Ruiz
      Aguilera' 'Cabezo de Torres' 'Murcia' 30110 8052602236 }

      Facturas{
            Factura '12012' '13/05/2012' {
                  Emisor ESB73261604
                  Cliente ESB12265645
                  LineaFactura{
                        Linea 'Puro Zumo de Naranja' '380' 1000 '0,85'
                        '850.20'
                        Linea 'Cocktail de Multifrutas' '380' 3000
                        '0,55' '1650.80'
                  }
                  Base '2501'
                  TipoIva 18
                  IVA '450.18'
                  Total '2951.18'
            }
      }
```

4.2.3 Transformación de datos a modelo con Schemol

Otra forma posible para inyectar modelos conformes al metamodelo Invoic es utilizar la herramienta Schemol. Para ello se ha escrito una transformación Schemol que ha establecido la correspondencia entre el esquema de las bases de datos relativo a las facturas y el metamodelo Invoic.

La Figura 34 muestra el esquema de la base de datos que representa las facturas. En este esquema se define la tabla “EnvioEDI” que almacena los envíos EDI realizados en la empresa y que contiene una referencia al cliente y emisor al que corresponden. También se define la tabla “Factura” que almacena los datos de la factura y que está compuesta por un conjunto de líneas de facturas representadas por la tabla “LineaFactura”. Además, la tabla Factura tiene referencias hacia el emisor y el cliente al que corresponden.

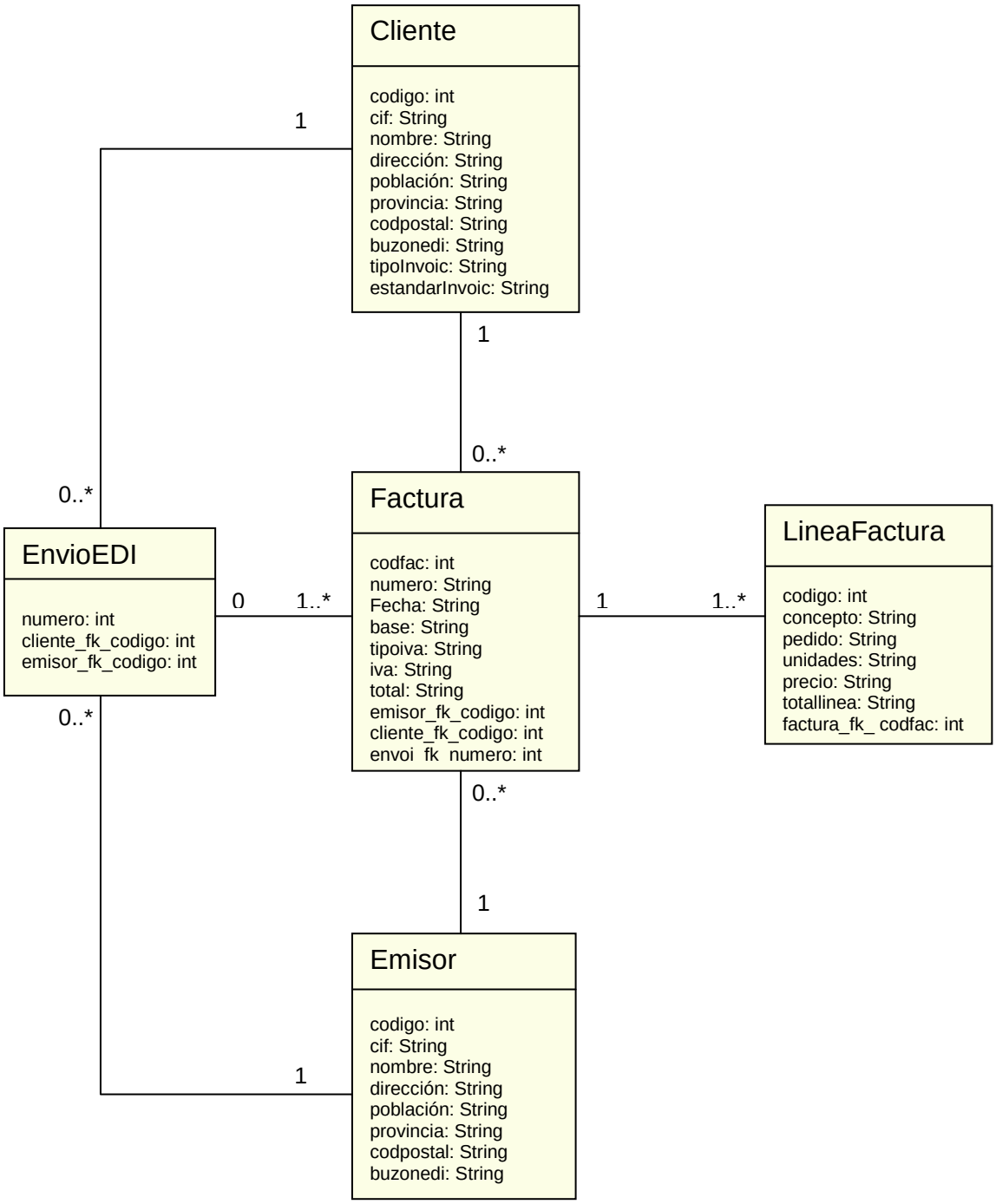


Figura 34. Esquema de la base de datos. (Fuente: propia)

La Figura 35 muestra la transformación Schemol que inyecta modelos conformes al metamodelo Invoic a partir de los datos de la base de datos que recibe como entrada.

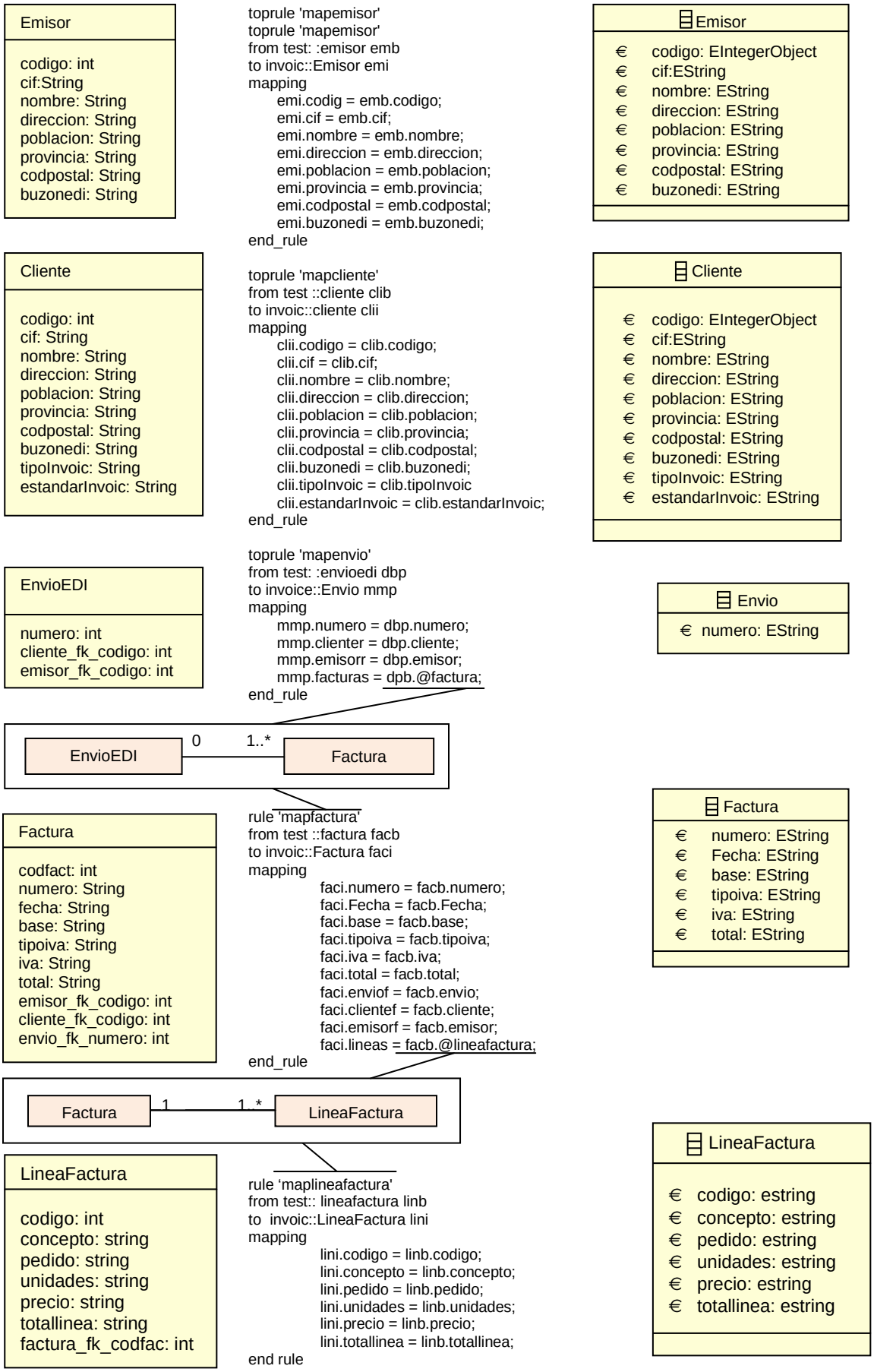


Figura 35. Transformación datos- a-modelo. (Fuente: propia)

Las reglas que marcan el punto de comienzo de la transformación y que son lanzadas directamente son las de tipo “toprule”, en este caso “mapemisor”, “mapcliente” y “mapenvio”.

La regla “mapemisor” de tipo “toprule” establece la correspondencia entre la tabla “Emisor” de la base de datos y la metaclase “Emisor” del metamodelo destino. En la sección *mapping* de esta regla se establecen las propiedades de la metaclase “Emisor” a partir de las propiedades de la tabla “Emisor”.

La regla “mapcliente” de tipo “toprule” establece la correspondencia entre la tabla “Cliente” de la base de datos y la metaclase “Cliente” del metamodelo destino.

La última regla de tipo “toprule”, “mapenvio”, establece la correspondencia entre la tabla “EnvioEDI” de la base de datos y la metaclase “Envio” del metamodelo destino. A partir de esta regla (a diferencia de las anteriores) se deriva la ejecución de la regla “mapfactura”.

La regla “mapfactura” de tipo “rule” establece la correspondencia entre la tabla “Factura” de la base de datos y la metaclase “Factura” del metamodelo destino. Esta regla es lanzada a través del binding contenido en la regla anterior, basándose en que la tabla “EnvioEDI” de la base de datos está compuesta por una o muchas facturas.

La regla “maplineafactura” de tipo “rule” establece la correspondencia entre la tabla “LineaFactura” de la base de datos y la metaclase “LineaFactura” del metamodelo destino. Esta regla se ejecuta a través del binding contenido en la regla “mapfactura” basado en la relación de las tablas “Factura” y “LineaFactura” de la base de datos.

4.2.4 Metamodelo Edifact

Se ha definido un metamodelo que representa los conceptos y relaciones del formato estructurado EDIFACT. Este metamodelo se ha denominado, Edifact, y se ha creado con la herramienta Emfatic. La Figura 36 muestra el metamodelo como un diagrama de clases y la Figura 37 muestra la definición textual con Emfatic.

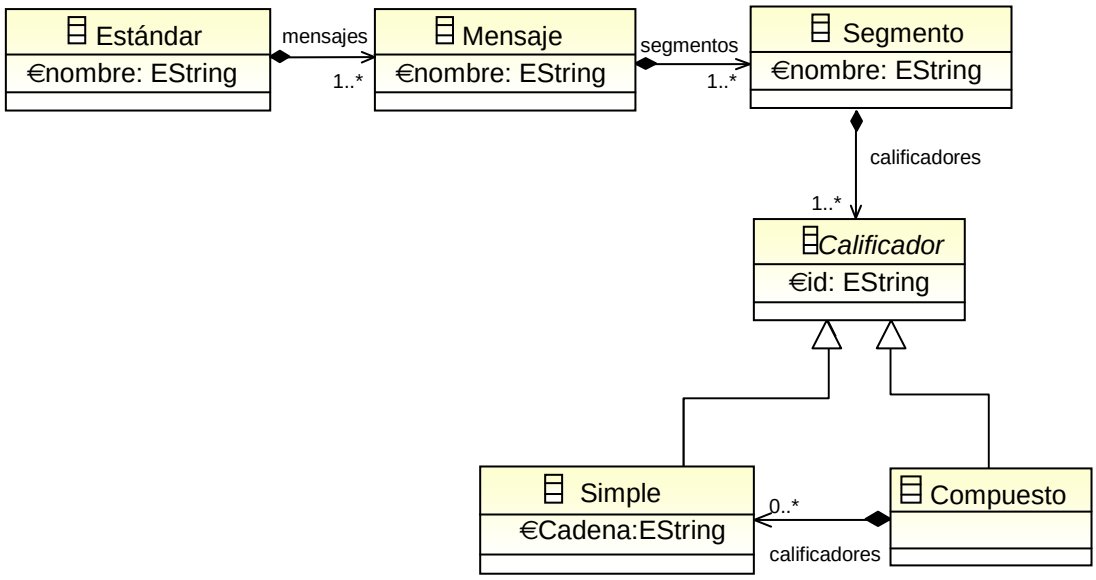


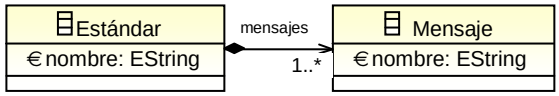
Figura 36. Metamodelo Edifact. (Fuente: propia)

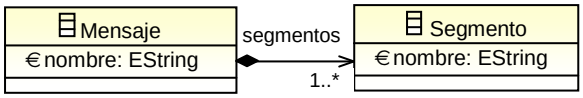
El metamodelo comienza con la metaclase “Estandar”. Esta metaclase representa los diferentes estándares que componen la especificación UN/EDIFACT (D93A, D96Am, etc.). Cada estándar está compuesto por un conjunto de mensajes (Invoic, Order, Desadv, etc.) representados por la metaclase “Mensaje” y por la relación de agregación entre las metaclases “Estandar” y “Mensaje”. A su vez, cada mensaje está formado por un conjunto de segmentos (UNH, BGM, etc.) representados por la metaclase “Segmento” y la relación de agregación entre las metaclases “Mensaje” y “Segmento”. Por último, cada segmento está compuesto por calificadores que pueden ser de dos tipos, simples y compuestos. A su vez, los calificadores compuestos pueden estar formados por calificadores simples representado por la relación de agregación entre las metaclases “Simple” y “Compuesto”.

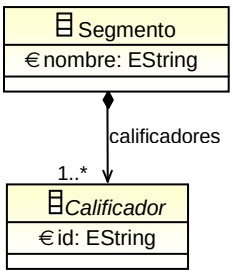
Este metamodelo Edifact es válido para cualquier mensaje definido a través de Edifact, aunque en este proyecto en particular se van a generar modelos que conforman al metamodelo para las facturas. En el Anexo A se encuentra detallada la estructura del fichero Invoic en el estándar D93A.

En la figura 37 se muestra la correspondencia entre la definición realizada en Emfatic para el metamodelo Edifact y las clases y relaciones que se generan en un diagrama de clases UML que representa estos conceptos y relaciones.

```
@namespace(uri="http://fmaster.es/finmaster/invoic",prefix="edifact")
package edifact;

class Estandar {
  attr String nombre;
  val Mensaje[+] mensajes;
}

class Mensaje {
  attr String nombre;
  val Segmento[+] segmentos;
}

class Segmento {
  attr String nombre;
  val Calificador[+] calificadores;
}

abstract class Calificador {
  attr String ~id;
}

class Simple extends Calificador {
  attr String cadena;
}

class Compuesto extends Calificador {
  val Simple[*] calificadores;
}
```







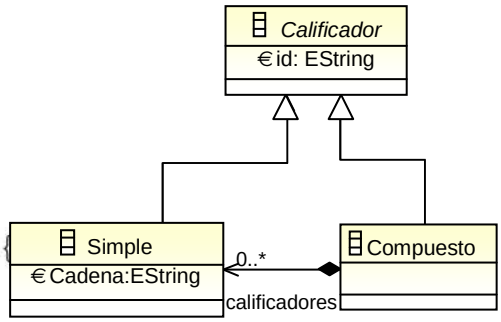


Figura 37. EMFatic para el Metamodelo Edifact. (Fuente: propia)

4.2.5 Sintaxis Concreta del metamodelo Edifact con EMFText

La definición de la sintaxis concreta o notación para el metamodelo Edifact se realiza en una primera fase del proyecto con el objetivo de poder inyectar modelos conformes al metamodelo Edifact pudiendo de esta manera probar las transformaciones modelo a texto definidas mediante MOFScript. Una vez el proyecto queda ensamblado, este inyección de modelos pierde importancia puesto que los modelos que conforman con este metamodelo son proporcionados mediante la transformación modelo a modelo definida con RubyTL. Por tanto, se presenta esta sección aunque en el esquema general de la solución no queda incluida.

Una vez definido el metamodelo, se pasa a definir su sintaxis concreta o notación. Al igual que para el metamodelo Invoic, se utiliza la herramienta EMFText que permite definir una sintaxis concreta textual para la sintaxis abstracta de un DSL.

A continuación, se muestra la gramática BNF de la que partimos.

Estandar ::= EstandarNombre (Mensaje)$^{+}$

Mensaje ::= MensajeNombre (Segmento)$^{+}$

Segmento ::= SegmentoNombre (Calificador)$^{+}$

Calificador ::= Simple | Compuesto

Simple ::= SimpleId SimpleCadena

Compuesto :: CompuestoId (Simple)*

A partir de la gramática anterior definimos las reglas con EMFText.

```
SYNTAXDEF Edifact
FOR <http://fmaster.es/finmaster/edifact>
START Estandar
OPTIONS {
      overridePluginXML = "false";
      overrideManifest = "false";
      reloadGeneratorModel = "true";
      generateCodeFromGeneratorModel = "true";
      generateTestAction = "true";
}
```

```
TOKENS {
DEFINE STRING_LITERAL $'\''('\\ '('n'|'r'|'t'|'b'|'f'|'"'|'\''|'\\
'|'>'|'u'('0' .. '9'|'a'.. 'f'|'A'.. 'F')('0' .. '9'|'a'.. 'f'|'A'..
'F')('0' .. '9'|'a'.. 'f'|'A'.. 'F')('0' .. '9'|'a'.. 'f'|'A'..
'F')|.) |~( '\''|'\\ '))('\\ '('n'|'r'|'t'|'b'|'f'|'"'|'\''|'\\
'|'>'|'u'('0' .. '9'|'a'.. 'f'|'A'.. 'F')('0' .. '9'|'a'.. 'f'|'A'..
'F')('0' .. '9'|'a'.. 'f'|'A'.. 'F')('0' .. '9'|'a'.. 'f'|'A'.. 'F')
|.) |~( '\''|'\\ '))*'\''$;
}
```

```
RULES {
Estandar::="Estandar" nombre[] "{" mensajes+ "}";

Mensaje::="Mensaje" nombre[] "{" segmentos+ "}";

Segmento::="Segmento" nombre[] "{" calificadores+ "}";

Simple::="CalificadorS" id[] (cadena[STRING_LITERAL])?;
Compuesto::="CalificadorC" id[] "{" calificadores* "}"; }
```

Por último, se muestra un ejemplo de modelo textual definido a partir de las reglas EMFText y que conforma con el metamodelo Edifact. Este modelo será almacenado en formato XMI para su uso posterior en las transformaciones modelo a texto.

```
Estandar D93A{
      Mensaje Invoic{
            Segmento UNB{
                  CalificadorC S001{
                        CalificadorS 0001 'UNOA'
                        CalificadorS 0002 '2'
                  }
                  CalificadorC S002{
                        CalificadorS 0004 'EMISOR'
                  }
                  CalificadorC S003{
                        CalificadorS 0004 'DESTINATARIO'
                  }
                  CalificadorC S004{
                        CalificadorS 0017 '990802'
                        CalificadorS 0019 '1557'
                  }
                  CalificadorS 0020 '9908021557'
            }
            Segmento UNH{
                  CalificadorS 0062 'INVOIC001'
                  CalificadorC S009{
                        CalificadorS 0065 'INVOIC'
                        CalificadorS 0052 'D'
                        CalificadorS 0054 '93A'
                        CalificadorS 0051 'UN'
                  }
            }
      }
}
```

4.2.6 Transformación de modelos Invoic a modelos EDIFACT

En la Figura 38 se pueden ver las definiciones de las transfomaciones modelo-a-modelo realizadas con RubyTL para transformar modelos que conforman al metamodelo Invoic a modelos que conformar con el metamodelo Edifact.

En primer lugar, nos definimos la regla “Envio2Estandar” de tipo “toprule”. Esta regla establece la correspondencia entre la metaclase “Envio” del metamodelo origen Invoic con la metaclase “Estandar” del metamodelo destino Edifact. Al ejecutar la regla se crea una instancia del elemento destino. En el *mapping* de esta regla establecemos el nombre del estándar accediendo a las propiedades del cliente, ya que, cada cliente almacena el tipo de formato estructurado que ha solicitado para el envío de facturas electrónicas. Por último, el motor de RubyTL al llegar a la segunda regla del mapping, ve que el tipo de la derecha

de la regla no es compatible con el tipo de la izquierda. Una vez realizada esta comprobación, el motor RubyTL busca reglas que permitan transformar la parte de la derecha en la parte de la izquierda. En esta búsqueda encuentra la regla , “Factura2Mensaje”.

```
top_rule 'Envio2Estandar' do

  from Invoic::Envio
  to Edifact::Estandar
  mapping do |en, ed|
    ed.nombre = en.clienter.estandarInvoic
    ed.mensajes = en.facturas
  end
end
```

La regla “Factura2Mensaje” crea un mensaje por cada factura que encuentra en el modelo origen. Por tanto, establece la relación entre la metaclase “Factura” y la metaclase “Mensaje”. En el *mapping* de esta regla se establece el nombre del mensaje, en concreto “Invoic” (dado que se está trabajando con la factura electrónica) y se ejecutan los dos últimos *binding* para lo que es necesario una búsqueda de reglas que permitan obtener segmentos a partir de la factura y de sus líneas.

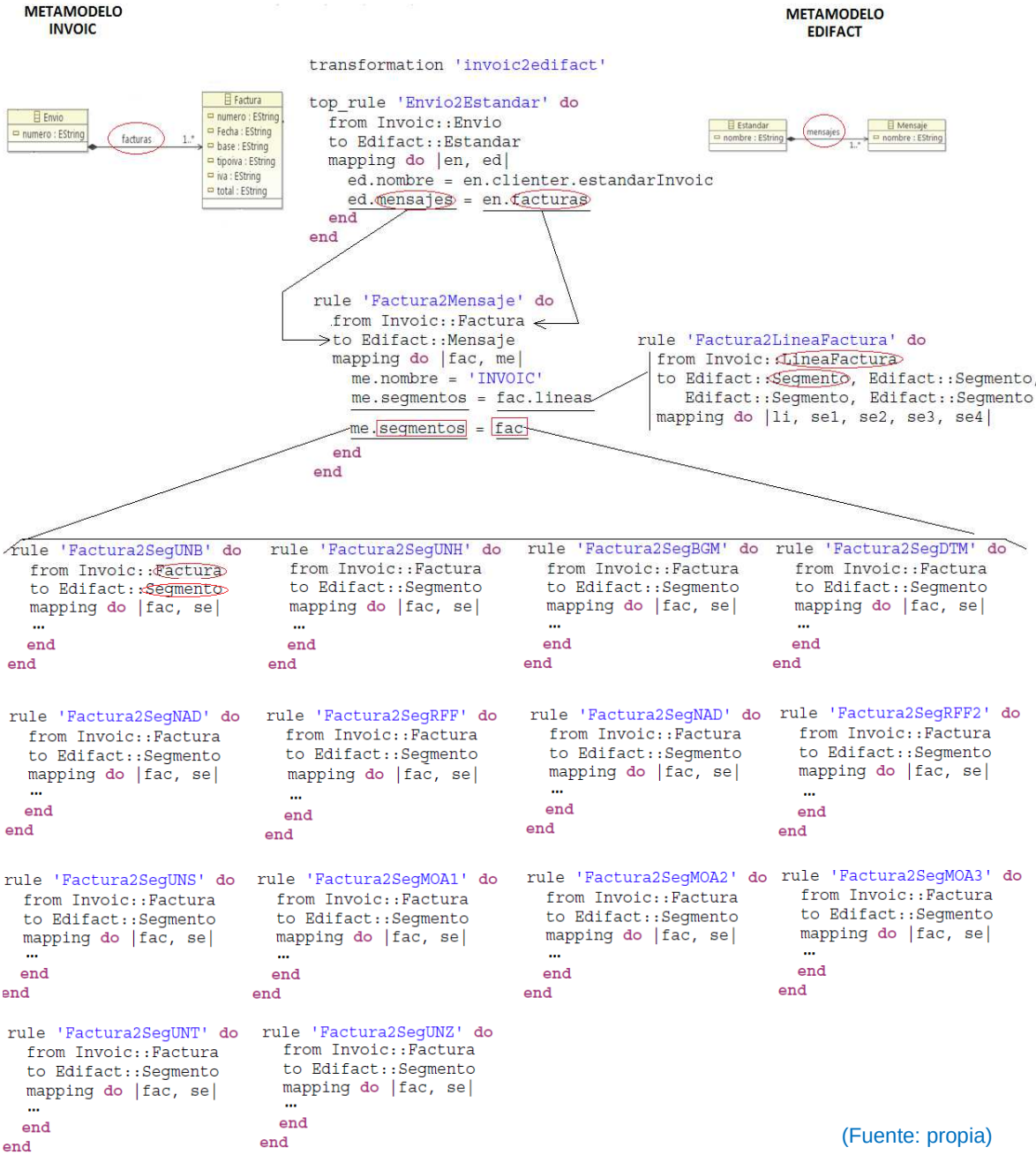
```
rule 'Factura2Mensaje' do

  from Invoic::Factura
  to Edifact::Mensaje
  mapping do |fac, me|
    me.nombre = 'INVOIC'
    me.segmentos = fac.lineas
    me.segmentos = fac
  end
end
```

De la primera búsqueda lanzada por el motor de RubyTL en la anterior regla, se buscan reglas que permitan transformar las líneas de una factura en segmentos. En esta búsqueda, se encuentra la regla “Factura2LineaFactura” que se puede ver a continuación. En el cuerpo de la misma se crean cuatro segmentos por cada línea de la factura. Esta composición de segmentos por línea de factura esta especificada en el estándar EDIFACT. Por cada segmento se crean calificadores simples y/o compuestos y se agregan al segmento siguiendo la especificación del estándar. Otra característica de esta regla, es la utilización de código Ruby en el cuerpo de la misma.

```
rule 'Factura2LineaFactura' do
  from Invoic::LineaFactura
  to Edifact::Segmento, Edifact::Segmento, Edifact::Segmento,
Edifact::Segmento
  mapping do |li, se1, se2, se3, se4|
      se1.nombre = 'LIN'
      se1.calificadores << Edifact::Simple.new(:id =>'1082', :cadena
      => li.codigo)
      se1.calificadores << Edifact::Simple.new(:id =>'1229', :cadena
      => '482')


      se2.nombre = 'IMD'
      se2.calificadores << Edifact::Simple.new(:id =>'7077', :cadena
      => 'F')
      se2.calificadores << Edifact::Simple.new(:id =>'7081', :cadena
      => 'M')
      co =  Edifact::Compuesto.new(:id => 'C273')
      co.calificadores << Edifact::Simple.new(:id =>'7009', :cadena =>
      '')
      co.calificadores << Edifact::Simple.new(:id =>'1131', :cadena =>
      '')
      co.calificadores << Edifact::Simple.new(:id =>'3055', :cadena =>
      '')
      co.calificadores << Edifact::Simple.new(:id =>'7008', :cadena =>
      li.concepto)
      se2.calificadores << co


      se3.nombre = 'QTY'
      co =  Edifact::Compuesto.new(:id => 'C186')
      co.calificadores << Edifact::Simple.new(:id =>'6063', :cadena =>
      '47')
      co.calificadores << Edifact::Simple.new(:id =>'6060', :cadena =>
      li.unidades)
      se3.calificadores << co


      se4.nombre = 'MOA'
      co =  Edifact::Compuesto.new(:id => 'C516')
      co.calificadores << Edifact::Simple.new(:id =>'5025', :cadena =>
      '6')
      co.calificadores << Edifact::Simple.new(:id =>'5004', :cadena =>
      li.totallinea)
      se4.calificadores << co
  end
end
```

De la segunda búsqueda lanzada a partir de la regla “Factura2Mensaje” se encuentran reglas que crean un segmento por cada factura. Todas ellas crean los segmentos según el estándar EDIFACT. En el cuerpo de estas reglas se utiliza código Ruby para crear los calificadores y agregarlos al segmento. A continuación se puede ver un ejemplo.

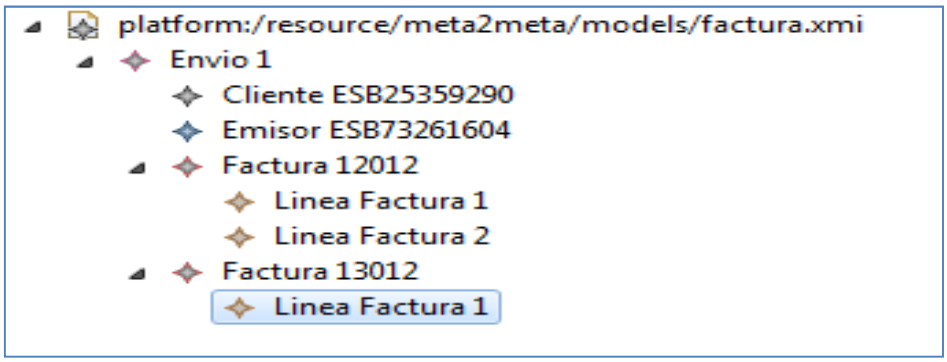
```
rule 'Factura2SegRFF' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='RFF'
    co =  Edifact::Compuesto.new(:id => 'C506')
    co.calificadores << Edifact::Simple.new(:id =>'1153', :cadena =>
'VA')
    co.calificadores << Edifact::Simple.new(:id =>'1154', :cadena =>
fac.emisorf.cif)
    se.calificadores << co
  end
end
```



(Fuente: propia)

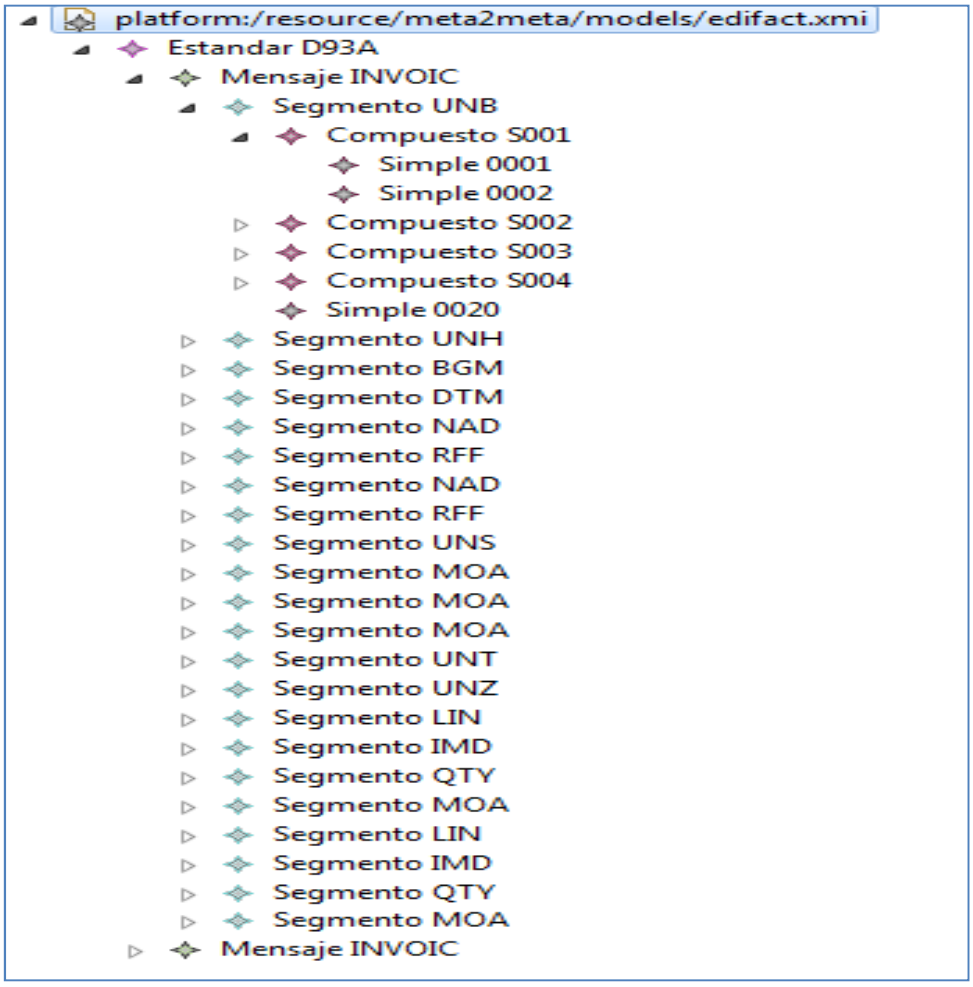
Figura 38. Transformación modelo a modelo con RubyTL.

A continuación, se muestra en la Figura 39 un ejemplo de modelo de entrada que conforma con el metamodelo Invoic y en la Figura 40 el modelo obtenido tras aplicar la transformación modelo a modelo mediante la herramienta RubyTL.



(Fuente: propia)

Figura 39. Modelo de entrada que conforma con el metamodelo Invoic



(Fuente: propia)

Figura 40. Ejemplo modelo de salida generado con RubyTL.

4.2.7 Transformación de modelos EDIFACT a documentos EDIFACT

El último paso de la cadena de transformación es generar los documentos en formato EDIFACT a partir de los modelos EDIFACT obtenidos de la transformación modelo a modelo explicada en el apartado anterior. Se ha usado el lenguaje MOFScript para escribir la transformación modelo-a-texto que se muestra en la Figura 42.

La transformación comienza con la regla principal “main”, indicando el elemento del metamodelo de entrada utilizado como punto de partida, en este caso la metaclase “Estandar” del metamodelo Edifact. Como se puede ver a continuación sobre la regla, utilizamos el iterador *forEach* para recorrer los mensajes que se encuentran dentro de la metaclase “Estandar”. Por cada elemento encontrado, se lanza la regla “mapMensaje”.

```
edifact.Estandar::main(){
    file(outputFile)
    'UNA:+,?\''
    newline(1)
    self.mensajes->forEach(c){ c.mapMensaje() }
}
```

La regla “mapMensaje” es ejecutada sobre el elemento “Mensaje” del metamodelo de entrada Edifact. Partiendo de la relación de agregación entre las metaclases “Mensaje” y “Segmento”, en el cuerpo de esta regla se recorren los segmentos que forman el mensaje llamando por cada uno de ellos a la regla “mapSegmento”.

La regla “mapSegmento” se ejecuta sobre la metaclase “Segmento” del metamodelo Edifact. A través del *foreach* recorremos todos los calificadores que componen el segmento que estamos tratando. Por cada calificador se ejecuta la sentencia “oclIsTypeOf()” para diferenciar los calificadores simples de los compuestos según la jerarquía de herencia que se puede ver en el metamodelo Edifact. En el caso de que se trate de un calificador simple, ejecutamos la regla “mapCalificadorS”. Por el contrario, si se trata de un calificador compuesto, se ejecuta la regla “mapCalificadorC”.

Como se puede ver sobre el metamodelo Edifact, un calificador compuesto puede estar formado por calificadores simples. La regla “mapCalificadorC” se

ejecuta sobre la metaclase “Compuesto” del metamodelo de entrada Edifact y recorre los calificadores simples que componen el calificador compuesto llamando por cada uno de ellos a la regla “mapCalificadorS”.

Por último, la regla “mapCalificadorS” es la encargada de generar el texto sobre el fichero de salida que corresponde a la factura electrónica. En el cuerpo de esta regla, se manda sobre el fichero de salida el contenido de la propiedad “cadena” de cada uno de los calificadores simples tratados. Sobre la Figura 41 se pueden ver tres ejemplos de calificadores simples y los valores de sus propiedades.

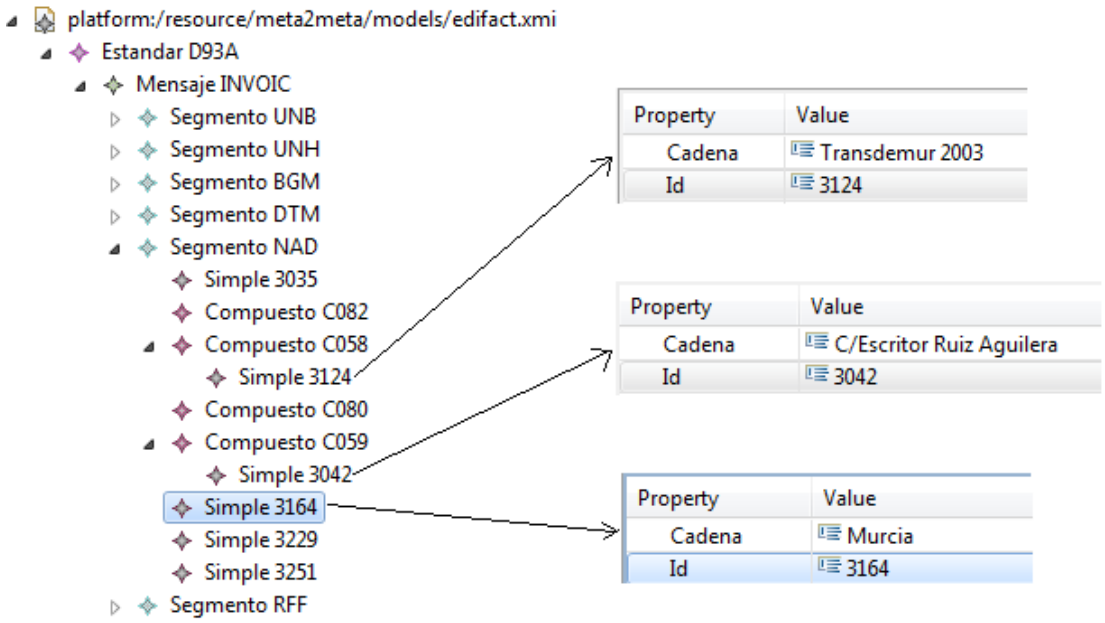


Figura 41. Ejemplo de modelo Edifact. (Fuente: propia)

Sobre la Figura 42 se puede ver la estructura general de la transformación así como los elementos del metamodelo de entrada sobre los que están basados mediante diagramas de clases UML.

```
texttransformation tf (in edifact:"http://fmaster.es/finmaster/edifact") {
  property outputFile : String = "invoic.edifact"

edifact.Estandar::main(){
  file(outputFile)
  'UNA:+,?\''
  newline(1)
  self.mensajes->forEach(c){ c.mapMensaje() }
}

edifact.Mensaje::mapMensaje() {
  self.segmentos->forEach(d){ d.mapSegmento() }
}

edifact.Segmento::mapSegmento() {
  self.nombre
  self.calificadores->forEach(e){
   '+'
    if (e.oclIsTypeOf(edifact.Compuesto)) {
      e.mapCalificadorC()
    } else {
      e.mapCalificadorS()
    }
  }
  '\'' newline(1)
}

edifact.Simple::mapCalificadorS() {
    var cad:string = "'"
    if (self.cadena.substring(0,1) = "'")
      cad = s self.cadena.substring(1,
      self.cadena.size()-1)
    else cad = self.cadena
    cad
}

edifact.Compuesto::mapCalificadorC() {
  var pri:boolean = true
  self.calificadores->forEach(x){
    if (pri = true){
      pri = false
      x.mapCalificadorS()
    }
    else ':'x.mapCalificadorS()
  }
 }
}
}
```

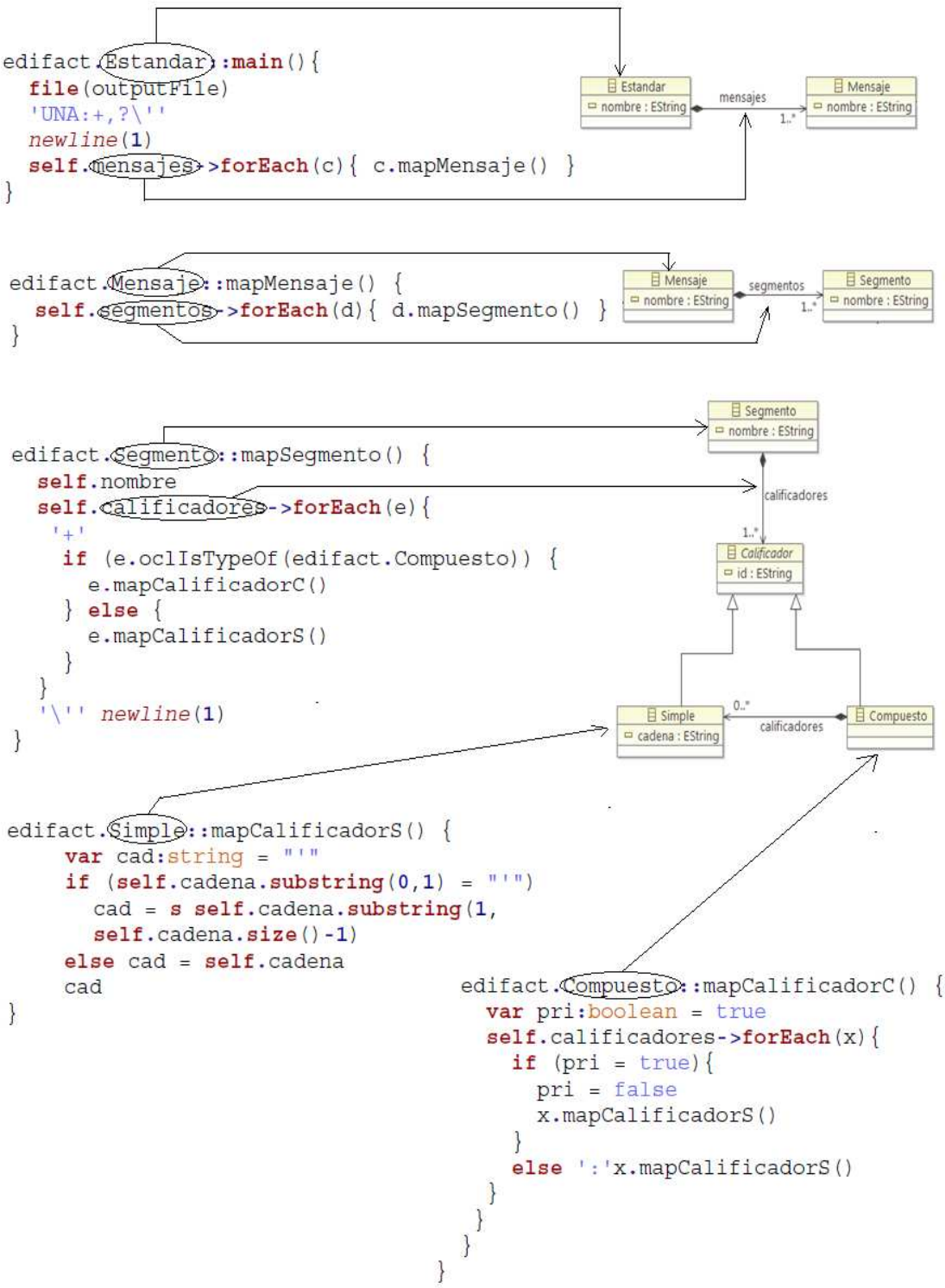


Figura 42. Transformación MOFScript. (Fuente: propia)

A continuación se puede ver el fichero obtenido tras ejecutar la transformación modelo a texto sobre el modelo de entrada que conforma con el metamodelo Edifact de la Figura 36. El fichero obtenido cumple con la sintaxis D93A del mensaje Invoic de Edifact.

```
UNA:+,?'
UNB+UNOA:2+8052602236+8052602748+120615:0931+1'
UNH+INVOIC1+INVOIC:D:93A:UN'
BGM+380+1'
DTM+137:13/05/2012:101'
NAD+EM++Transdemur 2003++C/Escritor Ruiz Aguilera+Murcia++30110'
RFF+VA:ESB73261604'
NAD+RE++TRANSPORTE SL++C/DEL MAR Nº5+MURCIA++30110'
RFF+VA:ESB12364525'
UNS+S'
MOA+125:2501'
MOA+176:450.18'
MOA+139:2951.18'
UNT+16+INVOIC1'
UNZ+1+1'
LIN+1+482'
IMD+F+M+:::Puro Zumo de Naranja'
QTY+47:1000'
MOA+6:850.20'
LIN+2+482'
IMD+F+M+:::Cocktail de Multifrutas'
QTY+47:3000'
MOA+6:1650.80'
UNB+UNOA:2+8052602236+8052602748+120615:0931+1'
UNH+INVOIC1+INVOIC:D:93A:UN'
BGM+380+1'
DTM+137:13/05/2012:101'
NAD+EM++Transdemur 2003++C/Escritor Ruiz Aguilera+Murcia++30110'
RFF+VA:ESB73261604'
NAD+RE++TRANSPORTE SL++C/DEL MAR Nº5+MURCIA++30110'
RFF+VA:ESB12364525'
UNS+S'
MOA+125:1000'
MOA+176:180.00'
MOA+139:1180.00'
UNT+16+INVOIC1'
UNZ+1+1'
LIN+1+482'
IMD+F+M+:::Don Juan'
QTY+47:1000'
MOA+6:1000.00'
```

4 Conclusiones y Trabajos Futuros

5.1 Conclusiones

El uso de los ficheros electrónicos genera un ahorro de costes a la vez que permite a las empresas ser más eficientes. Con el impulso de las diferentes organizaciones públicas, la implantación del intercambio de los ficheros electrónicos es cada vez más una realidad en las empresas que ven la oportunidad de automatizar tareas repetitivas destinando los recursos liberados a otras tareas y reduciendo los posibles errores derivados del factor humano.

En este proyecto, se aborda la implantación de los ficheros electrónicos desde la Ingeniería Dirigida por Modelos. Dicha ingeniería permite trabajar en un nivel de abstracción superior, trabajando con modelos y automatizando la generación de los ficheros electrónicos.

Este proyecto propone la integración MDE con una tecnología existente como es EDI, de manera que las técnicas MDE se apliquen sobre los documentos electrónicos.

El uso de Schemol nos da una gran potencia puesto que podemos implantar el uso de los modelos desde la base de datos, sin tener que crear los modelos, por ejemplo con el DSL creado con EMFText. El potencial de esta herramienta es claro, ya que podemos conectarnos directamente a la base de datos y generar una estructura totalmente transparente desde el inicio hasta el fin.

La solución aportada mediante este proyecto permite evitar las prácticas que algunas empresas han utilizado para intentar implantar el envío electrónico de datos. Algunos ejemplos de estas prácticas son el uso de un software que recorre la estructura de la base de datos, almacenando los datos en un fichero de formato txt que posteriormente es transformado por una empresa externa en otro fichero que cumple el formato EDIFACT. Otra práctica llevada a cabo por algunas empresas es generar el fichero electrónico desde la propia base de datos con la dificultad que conlleva el mantenimiento de dicha exportación ante los posibles cambios que se deban de afrontar.

Desde un punto de vista comercial, el empresario siempre pretende abarcar el mayor grado de automatización posible. El uso de ficheros electrónicos le permite no solo automatizar la factura electrónica, sino que puede abarcar todo el *workflow* de la empresa obteniendo beneficios muy claros.

Con los ficheros electrónicos que genera una empresa y envía a otra se reducen muchas tareas manuales como puede ser, en el caso de la factura electrónica, la impresión, el sobre o tener que ir a correos. Todas estas tareas quedan completamente eliminadas puesto que se puede llegar a un sistema en el que con procesos automáticos se pueden enviar todas las facturas realizadas durante una jornada de trabajo mediante un sistema automático que se ejecute al final del día.

Ahora bien, si con el envío se consiguen grandes ventajas, con la recepción de ficheros electrónicos no se consiguen menos. Esta recepción nos permite realizar tareas de forma automática como la conciliación o la contabilización. Por tanto, una vez la empresa ha instalado la herramienta, debe intentar conseguir que sus proveedores asuman el uso de los ficheros electrónicos como una tarea más dentro del proceso.

A continuación se muestran los beneficios que aporta la solución planteada en este proyecto.

Generación automática del documento electrónico a enviar

A partir de los datos almacenados en la base de datos y mediante la estructura planteada se genera de manera automática el documento electrónico que se pretende enviar aplicando transformaciones datos-a-modelo, modelo-a-modelo y modelo-a-texto.

Representación independiente a cualquier tecnología

Mediante un metamodelo se puede representar los conceptos y relaciones a modelar de forma independiente a cualquier tecnología. Por ejemplo, mediante el metamodelo Invoic se reflejan los conceptos y relaciones de las facturas en un nivel de abstracción que no implica el uso de una determinada tecnología.

Generación de documentos en múltiples formatos a partir de un modelo

Una vez que se cuenta con un modelo extraído de la base de datos que conforma con el metamodelo inicial, definiendo un metamodelo por cada formato estructurado y la transformación del modelo inicial al modelo del formato estructurado, podemos abarcar todos los formatos estructurados que se deseen y desde estos generar el documento electrónico de forma automática. Por tanto, añadir un nuevo formato estructurado solo implica definir el metamodelo y las transformaciones m2m y m2t.

Reutilización de los formatos estructurados al añadir nuevos documentos electrónicos

A través de la estructura planteada, las definiciones de los formatos estructurados mediante un metamodelo y su generación a texto mediante la definición de una transformación modelo a texto, son reutilizados cuando se pretende incorporar nuevos documentos electrónicos. Por ejemplo, si sobre la estructura planteada para la factura electrónica se quiere añadir el uso de pedidos electrónicos, se debería definir el metamodelo para los pedidos, la transformación d2m para extraer modelos desde la base de datos y la transformación m2m para transformar a modelos en los distintos formatos estructurados, pero el metamodelo Edifact y las transformaciones m2t serían las que se han desarrollado con anterioridad.

Definición de formatos estructurados personalizados

Cuando un cliente o proveedor exige la adaptación a un formato electrónico particular que no está recogido en ningún estándar, la estructura planteada permite añadir este formato personalizado como un nuevo formato estructurado y con las mismas implicaciones, definición del metamodelo, de la transformación m2m y de la transformación m2t. La única diferencia, es que la definición del metamodelo se hará en base a los requisitos del cliente.

Mantenimiento de los formatos estructurados

Al plantear modelos más cercanos al *workflow* de la empresa, cualquier aparición de nuevos formatos estructurados o modificación de los ya existentes

se puede abordar desde las transformaciones modelo-modelo y desde los metamodelos que describen los formatos estructurados sin necesidad de modificar los metamodelos cercanos al *workflow* de la empresa.

5.2 Trabajos Futuros

Cuando se intente abordar la implantación en los proveedores de la hipotética empresa que adquiere la herramienta, se encontrarán con muchos tipos de proveedores. Algunos de ellos por su tamaño ya tendrán implantados Sistemas de envío de ficheros electrónicos como puede ser EDI, otras asumirán la implantación como un requisito y la implantarán con mayor o menor dificultad, pero otras no podrán de ninguna manera acceder al uso de ficheros electrónicos. Es aquí, donde se podría continuar trabajando con este proyecto hasta generar una herramienta completa que la empresa donde se haya implantado pueda llegar a ofrecer a sus proveedores. De esta manera, ambas herramientas la del proveedor y la de la propia empresa se comunicarían y permitirían el envío y recepción de ficheros electrónicos. Las ventajas de conseguir que los proveedores usen los ficheros electrónicos como si utilizaran una impresora virtual redundan en un mayor beneficio hacia la propia empresa y permite la expansión de la herramienta.

Es decir, este proyecto podría convertirse en una herramienta comercial que una vez implantada en la empresa permita expandirse hacia los proveedores y clientes de la misma dado que el interés de la empresa debe dirigirse en ese sentido una vez estudiados los beneficios que pueden conseguirse.

En este proyecto se ha planteado el metamodelo Invoic pero se podría seguir trabajando para definir otros metamodelos que cubran el proceso de trabajo de la empresa. Como se ha podido ver a lo largo de este proyecto, el metamodelo Edifact podría ser alimentado mediante las transformaciones modelo-modelo con todos otros modelos que conformen con los diferentes metamodelos que cubran el proceso de trabajo y sin realizar ninguna modificación permitiendo obtener los diferentes ficheros electrónicos.

También se podría definir más metamodelos que cubran los diferentes formatos estructurados que existen en el mercado y que permitan que la

herramienta sea lo más flexible posible para permitir abarcar el mayor mercado posible.

Anexo A. Estructura mensaje EDIFACT D93A/INVOIC

```
     UNH, Message header                          M        1
     BGM, Beginning of message                    M        1
     DTM, Date/time/period                        M       35
     PAI, Payment instructions                    C        1
     ALI, Additional information                  C        5
     IMD, Item description                        C        1
     FTX, Free text                               C       10

--- Segment Group 1 ------------------------- C       10 -------+
RFF, Reference                                 M        1        |
DTM, Date/time/period                          C        5        |
---- ------------------------------------------------------------+

--- Segment Group 2 ------------------------- C       20 -------+
NAD, Name and address                          M        1        |
LOC, Place/location identification             C       25        |
FII, Financial institution information         C        5        |
                                                                 |
--- Segment Group 3 ------------------------- C     9999 ------+|
RFF, Reference                                 M        1       ||
DTM, Date/time/period                          C        5       ||
---- ------------------------------------------------------------+|
                                                                 |
--- Segment Group 4 ------------------------- C        5 ------+|
DOC, Document/message details                  M        1       ||
DTM, Date/time/period                          C        5       ||
---- ------------------------------------------------------------+|
                                                                 |
--- Segment Group 5 ------------------------- C        5 ------+|
CTA, Contact information                       M        1       ||
COM, Communication contact                     C        5       ||
---- ------------------------------------------------------------++

--- Segment Group 6 ------------------------- C        5 -------+
TAX, Duty/tax/fee details                      M        1        |
MOA, Monetary amount                           C        1        |
```

```
LOC, Place/location identification            C        5        |
---- ---------------------------------------------------------+

--- Segment Group 7 ------------------------- C        5 -------+
CUX, Currencies                               M        1        |
DTM, Date/time/period                         C        5        |
---- ---------------------------------------------------------+

--- Segment Group 8 ------------------------- C       10 -------+
PAT, Payment terms basis                      M        1        |
DTM, Date/time/period                         C        5        |
PCD, Percentage details                       C        1        |
MOA, Monetary amount                          C        1        |
---- ---------------------------------------------------------+

--- Segment Group 9 ------------------------- C       10 -------+
TDT, Details of transport                     M        1        |
                                                                 |
--- Segment Group 10 ------------------------ C       10 ------+|
LOC, Place/location identification            M        1       ||
DTM, Date/time/period                         C        5       ||
---- ---------------------------------------------------------++

--- Segment Group 11 ------------------------ C        5 -------+
TOD, Terms of delivery                        M        1        |
LOC, Place/location identification            C        2        |
---- ---------------------------------------------------------+

--- Segment Group 12 ------------------------ C     1000 -------+
PAC, Package                                  M        1        |
MEA, Measurements                             C        5        |
                                                                 |
--- Segment Group 13 ------------------------ C        5 ------+|
PCI, Package identification                   M        1       ||
RFF, Reference                                C        1       ||
DTM, Date/time/period                         C        5       ||
GIN, Goods identity number                    C        5       ||
---- ---------------------------------------------------------++
```

```
--- Segment Group 14 ----------------------- C       15 -------+
ALC, Allowance or charge                      M        1        |
ALI, Additional information                   C        5        |
                                                                |
--- Segment Group 15 ----------------------- C        5 ------+|
RFF, Reference                                M        1       ||
DTM, Date/time/period                         C        5       ||
---- ------------------------------------------------------------+|
                                                                |
--- Segment Group 16 ----------------------- C        1 ------+|
QTY, Quantity                                 M        1       ||
RNG, Range details                            C        1       ||
---- ------------------------------------------------------------+|
                                                                |
--- Segment Group 17 ----------------------- C        1 ------+|
PCD, Percentage details                       M        1       ||
RNG, Range details                            C        1       ||
---- ------------------------------------------------------------+|
                                                                |
--- Segment Group 18 ----------------------- C        2 ------+|
MOA, Monetary amount                          M        1       ||
RNG, Range details                            C        1       ||
---- ------------------------------------------------------------+|
                                                                |
--- Segment Group 19 ----------------------- C        1 ------+|
RTE, Rate details                             M        1       ||
RNG, Range details                            C        1       ||
---- ------------------------------------------------------------+|
                                                                |
--- Segment Group 20 ----------------------- C        5 ------+|
TAX, Duty/tax/fee details                     M        1       ||
MOA, Monetary amount                          C        1       ||
---- ------------------------------------------------------------++

--- Segment Group 21 ----------------------- C      100 -------+
RCS, Requirements and conditions              M        1        |
RFF, Reference                                C        5        |
```

```
DTM, Date/time/period                          C        5        |
FTX, Free text                                 C        5        |
---- ----------------------------------------------------------+

--- Segment Group 22 ------------------------ C   200000 -------+
LIN, Line item                                 M        1        |
PIA, Additional product id                     C       25        |
IMD, Item description                          C       10        |
MEA, Measurements                              C        5        |
QTY, Quantity                                  C        5        |
PCD, Percentage details                        C        1        |
ALI, Additional information                    C        5        |
DTM, Date/time/period                          C       35        |
GIN, Goods identity number                     C     1000        |
GIR, Related identification numbers            C     1000        |
QVA, Quantity variances                        C        1        |
FTX, Free text                                 C        5        |
                                                                 |
--- Segment Group 23 ------------------------ C        5 ------+|
MOA, Monetary amount                           M        1       ||
CUX, Currencies                                C        1       ||
---- ----------------------------------------------------------+|
                                                                 |
--- Segment Group 24 ------------------------ C       10 ------+|
PAT, Payment terms basis                       M        1       ||
DTM, Date/time/period                          C        5       ||
PCD, Percentage details                        C        1       ||
MOA, Monetary amount                           C        1       ||
---- ----------------------------------------------------------+|
                                                                 |
--- Segment Group 25 ------------------------ C       25 ------+|
PRI, Price details                             M        1       ||
API, Additional price information              C        1       ||
RNG, Range details                             C        1       ||
DTM, Date/time/period                          C        5       ||
---- ----------------------------------------------------------+|
                                                                 |
--- Segment Group 26 ------------------------ C       10 ------+|
```

```
RFF, Reference                                M        1       ||
DTM, Date/time/period                         C        5       ||
---- ---------------------------------------------------------+|
                                                               |
--- Segment Group 27 ----------------------- C       10 ------+|
PAC, Package                                  M        1       ||
MEA, Measurements                             C       10       ||
                                                               ||
--- Segment Group 28 ----------------------- C       10 -----+||
PCI, Package identification                   M        1      |||
RFF, Reference                                C        1      |||
DTM, Date/time/period                         C        5      |||
GIN, Goods identity number                    C       10      |||
---- ---------------------------------------------------------++|
                                                               |
--- Segment Group 29 ----------------------- C      100 ------+|
LOC, Place/location identification            M        1       ||
QTY, Quantity                                 C        1       ||
DTM, Date/time/period                         C        5       ||
---- ---------------------------------------------------------+|
                                                               |
--- Segment Group 30 ----------------------- C        5 ------+|
TAX, Duty/tax/fee details                     M        1       ||
MOA, Monetary amount                          C        1       ||
LOC, Place/location identification            C        5       ||
---- ---------------------------------------------------------+|
                                                               |
--- Segment Group 31 ----------------------- C       20 ------+|
NAD, Name and address                         M        1       ||
LOC, Place/location identification            C        5       ||
                                                               ||
--- Segment Group 32 ----------------------- C        5 -----+||
RFF, Reference                                M        1      |||
DTM, Date/time/period                         C        5      |||
---- ---------------------------------------------------------+||
                                                               ||
--- Segment Group 33 ----------------------- C        5 -----+||
DOC, Document/message details                 M        1      |||
```

```
DTM, Date/time/period                          C        5      |||
---- ------------------------------------------------------+||
                                                            ||
--- Segment Group 34 ------------------------ C        5 -----+||
CTA, Contact information                       M        1      |||
COM, Communication contact                     C        5      |||
---- ------------------------------------------------------++|
                                                             |
--- Segment Group 35 ------------------------ C       15 ------+|
ALC, Allowance or charge                       M        1       ||
ALI, Additional information                    C        5       ||
                                                            ||
--- Segment Group 36 ------------------------ C        1 -----+||
QTY, Quantity                                  M        1      |||
RNG, Range details                             C        1      |||
---- ------------------------------------------------------+||
                                                            ||
--- Segment Group 37 ------------------------ C        1 -----+||
PCD, Percentage details                        M        1      |||
RNG, Range details                             C        1      |||
---- ------------------------------------------------------+||
                                                            ||
--- Segment Group 38 ------------------------ C        2 -----+||
MOA, Monetary amount                           M        1      |||
RNG, Range details                             C        1      |||
---- ------------------------------------------------------+||
                                                            ||
--- Segment Group 39 ------------------------ C        1 -----+||
RTE, Rate details                              M        1      |||
RNG, Range details                             C        1      |||
---- ------------------------------------------------------+||
                                                            ||
--- Segment Group 40 ------------------------ C        5 -----+||
TAX, Duty/tax/fee details                      M        1      |||
MOA, Monetary amount                           C        1      |||
---- ------------------------------------------------------++|
                                                             |
--- Segment Group 41 ------------------------ C       10 ------+|
```

```
TDT, Details of transport                    M        1       ||
                                                              ||
--- Segment Group 42 ----------------------- C       10 -----+||
LOC, Place/location identification           M        1      |||
DTM, Date/time/period                        C        5      |||
---- ------------------------------------------------------++|
                                                               |
--- Segment Group 43 ----------------------- C        5 ------+|
TOD, Terms of delivery                       M        1       ||
LOC, Place/location identification           C        2       ||
---- ------------------------------------------------------+|
                                                               |
--- Segment Group 44 ----------------------- C      100 ------+|
RCS, Requirements and conditions             M        1       ||
RFF, Reference                               C        5       ||
DTM, Date/time/period                        C        5       ||
FTX, Free text                               C        5       ||
---- ------------------------------------------------------++
    UNS, Section control                          M        1
    CNT, Control total                            C       10

--- Segment Group 45 ----------------------- M      100 -------+
MOA, Monetary amount                         M        1        |
                                                               |
--- Segment Group 46 ----------------------- C        1 ------+|
RFF, Reference                               M        1       ||
DTM, Date/time/period                        C        5       ||
---- ------------------------------------------------------++

--- Segment Group 47 ----------------------- C       10 -------+
TAX, Duty/tax/fee details                    M        1        |
MOA, Monetary amount                         C        2        |
---- ------------------------------------------------------+

--- Segment Group 48 ----------------------- C       15 -------+
ALC, Allowance or charge                     M        1        |
ALI, Additional information                  C        1        |
MOA, Monetary amount                         C        2        |
```

```
      ---- ------------------------------------------------------------+
           UNT, Message trailer                          M        1
```

Segmento ISO9735/UNH

```
0062       MESSAGE REFERENCE NUMBER                 M   an1..14
S009       MESSAGE IDENTIFIER                       M
   0065    Message type                             M   an1..6
   0052    Message version number                   M   an1..3
   0054    Message release number                   M   an1..3
   0051    Controlling agency, coded                M   an1..3
   0057    Association assigned code                C   an1..6
   0110    Code list directory version number       C   an1..6
   0113    Message type sub-function identification C   an1..6
0068       COMMON ACCESS REFERENCE                  C   an1..35
S010       STATUS OF THE TRANSFER                   C
   0070    Sequence of transfers                    M   n1..2
   0073    First and last transfer                  C   a1..1
S016       MESSAGE SUBSET IDENTIFICATION            C
   0115    Message subset identification            M   an1..14
   0116    Message subset version number            C   an1..3
   0118    Message subset release number            C   an1..3
   0051    Controlling agency, coded                C   an1..3
S017       MESSAGE IMPLEMENTATION GUIDELINE IDENTIFICATION C
   0121    Message implementation guideline identification M   an1..14
   0122    Message implementation guideline version number C   an1..3
   0124    Message implementation guideline release number C   an1..3
   0051    Controlling agency, coded                C   an1..3
S018       SCENARIO IDENTIFICATION                  C
   0127    Scenario identification                  M   an1..14
   0128    Scenario version number                  C   an1..3
   0130    Scenario release number                  C   an1..3
   0051    Controlling agency, coded                C   an1..3
```

Anexo B. Modelo que conforma con el metamodelo Edifact

```
Estandar D93A{
      Mensaje Invoic{
            Segmento UNB{
                  CalificadorC S001{
                        CalificadorS 0001 'UNOA'
                        CalificadorS 0002 '2'
                  }
                  CalificadorC S002{
                        CalificadorS 0004 'EMISOR'
                  }
                  CalificadorC S003{
                        CalificadorS 0004 'DESTINATARIO'
                  }
                  CalificadorC S004{
                        CalificadorS 0017 '990802'
                        CalificadorS 0019 '1557'
                  }
                  CalificadorS 0020 '9908021557'
            }
            Segmento UNH{
                  CalificadorS 0062 'INVOIC001'
                  CalificadorC S009{
                        CalificadorS 0065 'INVOIC'
                        CalificadorS 0052 'D'
                        CalificadorS 0054 '93A'
                        CalificadorS 0051 'UN'
                  }
            }
            Segmento BGM{
                  CalificadorC C002{
                        CalificadorS 1001 '380'
                  }
                  CalificadorS 1004 '12/2012'
            }
            Segmento DTM{
                  CalificadorC C507{
                        CalificadorS 2005 '137'
                        CalificadorS 2380 '120113'
                        CalificadorS 2379 '101'
                  }
            }
            Segmento NAD{
                  CalificadorS 3035 'RE'
                  CalificadorC C082{
                  }
                  CalificadorC C058{
                        CalificadorS 3124 'Agencia Transdemur-2003 SL'
                  }
                  CalificadorC C080{
                  }
                  CalificadorC C059{
                        CalificadorS 3042 'C/Escritor Ruiz Aguilera,
18'
                  }
                  CalificadorS 3164 'Murcia'
                  CalificadorS 3229
                  CalificadorS 3251 '30110'
```

```
            }
            Segmento RFF{
                  CalificadorC C506{
                        CalificadorS 1153 'VA'
                        CalificadorS 1154 'ESB73261604'
                  }
            }
            Segmento NAD{
                  CalificadorS 3035 'EM'
                  CalificadorC C082{
                  }
                  CalificadorC C058{
                        CalificadorS 3124 'Transporte SL'
                  }
                  CalificadorC C080{
                  }
                  CalificadorC C059{
                        CalificadorS 3042 'C/Del Mar Nº5'
                  }
                  CalificadorS 3164 'Murcia'
                  CalificadorS 3229
                  CalificadorS 3251 '30110'
            }
            Segmento RFF{
                  CalificadorC C506{
                        CalificadorS 1153 'VA'
                        CalificadorS 1154 'ESB12364525'
                  }
            }
            Segmento LIN{
                  CalificadorS 1082 '1'
                  CalificadorS 1229 '482'
            }
            Segmento IMD{
                  CalificadorS 7077 'F'
                  CalificadorS 7081 'M'
                  CalificadorC C273{
                        CalificadorS 7009
                        CalificadorS 1131
                        CalificadorS 3055
                        CalificadorS 7008 'Viaje Descartes/Francia a
Murcia/España'
                  }
            }
            Segmento QTY{
                  CalificadorC C186{
                        CalificadorS 6063 '47'
                        CalificadorS 6060 '1'
                  }
            }
            Segmento MOA{
                  CalificadorC C516{
                        CalificadorS 5025 '66'
                        CalificadorS 5004 '1160'
                  }
            }
            Segmento UNS{
                  CalificadorS 0081 'S'
            }
            Segmento MOA{
                  CalificadorC C516{
```

```
                        CalificadorS 5025 '125'
                        CalificadorS 5004 '1160'
                  }
            }
            Segmento MOA{
                  CalificadorC C516{
                        CalificadorS 5025 '176'
                        CalificadorS 5004 '280,80'
                  }
            }
            Segmento MOA{
                  CalificadorC C516{
                        CalificadorS 5025 '139'
                        CalificadorS 5004 '1440,80'
                  }
            }

            Segmento UNT{
                  CalificadorS 0074 '16'
                  CalificadorS 0062 'INVOIC001'
            }
            Segmento UNZ{
                  CalificadorS 0036 '1'
                  CalificadorS 0020 '9908021557'
            }
      }
}
```

Anexo C. Configuración Schemol

En este anexo se refleja la configuración de Schemol mediante el fichero que se puede ver en la figura 43. Este fichero se encuentra localizado en este caso en src → schemol.example.skeleton → con el nombre de fichero ScheMolSimpleExample.java.

```
package schemol.example.skeleton;

import schemol.dbmanager.DBManager.DBManagerTypes;
import schemol.interpreter.ScheMoLConfiguration;
import schemol.interpreter.ScheMoLLauncher;

public class ScheMoLSimpleExample {
      public static void main(String[] args) {

            ScheMoLConfiguration configuration = new
ScheMoLConfiguration();

            configuration.setCsPath("./transformation/simple.db2m"); ①
            configuration.setMetamodelPath("./metamodel/Invoic.ecore");②
            configuration.setMetamodelPrefix("Invoic");③
            configuration.setDbType(DBManagerTypes.MYSQL);④
            configuration.setHost("localhost");
            configuration.setDatabase("test");
            configuration.setUser("root");
            configuration.setPassword("1234");
            configuration.setResultPath("./factura.xmi");⑤

            ScheMoLLauncher launcher = new
ScheMoLLauncher(configuration);
            launcher.launch();
      }
}
```

Figura 43. Configuración Schemol (Fuente: propia)

1. Se indica el fichero donde se encuentra definida la transformación
2. Ruta del metamodelo Invoic (fichero .ecore) con el cual lo modelos generados deben conformar.
3. Prefijo del metamodelo.
4. Configuración de la Base de Datos utilizada incluida localización, nombre, usuario y password.
5. Nombre y localización del modelo a generar.

Las transformaciones definidas mediante Schemol se almacenan en la carpeta “transformation” de la estructura del proyecto Schemol. Para ejecutar la transformación se abre el fichero de configuración “ScheMolSimpleExample.java” botón derecho “Run As” → “Java Application” y la herramienta, en el caso de que todo este correcto, genera el modelo “factura.xmi” según se le indicó en el fichero de configuración.

Anexo D. Configuración de RubyTL

Antes de poder ejecutar transformaciones con RubyTL, se debe instalar el intérprete de Ruby. Se recomienda instalar una versión estable como la versión de Ruby 1.8.6.

Una vez instalado el intérprete de Ruby, se debe configurar el entorno con el path de Ruby. Para ello, se accede a la opción del menú Windows → *Preferences* y seleccionamos Ruby → *Installed Interpreters* tal y como se muestra en la figura 44.

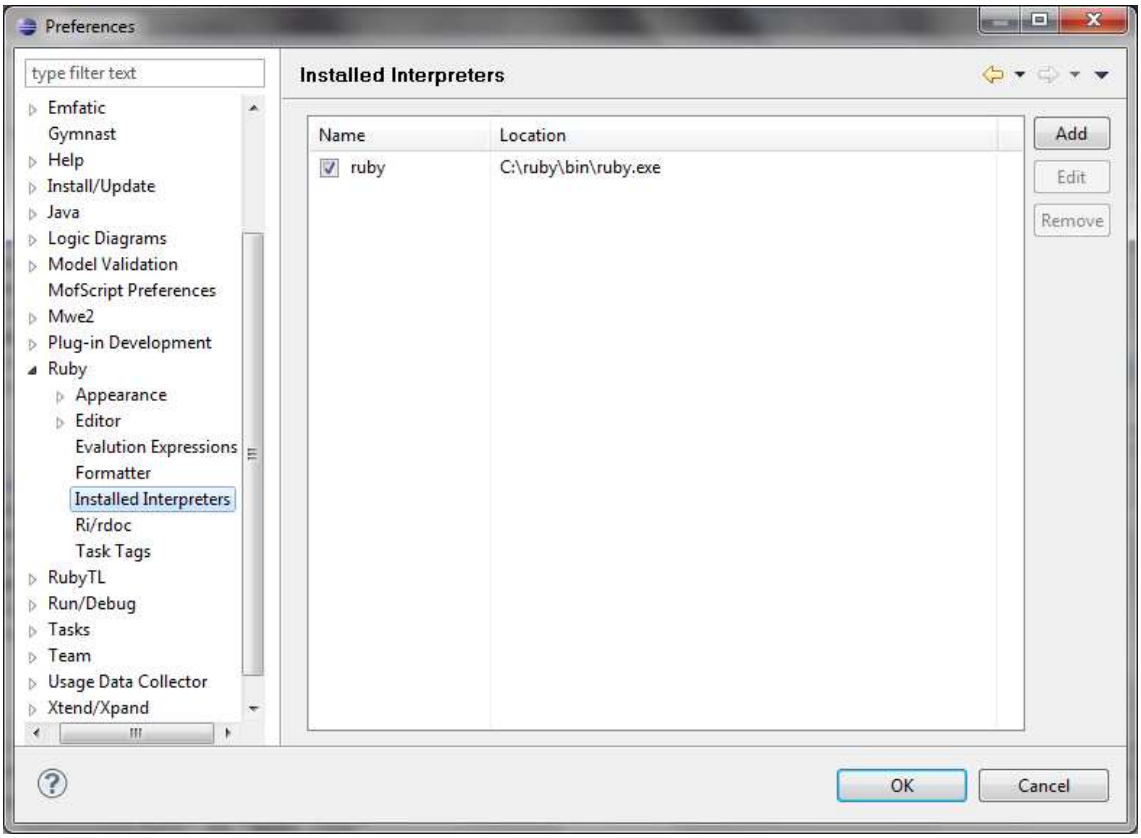


Figura 44. Configuración interprete Ruby. (Fuente: Eclipse)

Para ejecutar una transformación modelo-modelo hay que especificar cuáles son los modelos y metamodelos de entrada concretos que usa la definición de la transformación.

En el caso de RubyTL, se definen ficheros Rakefile (un sistema parecido a Make, pero embebido en Ruby) en los que se especifican tareas de

construcción. Así, hay tareas específicas para transformaciones modelo-modelo, modelo-código, etc.

A continuación se muestra el fichero Rakefile utilizado en este proyecto para ejecutar la transformación modelo-modelo.

```
model_to_model :invoic2edifact do |t|
  t.sources :package => 'Invoic',
            :metamodel => 'metamodels/Invoic.ecore',
            :model => 'models/factura.xmi'

  t.targets :package => 'Edifact',
            :metamodel => 'metamodels/Edifact.ecore',
            :model => 'models/edifact2.xmi'

  t.transformation 'transformations/m2m/i2e.rb'
end
```

Una tarea model_to_model está compuesta de los siguientes elementos:

- El nombre de la tarea. Por convención se prefija con ":", aunque puede especificarse como una cadena. En este caso ":invoic2edifact".
- Un conjunto de elementos sources, que representa modelos y metamodelos de entrada. Se especifica el espacio de nombres que espera la definición de la transformación (ej. Invoic), el fichero .ecore que contiene el metamodelo, y un modelo que conforma al metamodelo.
- Un conjunto de elementos targets, que representan modelos y metamodelos de salida. Se especifica el espacio de nombres que espera la definición de la transformación (ej. Edifact), el fichero .ecore que contiene el metamodelo, y un modelo que conforma al metamodelo. Si el modelo no existe, al finalizar la transformación se creará nuevo, si ya existe será reemplazado por el nuevo.
- La ruta de la transformación que se ejecutará.

Anexo E. Transformación de modelos Invoic a modelos EDIFACT.

```
rule 'Factura2SegUNB' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|

    se.nombre = 'UNB'
    co =  Edifact::Compuesto.new(:id => 'S001')
    co.calificadores << Edifact::Simple.new(:id =>'0001', :cadena =>
'UNOA')
    co.calificadores << Edifact::Simple.new(:id =>'0002', :cadena =>
'2')
    se.calificadores << co
    co =  Edifact::Compuesto.new(:id => 'S002')
    co.calificadores << Edifact::Simple.new(:id =>'0004', :cadena =>
fac.emisorf.buzonedi)
    se.calificadores << co
    co =  Edifact::Compuesto.new(:id => 'S003')
    co.calificadores << Edifact::Simple.new(:id =>'0004', :cadena =>
fac.clientef.buzonedi)
    se.calificadores << co
    co =  Edifact::Compuesto.new(:id => 'S004')
    co.calificadores << Edifact::Simple.new(:id =>'0017', :cadena =>
Time.now.strftime("%y%m%d"))
    co.calificadores << Edifact::Simple.new(:id =>'0019', :cadena =>
Time.now.strftime("%I%M"))
    se.calificadores << co
    se.calificadores << Edifact::Simple.new(:id =>'0020', :cadena =>
fac.enviof.numero)

   end
end

rule 'Factura2SegUNH' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre = 'UNH'
    se.calificadores << Edifact::Simple.new(:id =>'0062', :cadena =>
'INVOIC' + fac.enviof.numero)
    co =  Edifact::Compuesto.new(:id => 'S009')
    co.calificadores << Edifact::Simple.new(:id =>'0065', :cadena =>
'INVOIC')
    co.calificadores << Edifact::Simple.new(:id =>'0052', :cadena =>
'D')
    co.calificadores << Edifact::Simple.new(:id =>'0054', :cadena =>
'93A')
    co.calificadores << Edifact::Simple.new(:id =>'0051', :cadena =>
'UN')
    se.calificadores << co
  end
end

rule 'Factura2SegBGM' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
```

```
    se.nombre ='BGM'
    co =  Edifact::Compuesto.new(:id => 'C002')
    co.calificadores << Edifact::Simple.new(:id =>'1001', :cadena =>
'380')
    se.calificadores << co
    se.calificadores << Edifact::Simple.new(:id =>'1004', :cadena =>
fac.enviof.numero)
  end
end

rule 'Factura2SegDTM' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='DTM'
    co =  Edifact::Compuesto.new(:id => 'C507')
    co.calificadores << Edifact::Simple.new(:id =>'2005', :cadena =>
'137')
    co.calificadores << Edifact::Simple.new(:id =>'2380', :cadena =>
fac.Fecha)
    co.calificadores << Edifact::Simple.new(:id =>'2379', :cadena =>
'101')
    se.calificadores << co
  end
end

rule 'Factura2SegNAD' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='NAD'
    se.calificadores << Edifact::Simple.new(:id =>'3035', :cadena =>
'EM')
    co =  Edifact::Compuesto.new(:id => 'C082')
    se.calificadores << co
    co =  Edifact::Compuesto.new(:id => 'C058')
    co.calificadores << Edifact::Simple.new(:id =>'3124', :cadena =>
fac.emisorf.nombre)
    se.calificadores << co
    co =  Edifact::Compuesto.new(:id => 'C080')
    se.calificadores << co
    co =  Edifact::Compuesto.new(:id => 'C059')
    co.calificadores << Edifact::Simple.new(:id =>'3042', :cadena =>
fac.emisorf.direccion)
    se.calificadores << co
    se.calificadores << Edifact::Simple.new(:id =>'3164', :cadena =>
fac.emisorf.provincia)
    se.calificadores << Edifact::Simple.new(:id =>'3229', :cadena =>
'')
    se.calificadores << Edifact::Simple.new(:id =>'3251', :cadena =>
fac.emisorf.codpostal)
  end
end

rule 'Factura2SegRFF' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='RFF'
    co =  Edifact::Compuesto.new(:id => 'C506')
```

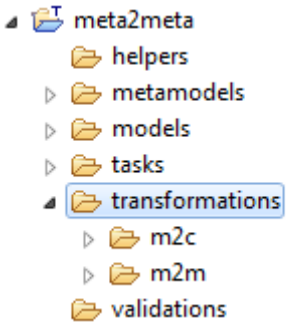
```
    co.calificadores << Edifact::Simple.new(:id =>'1153', :cadena =>
'VA')
    co.calificadores << Edifact::Simple.new(:id =>'1154', :cadena =>
fac.emisorf.cif)
    se.calificadores << co
  end
end

rule 'Factura2SegNAD' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='NAD'
    se.calificadores << Edifact::Simple.new(:id =>'3035', :cadena =>
'RE')
    co =  Edifact::Compuesto.new(:id => 'C082')
    se.calificadores << co
    co =  Edifact::Compuesto.new(:id => 'C058')
    co.calificadores << Edifact::Simple.new(:id =>'3124', :cadena =>
fac.clientef.nombre)
    se.calificadores << co
    co =  Edifact::Compuesto.new(:id => 'C080')
    se.calificadores << co
    co =  Edifact::Compuesto.new(:id => 'C059')
    co.calificadores << Edifact::Simple.new(:id =>'3042', :cadena =>
fac.clientef.direccion)
    se.calificadores << co
    se.calificadores << Edifact::Simple.new(:id =>'3164', :cadena =>
fac.clientef.provincia)
    se.calificadores << Edifact::Simple.new(:id =>'3229', :cadena =>
'')
    se.calificadores << Edifact::Simple.new(:id =>'3251', :cadena =>
fac.clientef.codpostal)
  end
end

rule 'Factura2SegRFF2' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='RFF'
    co =  Edifact::Compuesto.new(:id => 'C506')
    co.calificadores << Edifact::Simple.new(:id =>'1153', :cadena =>
'VA')
    co.calificadores << Edifact::Simple.new(:id =>'1154', :cadena =>
fac.clientef.cif)
    se.calificadores << co
  end
end

rule 'Factura2LineaFactura' do
  from Invoic::LineaFactura
  to Edifact::Segmento, Edifact::Segmento, Edifact::Segmento,
Edifact::Segmento
  mapping do |li, se1, se2, se3, se4|
      se1.nombre = 'LIN'
      se1.calificadores << Edifact::Simple.new(:id =>'1082', :cadena
=> li.codigo)
      se1.calificadores << Edifact::Simple.new(:id =>'1229', :cadena
=> '482')
```

```
      se2.nombre = 'IMD'
      se2.calificadores << Edifact::Simple.new(:id =>'7077', :cadena
=> 'F')
      se2.calificadores << Edifact::Simple.new(:id =>'7081', :cadena
=> 'M')
      co =  Edifact::Compuesto.new(:id => 'C273')
      co.calificadores << Edifact::Simple.new(:id =>'7009', :cadena =>
'')
      co.calificadores << Edifact::Simple.new(:id =>'1131', :cadena =>
'')
      co.calificadores << Edifact::Simple.new(:id =>'3055', :cadena =>
'')
      co.calificadores << Edifact::Simple.new(:id =>'7008', :cadena =>
li.concepto)
      se2.calificadores << co


      se3.nombre = 'QTY'
      co =  Edifact::Compuesto.new(:id => 'C186')
      co.calificadores << Edifact::Simple.new(:id =>'6063', :cadena =>
'47')
      co.calificadores << Edifact::Simple.new(:id =>'6060', :cadena =>
li.unidades)
      se3.calificadores << co


      se4.nombre = 'MOA'
      co =  Edifact::Compuesto.new(:id => 'C516')
      co.calificadores << Edifact::Simple.new(:id =>'5025', :cadena =>
'6')
      co.calificadores << Edifact::Simple.new(:id =>'5004', :cadena =>
li.totallinea)
      se4.calificadores << co
  end
end

rule 'Factura2SegUNS' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='UNS'
    se.calificadores << Edifact::Simple.new(:id =>'0081', :cadena =>
'S')
  end
end

rule 'Factura2SegMOA1' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre = 'MOA'
    co =  Edifact::Compuesto.new(:id => 'C516')
    co.calificadores << Edifact::Simple.new(:id =>'5025', :cadena =>
'125')
    co.calificadores << Edifact::Simple.new(:id =>'5004', :cadena =>
fac.base)
    se.calificadores << co
  end
end
```

```
rule 'Factura2SegMOA2' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='MOA'
    co =  Edifact::Compuesto.new(:id => 'C516')
    co.calificadores << Edifact::Simple.new(:id =>'5025', :cadena =>
'176')
    co.calificadores << Edifact::Simple.new(:id =>'5004', :cadena =>
fac.iva)
    se.calificadores << co
  end
end

rule 'Factura2SegMOA3' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='MOA'
    co =  Edifact::Compuesto.new(:id => 'C516')
    co.calificadores << Edifact::Simple.new(:id =>'5025', :cadena =>
'139')
    co.calificadores << Edifact::Simple.new(:id =>'5004', :cadena =>
fac.total)
    se.calificadores << co
  end
end

rule 'Factura2SegUNT' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='UNT'
    se.calificadores << Edifact::Simple.new(:id =>'0074', :cadena =>
'16')
    se.calificadores << Edifact::Simple.new(:id =>'0062', :cadena =>
'INVOIC' + fac.enviof.numero)
  end
end

rule 'Factura2SegUNZ' do
  from Invoic::Factura
  to Edifact::Segmento
  mapping do |fac, se|
    se.nombre ='UNZ'
    se.calificadores << Edifact::Simple.new(:id =>'0036', :cadena =>
'1')
    se.calificadores << Edifact::Simple.new(:id =>'0020', :cadena =>
fac.enviof.numero)
  end
end
```

Anexo F. Configuración MOFScript

Se ha partido del proyecto creado para aplicar las transformaciones modelo a modelo. Dentro de la carpeta *transformations* se crean dos carpetas m2c y m2t para separar los dos tipos de transformaciones, modelo-modelo y modelo-texto, quedando una estructura de proyecto como la que se puede ver en la figura 45.



(Fuente: Eclipse)

Figura 45. Estructura del proyecto para definir transformaciones.

Antes de comenzar a definir y ejecutar transformaciones MOFScript, se tiene que configurar el entorno. Es muy importante comprobar el campo que indica la localización de los metamodelos que se utilizarán en la definición de las transformaciones. Para poder comprobar dicho campo se hace desde la opción del menú Windows → Preferences y aparece la pantalla de la figura 46. El primer campo que se puede contemplar es “Metamodels path”. Es conveniente comprobar el resto de campos, para ello se hace click en cada campo y comprobamos si existe mensaje de error.

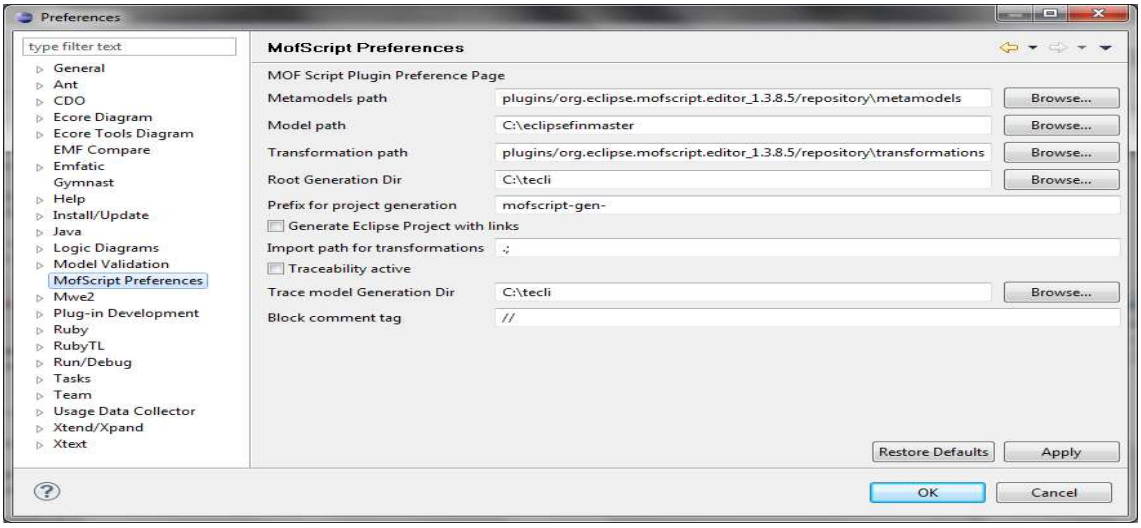


Figura 46. Configuración MOFScript. (Fuente: Eclipse)

Para evitar tener que estar cambiando la ruta de los metamodelos con los que se trabajan, se puede usar la opción de registro de metamodelos ecore en la plataforma Eclipse. Con este método, se puede registrar un metamodelo, y las transformaciones que lo utilizan funcionarán correctamente sin la necesidad de configurar las rutas de los metamodelos.

Una vez ejecutado el fichero rakefile, los modelos generados se almacenan en la carpeta definida sobre cada transformación individualmente. Para cambiar la ruta de almacenamiento, se hace click en el botón derecho sobre el fichero m2t y se pulsa sobre *Properties* apareciendo la pantalla de la figura 47. En dicha pantalla se puede apreciar la entrada "*Select Root Output Directory*" que se puede modificar para indicar una nueva ruta.

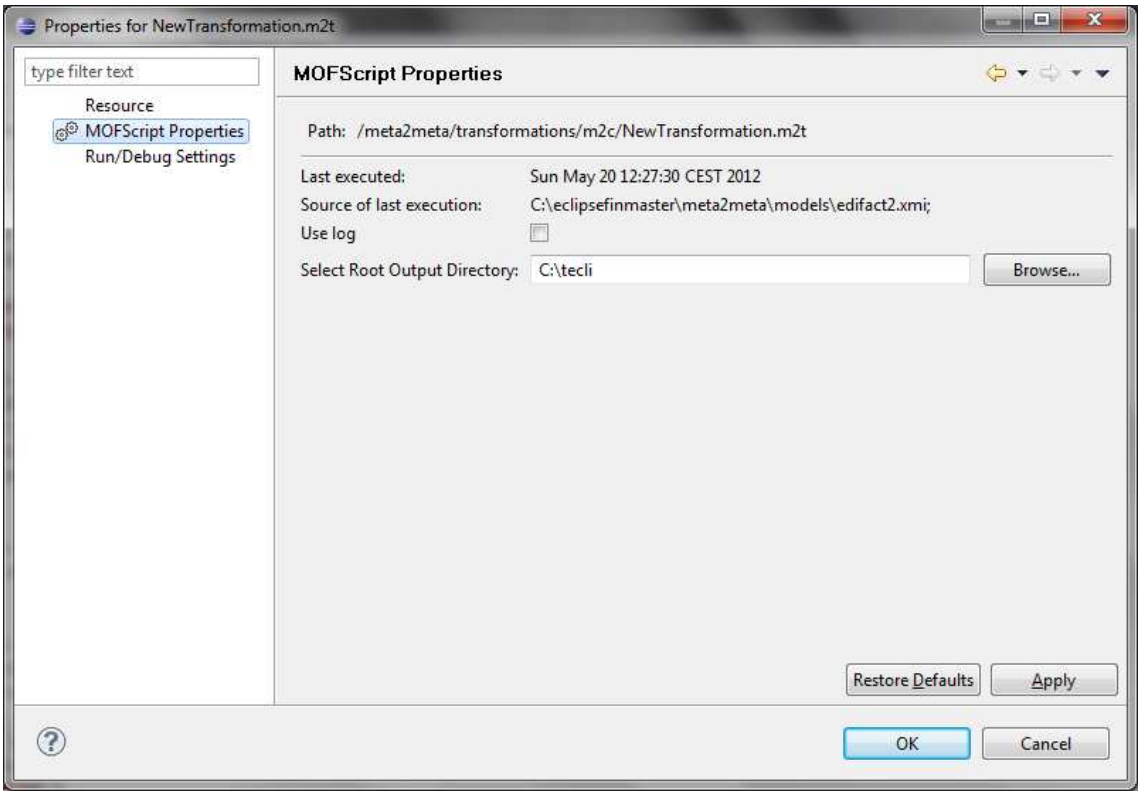


Figura 47. Configuración de la ruta de salida. (Fuente: Eclipse)

Una vez el entorno está configurado, se puede pasar a definir las transformaciones m2t.

Lista de acrónimos

AECOC: Asociación Española de Codificación Comercial

CCI: Centro de Cooperación Interbancaria

CDO: Connected Data Objects

CODICE: Componentes y documentos interoperables para la contratación Electrónica

D2M: Transformaciones datos-a-modelo

DSL: Domain Specific Language

EBXML: Electronic Business XML

EDI: Intercambio electrónico de datos

EDIFACT: Electronic Data Interchange For Administration, Commerce and Transport

EMF: Eclipse Modeling Framework

JDT: Java Development Toolkit

OASIS: Organization for the Advancement of Structured Information Standards)

OMG: Object Management Group

MDA: Model-Driven Architecture

MDD: Desarrollo de Software Dirigido por Modelos

MDE: Ingeniería de Modelos

M2M: Transformaciones modelo a modelo

M2T: Transformaciones modelo a texto

PDE: Plug-in Development Environment

SDK: Eclipse Software Development Kit

SGML: Estándar Generalized Markup Language

SQL: Structured query language

TSA: Prestadores de Servicios de Sellado de Tiempo

UBL: Universal Business Language

UN/EDIFACT: United Nations/Electronic Data Interchange For Administration, Commerce and Transport

UNECE: United Nations Economic Commission for Europe

UML: Unified Modeling Language

USH: Security header

UST: Security tráiler

VA: Prestadores de Servicios de Validación

VAN: Value Added Network

XMI: XML Metadata Interchange

Referencias

1. Selic, Bran; Manifestaciones sobre MDA, en [38]
2. Bézivin, J., Barbero, M., Jouault, F.: "On the Applicability Scope of Model Driven Engineering". MOMPES 2007: 3-7.
3. MDA, http://www.omg.org/mda/
4. Kelly, S., Tolvanen, J.P.: *Domain Specific Languages*, John Wiley, 2008.
5. Blair, G., Bencomo, N., France, R.B., Models@run.time, IEEE Computer, vol. 42, num. 10, pp. 22-27, October 2009.
6. Clark, T., Sammut, P., Willans. J.; *Applied Metamodelling A Foundation for Language Driven Development*, Second edition Xactium, 2004.
7. Kleppe, A., Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, Addison-Wesley, 2008.
8. Warmer, J., Kleppe, A., "*The Object Constraint Language*. Second Edition", Addison-Wesley. 2003.
9. Gronback, R.C., Merks, E.; "Arquitectura Dirigida por Modelos en Eclipse EMF" en [38].
10. MOF, *MetaObject Facility*, http://www.omg.org/spec/MOF/2.0/
11. EMFText, http://www.emftext.org/index.php/EMFText
12. xText (2011). xText 2.0. http://www.xtext.org
13. DSL Tools, http://www.microsoft.com/download/en/details.aspx?id=2379
14. MetaEdit+, http://www.metacase.com/
15. Fowler, M. with Rebecca Parsons, *Domain-Specific Languages*. Addison-Wesley, 2010.
16. ATL (Atlas Transformation Language), http://www.eclipse.org/atl/
17. QVT, Query/View/Transformation, http://www.omg.org/spec/QVT/1.0/PDF/
18. Acceleo, http://www.eclipse.org/acceleo/
19. Mof2Text, http://www.omg.org/cgi-bin/doc?ad/2004-4-7
20. MofScript, http://www.eclipse.org/gmt/mofscript/
21. XPand, openArchitectureWare, http://www.openarchitectureware.org/
22. Atkinson, C., Kühne, T.: "Model-Driven Development: A Metamodeling Foundation". IEEE Software 20(5): 36-41 (2003)
23. Gonzalez-Perez C., Henderson-Sellers, B., *Metamodelling for Software Engineering*, John Wiley & Sons, 2008.

24. Kleppe, A.; Warmer, J., Bast, W.; *MDA Explained*, Addison-Wesley, 2003.
25. Canovas, J.L., García-Molina,J.; "A Domain Specific Language for Extracting Models in Software Modernization", 5th European Conference on MDA, LNCS, vol. 5562, 2009.
26. Díaz, O., Cánovas, J.L., Puente, G., García-Molina, J.: *Harvesting Models from Web 2.0 Databases*, Software and Systems Modeling, enero 2011. DOI: 10.1007/s10270-011-0194-z
27. Cánovas, J.L., Jouault, F., Cabot, J., García-Molina, J.: "API2MoL: Automating the building of bridges between APIs and Model-Driven Engineering", Information and Software Technology (October 2011), doi:10.1016/j.infsof.2011.09.006
28. MetaDepth, *A framework for deep meta-modeling*, http://astreo.ii.uam.es/~jlara/metaDepth/
29. Gonzalez-Perez C., Henderson-Sellers, B., *Metamodelling for Software Engineering*, John Wiley & Sons, 2008.
30. ADM, http://adm.omg.org
31. Cánovas, J.L., García-Molina, J.; "An Architecture-Driven Modernization Tool for Calculating Metrics", IEEE Software, julio 2009.
32. XMI, *XML Metadata Interchange*, http://www.omg.org/spec/XMI/2.1.1/
33. Emfatic, http://wiki.eclipse.org/Emfatic
34. OCLinEcore, http://wiki.eclipse.org/MDT/OCLinEcore
35. Espinazo, J., Sánchez Cuadrado, J., García Molina, J.: Morsa: A Scalable Approach for Persisting and Accessing Large Models. MoDELS 2011: 77-92
36. GMF (Graphical Modeling Framework), http://www.eclipse.org/modeling/gmp/
37. Modisco, http://www.eclipse.org/MoDisco/
38. Bezivin, J., Vallecillo, A., García-Molina, J., Rossi, G.; monografía sobre "Desarrollo de Software Dirigido por Modelos", Novática, num. 192, marzo-abril, 2008.
39. Agile Generative Environment (AGE), http://gts.inf.um.es/trac/age
40. Simón González González, Construcción de un Generador de Escenas usando técnicas DSDM.
41. RubyTL, http://rubytl.rubyforge.org/

42. Jesús García Molina, Fundamentos del Desarrollo de Software Dirigido por Modelos.
43. Erich Gamma, Lee Nackman, John Wiegand, “EMF Eclipse Modeling Framework”.
44. Facturae, http://www.facturae.es/es-ES/Paginas/principal.aspx
45. UN/Edifact, http://www.unece.org/cefact/edifact/welcome.html
46. UBLXML, http://www.ubl.org.es/
47. Jesús J. García Molina, Capítulos 1 y 3 en "Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas", editado por Jesús J. García Molina, Félix O. García, Vicente Pelechano, Antonio Vallecillo, Juan Manuel Vara, Cristina Vicente-Chicote. Editorial RA-MA, octubre 2012 (el autor nos ha permitido acceder a sus dos capítulos).
48. Schemol, http://modelum.es/trac/schemol/
49. Red.es, ASIMELEC: Manuales Plan Avanza La factura electrónica.
50. AECOC, http://www.aecoc.es/
51. Gra2MoL, http://modelum.es/trac/gra2mol/
52. Openarchitectureware, http://www.openarchitectureware.org/staticpages/index.php/documentation
53. Javier Luis Cánovas Izquierdo, Lenguajes específicos del dominio para la extracción de modelos desde los espacios tecnológicos del grammarware, dataware y apiware.
54. Api2MoL, http://modelum.es/trac/api2mol/