Generación Automática del Modelo de Diseño desde el Modelo de Análisis a través de Reglas QVT

Ariel Arsaute, Fabio Zorzan, Marcela Daniele, Paola Martellotto

Dpto. de Computación, Facultad de Ciencias Exactas, Fco-Qcas y Naturales
Universidad Nacional de Río Cuarto
Río Cuarto, Córdoba, Argentina
{aarsaute, fzorzan, marcela, paola}@dc.exa.unrc.edu.ar

Abstract. Model Driver Architecture (MDA) define un proceso de construcción del software basado en producción y transformación de modelos. MDA se fundamenta en los principios de abstracción, automatización y estandarización. Vinculado con MDA, la Object Management Group (OMG) ha definido el estándar Query/View/Transformation (QVT) para la definición y transformación de modelos de software. Por otro lado, el Proceso Unificado (PU), también define un proceso de construcción del software generando distintas vistas o modelos. En este trabajo se sientan las bases para la integración de MDA y el PU. Se propone un conjunto de reglas QVT que establecen una transformación de forma automática entre los modelos producidos en las etapas de Captura de Requisitos, Análisis Y Diseño. El objetivo del trabajo es la definición del conjunto de reglas QVT que posibiliten la transición desde la etapa de Análisis a la etapa de Diseño considerando la tecnología de implementación RemoteMethodInvocation (RMI).

Keywords:ProcesoUnificado, MDA, Relations, QVT.

1 Introducción

La dinámica propia de la Ingeniería de Software implica el surgimiento constante de nuevas tecnologías que den soporte al proceso de construcción de sistemas de información. Todo esto implica un arduo trabajo de integración que posibilite la co-existencia de las tecnologías involucradas.

MDA [1] define un proceso de construcción de software basado en la producción y transformación de modelos. MDA se fundamenta en los principios de abstracción, automatización y estandarización. La idea principal subyacente en MDA es abstraer propiedades y características de los sistemas de información en un modelo abstracto independiente de los cambios producidos en las tecnologías.

En sintonía con MDA, la OMG, ha definido el estándar QVT [2]. QVT consta de consultas, vistas, y reglas de transformación. El componente de consultas de QVT recibe como entrada un modelo, y selecciona elementos específicos. Una vista puede verse como el resultado de una consulta sobre un modelo que proporciona un punto

de vista particular. El componente de transformaciones de QVT permite definir transformaciones entre modelos. Dichas transformaciones describen relaciones entre dos metamodelos: fuente y objetivo. Ambos metamodelos deben ser especificados en Meta ObjectFacility (MOF) [3]. Una vez definida y aplicada la transformación, se obtiene el modelo instancia del metamodelo objetivo, a partir de un modelo fuente.

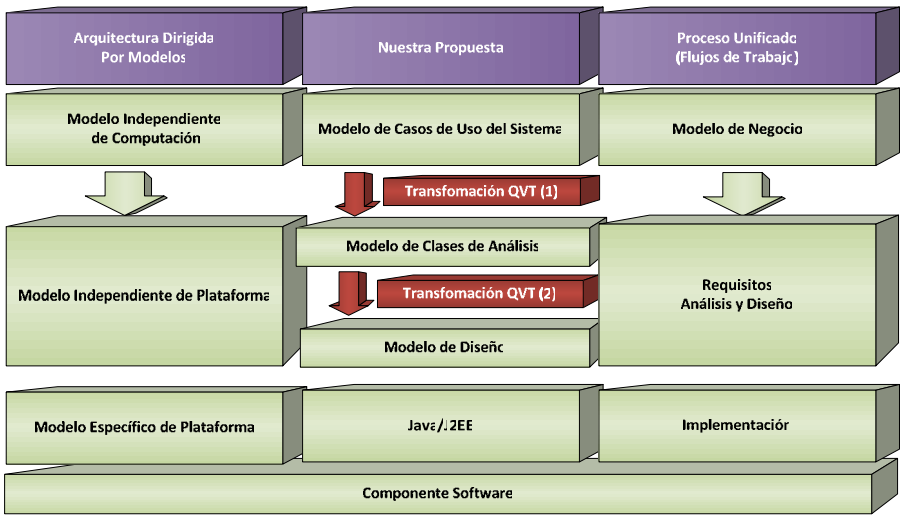

Fig. 1. Propuesta de Transformación.

Por otro lado, el Proceso Unificado [4] es una metodología de desarrollo de software que define tareas y responsabilidades para la construcción de un producto de software. La metodología divide el desarrollo en etapas, cada una de las cuales produce diferentes modelos o vistas del sistema. Las primeras etapas del proceso, centran sus esfuerzos en la comprensión del problema, las tecnologías a utilizar, la delimitación del ambiente del proyecto, el análisis de los riesgos, y la definición de la arquitectura del proyecto. Las actividades centrales son aquellas encargadas de modelar el negocio.

El objetivo final de esta línea de investigación, es lograr la automatización parcial de las actividades del PU, desde la Captura de Requisitos hasta la Implementación, aplicando reglas de transformación de modelos QVT. En este artículo se presenta el segundo paso para lograr este objetivo, que corresponde a la Transformación QVT (2) expresada en la Fig.1. Esta transformación contiene las reglas QVT que permitirán pasar del modelo de análisis al modelo de diseño. El modelo de análisis, nuestro modelo fuente, corresponde a una instancia del metamodelo de UnifiedModelingLanguage (UML) [5], el cual se obtuvo como resultado de aplicar la Transformación QVT (1) definida en [6] por los autores de este trabajo, y el modelo de diseño, nuestro modelo objetivo, también corresponde a una instancia del metamodelo UML. Como resultado, la aplicación de las reglas QVT producen un diagrama de clases de diseño (etapa de diseño) a partir de un diagrama de clases de análisis (etapa de análisis).

El artículo está organizado de la siguiente forma: En la sección 2 se presenta MDA, en la sección 3 se referencia al estándar QVT definido por la OMG. En la sección 4 se presenta la metodología de desarrollo del Proceso Unificado [4] como también tecnología y patrones que tuvimos en cuenta en el diseño. En la sección 5 se presenta la propuesta junto con un ejemplo de aplicación, y finalmente, en la sección 6 se expone las conclusiones.

2 Model Driver Architecture (MDA)

MDA define un proceso de construcción de software basado en la producción y transformación de modelos. Los principios en los cuales se fundamenta MDA son: abstracción, automatización y estandarización. El proceso central de MDA es la transformación de modelos. La idea principal subyacente es utilizar modelos, de manera que las propiedades y características de los sistemas queden contenidas en un modelo abstracto independiente de los cambios producidos en la tecnología. MDA proporciona una serie de guías o patrones expresadas como modelos.

Con respecto a la automatización, MDA favoreció al surgimiento de nuevas herramientas CASE con funcionalidades específicas destinadas al intercambio de modelos, verificación de consistencia, transformación de modelos y manejo de metamodelos, entre otras.

3 Query/View/Transformation (QVT)

La OMG [7] ha definido el estándar QVT [2] para trabajar con modelos de software. QVT consta de consultas, vistas, y transformaciones. El componente de consultas de QVT toma como entrada un modelo, y selecciona elementos específicos del mismo. Para la resolución de las consultas, se propone el uso de una versión extendida de OCL 2.0 [8]. Una vista es una proyección realizada sobre un modelo, creada mediante una transformación. Una vista puede verse como el resultado de una consulta sobre un modelo. En esta sección se presenta el componente de transformaciones de QVT que tiene como objetivo definir transformaciones entre modelos. Estas transformaciones describen relaciones entre un metamodelo fuente F y un metamodelo objetivo O, ambos metamodelos deben ser especificados en MOF [3]. Una vez definida la transformación, es utilizada para obtener un modelo objetivo, una instancia del metamodelo O, a partir de un modelo fuente, que es una instancia del metamodelo F. También la transformación puede ser utilizada para chequear la correspondencia entre dos modelos. La especificación de QVT 1.1 [2] tiene una naturaleza híbrida declarativa/imperativa. En este trabajo interesa el lenguaje Relations que tiene naturaleza declarativa.

3.1 Lenguaje Relations

El lenguaje Relations es una especificación declarativa de las relaciones entre metamodelos MOF. Una transformación especifica un conjunto de relaciones que deben cumplir los elementos de los modelos involucrados. La ejecución de una transformación en una dirección particular se realiza seleccionando el metamodelo objetivo de la transformación. Una relación especifica la relación entre elementos de los modelos candidatos. La relación involucra dos o más dominios, y dos restricciones denominadas cláusulas guard (o cláusula when) y cláusula where. La cláusula guard de la relación especifica la condición bajo la cual la relación debe ser cumplida. La cláusula where define la condición que deben cumplir los elementos intervinientes en la relación. Una relación puede ser declarada en modo sólo chequeo (checkonly) o forzado (enforced). Si un domino es marcado como sólo chequeo, cuando se ejecute la transformación sólo será chequeado para ver si existe una correspondencia válida con otro dominio perteneciente al modelo objetivo; en cambio, si el modelo está marcado como forzado, cuando una relación no se satisface, los elementos del modelo objetivo serán creados, borrados o eliminados en el modelo para satisfacer la relación. En la actualidad existen herramientas bastantes maduras que implementan el lenguaje Relations, una de estas es MediniQVT[9].

3.2 MediniQVT

Esta herramienta implementa la especificación QVT/Relations de la OMG en un poderoso motor QVT. Está diseñada para transformaciones de modelos permitiendo un rápido desarrollo, mantenimiento y particularización de reglas de transformación de procesos específicos. La herramienta está integrada a Eclipse y utiliza EMF para la representación de modelos. Además, posee un editor con asistente de código y un depurador de relaciones.

4 Proceso Unificado: Del Análisis al Diseño

La metodología del Proceso Unificado [4]divide el desarrollo en etapas, cada una de las cuales produce diferentes modelos o vistas del sistema. Las primeras etapas del proceso, centran sus esfuerzos en la comprensión del problema, las tecnologías a utilizar, la delimitación del ambiente del proyecto, el análisis de los riesgos, y la definición de la arquitectura del proyecto. Para la arquitectura del proyecto eneste trabajo se asume las siguientes consideraciones arquitecturales que permiten definir el diseño a generar: Arquitectura distribuida utilizando la tecnología Java RMI [10], uso de patrón de Arquitectura Layer [11], patrones de diseño Mediador [12] y DTO [13].

4.1 Remote Method Invocation (RMI)

RemoteMethodInvocation (RMI) [10] es un mecanismo que permite realizar llamadas a métodos de objetos remotos situados en distintas (o la misma) máquinas

virtuales de Java, compartiendo así recursos y carga de procesamiento a través de varios sistemas.

Toda aplicación RMI normalmente se descompone en 2 partes:

- **Un servidor**, que crea algunos objetos remotos, crea referencias para hacerlos accesibles, y espera a que el cliente los invoque.
- **Un cliente**, que obtiene una referencia a objetos remotos en el servidor, y los invoca.

Las aplicaciones Java que utilizan RMI deben contener básicamente del lado del servidor una clase de control. Esta clasedebe ofrecer métodos a ser invocados remotamente por un cliente, estas clases de control deben implementar Interfaces remotas. Las interfaces remotas son las utilizadas para la comunicación entre el cliente y servidor. RMI permite desarrollar aplicaciones distribuidas muy fácilmente. Entre otras ventajas de utilizar de RMI en aplicaciones Java se pueden nombrar: permite una separación entre la interface y la implementación, es posible la descarga dinámica de código y permite utilizar protocolos seguros de comunicación como SSL y HTTPS.

4.2 Data Transfer Object (DTO)

Data Transfer Object (DTO) es un patrón de diseño [11] que ofrece una solución para distribución de objetos, en particular para el intercambio de datos en llamadas remotas costosas.

Cuando se trabaja con una interfaz remota, como RMI, cada llamada remota es costosa. Como resultado, usted necesita reducir el número de llamadas, y eso significa que usted necesita para cada llamada transferir más datos.

La solución es crear un objeto de transferencia de datos (DTO) que puede contener todos los datos de la llamada. Un DTO es un objeto serializable que puede viajar fácilmente a través de la red y que contiene generalmente información perteneciente a varios objetos de dominio. Por lo general, se utiliza un ensamblador en el lado del servidor para transferir datos entre el DTO y los objetos de dominio.

El patrón Data Transfer Object [13] permite crear el objeto que será serializado para transferir información entre la interfaz del usuario (frontend) y el servidor (back end) del sistema.De esta manera se consigue un objeto serializable para pasar entre las dos capas, con la información necesaria para reducir el número de llamadas entre ellas.

5 Automatización del mapeo entre clases deanálisis a clases de diseño

El objetivo final de esta línea de investigación es lograr la automatización parcial de las actividades del PU, es decir, desde la “Captura de Requisitos” hasta la “Implementación”, aplicando reglas de transformación de modelos QVT. En este trabajo se presenta la segunda etapa que consiste en la transformación del modelo de

análisis al modelo de diseño como continuidad del trabajo previo [6], donde se realizó la transformación del modelo de CU de sistema al modelo de análisis. Como se mencionó anteriormente, una transformación en QVT requiere al menos dos modelos, uno fuente y otro objetivo. En este caso, el fuente es una instancia del metamodelo UML correspondiente al modelo de análisis y el objetivo también corresponde a una instancia del metamodelo UML, el cual es el modelo de diseño. Los modelos genéricos de análisis y diseño se muestran en las Fig. 2 y 3respectivamente.

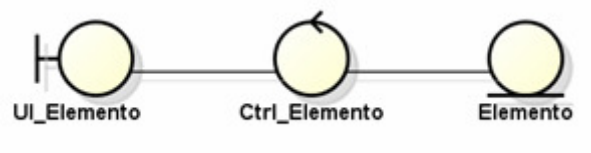

Fig. 2. Modelo genérico de análisis fuente de la transformación

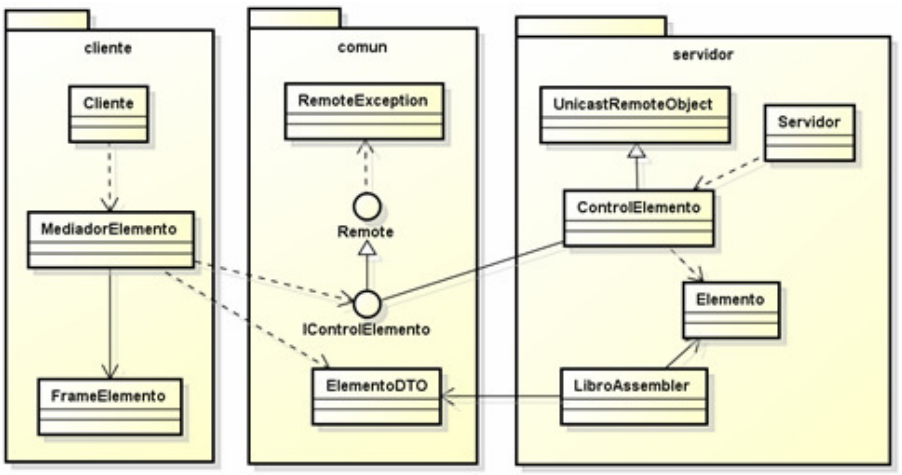

Fig. 3. Modelo genérico de diseño generado por la aplicación de la transformación

La herramienta CASE utilizada para la definición de las reglas QVT es MediniQVT [9]. Esta herramienta fue presentada en la seccion 3.2 de este trabajo.

La Fig. 4 muestra, en forma abreviada, la sintaxis gráfica de la relación packageAnalisisToPackageDisenio, esta relación es la más importante de la transformación entre los metamodelos UML de análisis y diseño

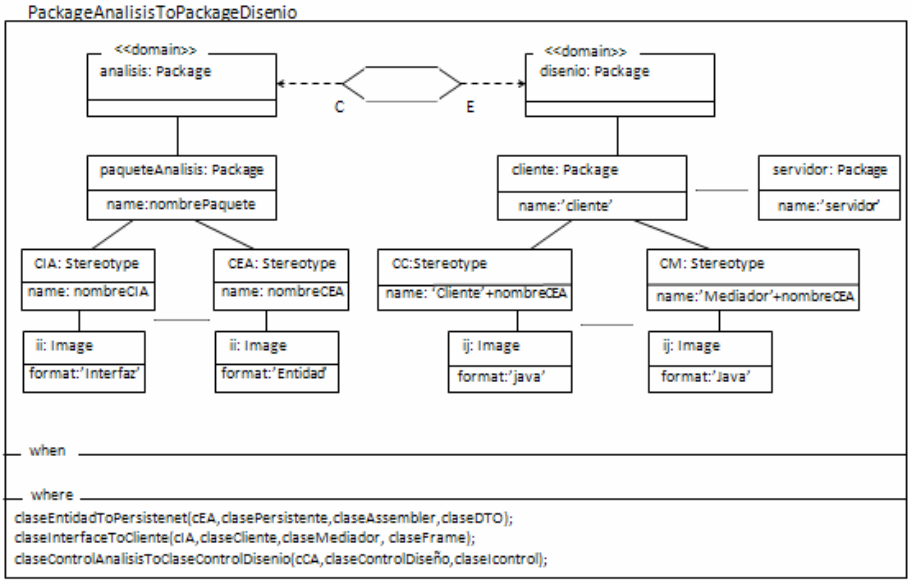

Fig. 4. Sintaxis Gráfica de laRelación

La transformación QVT se define a nivel Metamodelos y se aplica a nivel modelos. La transformación se define en Relations de la siguiente manera:

```
transformationAnalisisToDisenio
(modeloAnalisis:uml ,modeloDisenio:uml)
```

Esta transformación toma un modelo de análisis (modeloAnalisis), que es una instancia del metamodelo UML y un modelo de diseño (modeloDisenio) que también es una instancia del metamodelo UML.

A continuación se muestra la definición en Relations de la relación más importante de la transformación la cual se denominaPackageAnalisisToPackageDisenio.Cabe aclarar que en la definición presentada sólo se expone en forma completa la obtención del paquete *cliente*del diseño generado, por una cuestión de tamaño de la relación, sólo se muestra una parte de la definición de los paquetes *comun*y *servidor*.

```
toprelationPackageAnalisisToPackageDisenio {
nombrePaquete : String;
checkonlydomainmodeloAnalisisanalisis : uml::Package {
packagedElement =paquete_analisis:uml::Package
{name=nombrePaquete,
 // cIA clase Interfaz de Analisis
packagedElement = cIA:uml::Stereotype
     {name= nombreClaseInterfaz,
     icon = ii :uml::Image {format = 'Interfaz'}},
 //cCAclase Control de Analisis
```

```
packagedElement = cCA : uml::Stereotype
        {name= nombreClaseControl,
     icon = ic :uml::Image {format ='Control'}},
//cEAclase Entidad de Analisis
packagedElement = cEA: uml::Stereotype
        {name= nombreClaseEntidad,
     icon = ie :uml::Image {format = 'Entidad'}},
name = nombrePaquete
   };
enforcedomainmodeloDiseniodisenio: uml::Package {
packagedElement =cliente : uml::Package{name='cliente',
//cC clase control
packagedElement = cC : uml::Stereotype
{name='Cliente',
icon = ij :uml::Image {format = 'Java'}},
//cM clase Mediador
packagedElement = cM : uml::Stereotype
 {name='Mediador'+nombreClaseEntidad,
icon = ij :uml::Image {format = 'Java'}},
//cFclase Frame
packagedElement = cF : uml::Stereotype
{name='Frame'+nombreClaseEntidad,
icon = ij :uml::Image {format = 'Java'}},
packagedElement =comun : uml::Package
 {name='comun',
       packagedElement =claseRemoteException:uml::Stereotype
              :
              :     },
packagedElement =servidor:uml::Package
     {name='servidor'
:  },
name = nombrePaquete
};
where {
claseEntidadToPersistente(cEA,clasePersistente,
claseAssembler,claseDTO);
claseInterfaceToCliente(cIA,claseCliente,claseMediador,
claseFrame);
claseControlAnalisisToClaseControlDisenio(cCA,
claseControlDiseño,claseIcontrol);
}
}
```

La relación PackageAnalisisToPackageDisenio define la transformacióndel paquete UML del modelo de análisis al modelo de diseño.

En la especificación de la relación se define la transformación de cada paquete de análisis a su correspondiente paquete de diseño, siendo ambos paquetes UML. El paquete de diseño se organiza a su vez en tres paquetes que son:*cliente, común* y *servidor*; que según las consideraciones del diseño y a la experiencia de éste equipo con aplicaciones RMI son los más adecuados

A partir del modelo de análisis con clases de tipo Interfaz, control y entidad, se generan 3 paquetes de diseño, estos son: *cliente*, *comun* y *servidor*. Estos paquetes se generan debido a que la tecnología de implementación será RMI, y como se explicó en la sección 4.1, estos paquetes son necesarios. Dentro del paquete cliente se construirá una única clase Cliente, que es la encargada de iniciar el programa cliente de la aplicación y recuperar todas las interfaces remota para poder interactuar con el servidor. También, se generan en el paquete, por cada clase Interfaz del análisis, una clase Frame y Mediador, la clase Frame es la encargada de interactuar con el usuario, y la clase Mediador es la encargada de mediar entre la Clase Frame y las clases de control del servidor. Dentro del paquete *comun* se generan, por cada clase de tipo control de análisis, una Interfaz Java. Estas interfaces son las que se publicarán por medio del demonio RMI cuando se inicialice el servidor de la aplicación. Además en el paquete *comun* de diseño, por cada clase Entidad de Análisis se genera una clase Java DTO que es la encargada de llevar y traer información del entre el cliente y servidor (ver sección 4.2). Por último, en el paquete *servidor,* se genera una única clase Servidor que es la encargada de iniciar la aplicación servidora publicando las interfaces RMI para que las aplicaciones cliente se puedan conectar. También, se generan, por cada clase de tipo Entidad del análisis, una clase Persistente y otra Assembler que es la encargada de hacer la traducción entre la clase Persistente y la DTO correspondiente del paquete *comun*. Las últimas clases generadas en el paquete servidor son las clases de control, estas clases se generan a partir de cada clase de tipo control del análisis.

Las diferentes relaciones entre las clases son generadas mediante las relaciones `claseEntidadToPersistente`, `claseInterfaceToCliente` y `claseControlAnalisisToClaseControlDisenio`que son parte de la cláusula*where* de la relación.

5.1 Ejemplo de Aplicación

En esta sección se presenta un ejemplo de aplicación de la transformación QVT definidas en el apartado anterior.

En la Fig. 5 se muestra parte del diagrama de clases de análisisdel caso de uso gestión de libros obtenido del ejemplo de aplicación del trabajo previo, y que es modelo fuente de la transformación del presente trabajo. En la Fig.6 se muestra el modelo de análisis en la vista generada por el EcoreModel Editor. En la Fig. 7 se presenta el diagrama de clases de diseño obtenido por aplicación de la transformación al modelo de análisis de la gestión de Libros.

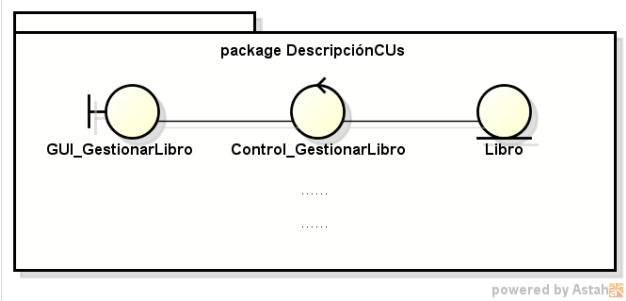

Fig. 5. Diagrama de clases de Análisis - Caso de Uso gestionar

Para llevar a cabo esta transformación se utilizó la herramienta MediniQVT. Teniendo en cuenta que MediniQVT utiliza EMF para representar los modelos/metamodelos involucrados en las transformaciones. Cabe aclarar que el metamodelo UML en formato EMF, que es el metamodelo fuente y objetivo en la definición de la transformación, viene provisto por la herramienta. Para poder aplicar la transformación, que es lo más importante del caso de estudio, fue necesario obtener el modelo fuente, el cual es el resultado de aplicar la transformación realizada en [6]. Luego de obtener los modelos/metamodelos, se aplicó la transformación al modelo fuente y se obtuvo el modelo objetivo. Este modelo corresponde a una especificación UML de diagrama de clases de diseño.En la Fig. 8 se muestra el modelo de diseño en la vista generada por el EcoreModel Editor.

Fig. 6. Diagrama de clases de Análisis - Caso de Uso gestionar Libro, vista generada por el EcoreModel Editor.

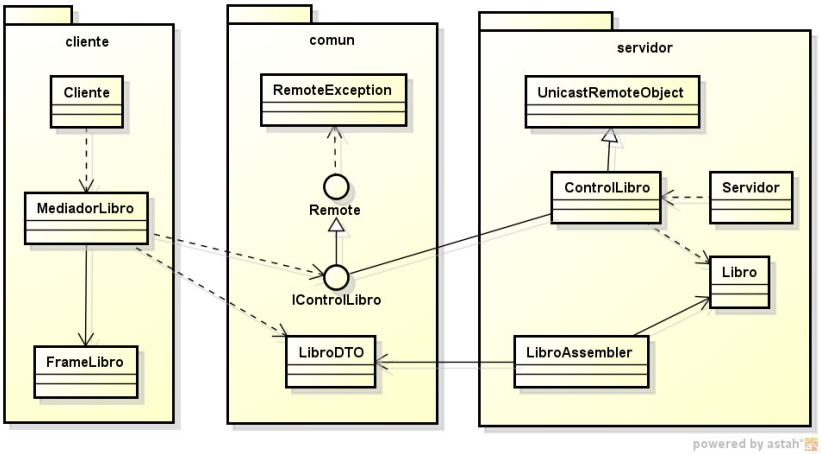

Fig. 7. Diagrama de clases de Diseño - Caso de Uso "Gestionar Libro".

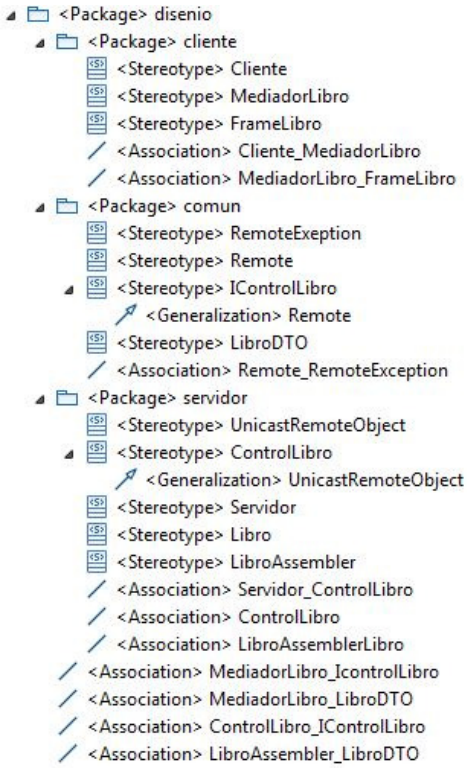

Fig. 8. Diagrama de clases de Diseño obtenido por la aplicación de la transformación, vista generada por el EcoreModel Editor.

6 Conclusiones

Este trabajo tiene como objetivo hacer una contribución a la mejora de los procesos de desarrollo de software. La elaboración y culminación exitosa de este caso de estudio permitió la utilización de una especificación de captura de requerimiento para dar inicio a la construcción del software de manera automática en el marco de MDA utilizando reglas de transformación QVT.

El beneficio de esta transformación se refleja también en el dinamismo de los cambios en los requisitos durante toda la vida del software desarrollado, es decir, cualquier cambio en la especificación de la captura de requerimientos podrá ser propagado automáticamente a los distintos artefactos producidos en el proceso de desarrollo del software, esto debido a que se definió una transformación que puede ser ejecutada con una herramienta permitiendo así adaptar rápidamente los cambios de la captura de requerimiento a la especificación del análisis y diseño.

En esta línea de Investigación, hasta el momento se ha avanzado en la transformación que convierte la especificación de Captura de Requerimientos a una especificación UML correspondiente al artefacto de diseño, pasando por sus respectivas transformaciones. El próximo objetivo es realizar la transformación hacia el modelo final correspondiente al componente de software y de esta manera cubrir con todas las etapas del PU.

Por último, este trabajo intenta promover el uso eficiente de modelos de sistemas en el proceso de desarrollo de software (desarrollo de software dirigido por modelos) que es uno de los principales objetivos de MDA, en el marco de la Ingeniería de Software dirigida por modelos, representando para los desarrolladores, una nueva manera de organizar y administrar arquitecturas empresariales, basada en la utilización de herramientas de automatización de etapas en el ciclo de desarrollo del software. De esta forma, permitir definir los modelos y facilitar trasformaciones paulatinas entre diferentes modelos.

Referencias

1. Miller, J., Mukerji, J., MDA Guide Version 1.0.1 Document number omg/2003-06-01, Disponible en: http://www.omg.com/mda, 2003.
2. Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, OMG Document Number: formal/2011-01-01, Standard Document URL: http://www.omg.org/spec/QVT/1.1/PS/, últimoaccesoNoviembre 2012.
3. Object Management Group Meta Object Facility (MOF) Core Specfication OMG Available Specication. Versión 2.0. formal/06-01-01, http://www.omg.org/docs/formal/06-01-01.pdf, último acceso Febrero 2013.
4. Jacobson, I. El Proceso Unificado de Desarrollo de software. Addison-Wesley, EE.UU., 2000.

5. Booch G., Rumbaugh J., Jacobson I., The Unified Modeling Language. Addison Wesley, Second Edition. 2005.

6. Ariel Arsaute, Marcelo Uva, Fabio Zorzan, y otros, “Hacia una integración de MDA y el Proceso Unificado a través de reglas de transformación QVT”. 41 Jornadas Argentinas de Informática – JAIIO 2012.

7. Object Management Group, http://www.omg.org,últimoaccesoAbril 2013.

8. Object Management Group Object Constraint Language Version 2.0. OMG DocumentNumber: formal/06-05-01, Standard Document URL:http://www.omg.org/cgi-bin/doc?omg/03-06-01, último acceso Marzo 2013.

9. ikv++: medini QVT. http://www.ikv.de/, ultimo acceso Agosto 2012.

10. ORACLE, Java SE Documentation, Java RemoteMethodInvocation URL: http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html, último acceso Noviembre 2012.

11. Frank Buschmann&otros, Pattern-Oriented Software Architecture, Wiley; Volume 1 edition (August 8, 1996).

12. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissedes. Design Patterns, Elements of Reusable Object/Oriented Software- Addison-Wesley, 1995.

13. Fowler, Martin, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.