MDSD multi-plataforma: más allá de la vista funcional*

*Juan Camilo Jiménez***
*Jesús Andrés Hincapié Londoño****
*Juan Bernardo Quintero*****

Recibido: 15/12/2015 • Aceptado: 14/07/2016
DOI: 10.22395/rium.v15n29a9

Resumen

Los enfoques de desarrollo de software dirigido por modelos (MDSD, por sus siglas en inglés) se han basado tradicionalmente en la vista funcional obteniendo resultados positivos durante los últimos años; no obstante, presentan restricciones en el soporte para la generación hacia múltiples plataformas. Este artículo propone un enfoque multivistas para MDSD que permite el modelado de la plataforma (vistas lógica y física de un sistema de software), de tal manera que se puedan expresar y reutilizar arquitecturas de software mediante el uso de modelos.

Palabras clave: modelado de plataforma, MDSD, vista lógica, vista física, enfoque multi-vistas MDSD.

* Artículo derivado de la investigación Metáfora: generación de aplicaciones web y móviles basadas en procesos de negocio, usando ingeniería dirigida por modelos, financiada por Universidad de Medellín, ABC-Flex Ltda, Perceptio S.A.S, Inteligencia Móvil S.A.S y Universidad EAFIT y ejecutada entre febrero de 2013 y junio de 2015

** Ingeniero de sistemas, Msc. en Ingeniería de Software, Universidad de Medellín. Calle 75 Sur #35 – 51, Sabaneta, Antioquia +57 (4) 4171742 / +57 3024588699. jjimenez@kindermusik.com

*** Ingeniero de sistemas, Msc. Universidad de Antioquia, Profesor de tiempo completo Universidad de Medellín. Cr. 87 N.° 30-65. Teléfono (57-4)3405660. jehincapie@udem.edu.co. Fax: (57-4) 3405216

**** Ingeniero de sistemas, PhD. Universidad de Antioquia, Profesor de tiempo completo Universidad de Medellín. Cr. 87 N.° 30-65. Teléfono (57-4)3405660. jbquintero@udem.edu.co. Fax: (57-4) 3405216

Multi-platform MDSD: beyond the functional view

Abstract

Traditional Model Driven Software Development (mdsd) approaches have traditionally been based on the functional view and have yielded positive results in recent years; however, they present support restrictions for generation in multiple platforms. This article proposes a multi-view approach for mdsd that allows to model the platform (views, logics and physics of a software system) in such way that software architectures may be expressed and reused by using models.

Key words: platform modeling, mdsd, logical view, physical view, multi-view mdsd approach.

INTRODUCCIÓN

El desarrollo de software dirigido por modelos ha estado dominado en los últimos años por enfoques simples que se basan exclusivamente en el modelado de la vista funcional del sistema, mientras que la lógica de plataforma depende de la herramienta que se esté utilizando, lo que tiene como efecto que la generación de aplicaciones sea poco flexible al estar mayormente ligada a las plantillas utilizadas para la generación de modelo a texto [1]. Tal es el caso de enfoques como Object Oriented Web Solutions (OOWS) [2], UML-Based Web Engineering (UWE) [3], Web Modeling Language (WebML) [4], Hypertext Modeling Method of MIDAS [5], DSL for the implementation of dynamic web applications (WebDSL) [6], DSL for generating Web application (MarTE/Quorra) [7] y Web Software Architecture (WebSA) [8]. Todas las iniciativas listadas anteriormente definen tres tipos de modelos: estructura, navegación y presentación, pero también hay otros casos en que se usan algunos modelos extra, incluso de forma implícita como en el caso de MarTE/Quorra.

La mayoría de estos enfoques no considera un mecanismo para describir la arquitectura de destino de manera separada de las transformaciones, sino que los autores deciden primero la arquitectura y la tecnología en la que se va a generar la aplicación, de tal manera que la lógica de modelado de la plataforma queda combinada con las transformaciones. Esto lleva a que la arquitectura de destino sea totalmente dependiente del método y no se pueda reusar para diferentes plataformas. No obstante, en WebSA [8], los autores aseguran que su enfoque incluye la modularización de la plataforma y las transformaciones para soportar diferentes arquitecturas.

Para solucionar estas carencias, este artículo propone modelar vistas para definir los elementos propios de la plataforma dentro de un enfoque multi-vistas MDSD en el marco de un proyecto llamado *Metáfora* [1]. Este enfoque debe permitir la expresión de la arquitectura tanto lógica como física (configuración de plataforma), la cual puede ser usada en diversos procesos de generación de aplicaciones. La generación hacia múltiples plataformas se puede alcanzar cuando se usan los mismos modelos de una aplicación, combinándolos con distintas configuraciones, para generar versiones de la misma aplicación en diferentes plataformas. Lo anterior sugiere que exista un repositorio o mecanismo que permita el almacenamiento y reuso de las configuraciones de plataforma en diferentes proyectos de desarrollo de software; inclusive se pueden aplicar varias configuraciones de plataforma en un mismo proyecto, en donde, por ejemplo, parte del sistema se despliega en la web y otra parte se despliega en dispositivos móviles.

Para estructurar esta propuesta de modelado de plataforma, se debe elaborar un lenguaje específico de dominio (DSL, por sus siglas en inglés) [9] a partir de las siguientes tareas que constituyen las secciones del presente artículo:

1. Identificar las características y responsabilidades de los elementos que hacen parte del diseño arquitectónico.
2. Definir un metamodelo que consolide las características de la plataforma con relevancia para la generación de código fuente.
3. Especificar mecanismos para modelar y transformar los elementos conforme al meta-modelo de la plataforma.
4. Detallar las reglas de integración entre el modelo funcional y el modelo de plataforma e implementar la generación de código fuente con dichos modelos.

La tecnología elegida para desarrollar esta propuesta es la plataforma Eclipse Modeling Framework (EMF) de *Eclipse* con la implementación particular del *framework* de modelado *Epsilon* dada la variedad y madurez de sus lenguajes y herramientas de modelado. El progreso de esta propuesta se ilustrará con un estudio de caso [1] basado en el proceso de gestión de incidentes (IcM, por sus siglas en inglés) de la biblioteca de Infraestructura de Tecnologías de la Información (ITIL, por sus siglas en inglés).

1. CARACTERIZACIÓN DE LOS ELEMENTOS DEL MODELO DE PLATAFORMA

El modelado de la plataforma, también conocido como el diseño arquitectónico, comprende todos aquellos aspectos no funcionales tanto en el nivel lógico o de patrones de diseño como en el nivel físico o de despliegue [10]; corresponde a las vistas lógica y física del sistema y los enlaces entre ellas. Por consecuente, para la construcción del metamodelo de plataforma de esta propuesta se deben puntualizar las responsabilidades y características de cada una de estas vistas.

La vista lógica: es la encargada de mostrar los principales elementos de diseño y sus relaciones de forma independiente de los detalles técnicos y de cómo la funcionalidad será implementada en la plataforma de ejecución; en palabras de Kruchten: "describe la organización estática del software en su ambiente de desarrollo" [11]. En esta vista se expresan los patrones de diseño arquitectónico, en especial el patrón de capas lógicas (*layers*), y en su mayor parte se enfoca en la facilidad de desarrollo, gestión de la configuración y reuso o funcionalidades comunes. De esta vista del sistema se pueden derivar los siguientes aspectos importantes en el código fuente:

- La estructura de carpetas de la solución.
- La distribución en proyectos o paquetes y las relaciones entre ellos.
- Las referencias (*import* y *access*) entre componentes internos y externos.
- El flujo de datos de la solución entre cada una de las capas.

La vista física: describe el mapeo del *software* en el *hardware* dentro del diseño arquitectónico [11]. Esta vista abarca los nodos (*tiers*) que forman la topología de hardware en la que la aplicación se ejecuta. Se enfoca en la distribución, comunicación y aprovisionamiento [12], y se construye desde la perspectiva de un ingeniero de plataforma que se preocupa por el *hardware* y las comunicaciones con sus requisitos no funcionales asociados, tales como disponibilidad, confiabilidad, rendimiento, carga de trabajo y escalabilidad [13]. Esta vista muestra los artefactos propios del despliegue de la aplicación y sus dependencias en tiempo de ejecución. Dichos artefactos se mapean a los componentes de la vista lógica para conformar una representación completa de la plataforma. Los elementos del código fuente producto de esta vista son:

- Las tecnologías y versiones utilizadas.
- Los archivos (artefactos) específicos.
- Las referencias entre artefactos.
- La caracterización de los artefactos.
- Los nombres y extensiones de los archivos.

Los *artefactos* de la vista física tienen diferentes características que deben ser analizadas desde diferentes puntos de vista, como se aprecia en la figura 1:

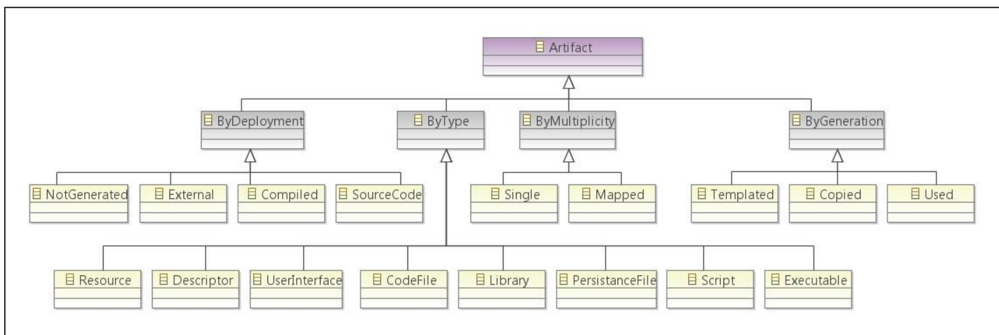


Figura 1. Taxonomía de artefactos desde diferentes puntos de vista

Fuente: elaboración propia

- *Según su multiplicidad*: un artefacto puede tener una sola instancia, como es el caso de los archivos de configuración, o múltiples instancias mapeadas a varios elementos de una aplicación, como por ejemplo, las interfaces de usuario.
- *Según su generación*: un artefacto puede ser concebido dentro de la vista física de tres formas: puede responder a una plantilla, puede ser copiado directamente de una ubicación o simplemente estar publicado en una ubicación externa y ser referenciado en tiempo de ejecución.

- *Según el despliegue*: considerando la representación de los artefactos en el diagrama de despliegue encontramos los siguiente tipos de artefactos: código fuente, artefactos compilados, artefactos externos que no están alojados en el repositorio local y artefactos no generados que se representan al nivel de diseño, pero no corresponden a ninguna instancia en el código fuente, sino típicamente a entidades en tiempo de ejecución.

- *Según su naturaleza*: muchos son los tipos de artefactos en los que se pueden agrupar según sus características y valor semántico para llegar a una generalización independiente de la plataforma. Para la tipificación de los artefactos de código fuente, se consultó con varios desarrolladores y arquitectos con experiencia en ocho diferentes tecnologías de vanguardia: Java, .NET (ASP, MVC, WinForms, WPF), PHP, Ruby on Rails, Objective C (iOS), Windows Phone, Android y Visual Basic 6. Se les pidió que identificaran y retroalimentaran sobre los tipos de artefactos que se utilizan en estas plataformas y a partir de su experiencia conjunta se generó la taxonomía de artefactos según su naturaleza sintetizada en siete grandes tipos: recursos, descriptores, interfaces de usuario, archivos de código, librerías, archivos de persistencia, *scripts* y ejecutables.

Para enlazar los elementos de las vistas lógica y física se aprovecha el concepto de *manifestación* [14], en el que una *layer* de la vista lógica se realiza en una o varias *tiers* de la vista física a través de relaciones de manifestación que implican el uso de artefactos o componentes. Incluso algunos autores [14] mencionan la necesidad de un diagrama intermedio llamado el *diagrama de manifestación*, el cual aún no hace parte de la especificación oficial del lenguaje unificado de modelado (UML, por sus siglas en inglés); no obstante, el concepto de manifestación se ajusta para mapear los componentes lógicos con los físicos; en este caso, podemos concluir que una *layer* se manifiesta en una o varias *tiers* a través de uno o más artefactos.

2. META-MODELO DE LA PLATAFORMA

Los elementos que permiten modelar la plataforma se representan en un metamodelo usando *Ecore* el formato propio de EMF; este metamodelo se basa en uno de los patrones más populares en el diseño de software como es el diseño por capas, caracterizando los contenedores de la vista lógica en *layers* que representan los paquetes, y los de la vista física, en *tiers* [15] dentro de las cuales están contenidos los artefactos que enlazan las vistas por medio de *manifestaciones* (ver figura 2). Cabe resaltar que como las *layers* se pueden anidar jerárquicamente, se implementa el patrón *composite*, lo que da como resultado paquetes simples y compuestos [16].

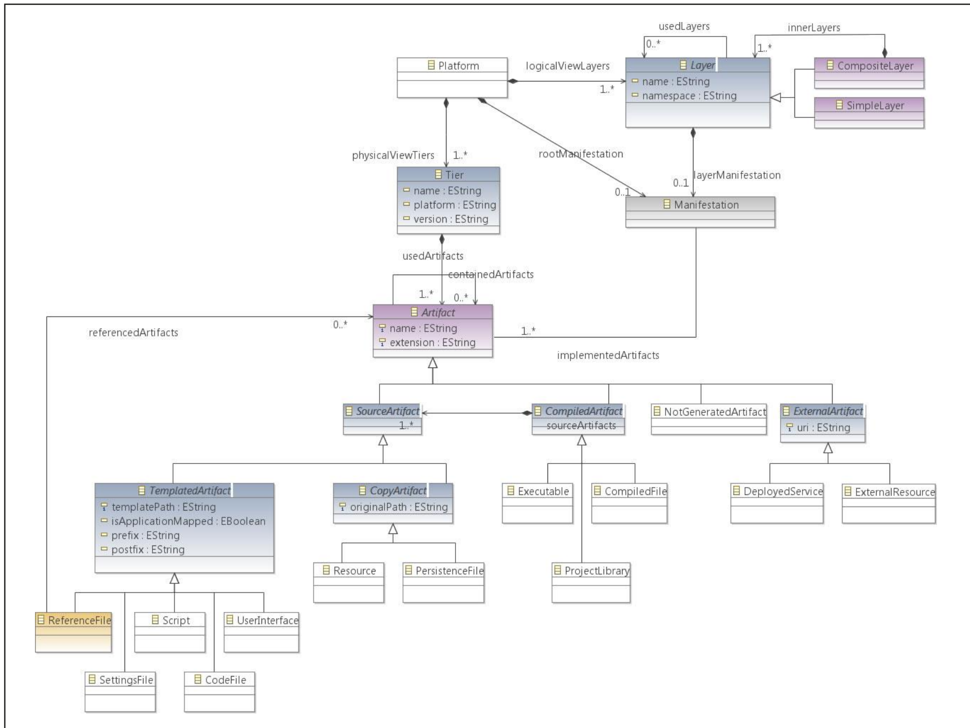


Figura 2. Meta-modelo de la plataforma
Fuente: elaboración propia

Los artefactos poseen diferentes particularidades con un gran valor semántico de cara a la generación de código fuente; es por esto que el metamodelo de la plataforma es extendido mediante la clasificación de artefactos en función a la taxonomía descrita en el capítulo anterior. Según la manera en la que se despliegan los artefactos, podemos definir los cuatro tipos principales: *fuente*, *compilado*, *externo* y *no generado*. Para los artefactos compilados debe existir una correspondencia con varios archivos fuente contenidos. A su vez, los artefactos de tipo fuente pueden ser copiados directamente de otra ubicación o responder a una plantilla de generación. Para finalizar la clasificación de artefactos se incluye la tipificación según la naturaleza derivada de la taxonomía del apartado anterior; de esta manera se facilita al usuario la asignación de tipos.

3. MECANISMOS PARA MODELAR LA PLATAFORMA

Las entradas del modelo de plataforma son: el modelo de *weaving* de plataforma y las vistas lógica y física, que se construyen con un editor basado en el metamodelo de UML versión 2, como, por ejemplo, *Papyrus*, se concretan en un diagrama de *paquetes de desarrollo* (ver figura 3) y uno de *despliegue* (ver figura 4).

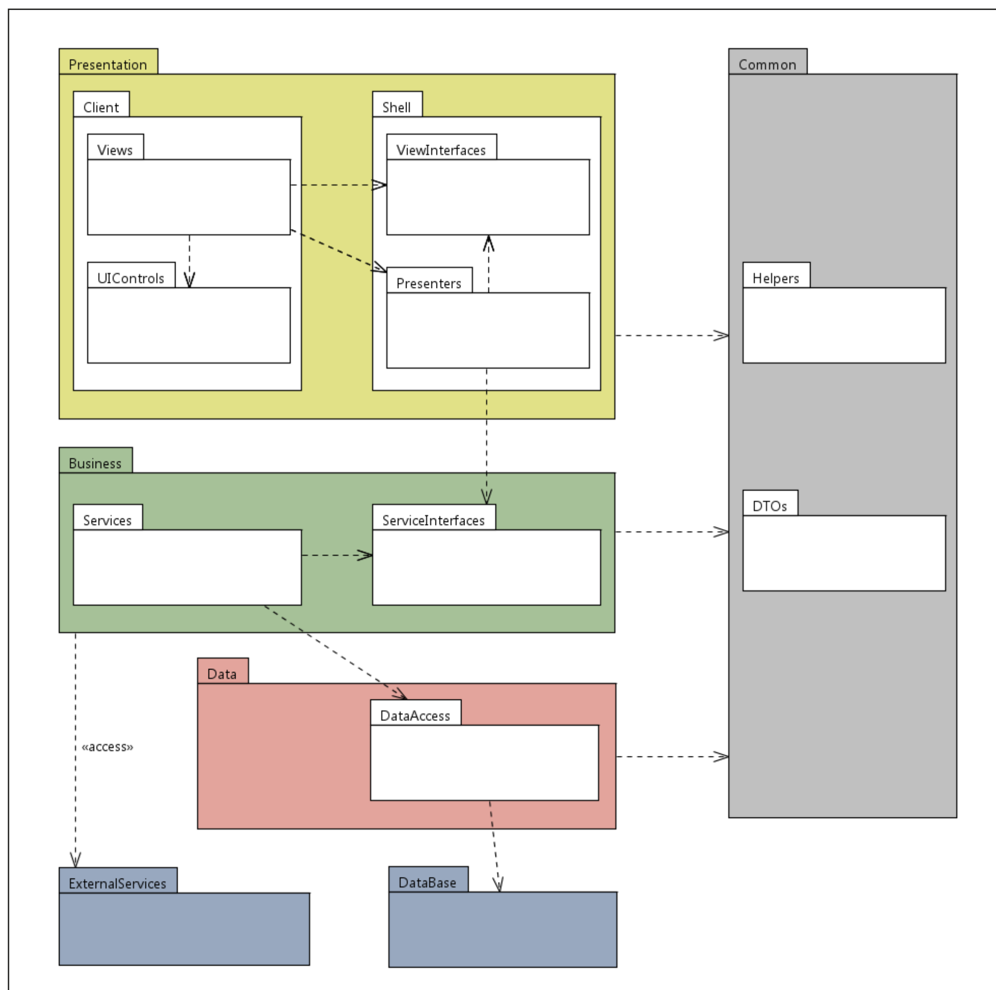


Figura 3. Diagrama de paquetes de IcM implementando el patrón MVP en papyrus
Fuente: elaboración propia

Para enlazar las vistas lógica y física se requiere un mecanismo que permita la combinación automática de modelos basado en una serie de correspondencias preestablecidas. Esto tiene un número de aplicaciones en el proceso de MDSD [17]. El *model weaving* [18] es una operación genérica que establece correspondencias directas al nivel de granularidad detallado entre varios modelos que típicamente responden a diferentes metamodelos; en otras palabras, la práctica del weaving permite generar mapeos directos entre dos o más modelos. En el caso del modelado de la plataforma es necesario que el usuario defina cuáles artefactos manifiestan cada *layer*.

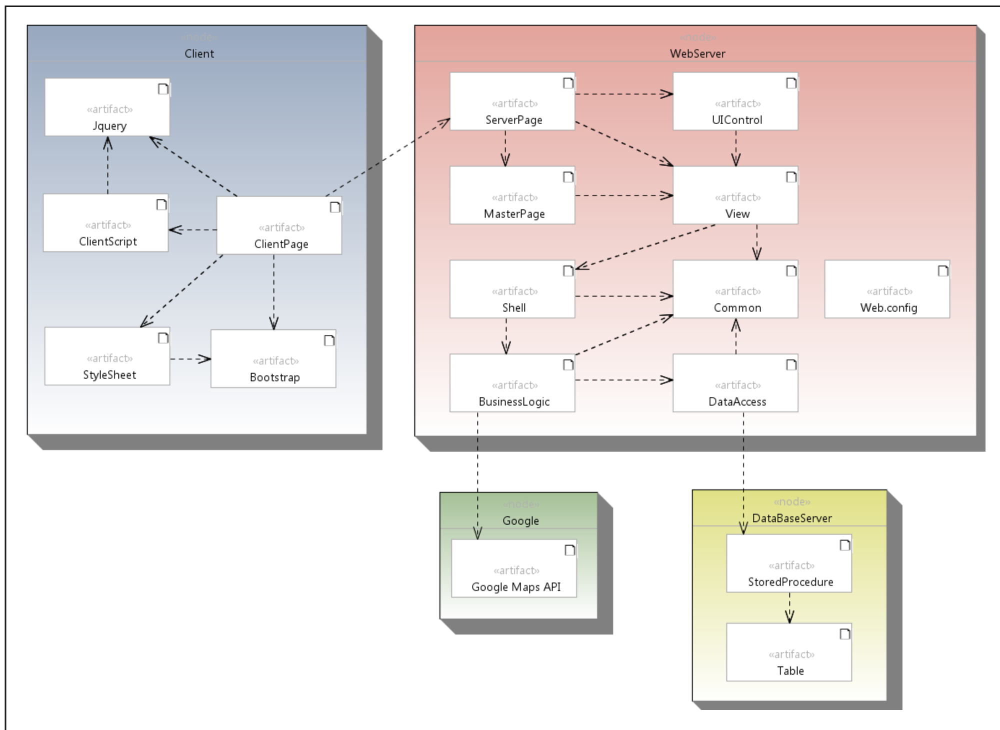


Figura 4. Diagrama de despliegue de IcM representando una arquitectura ASP.NET en papyrus
Fuente: elaboración propia

Es necesario, entonces, definir un *meta-modelo intermedio de weaving para la plataforma* (ver figura 5) muy similar a una parte del modelo de plataforma y que contenga los elementos comunes con los que se puedan expresar las manifestaciones. Asimismo, y para aprovechar la interacción del usuario, este modelo debe permitir capturar información adicional que sea requerida para completar el modelo de plataforma y que no esté expresada en los diagramas UML de paquetes de desarrollo y despliegue. El weaving de plataforma debe cumplir con los siguientes objetivos:

- Ligar paquetes con artefactos (manifestaciones).
- Tipificar artefactos.
- Desglosar artefactos compilados en sus artefactos fuente.
- Agregar nuevos artefactos que no hagan parte del diagrama de despliegue.
- Definir las plantillas o URIs de los artefactos.
- Definir propiedades adicionales de los artefactos.

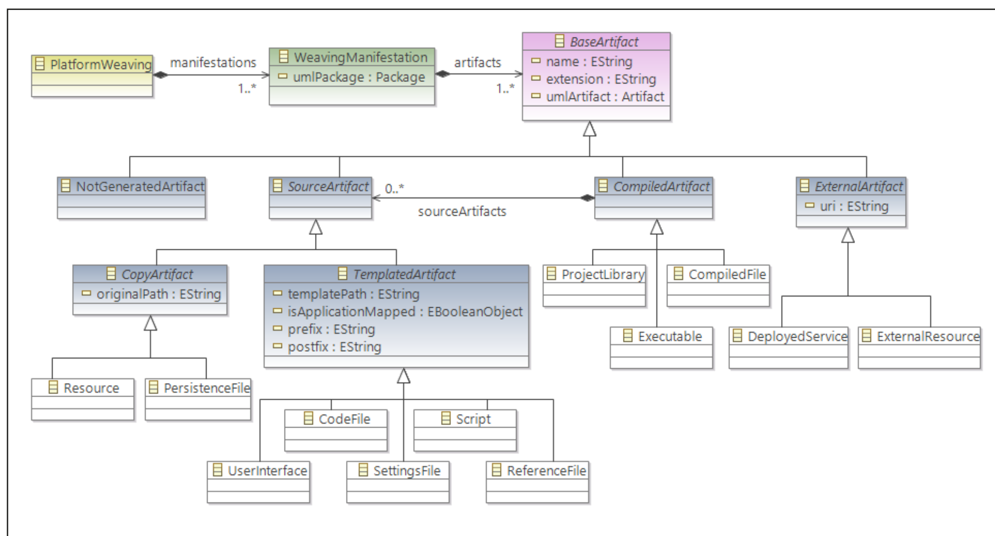


Figura 5. Meta-modelo de weaving de plataforma

Fuente: elaboración propia

La plataforma *Epsilon* de *Eclipse* provee una herramienta llamada *ModeLink* (ver figura 6) que permite la edición en paralelo de varios modelos EMF con la facilidad de arrastrar y soltar elementos para crear enlaces entre los modelos. De esta manera se pueden cargar en paralelo los modelos UML de paquetes de desarrollo y despliegue junto con el modelo de *weaving* de la plataforma para generar los enlaces necesarios

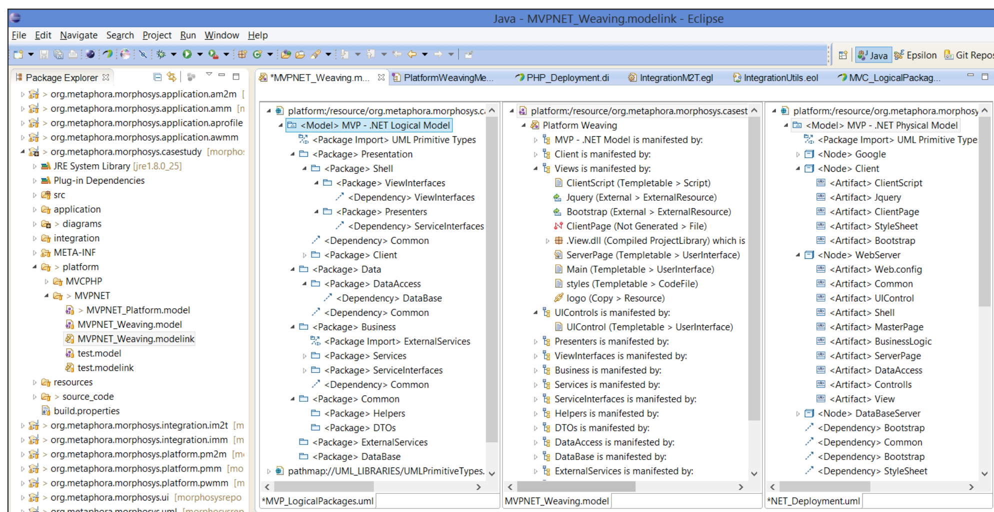


Figura 6. Weaving de plataforma de IcM en el editor modelink de épsilon

Fuente: elaboración propia

y permitir al usuario especificar la información adicional requerida de cara a la generación del modelo de plataforma.

Una vez definidos los mecanismos para construir las tres entradas del modelo de plataforma ya se puede proceder a la generación del mismo (ver figura 7). Para combinar los tres modelos de origen no es suficiente aplicar una transformación de modelo a modelo (M2M, por sus siglas en inglés) de mapeo tradicional, sino que se debe aplicar una transformación de modelos, basada en los enlaces especificados en el modelo de *weaving* de plataforma.

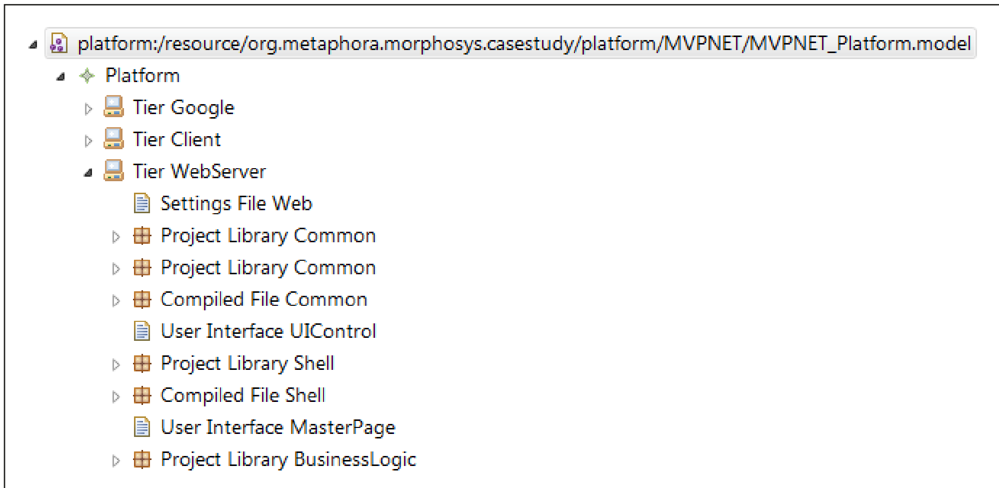


Figura 7. Instancia del modelo de plataforma MVP / ASP.NET para IcM
Fuente: elaboración propia

Algunas investigaciones [19] [20] han demostrado que la combinación de modelos se puede descomponer en cuatro fases: comparación, conformidad de verificación, mezcla y reestructuración. Dentro de la familia de lenguajes de *Epsilon* existe soporte para cada una de estas etapas. El lenguaje de comparación de *Epsilon* (ECL, por sus siglas en inglés) permite definir los criterios de comparación entre los elementos enlazados; el lenguaje de validación de *Epsilon* (EVL, por sus siglas en inglés) sirve para validar dichos elementos, mientras que el lenguaje de mezcla de *Epsilon* (EML, por sus siglas en inglés) es un lenguaje declarativo que permite aplicar reglas de transformación a elementos equivalentes. Para el caso de elementos en los que no se encuentra equivalencia, se crean reglas de transformación similares a las de EML, pero usando el lenguaje de transformación de *Epsilon* (ETL, por sus siglas en inglés) de manera que se realiza transformación M2M de mapeo directo.

4. INTEGRACIÓN DEL MODELO FUNCIONAL Y GENERACIÓN DE CÓDIGO

El modelo de la plataforma sintetiza toda la información arquitectónica relevante tanto desde el punto de vista lógico (patrones y capas) como desde el punto de vista físico (estructura de carpetas y archivos que se deben generar para cada arquitectura). La generación de código fuente o transformación de modelo a texto (M2T, por sus siglas en inglés) consiste entonces en la combinación del modelo de aplicación (que hace parte de otra iniciativa dentro del proyecto Metáfora [1]) y el modelo de plataforma teniendo en cuenta que:

- La estructura de carpetas de la solución se genera en función de la jerarquía de *layers* del modelo de plataforma.
- La generación de los artefactos se realiza dependiendo de los tipos definidos en los modelos de plataforma.
- Los artefactos de tipo no generado, como su nombre lo indica, son ignorados por el proceso de transformación M2T.
- Los artefactos de tipo externo tampoco son generados y solo se utilizan como parte del texto de las plantillas de los artefactos fuente.
- Los artefactos compilados no se generan directamente, sino a través de sus artefactos fuente contenidos.
- Los artefactos fuente de tipo copia no responden a una plantilla sino que se realiza una copia directa de algún archivo según la ruta especificada.
- Los artefactos con plantilla son generados usando el lenguaje de generación de *Epsilon* (EGL, por sus siglas en inglés).
- Los artefactos con plantilla que no están mapeados se generan una sola vez, mientras que los que están mapeados se generan uno por cada bloque presente en el modelo de aplicación.
- La lógica de aplicación, tal como la distinción de los diferentes tipos de interfaz de usuario, no es desarrollada en esta iniciativa.
- Los paquetes que no hacen parte de una manifestación, directamente o a través de uno de sus paquetes contenidos no se pueden mapear a la aplicación.
- Los artefactos de código fuente que no hacen parte de una manifestación no se generan.

Para cada arquitectura que se implementa se deben generar las respectivas plantillas EGL usando sentencias del lenguaje de objetos de *Epsilon* (EOL, por sus siglas en inglés) que contarán con la disponibilidad de las siguientes variables de los modelos de aplicación y plataforma:

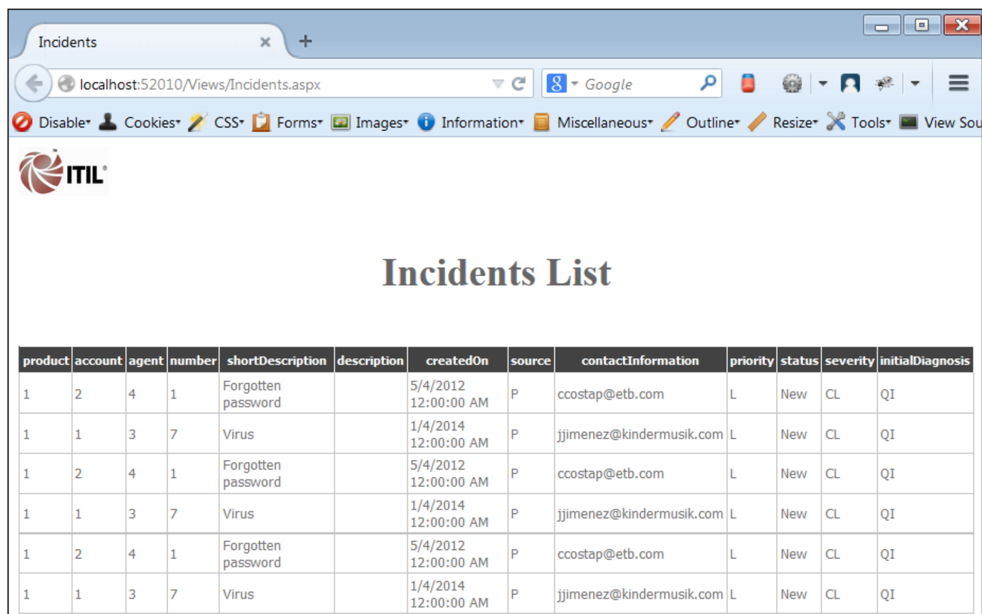
- Nodo raíz del modelo de aplicación.
- Información del artefacto que se está generando, incluyendo su ruta completa.
- Bloque de aplicación específico en el caso de los artefactos mapeables.

La intención de esta propuesta es generar un repositorio de plantillas para diferentes arquitecturas, de modo que cada modelo de plataforma se pueda reutilizar para varios modelos de aplicación y, de manera recíproca, que el mismo modelo de aplicación se pueda generar automáticamente en varias plataformas (ver figura 8 y figura 9).

Figura 8. Ejemplo de código fuente de la aplicación para IcM generada con Morphosys corriendo en Visual Studio

Fuente: elaboración propia

El proceso de modelado de plataforma presentado en este artículo se puede resumir entonces en la figura 10.



product	account	agent	number	shortDescription	description	createdOn	source	contactInformation	priority	status	severity	initialDiagnosis
1	2	4	1	Forgotten password		5/4/2012 12:00:00 AM	P	ccostap@etb.com	L	New	CL	QI
1	1	3	7	Virus		1/4/2014 12:00:00 AM	P	jjimenez@kindermusik.com	L	New	CL	QI
1	2	4	1	Forgotten password		5/4/2012 12:00:00 AM	P	ccostap@etb.com	L	New	CL	QI
1	1	3	7	Virus		1/4/2014 12:00:00 AM	P	jjimenez@kindermusik.com	L	New	CL	QI
1	2	4	1	Forgotten password		5/4/2012 12:00:00 AM	P	ccostap@etb.com	L	New	CL	QI
1	1	3	7	Virus		1/4/2014 12:00:00 AM	P	jjimenez@kindermusik.com	L	New	CL	QI

Figura 9. Ejemplo de ejecución de la aplicación de IcM generada con Morphosys
Fuente: elaboración propia

5. CONCLUSIONES

MDSD está ofreciendo un panorama prometedor para mejorar la eficiencia del desarrollo de software, pero, a su vez, carece en considerar aspectos por fuera de la funcionalidad del sistema. Queda demostrado en este trabajo que aparte de modelar la aplicación, también es posible y necesario modelar aspectos concernientes a la plataforma de ejecución para enriquecer el poder expresivo de los modelos.

El desarrollo de esta propuesta permite la reutilización de los diagramas UML de la fase de diseño de software, transformándolos en un insumo importante para expresar características de diseño arquitectónico por medio de un modelo consolidado de la plataforma que, además, puede ser construido de manera independiente por el usuario arquitecto, y guardado en un repositorio para reutilizarlo con diferentes modelos de aplicación. Estos diagramas se enlazan a través de un modelo intermedio de *weaving* de plataforma que permite expresar la manifestación de paquetes de la vista lógica a través de artefactos. Dichos artefactos se clasifican de una manera generalizada que guía la generación de modelo a texto según su multiplicidad, generación, la manera en que se despliegan o su tipo. En este último aspecto es importante considerar la definición de una ontología [21] que permita tener un lenguaje común para referirse a los diferentes artefactos que pueden conformar una plataforma.

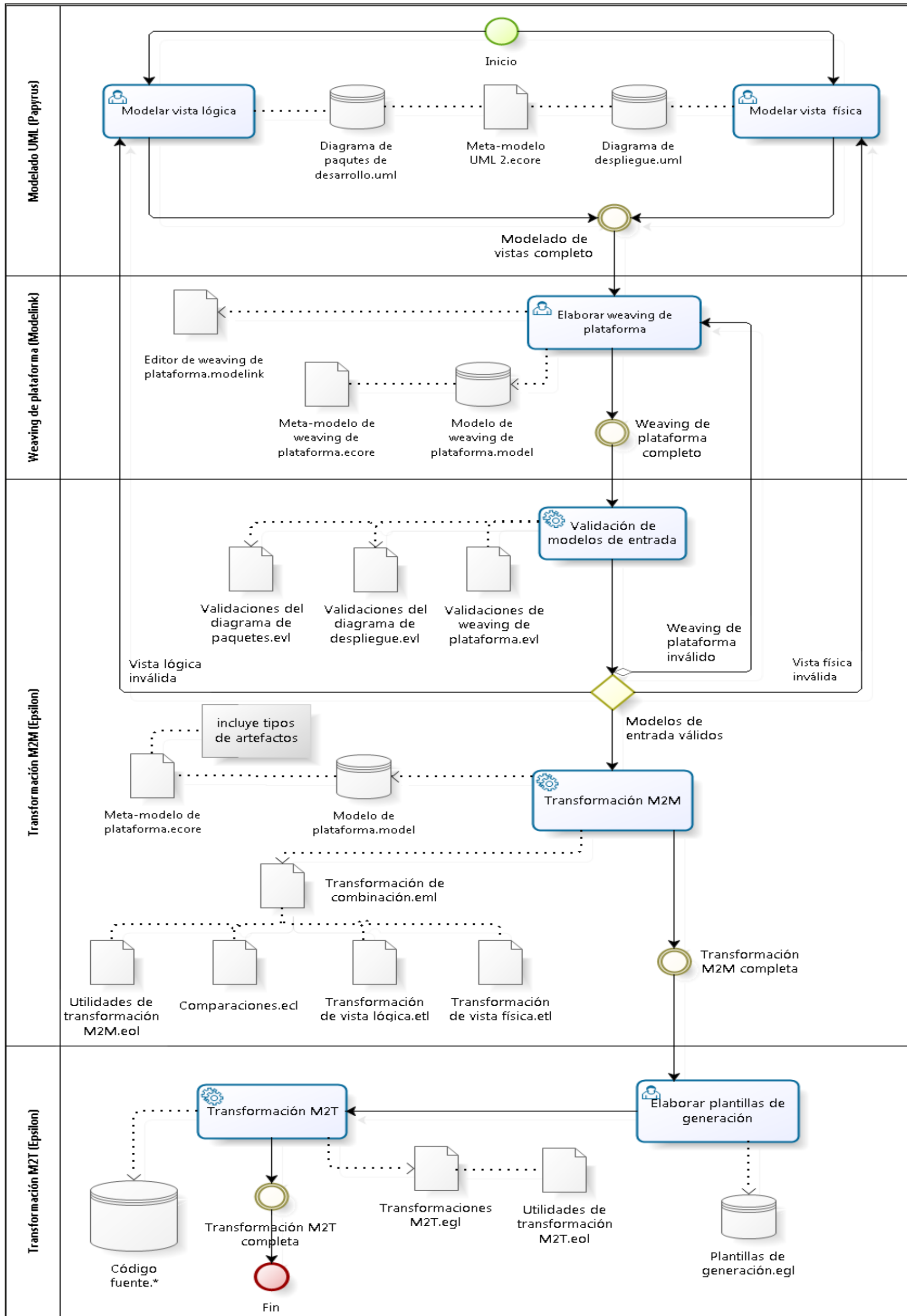


Figura 10. Proceso completo de modelado de plataforma
Fuente: elaboración propia

El nivel de detalle utilizado en los diagramas UML facilita la construcción del modelo de *weaving* de plataforma y, por consecuente, las transformaciones M2M y M2T. Esto se pudo evidenciar en el caso de los diagramas del patrón modelo vista presentador (MVP, por sus siglas en inglés) y .NET. Asimismo es importante resaltar que gran parte de la información semántica requerida para la ejecución reposa en el modelo de *weaving* de plataforma y las plantillas de generación, por lo que se puede concluir que la plataforma no solo está constituida por su modelo, sino también por las plantillas que son propias de cada tecnología.

Al modelar la plataforma por separado (con su propio metamodelo) se desarraigan los elementos de las vistas lógica y física de la herramienta de modelado específico, ya que los patrones de diseño se pueden generalizar y son totalmente independientes del modelado de la aplicación. Es por esto que se puede aprovechar la misma vista lógica para diferentes modelos de aplicación o, de manera recíproca, utilizar diferentes arquitecturas lógicas al mismo modelo de aplicación, lo que se traduce en ahorro de esfuerzos al modelar.

REFERENCIAS

[1] J. B. Quintero, J. A. Hincapié y R. Anaya, «Hacia enfoques multi-vistas en el desarrollo dirigido por modelos», 10 06 2015. [En línea]. Available: http://metafora.abcflex.net/. [Último acceso: 25 8 2015].

[2] O. Pastor, J. Fons y P. Vicente, «OOWS: A Method to Develop Web Applications from Web-Oriented Conceptual», *Department of Information Systems and Computation,* 2006.

[3] N. Koch y A. Kraus, «The Expressive Power of UML-based Web Engineering1,» *Ludwig-Maximilians-Universität München. Germany,* 2007.

[4] M. Brambilla, S. Comai, P. Fraternali y M. Matera, «Designing web applications with WebML and WebRatio», *Dipartimento di Elettronica e Informazione, Politecnico di Milano,* 2013.

[5] P. Cáceres, V. de Castro y E. Marcos, «Model Transformations for Hypertext modeling on Web Information Systems», *Rey Juan Carlos University,* 2006.

[6] E. Visser, «WebDSL: A Case Study in Domain-Specific Language Engineering», Delft University of Technology, 2008.

[7] J. Cadavid, D. López, J. Hincapie y J. Quintero, «DSL for generating Web application (MarTE/Quorra)», Archetypus Inc., EAFIT, 2009.

[8] S. Meliá y J. Gómez, «The WebSA approach: Applying model driven engineering to web applications», *Journal of Web Engineering,* vol. 5, N.° 2, pp. 121-149, 2006.

[9] M. Brambilla, J. Cabot and M. Wimmer, Model-driven Software Engineering in Practice, Morgan & Claypool, 2012.

[10] C. B. Reynoso, «Introducción a la arquitectura de software», Universidad de Buenos Aires, 2004.

[11] P. Kruchten, "Architectural Blueprints—The "4+1" View (Model of Software Architecture)", *IEEE Software 12,* 1995.

[12] FCGSS, *Applying 4+1 View Architecture with UML 2,* FCGSS, 2007.

[13] D. Ameller, X. Franch and J. Cabot, "Dealing with Non-Functional Requirements in Model-Driven Development", Report - Universitat Politècnica de Catalunya, 2010.

[14] «The Unified Modeling Language», 2009-2014. [En línea]. Available: http://www.uml-diagrams.org/. [Último acceso: 16 4 2013].

[15] P. &. P. T. Microsoft, Microsoft® Application Architecture Guide (Patterns & Practices), Microsoft Press, 2009.

[16] B. Sanders y C. Cumaranatunge, ActionScript 3.0 Design Patterns, O'REILLY, 2008.

[17] D. Kolovos, L. Rose , A. García y R. Paige, The Epsilon book, Epsilon, 2013.

[18] A. Jossic, M. Didonet Del Fabro, J.-P. Lerat, J. Bézivin y F. Jouault, «Model Integration with Model Weaving: a Case Study in System Architecture», de *International Conference on Systems Engineering and Modeling*, Haifa, 2007.

[19] R. A. Pottinger y P. A. Bernstein, «Merging Models Based on Given Cor-respondences», University of Washington,, 2003.

[20] C. Batini, S. Lenzerini y S. Navathe, «A Comparative Analysis of Methodologies for Database Schema Integration», *ACM Computing Surveys,* 1986.

[21] C. M. Zapata Jaramillo, G. L. Giraldo y G. A. Urrego Giraldo, «Las ontologías en la ingeniería de software: un acercamiento de dos grandes áreas del conocimiento», *Revista Ingenierías Universidad de Medellín,* vol. 9,N.° 16, pp. 91-99, julio 2011.