LENGUAJE ESPECÍFICO DE DOMINIO PARA LA DEFINICIÓN DE LA PLATAFORMA EN EL DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

JUAN CAMIO JIMENEZ DORADO
supercamilo@gmail.com

UNIVERSIDAD DE MEDELLÍN
FACULTAD DE INGENIERÍA
MAESTRÍA EN INGENIERÍA DE SOFTWARE
MEDELLÍN
2014

LENGUAJE ESPECÍFICO DE DOMINIO PARA LA DEFINICIÓN DE LA PLATAFORMA EN EL DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

JUAN CAMIO JIMENEZ DORADO

supercamilo@gmail.com

Trabajo de grado para optar al título de Magister en Ingeniería de Software

Asesor temático: Jesús Andrés Hincapié – Magíster en Ingeniería Informática

Asesor metodológico: Juan Bernardo Quintero – Magíster en Ingeniería Informática

UNIVERSIDAD DE MEDELLÍN

FACULTAD DE INGENIERÍA

MAESTRÍA EN INGENIERÍA DE SOFTWARE

MEDELLÍN

2014

CONTENIDO

LISTA DE ILUSTRACIONES

LISTA DE TABLAS

RESUMEN DE TRABAJO DE GRADO: LENGUAJE ESPECÍFICO DE DOMINIO PARA LA DEFINICIÓN DE LA PLATAFORMA EN EL DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

Autor: Juan Camilo Jiménez Dorado
Programa: Maestría en Ingeniería de Software,
Asesores: Juan Bernardo Quintero, Jesús Andrés Hincapié
Email: supercamilo@gmail.com

Universidad de Medellín.
Medellín
2014

Los enfoques tradicionales de Desarrollo de Software Dirigido por Modelos (MDSD) basados en la vista funcional han arrojado resultados positivos en los últimos años, no obstante, han presentado dificultades en aspectos como: el soporte multi-plataforma, la expresividad de los modelos y la intervención de las transformaciones. Este trabajo presenta una propuesta de modelado de la plataforma (vistas lógica y física de una aplicación), dentro de un enfoque multi-vistas para MDSD, de tal forma que se puedan expresar y reutilizar arquitecturas de software mediante el uso de modelos.

Lo anterior se logra a través de la elaboración de un Lenguaje Específico de Dominio (DSL) que hace parte del desarrollo de una herramienta de modelado MDSD ejemplificada a través de la aplicación de un estudio de caso del proceso gestión de incidentes de la Biblioteca de Infraestructura de Tecnologías de Información (ITIL).

Esta propuesta inicia con la identificación de los elementos propios de la vista lógica y física, incluyendo una clasificación detallada de los tipos de artefactos de código. De aquí se elicitan los requisitos con los que se construye el meta-modelo del DSL de plataforma y se procede a desarrollar los mecanismos para su instanciación.

Las vistas lógica y física se instancian a través de un modelador gráfico de Lenguaje de Modelado Unificado (UML), y luego se enlazan a través de manifestaciones que se definen en un editor de combinación de modelos (weaving, en inglés). Con los dos modelos UML más el modelo de weaving se definen varias reglas de Transformación de Modelo a Modelo (M2M), incluyendo operaciones de comparación, validación y combinación, que los traducen en una instancia del meta-modelo de plataforma. Por

último, se definen las reglas de Transformación de Modelo a Texto (M2T) que producen el código fuente. El proceso completo se puede apreciar en la siguiente ilustración:

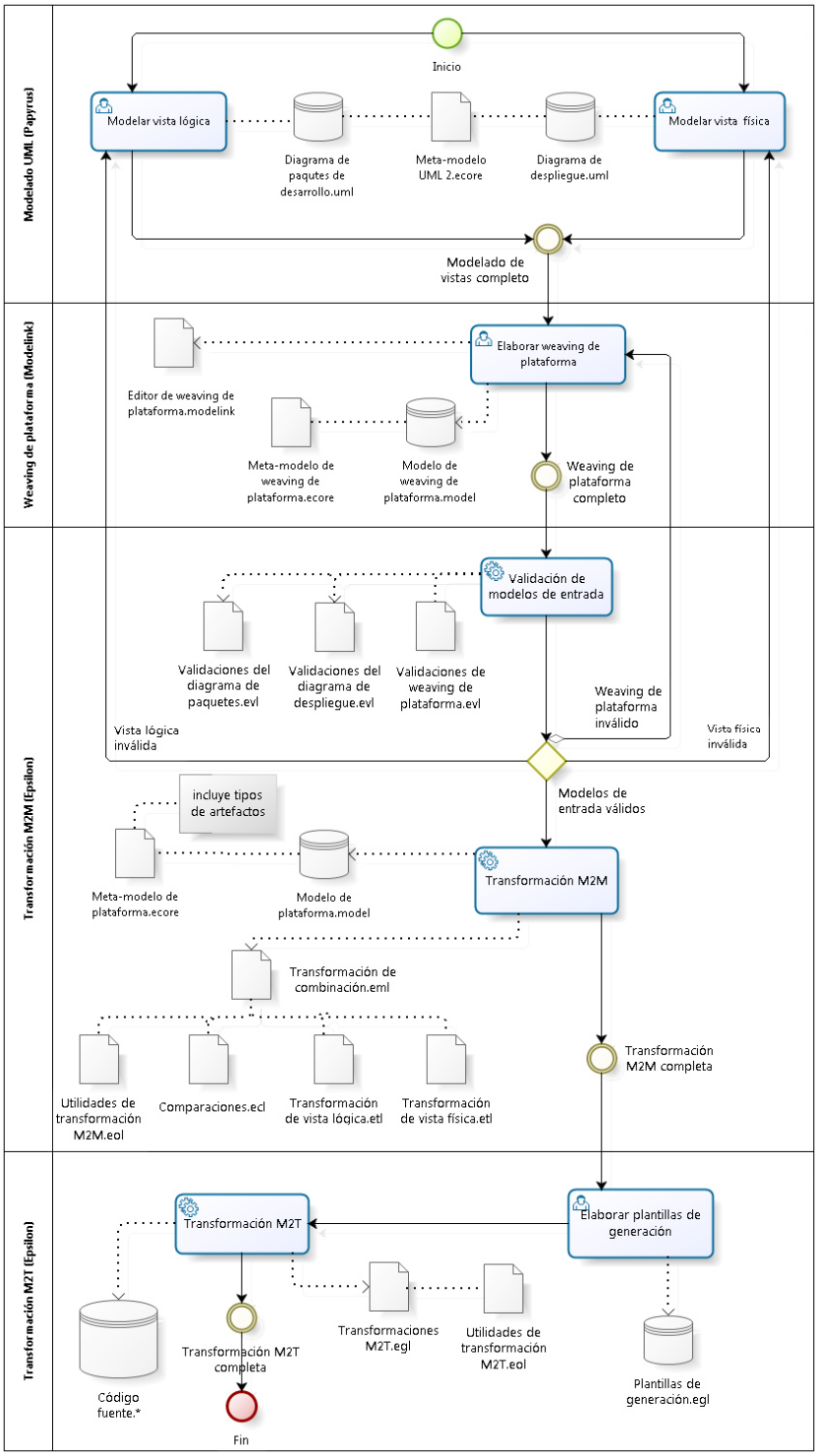


Ilustración 1: Proceso completo de modelado de plataforma en Morphosys

INTRODUCCIÓN

El Desarrollo de Software Dirigido por Modelos (MDSD) es una de las más recientes propuestas de la Ingeniería de Software que busca la generación semi-automática de software a partir del modelado de un sistema o parte de él.

Las aplicaciones en este campo han demostrado su viabilidad sin embargo se han quedado cortas en el producto final generado [1], en especial por los 3 problemas presentados en la Ilustración 1:

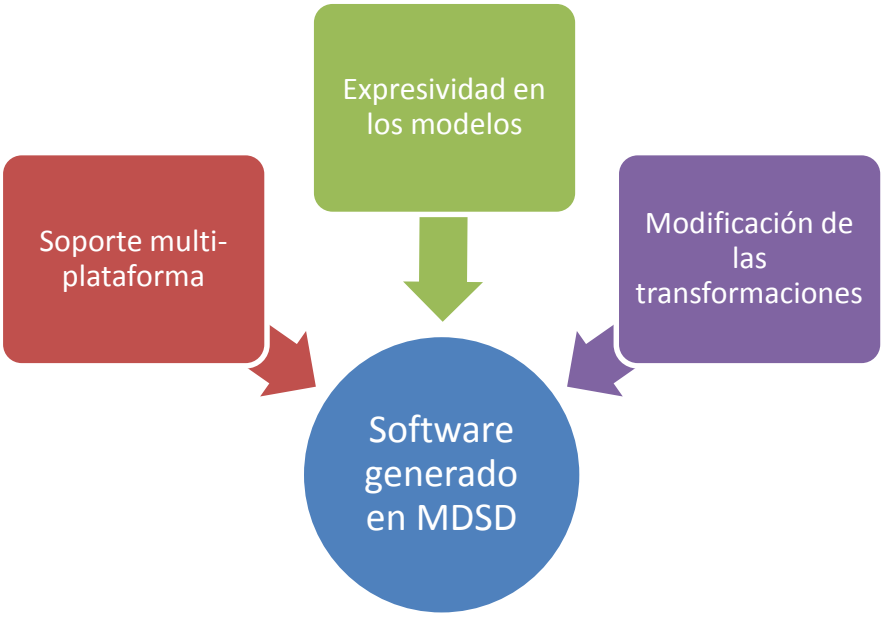


Ilustración 1: Principales problemas de MDSD [1]

Estas falencias se presentan por diversas limitaciones, y en mayor medida, porque la construcción de los modelos se basa exclusivamente en la vista funcional sin tener en cuenta que un sistema software (hardware + software) debe ser concebido desde varios puntos de vista [2] para abarcar otros aspectos, tales como: rendimiento, portabilidad, escalabilidad, etc. La mayoría de las aproximaciones MDSD existentes modelan sólo la vista funcional usando diagramas de clase como el estándar de facto [3], pero este tipo de enfoque no permite expresar los aspectos no funcionales, mientras que las soluciones que sí

tienen en cuenta estos aspectos, lo hacen enfocadas en uno solo de ellos y sin integración con la vista funcional [4].

Lo anterior sugiere que la inclusión de otras vistas, o sea la aplicación de un modelo multi-vistas ayuda a mitigar estas carencias y por consiguiente mejora el software generado en MDSD. Esta es la idea que motiva el proyecto <**Metáfora**> [5], un enfoque multi-vistas para MDSD, del que se derivan varias iniciativas, entre las cuales está el presente trabajo que se encarga del frente de **modelado de la plataforma** y parte de la **generación de código fuente (Transformación M2T),** que corresponde a la división por capas en múltiples archivos de código fuente y no aquellas características ligadas al modelo de aplicación**.** Estas actividades se capitalizan en el desarrollo de la herramienta <**Morphosys**> que hace parte del mismo proyecto.

Para abordar el desarrollo de la presente iniciativa, este documento se organiza de la siguiente manera: en el capítulo 1 se presentan el problema de la presente iniciativa junto con sus objetivos; en capítulo 2 se expone el marco referencial a modo de glosario para facilitar la comprensión de algunos conceptos al lector, luego en el capítulo 3 se describe a groso modo el proceso propuesto por metáfora y la delimitación del presente trabajo junto con la motivación para construir un DSL; en el capítulo 4 se relacionan los contenidos existentes en cuanto a enfoques MDSD y arquitectura multi-vistas; en el capítulo 5 se presenta el caso de estudio que se implementará en paralelo a modo de ilustración práctica del desarrollo del proyecto, junto con las tecnologías a utilizar en el desarrollo de Morphosys; en el capítulo 6 se describen las vistas arquitectónicas involucradas en el modelado de la plataforma y la responsabilidad de cada una dentro del proceso de desarrollo de software; en el capítulo 7 se explican la construcción del DSL en cuestión, detallando cada uno de sus componentes; en el capítulo 8 se trata la integración de las vistas del modelado de plataforma; luego en el capítulo 9 se muestra la generación del código fuente como paso final del proceso; y por último en el capítulo 10 se detallan las conclusiones y el trabajo futuro resultante de la presente iniciativa.

1. PROBLEMA, OBJETIVOS Y METODOLOGÍA

Esta iniciativa se encarga de desarrollar el flujo de modelado de la plataforma en MDSD que hace parte del proyecto Metáfora, el cual se describe en el siguiente capítulo junto con la delimitación del alcance del presente trabajo.

1.1. Preguntas de investigación

Es posible expresar los aspectos físicos de un sistema de software a través de un DSL?

La representación de la plataforma (vista lógica y fpisica) dentro de un enfoque multi-vistas ayuda a que MDSD soporte verdadera multi-plataforma?

1.2. Hipótesis

La aplicación de un DSL para la plataforma de un sistema (dentro de un enfoque arquitectónico multi-vistas) permite representar los aspectos lógicos y físicos del software generado en MDSD y por consiguiente aliviar la escasez de multi-plataformidad.

1.3. Justificación

A través de la historia del software, la lucha de la academia y la industria ha sido en pro de aumentar el nivel de abstracción para generar software de mejor calidad a un costo más bajo, a raíz de lo cual han surgido infinidad de métodos, patrones y herramientas en su escala evolutiva natural que han aportado enormemente para tal fin. El Desarrollo de Software Dirigido por Modelos sigue esta tendencia, escogiendo el concepto de modelo para lograr una abstracción a un nivel mucho más alto, lo que en esencia es la razón de que nuestra rama sea una verdadera Ingeniería y motiva mi participación en un tema tan prometedor y actual pero aún en etapa de madurez.

Como toda nueva propuesta tecnológica, el Desarrollo de software Dirigido por Modelos debe cumplir un ciclo hasta llegar a un punto de madurez suficientemente

robusto para su aceptación en la industria (ver Ilustración 2) y aunque este nuevo enfoque en primera instancia asuste y de la impresión de un acercamiento al "santo grial" del software, es tan valioso que vale la pena explorarlo y sumar a las iniciativas existentes, recogiendo sus experiencias para fortalecer sus falencias.

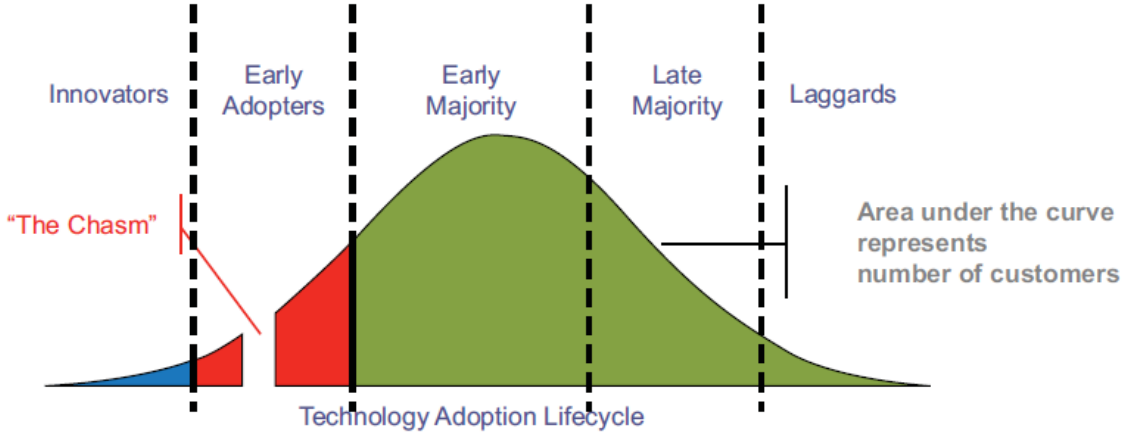


Ilustración 2: Curva de adopción de la tecnología [6]

Este es el caso de los estudios realizados recientemente por colegas [1] que identifican la carencia de soporte a los requerimientos no funcionales del sistema porque el modelado se enfoca exclusivamente en la vista funcional y dejan de lado los detalles de despliegue y el proyecto <**Metáfora**> que está en pleno desarrollo y surge como propuesta de solución a dicha carencia. La presente tesis busca aportar a dicho macro proyecto.

A lo largo de mi carrera he tenido un interés marcado en el meta-modelado y el Desarrollo de Software Dirigido por Modelos se muestra como candidato ideal para dejar de enfocarse en el árbol y pasar a mirar el bosque de los problemas a los que da solución la Ingeniería de Software.

No hay mayor realización para un aspirante al nivel de maestría que saber que su tesis hace una contribución real al campo investigativo, máxime cuando se trata de un tema de vanguardia y alto valor para la comunidad.

1.4. Objetivo General

Elaborar un Lenguaje Específico de Dominio para la representación de la plataforma en un enfoque arquitectónico multi-vistas enmarcado en el Desarrollo de Software Dirigido por Modelos.

1.5. Objetivos específicos

- Explorar los enfoques arquitectónicos y los avances en el Desarrollo de Software Dirigido por Modelos con respecto a los aspectos no funcionales.

- Caracterizar los componentes de un Lenguaje Específico de Dominio para el diseño arquitectónico.

- Desarrollar un Lenguaje Específico de Dominio para representar la plataforma en el Desarrollo de Software Dirigido por Modelos.

- Aplicar el Lenguaje Específico de Dominio de la plataforma en un caso de estudio de Desarrollo de Software Dirigido por Modelos.

- Verificar el soporte multiplataforma en el software generado usando Desarrollo Dirigido por Modelos cuando se representan las vistas física y lógica.

1.6. Metodología

OMG define dos posibilidades a la hora de definir lenguajes específicos de dominio, y que se corresponden con las dos situaciones mencionadas antes: o bien se define un nuevo lenguaje (alternativa de UML), o bien se extiende el propio UML, especializando algunos de sus conceptos y restringiendo otros, pero respetando la semántica original de los elementos de UML (clases, asociaciones, atributos, operaciones, transiciones, etc.) [7].

La primera opción es la adoptada por lenguajes como CWM [8], puesto que la semántica de algunos de sus constructores no coincide con la de UML. Para definir un nuevo lenguaje, sólo hay que describir su meta-modelo utilizando MOF. Por otro lado, hay situaciones en las que es suficiente con extender el lenguaje UML utilizando una serie de mecanismos recogidos en lo que se denominan Perfiles UML (en inglés, UML Profiles) [7].

Cada una de estas dos alternativas presenta ventajas e inconvenientes. Así, definir un nuevo lenguaje ad-hoc permite mayor grado de expresividad y correspondencia con los conceptos del dominio de aplicación particular. Sin embargo, y aun cuando esos nuevos lenguajes ad-hoc se describan con MOF, el hecho de no respetar el meta-modelo estándar de UML va a impedir que las herramientas UML existentes en el mercado puedan manejar sus conceptos de una forma natural [7].

Dada la necesidad de incrementar el poder expresivo de los modelos, se decide realizar un nuevo lenguaje y no utilizar un perfil UML. A modo de ilustración del proceso de construcción del DSL de esta tesis, se puede utilizar el modelo M3 de MOF (Ilustración 3):

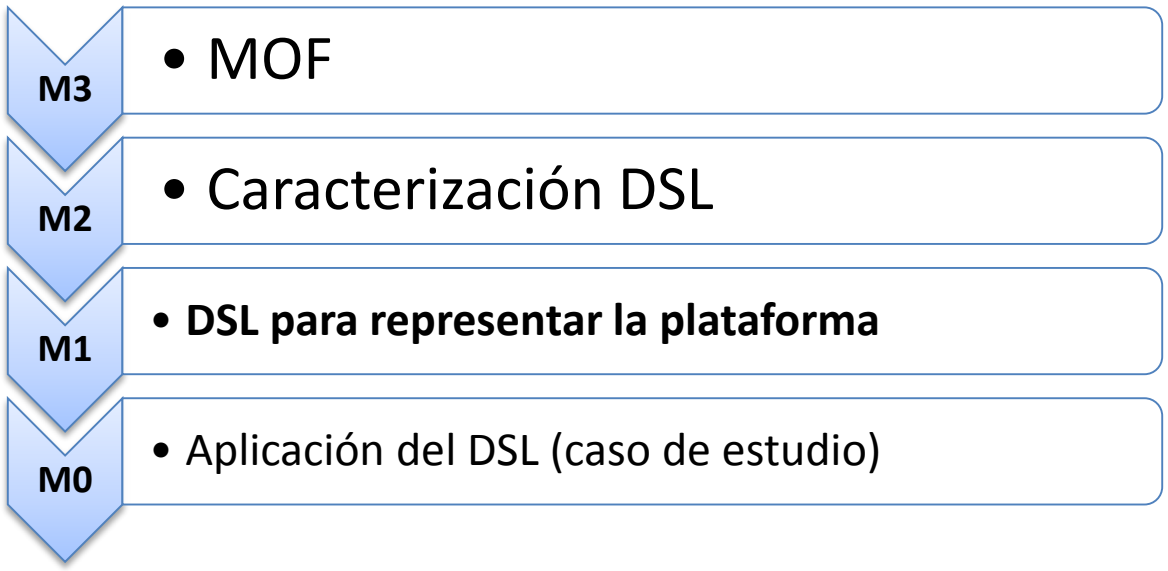


Ilustración 3: Modelo M3 del DSL para modelar la plataforma

La Tabla 1 muestra los objetivos desarrollados en el presente trabajo y cómo se dio cumplimiento a cada uno de ellos:

Objetivo	Técnicas	Productos
Explorar los enfoques arquitectónicos y los avances en el Desarrollo de Software Dirigido por Modelos con respecto a los aspectos no funcionales.	Revisión de literatura [9]	Marco conceptual sintetizado sobre modelos y MDSD presentado junto con la propuesta de trabajo de grado. Estado del arte de MDSD con respecto a los aspectos no funcionales y sus diferentes enfoques.
Caracterizar los componentes de un Lenguaje Específico de Dominio para el diseño arquitectónico.	Revisión de literatura [9] Desarrollo de la presente iniciativa	Articulo – “MDSD Multiplataforma más allá de la vista funcional”
Desarrollar un Lenguaje Específico de Dominio para representar la plataforma en el Desarrollo de Software Dirigido por Modelos.	Diseño del lenguaje Documentación del Lenguaje	DSL Sintaxis abstracta Sintaxis concreta Semántica Código fuente de Morphosys implementado en Eclipse + Epsilon + Papyrus (incluyendo combinación de modelos, transformaciones M2M y M2T y plantillas de generación)

Aplicar el Lenguaje Específico de Dominio de la plataforma en un caso de estudio de Desarrollo de Software Dirigido por Modelos.	Simulación	Caso de estudio funcional implementado en Eclipse + Epsilon + Papyrus
Verificar el soporte multiplataforma en el software generado usando Desarrollo Dirigido por Modelos cuando se representan las vistas física y lógica.	Pruebas	Ejecución y demostración del caso de estudio

Tabla 1: Diseño Metodológico

2. MARCO REFERENCIAL

2.1. Modelos

2.1.1. Modelo

Un modelo es una descripción de un sistema (o parte de él) escrita en un lenguaje bien definido, que está normalmente presentado como una combinación entre dibujos y texto [10].

Los modelos son un elemento clave en la ingeniería (en general), dónde se usan para analizar y diseñar sistemas complejos como una abstracción de un sistema y su ambiente, ya que permiten simular el sistema antes de llevarlo a la realidad [11], lo que tiende a significar ahorros en costo y tiempo, y aumento en la calidad.

En la ingeniería de software en particular, los modelos no son nada nuevo, se han usado en las últimas décadas en números análisis y métodos de diseño, cada uno con sus propios enfoques y notaciones de modelado [11].

2.1.2. Meta-modelo

Así como los modelos son un mecanismo para describir sistemas, también se hace necesario un mecanismo para describir los modelos, a esto se le llama meta-modelo, que no es más que el lenguaje para expresar un modelo [12]. En una manera abstracta, define los conductos de un lenguaje de modelado y sus relaciones, así como las restricciones y las reglas pero no la sintaxis concreta del lenguaje sino una sintaxis abstracta [11]. En una definición más simple, un meta-modelo es un modelo de modelos [10].

2.1.3. Meta-meta-modelo

Continuando con la analogía meta-modelo -> modelo -> sistema, se hace necesario agregar un cuarto nivel para definir las pautas sobre las cual se construyen los meta-modelos al nivel de abstracción más alto, en otras palabras, un meta-meta-modelo.

2.1.4. Meta-Object Facility (MOF)

Meta-Object Facility es una especificación del Object Management Group (OMG) para la definición y gestión de modelos que proporciona un lenguaje con los constructores y mecanismos mínimos necesarios para describir meta-modelos de lenguajes de modelado. El modelo M3 de MOF es un meta-meta modelo que se define a sí mismo y se expresa en 3 niveles de la siguiente manera [7] (Ilustración 4):

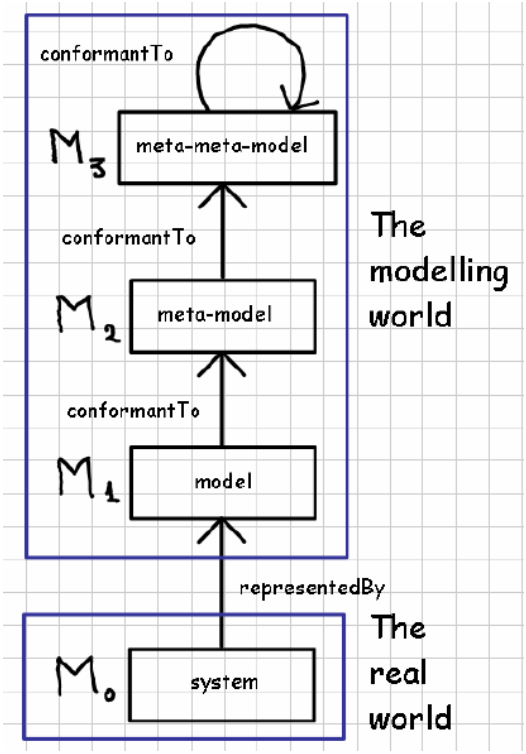


Ilustración 4: Modelo de 3 niveles de MOF

Nivel M0 (instancias): modela el sistema real, y sus elementos son las instancias que componen dicho sistema. Ejemplos de dichos elementos son el cliente “Juan” que vive en “Paseo de la Castellana, Madrid” y ha comprado el ejemplar número “123” del libro “Ulises”.

Nivel M1 (modelo del sistema): modelos de los sistemas concretos. En el nivel M1 es donde se definen, por ejemplo, los conceptos de “Cliente”, “Compra” y

“Libro”, cada uno con sus correspondientes atributos (“dirección”, “numero de ejemplar”, “título”, etc.). Existe una relación muy estrecha entre los niveles M0 y M1: los conceptos del nivel M1 definen las

Clasificaciones de los elementos del nivel M0, mientras que los elementos del nivel M0 son las instancias de los elementos del nivel M1.

Nivel M2 (meta-modelo): lenguajes de modelado. El nivel M2 define los elementos que intervienen a la hora de definir un modelo del nivel M1. En el caso de un modelo UML de un sistema, los conceptos propios del nivel M2 son “Clase”, “Atributo”, o “Asociación”. Al igual que entre los niveles M0 y M1, aquí también existe una gran relación entre los conceptos de los niveles M1 y M2: los elementos del nivel superior definen las clases de elementos válidos en un determinado modelo de nivel M1, mientras que los elementos del nivel M1 pueden ser considerados como instancias de los elementos del nivel M2.

Nivel M3 (meta-meta-modelo): elementos que constituyen los distintos lenguajes de modelado. De esta forma, el concepto de “clase” definido en UML (que pertenece al nivel M2) puede verse como una instancia del correspondiente elemento del nivel M3, en donde se define de forma precisa ese concepto, así como sus características y las relaciones con otros elementos (por ejemplo, una clase es un clasificador, y por tanto puede tener asociado un comportamiento, y además dispone de un conjunto de atributos y de operaciones).

2.1.5. Transformación de Modelos

Los modelos pueden relacionarse entre sí a través de transformaciones. Estas se definen a nivel de meta-modelo y se aplican a nivel de modelo que se ajuste a dicho meta-modelo, en otras palabras, la transformación se ejecuta entre dos modelos pero se define de acuerdo a sus respectivos meta-modelos [6].

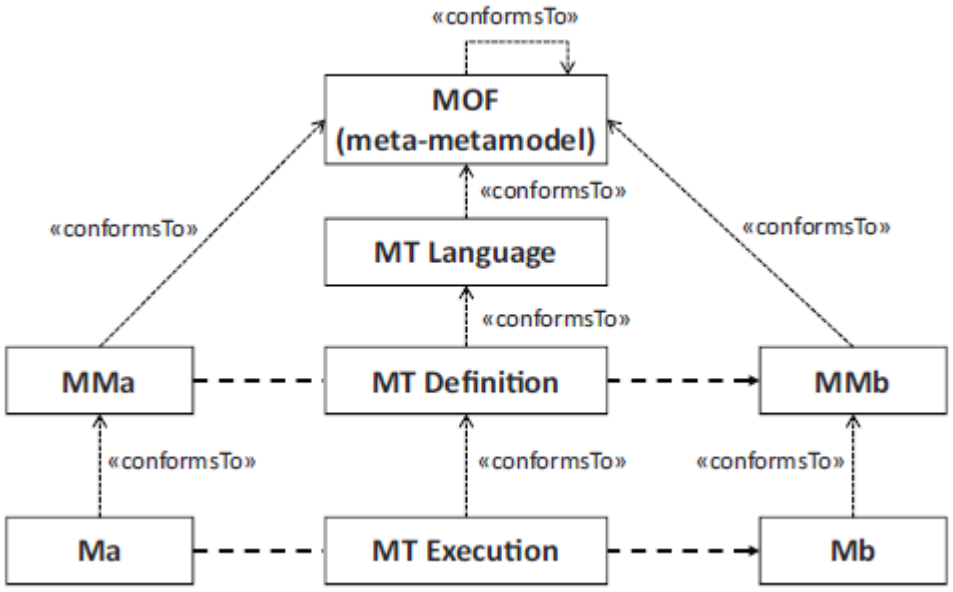


Ilustración 5: Definición y rol de las transformaciones entre modelos [6].

Los modelos necesitan ser fusionados, alineados, refactorizados, refinados y traducidos a otros lenguajes o representaciones. Todas estas operaciones son implementadas como transformaciones de modelos [13].

Al hablar de transformaciones de modelos se distinguen dos tipos principales: de modelo a modelo y de modelo a texto.

2.1.5.1. Transformaciones de Modelo a Modelo (M2M)

Una transformación de modelo a modelo (en inglés Model to Model, M2M) es un programa que recibe uno o más modelos como entrada y devuelve uno o más modelos como salida, dando como resultado en la práctica transformaciones: uno a uno, uno a muchos, muchos a uno o aún muchos a muchos [6].

Las transformaciones M2M también se clasifican en exógenas, cuando los modelos están definidos en distintos lenguaje o endógenas cuando lo están en el mismo lenguaje.

2.1.5.2. Transformaciones de Modelo a Texto (M2T)

Una transformación de modelo a texto (en inglés Model to Text, M2T) es un programa que recibe uno o más modelos para automatizar la generación de un modelo representado como texto.

Las transformaciones M2T se han utilizado ampliamente en la Ingeniería de Software para la generación de documentación, listas de tareas y, por supuesto, el código fuente que ha sido el principal artefacto de esta disciplina durante los últimos tiempos y que se acompaña de otros tipos de código como los casos de prueba, los scripts de despliegue, etc.

2.1.6. Modelos en el desarrollo de software

Los modelos han jugado un rol muy importante en el desarrollo de software y tienen estrecha relación con el código fuente en diferentes enfoques de la Ingeniería de Software (Ilustración 6):

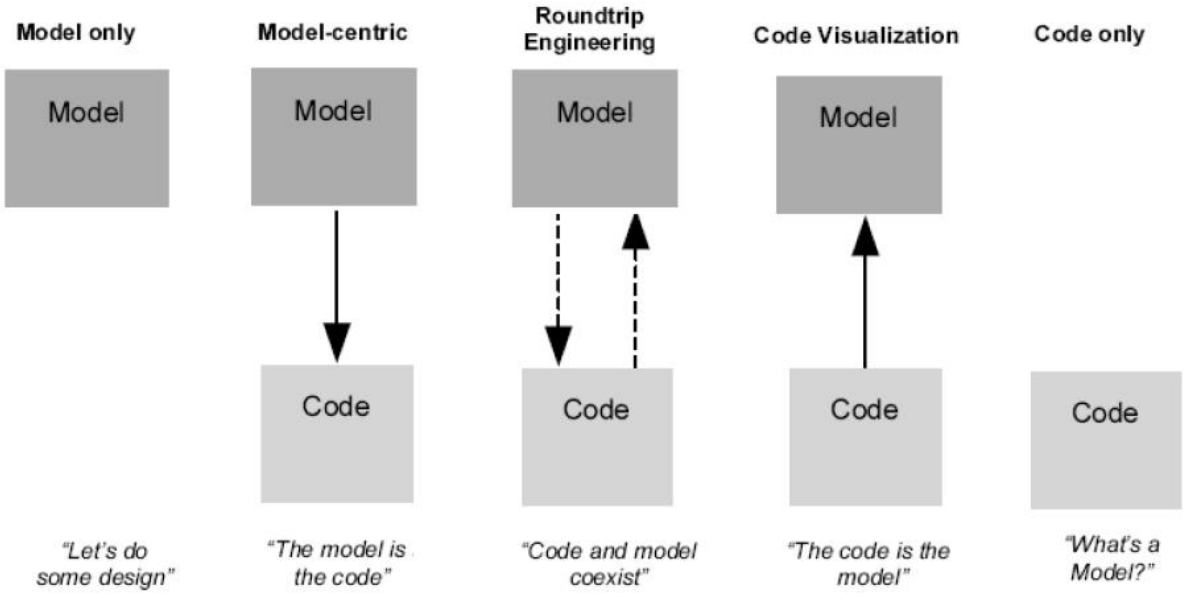


Ilustración 6: Modelos vs código [14]

2.2. Lenguajes de Modelado

Para construir modelos es necesario expresarlos de alguna manera. Los lenguajes de modelado permiten plasmar la información o el conocimiento de un sistema o estructura definida por un conjunto de reglas consistentes. Estas reglas son usadas para interpretar el significado de los componentes de dicha estructura. Los lenguajes de modelado son herramientas conceptuales para que los diseñadores formalicen sus pensamientos y conceptualicen la realidad de forma explícita [6].

2.2.1. Componentes de un Lenguaje de Modelado

Un lenguaje de modelado está compuesto por tres elementos: sintaxis abstracta, sintaxis concreta y semántica (Ilustración 7).

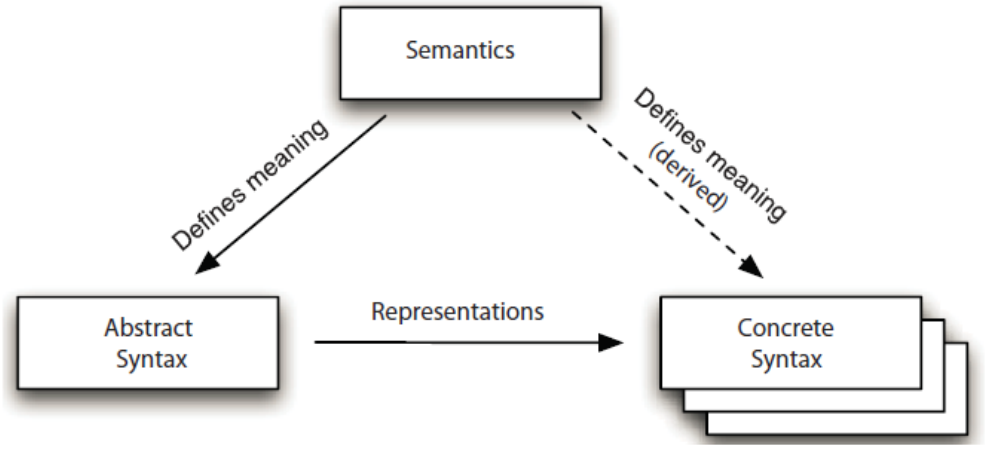


Ilustración 7: Componentes de un Lenguaje de Modelado [6].

2.2.1.1. Sintaxis abstracta (meta-modelo)

La sintaxis abstracta describe la estructura del lenguaje y la forma en que sus elementos se combinan, independientemente de una representación particular [6].

Para representar la sintaxis abstracta de un lenguaje de modelado se usa un meta-modelo que sigue el estándar MOF.

2.2.1.2. Sintaxis concreta (perfil)

La sintaxis concreta describe una representación específica de un lenguaje de modelado, cubriendo aspectos visuales o de codificación. Una sintaxis concreta puede ser gráfica o textual y sirve como referencia para el diseñador para sus actividades de modelado.

La sintaxis concreta es parte esencial de un lenguaje de modelado porque está directamente ligada con la expresividad y complejidad de los modelos. A la hora de construir una sintaxis concreta se deben tener en cuenta los siguientes aspectos: capacidad de escritura, capacidad de lectura, facilidad de aprendizaje y efectividad [15].

2.2.1.3. Semántica

La semántica es la interpretación de un lenguaje, lo que le da sentido a la sintaxis concreta construida a partir de la sintaxis abstracta.

2.2.2. Lenguaje de Propósito General (GPL)

Existen varios tipos de lenguajes de modelado, a saber: lenguajes visuales, lenguajes textuales o de tipo más específico. Estos últimos son los usados en las ciencias de la computación y entre ellos encontramos los lenguajes de propósito general y los lenguajes específicos de dominio.

Los Lenguajes de Propósito General (en inglés General Purpose Language, GPL) son lenguajes que pueden ser aplicados a cualquier dominio y no necesariamente a uno en particular.

2.2.3. Lenguaje de Modelado Unificado (UML)

El Lenguaje de Modelado Unificado (en inglés Unified Modeling Language, UML) es un GPL controlado por OMG que constituye una familia de notaciones gráficas, soportadas en el mismo meta-modelo, que ayudan en la descripción y diseño de sistemas de software. UML tiene estrecha relación con la orientación a objetos y

se ha convertido en el estándar de facto para el diseño de software alrededor de todo el mundo [16].

UML nació en 1997 como un compendio de lenguajes de modelado orientados a objetos utilizados en las décadas de los 80s y 90s y ha ido evolucionando hasta su última versión oficial 2.4.1 en 2011, sin contar con la versión 2.5 liberada en 2012 que todavía se cataloga como "en proceso". La Ilustración 8 muestra la evolución de UML hasta su versión base 2.0 en la cual se soporta MDA:

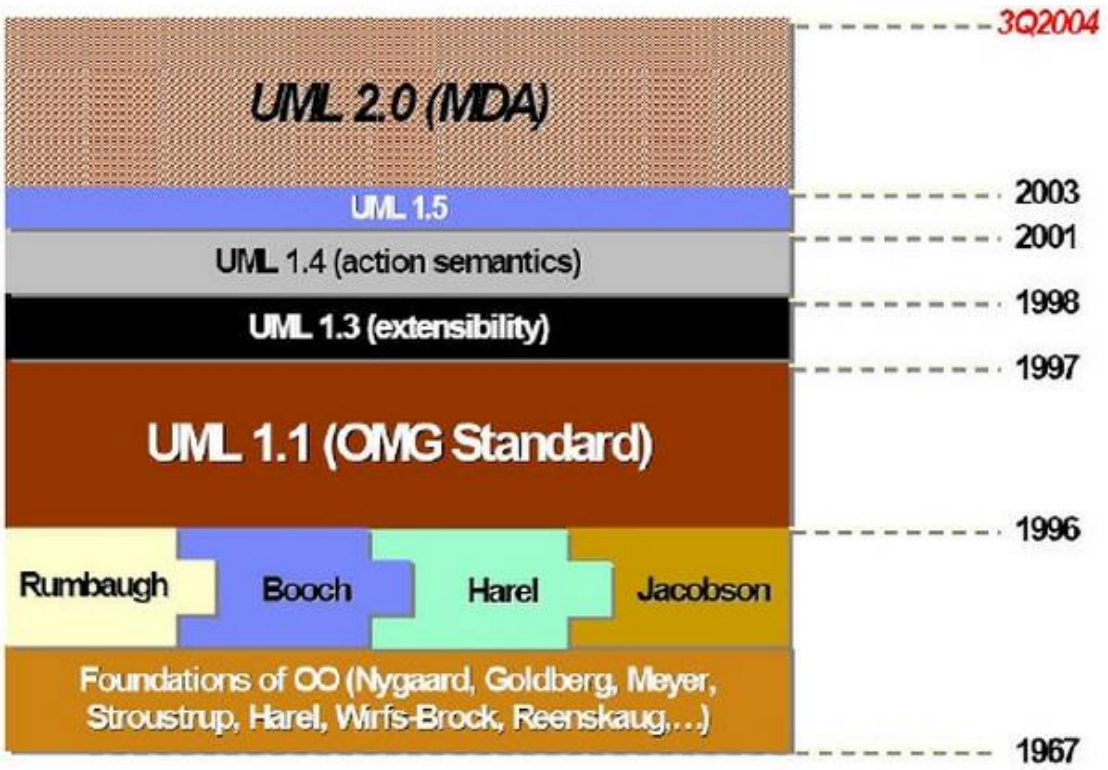


Ilustración 8: Evolución de UML

UML ha sido ampliamente utilizado para modelar sistemas de información y en los últimos años está retomando el protagonismo gracias a la Ingeniería Dirigida por Modelos, ya que es el lenguaje preferido por los enfoques más populares de esta rama.

2.2.3.1. Diagramas UML

UML define distintos tipos de diagramas para distintos tipos de escenarios y los clasifica en 3 grupos: estructurales, de comportamiento y de interacción.

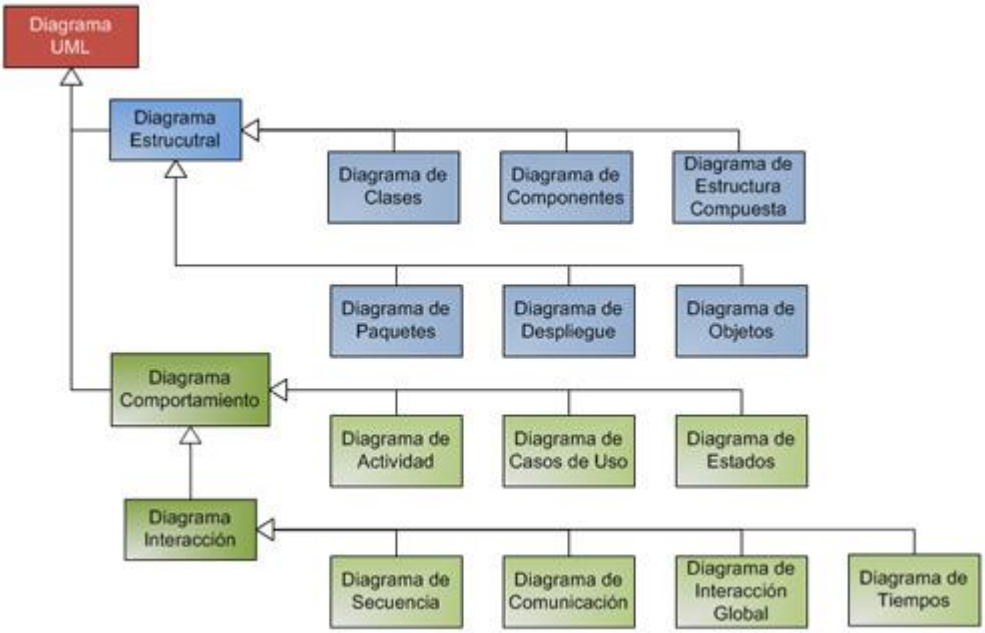


Ilustración 9: Clasificación de los diagramas UML [17]

2.2.3.2. Perfiles UML

El hecho de que UML sea un lenguaje de propósito general proporciona una gran flexibilidad y expresividad a la hora de modelar sistemas. Sin embargo, hay numerosas ocasiones en las que es mejor contar con algún lenguaje más específico para modelar y representar los conceptos de ciertos dominios particulares. Esto sucede, por ejemplo, cuando la sintaxis o la semántica de UML no permiten expresar los conceptos específicos del dominio, o cuando se desea restringir y especializar los constructores propios de UML, que suelen ser demasiado genéricos y numerosos [7].

Los Perfiles UML constituyen el mecanismo que proporciona el propio UML para extender su sintaxis y su semántica para expresar los conceptos específicos de un determinado dominio de aplicación [7].

Un perfil se define en un paquete UML, estereotipado «profile» (Ilustración 10), que extiende a un meta-modelo o a otro Perfil. Tres son los mecanismos que se utilizan para definir perfiles: estereotipos (en inglés, stereotypes), restricciones (en inglés, constraints), y valores etiquetados (en inglés, tagged values).

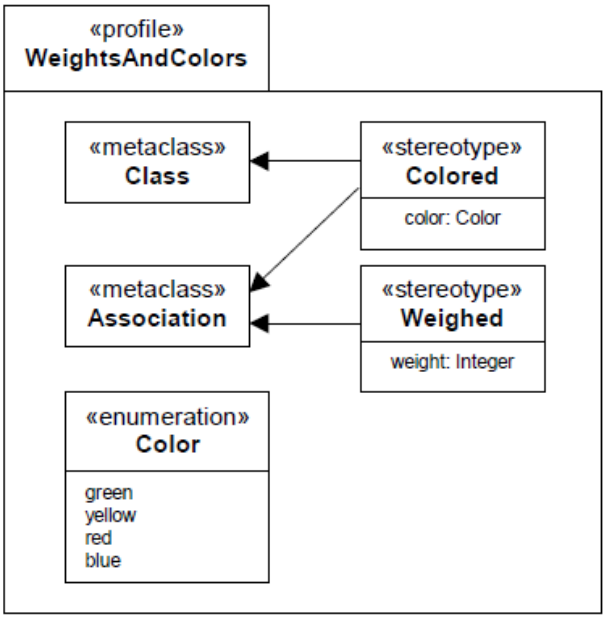


Ilustración 10: Ejemplo de perfiles UML [7]

2.2.4. XML para Intercambio de Meta-datos (XMI)

XML para Intercambio de Meta-datos (en inglés XML Meta-data Interchange, XMI) es una especificación de OMG definida con el objetivo de facilitar el intercambio de modelos entre herramientas de modelado y repositorios de meta datos basados en dicha especificación [18]. XMI es utilizad por los desarrolladores de software para el intercambio de modelos UML.

XMI está basada en XML (en inglés Extensible Markup Language) que es un GPL que permite definir la gramática de lenguajes específicos. La Ilustración 11 muestra la estructura de un archivo XMI.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <XMI xmi.version="1.2" xmlns:UML="org.omg.xmi.namespace.UML" timestamp="Sat May 27 16:17:22 COT 2006">
  - <XMI.header>
    - <XMI.documentation>
        <XMI.exporter>Netbeans XMI Writer</XMI.exporter>
        <XMI.exporterVersion>1.0</XMI.exporterVersion>
      </XMI.documentation>
    </XMI.header>
  - <XMI.content>
    - <UML:Model xmi.id="sm$-1784e56e:10b5d04bb81:-7e8a" name="Agenda" isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false">
      - <UML:Namespace.ownedElement>
        + <UML:Class xmi.id="sm$-1784e56e:10b5d04bb81:-7ff2" name="Proyecto" visibility="public" isSpecification="false" isRoot="false" isLeaf="false"
            isAbstract="false" isActive="false">
        + <UML:Class xmi.id="sm$-1784e56e:10b5d04bb81:-7fae" name="Persona" visibility="public" isSpecification="false" isRoot="false" isLeaf="false"
            isAbstract="false" isActive="false">
        + <UML:Class xmi.id="sm$-1784e56e:10b5d04bb81:-7f6c" name="Actividad" visibility="public" isSpecification="false" isRoot="false" isLeaf="false"
            isAbstract="false" isActive="false">
        + <UML:Class xmi.id="sm$-1784e56e:10b5d04bb81:-7f29" name="Participacion" visibility="public" isSpecification="false" isRoot="false" isLeaf="fal
            isAbstract="false" isActive="false">
        + <UML:Package xmi.id="sm$-1784e56e:10b5d04bb81:-7e89" name="java" isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false">
        + <UML:Association xmi.id="sm$-1784e56e:10b5d04bb81:-7ed8" isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false">
        + <UML:Association xmi.id="sm$-1784e56e:10b5d04bb81:-7ebe" isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false">
        + <UML:Association xmi.id="sm$-1784e56e:10b5d04bb81:-7ea6" isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false">
        + <UML:Stereotype xmi.id="sm$10cc53e:10b61644bcf:-7ff0" name="entity" visibility="public" isSpecification="false" isRoot="false" isLeaf="false"
            isAbstract="false">
        </UML:Namespace.ownedElement>
      </UML:Model>
    </XMI.content>
  </XMI>
```

Ilustración 11: Ejemplo de archivo XMI

2.2.5. Lenguaje Específico de Dominio (DSL)

Un Lenguaje Específico de Dominio (en inglés Domain-Specific Language, DSL), como su nombre lo indica, está ideado para satisfacer las necesidades de un dominio específico, lo que los hace más simples en su enfoque. Dentro de los DSLs más populares en la Ingeniería de Software encontramos el Lenguaje de Consulta Estructurado (SQL) y el Lenguaje de Marcado Hipertextual (HTML) que definen elementos y reglas para modelar bases de datos y páginas web respectivamente.

Se distingue también un tipo específico que es el Lenguaje Específico de Modelado de Dominio (en inglés Domain Specific Modeling Language, DSML) que es un DSL que se representa a través de modelos. Sin embargo, los términos DSL y DSML son utilizados indistintamente dentro del mundo de la Ingeniería Basada en Modelos [1].

[1] La Ingeniería Basada en Modelos es una rama de la Ingeniería del Software que se describe más adelante en este marco conceptual.

Al tratarse de un lenguaje de modelado, un DSL tiene los mismos 3 componentes (Ilustración 12):

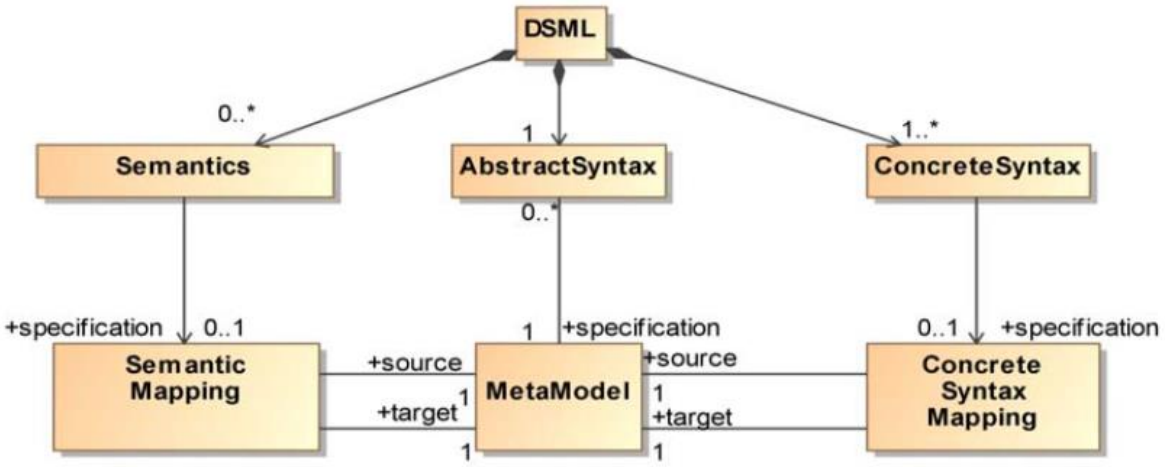


Ilustración 12: Componentes de un DSL

Principios de un DSL [6]

- Proveer buenas abstracciones para el desarrollador, ser simple, intuitivo y facilitar el trabajo.
- No depender de la experticia de una sola persona para su adopción y uso.
- Debe evolucionar y mantenerse actualizado respecto a las necesidades del usuario y el contexto.
- Debe acompañarse de herramientas y métodos para maximizar la productividad del los expertos en el dominio.
- Estar abierot a extensiones pero cerrado a modificaciones.

Los DSLs se pueden clasificar de la siguiente manera (Ilustración 13) [6]:

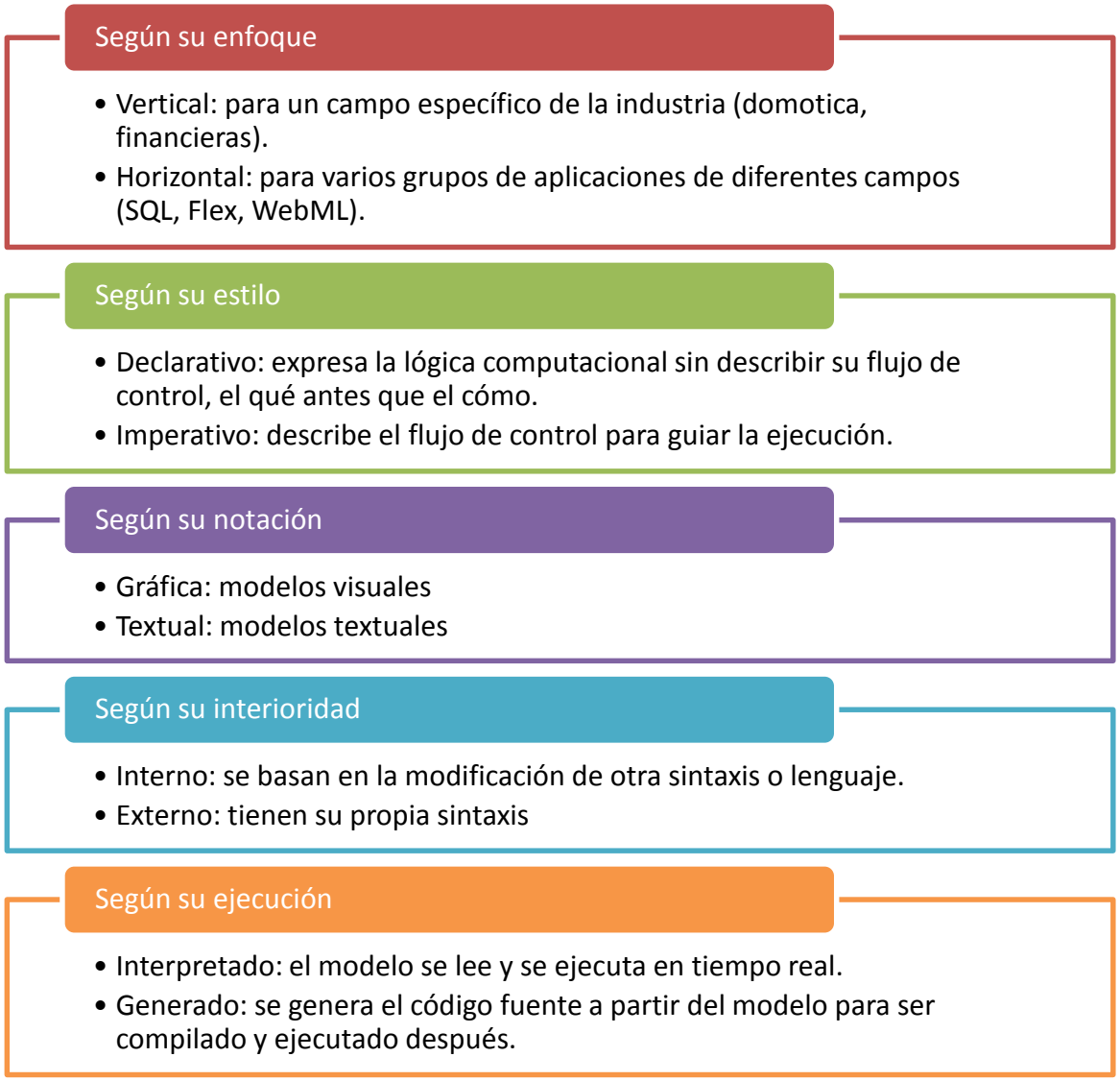


Ilustración 13: Clasificación de los DSLs.

2.2.6. Lenguaje de Restricción de Objetos (OCL)

Cómo los lenguajes de meta-modelado deben mantener su simplicidad para reducir la curva de aprendizaje, se limitan a la definición de elementos básicos, es aquí donde se hace necesario un lenguaje para definir restricciones.

El Lenguaje de Restricción de Objetos (en inglés Object Contraint Language, OCL) es un GPL textual adoptado como un estándar por la OMG [19] de naturaleza tipada, declarativa y libre de efectos secundarios. OCL tiene sus propios tipos que se rigen por reglas y operaciones específicas que pueden consultar o restringir el comportamiento de un sistema pero no lo modifican.

OCL se usa para complementar los meta-modelos basados en MOF con un conjunto de reglas textuales, también conocidas como reglas “bien formadas” [6] que tienen una estructura específica que se debe cumplir y que expresan condiciones que los elementos del meta-modelo no pueden expresar. Las restricciones se asocian típicamente reglas de negocio de un sistema.

2.3. Ingeniería Dirigida por Modelos (MDE)

La Ingeniería Dirigida por Modelos (en inglés Model-Driven Engineering, MDE) es el término con el que la se conoce en la comunidad a uno de las más recientes propuestas de la Ingeniería de Software en el que los modelos son considerados el elemento central del proceso de desarrollo y que define los mecanismos para utilizarlos en automatización tareas, tales como:

- Generación de código
- Refinamiento
- Refactoría
- Traducción a otros lenguajes o plataformas
- Análisis y Diseño
- Configuración
- …

MDE es un subconjunto de la Ingeniería Basada en Modelos (en inglés Model-Based Engineering, MBE), con la diferencia de que en MBE, generalmente hablando, los modelos cumplen un rol importante pero secundario, normalmente son usados para la fase de diseño y como artefactos de documentación, mientras que en MDE los modelos son el principal insumo para generar el código fuente de la aplicación [11].

Existe una metáfora que resume el objeto de MDE (Ilustración 14) donde se compara la construcción de una casa con la construcción de software [20]:

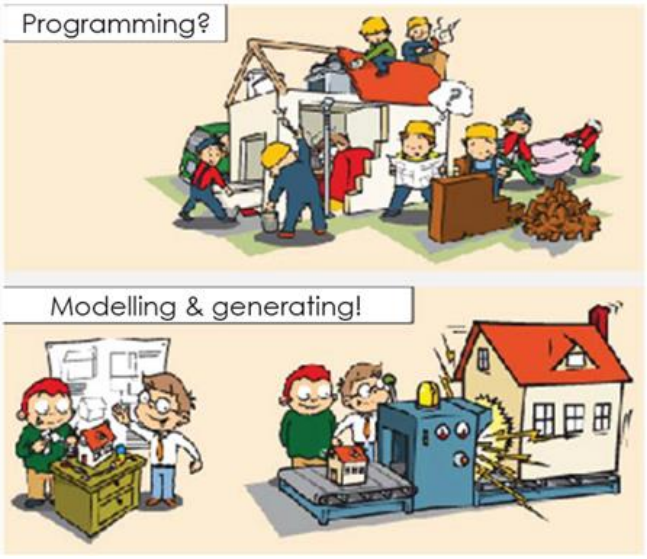


Ilustración 14: Una metáfora para MDE [20]

La ilustración muestra como en la programación tradicional (parte superior), alguien diseña y le da las instrucciones a los demás para construirla, mientras que en MDE (parte inferior) se introduce el modelo en un generador automático.

Principios de MDE

- Un paradigma que modela para MDE se considera eficaz si sus modelos tienen sentido desde el punto de vista del usuario y pueden servir como base para poner sistemas en ejecución.

- Los modelos se desarrollan con una exhaustiva comunicación entre encargados de producto, diseñadores, y miembros del equipo del desarrollo.

- Los modelos se definen a varios niveles manteniendo la coherencia entre los puntos de vista de los diferentes implicados en él.

Retos de MDE

A pesar de que MDE se presenta como la siguiente evolución natural de la Ingeniería de Software, como con cualquier nueva propuesta tiene sus detractores y se enfrenta a una serie de retos importantes que son expuestos por Martin Fowler en su libro UML Distilled [16]:

- El verdadero objetivo de la Ingeniería de Software es aumentar la productividad, reducir los errores y acotar el código. Mientras MDE no produzca estos efectos a corto plazo siempre va a ser sujeto de sospechas.

- MDE es percibido como el uso de modelos para generar código fuente pero se debe concebir mejor como un nivel de abstracción para todas las tareas del proceso de desarrollo de software y no sólo el código.

- Desde el punto de vista de los programadores, los modelos aún son vistos como simples dibujos para documentar el sistema.

- Los modelos son percibidos como artefactos de poco uso que nadie aprecia, incluyendo a los usuarios que solo esperan que el software se ejecute y funcione bien. En otras palabras, los modelos y las aplicaciones son vistos como competidores.

Acrónimos relacionados con MDE

Se puede decir que MBE es una disciplina que contiene a MDE y que a su vez deriva en enfoques más específicos como MDD y MDA [6], pero algunos autores se refieren al tema en general como MD* para alivianar la confusión. La Ilustración 15 aclara la relación entre los diferentes acrónimos.

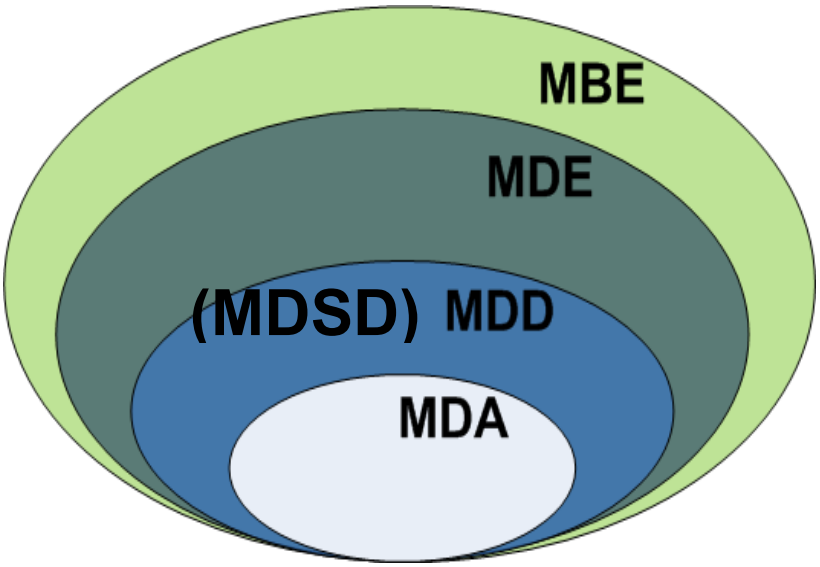


Ilustración 15: La jungla de acronimos MD* [6]

MDD y MDA son acrónimos registrados por el OMG para referirse a los estándares propuestos por esta organización, mientras que otras industrias, firmas de consultoría y universidades se refieren a MDE y MDSD como términos más precisos para referirse al tema [21].

2.3.1. Desarrollo de Software Dirigido por Modelos (MDSD)

El Desarrollo de Software Dirigido por Modelos (en inglés Model-Driven Software Development, MDSD), también conocido como Desarrollo Dirigido por Modelos (en inglés Model-Driven Development, MDD) es un nuevo paradigma de desarrollo de software que propone la generación semi-automática de software a partir del modelado de un sistema o parte de él.

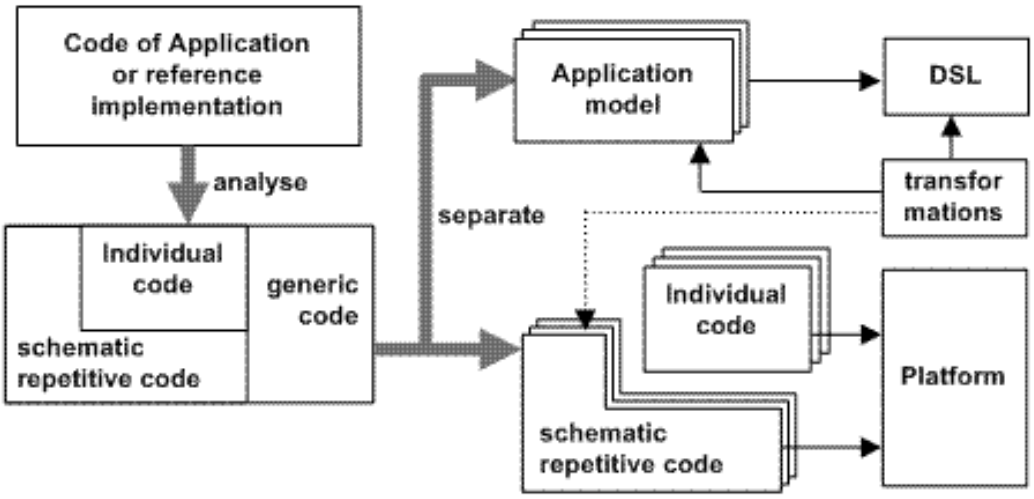


Ilustración 16: Las ideas básicas detrás de MDSD [11]

Se dice que la generación de código en MDSD es semiautomática porque el código fuente se puede descomponer en 3 partes (ver Ilustración 16) [11]:

- Código genérico: que es idéntico para todas las aplicaciones. Se refiere funciones que son administradas por la plataforma. (ej. Garbage collector de .NET, acceso a datos en disco).

- Código esquemático repetitivo: no es idéntico para todas las aplicaciones pero tiene la misma semántica (CRUD, Vistas maestro-detalle, generación de informes), por lo cual se puede generalizar y es al que le apunta MDSD, lo que permite deducir que MDSD no es el reemplazo definitivo para la programación tradicional y no implica cero programación manual.

- Código individual: el que no puede ser generado automáticamente porque es propio de la aplicación y sólo de ella. Este código puede no existir en algunas aplicaciones.

MDSD le da a los modelos un rol principal frente a las propuestas tradicionales basadas en lenguajes de programación, plataformas de objetos y componentes software que han reinado en los últimos años (Ilustración 17) y que presentan

falencias como: demasiado bajo nivel, poca expresividad y programas muy complejos.

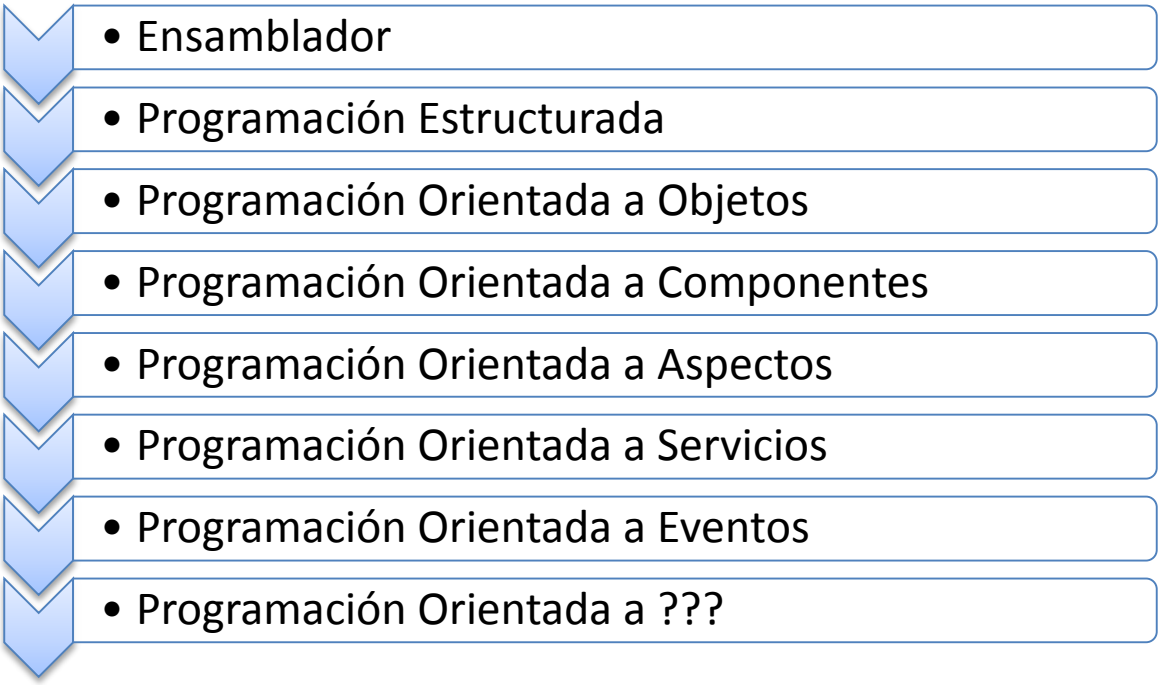


Ilustración 17: Evolución de los enfoques de programación tradicionales

Para remediar estas falencias, MDSD propone elevar el nivel de abstracción del proceso de desarrollo de software convirtiendo a los modelos (y las transformaciones entre ellos) en el principal artefacto durante todas las fases del proceso de desarrollo de software: captura y gestión de los requisitos, diseño, análisis, implementación, despliegue, configuración, mantenimiento, evolución,… [22].

Objetivos de MDSD

- Aumentar la productividad
- Reducir el coste
- Aumentar la calidad del software generado
- Maximizar la compatibilidad entre los sistemas
- Facilitar el re-uso e implementación de otras tecnologías
- Facilitar la evolución y el mantenimiento

- Conservar la inversión en el modelado del dominio y sus reglas de negocio

2.3.2. Arquitectura Dirigida por Modelos (MDA)

La Arquitectura Dirigida por Modelos (en inglés Model-Driven Architecture, MDA) es una visión específica de MDSD creada por OMG (basada en sus estándares) y que difiere de MSDS por ser más restrictiva al enfocarse exclusivamente en lenguajes de modelado basados en UML a través de perfiles [11].

Principios de MDA

- Representación Directa: Enfocándose en el dominio del problema y no de la tecnología.

- Automatización: Herramientas que apoyen y facilitan las labores mecánicas.

- Estándares abiertos: Que eliminen la diversidad y formalicen el desarrollo en cada plataforma.

Niveles de abstracción de MDA

MDA propone los siguientes niveles de abstracción (Ilustración 18):

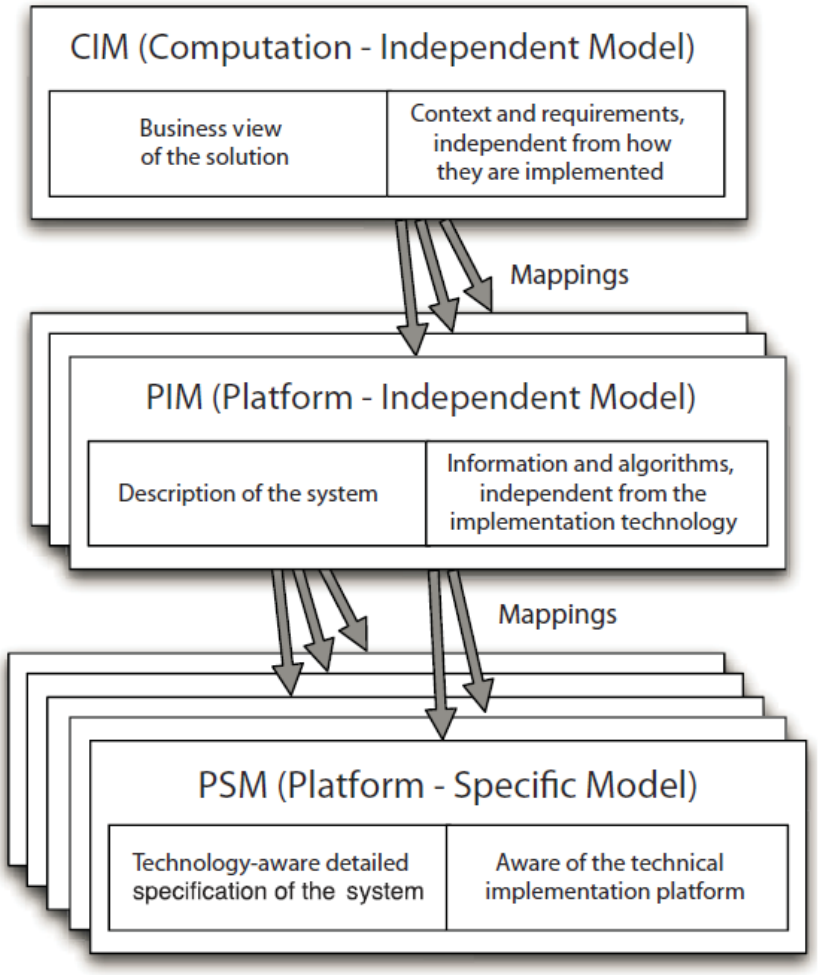


Ilustración 18: Los 3 niveles de modelado y abstracción en MDA [6]

2.3.2.1. Modelo Independiente de la Computación (CIM)

Un Modelo Independiente de la Computación (en inglés Computational Independent Model, CIM) describe los requerimientos de un sistema y el contexto de negocio del mismo. Los CIMs son expresados en términos del negocio o del dominio específico y solo hacen referencias limitadas a las Tecnologías de la Información TI en caso de que sean parte del contexto [23].

2.3.2.2. Modelo Independiente de la Plataforma (PIM)

Un Modelo Independiente de la Plataforma (en inglés Platform Independent Model, PIM) describe como se construirá el sistema en términos de TI pero sin hacer referencia a una plataforma específica (J2EE, .NET). Se modela usando un DSL, típicamente UML adaptado para el dominio específico usando perfiles [11] [23].

2.3.2.3. Modelo Específico a la Plataforma (PSM)

Un Modelo Específico a la Plataforma (en inglés Platform Specific Model, PSM) describe una solución para una plataforma específica, incluye los detalles del PIM para una implementación en una plataforma en particular [23] [10].

2.3.3. Herramientas MDE

Cómo cualquier enfoque de la Ingeniería del Software, MDE ya cuenta con variedad de herramientas de apoyo para sus diferentes procesos (Ilustración 19):

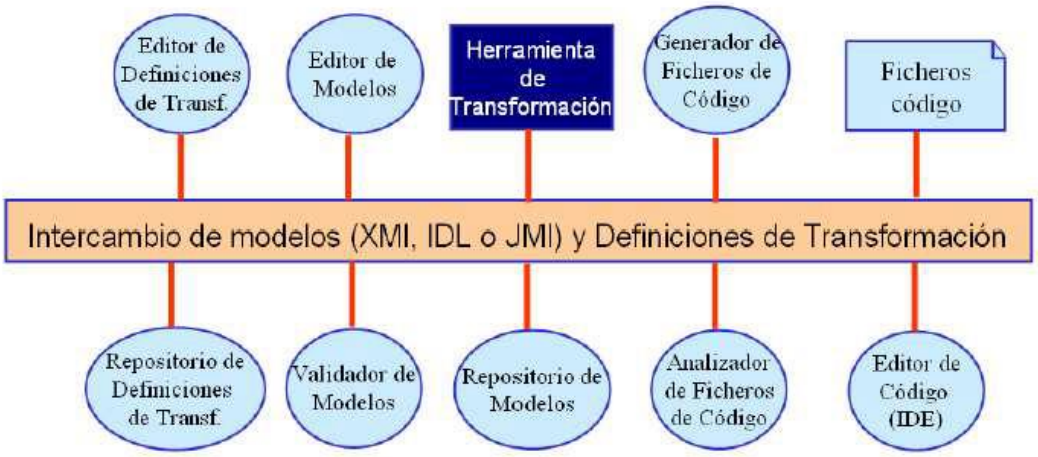


Ilustración 19: Tipos de herramientas MDE

En la sección de estado del arte de esta tesis se muestran las herramientas MDE más populares en la actualidad.

2.3.4. Ingeniería de Ida y Vuelta (RTE)

La Ingeniería de Ida y Vuela (en inglés Round-trip Engineering, RTE) se refiere al paradigma MDE de actualizar el código fuente basado en un modelo (como lo hace MDSD) y viceversa. RTE propone generar parte del código, complementarlo manualmente y tener una actualización recíproca (modelo-código-modelo) [11].

2.4. Modelos de vistas

Un Modelo de Vistas (en inglés View Model) también conocido como marco de referencia arquitectónica (en inglés View Framework) es un tipo de enfoque arquitectónico usado en Ingeniería de Software donde se concibe un sistema desde varias perspectivas definidas como un conjunto de elementos con un sentido específico.

Vista

Una vista es, para definirla sucintamente, un subconjunto resultante de practicar una selección o abstracción sobre una realidad, desde un punto de vista determinado [24].

Punto de vista

Si bien una vista no es más que una representación de todo el sistema software desde una determinada perspectiva, un punto de vista se define como un conjunto de reglas (o normas) para realizar y entender las vistas.

Sistema de Software

Los Modelos de Vistas surgen a partir del principio de “Sistema de Software”, que concibe el software no como una unidad solitaria sino que incluye el hardware en el que corre (Ilustración 20).

Ilustración 20: Sistema de Software

Esto da origen al modelado de sistemas desde varios puntos de vista, en otras palabras, el mismo sistema se puede expresar desde varias perspectivas, que son únicas y a la vez complementarias para lograr expresar un verdadero sistema de software integral.

Marcos de referencia

Tanto los marcos arquitectónicos como las metodologías de modelado de los organismos acostumbran ordenar las diferentes perspectivas de una arquitectura en términos de vistas. La mayoría de los frameworks y estrategias reconoce entre tres y seis vistas, que son las que se incluyen en el cuadro [25]. Los principales marcos de referencia arquitectónica son:

Zachman (Niveles)	**TOGAF (Arquitecturas)**	**4+1 (Vistas)**	**UML (Vistas)**	**POSA (Vistas)**	**Microsoft (Vistas)**
Scope	Negocios	Lógica	Diseño	Lógica	Lógica
Empresa	Datos	Proceso	Proceso	Proceso	Conceptual
Sistema lógico	Aplicación	Física	Implementación	Física	Física
Tecnología	Tecnología	Desarrol	Despliegu	Desarrol	

		lo	e	lo	
Representación		Casos de uso	Casos de uso		
Funcionamiento					

Tabla 2: Vistas en los marcos de referencia [25]

En la Tabla 2 se presentan los marcos de referencia con las vistas del modelado de la plataforma resaltadas. Como se puede apreciar, aunque los marcos tienen una percepción de puntos de vista diferentes, todos coinciden en la separación de los aspectos funcionales y la plataforma. Con base en estos marcos es que el proyecto Metáfora realiza su separación de conceptos en 4 vistas muy similares: las vistas **Conceptual** y de **Escenarios** que son del dominio del modelado de la aplicación (por fuera del alcance de la presente iniciativa) y las vistas **Lógica** y **Física** que son del dominio del modelado de la plataforma.

3. ENFOQUE MULTI-VISTAS DE METÁFORA Y MODELADO DE LA PLATAFORMA

Diversos entornos que siguen estrategias de desarrollo dirigido por modelos [26] [27] [28], tienen inicialmente como su principal actividad, la construcción de un diagrama de clases de UML, el cual posteriormente se va complementando y transformando, hasta llegar a una aplicación computacional en lo que diferenciaremos como método de enfoque MDSD simple (Ilustración 21):

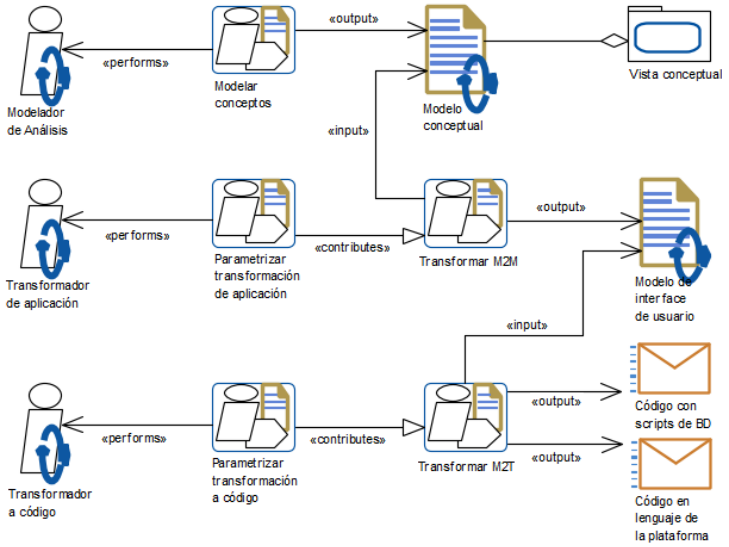


Ilustración 21: Detalle del método en el enfoque MDD simple [29]

Este método simple se puede resumir en los siguientes pasos: construir un modelo de origen, parametrizarlo y transformarlo en un modelo destino, y a partir de la parametrización de este, generar el código de la aplicación. Sin embargo, en un enfoque como el usado en el ejemplo anterior quedan muchos detalles por resolver, para cubrir estos detalles es necesario reflexionar acerca de cómo afrontar cada una de las problemáticas enunciadas en la descripción del problema del capítulo anterior. El enfoque multi-vistas de Metáfora argumenta la solución de los 3 problemas de MDSD expuestos al inicio de este documento de la siguiente manera [29]:

En el frente de las carencias en la expresividad de los modelos: generalmente los proyectos de desarrollo de software involucran grupos de personas que desempeñan diversos roles, cada rol necesita expresar diferentes aspectos del

sistema de información. Cuando un enfoque MDSD, usa como origen de la aplicación un solo modelo, los implicados se ven imposibilitados para plasmar algunos detalles. Al involucrar múltiples vistas en el desarrollo, y por ende múltiples modelos, se posibilita expresar estos detalles que se hubieran quedado por fuera y la vista física en particular permite articular aspectos del procesamiento entre los distintos equipos que conforman la solución.

En el frente de la complejidad en la intervención de las transformaciones: un factor clave para el éxito en la adopción de enfoques dirigidos por modelos, es posibilitar que las transformaciones sean parametrizables e intervenibles, permitiendo a los implicados expresar esos detalles que desean en el código y la aplicación. En este sentido, un enfoque multi-vistas debe ejecutar las transformaciones a través de procesos asistidos y herramientas que posibiliten la personalización de los modelos y el código generado.

En el frente de la escasez en multi-plataformidad: un enfoque multi-vistas, permite que una vista o conjunto de vistas, se utilicen para definir los elementos propios de la plataforma. Estas deben permitir expresar tanto la arquitectura lógica, como la arquitectura física de la configuración de plataforma, la cual puede ser utilizada en diversos procesos de generación de aplicaciones. La multi-plataformidad se puede alcanzar cuando se usan los mismos modelos de una aplicación, combinándolos con distintas configuraciones, para generar diversas versiones de la misma aplicación en distintas plataformas. Lo anterior sugiere que exista un repositorio o mecanismo que permita el almacenamiento y re-uso de las configuraciones de plataforma en diferentes proyectos de desarrollo de software, inclusive se pueden usar varias configuraciones plataforma para un mismo proyecto, en donde por ejemplo parte del sistema se despliega en la web y otra parte se despliega en dispositivos móviles.

Para capitalizar estas estrategias, Metáfora propone la posibilidad de definir un método que involucra 3 vistas adicionales con sus respectivos actores, además de una serie de actividades para enriquecer el proceso de generación de software a partir de MDSD (Ilustración 22):

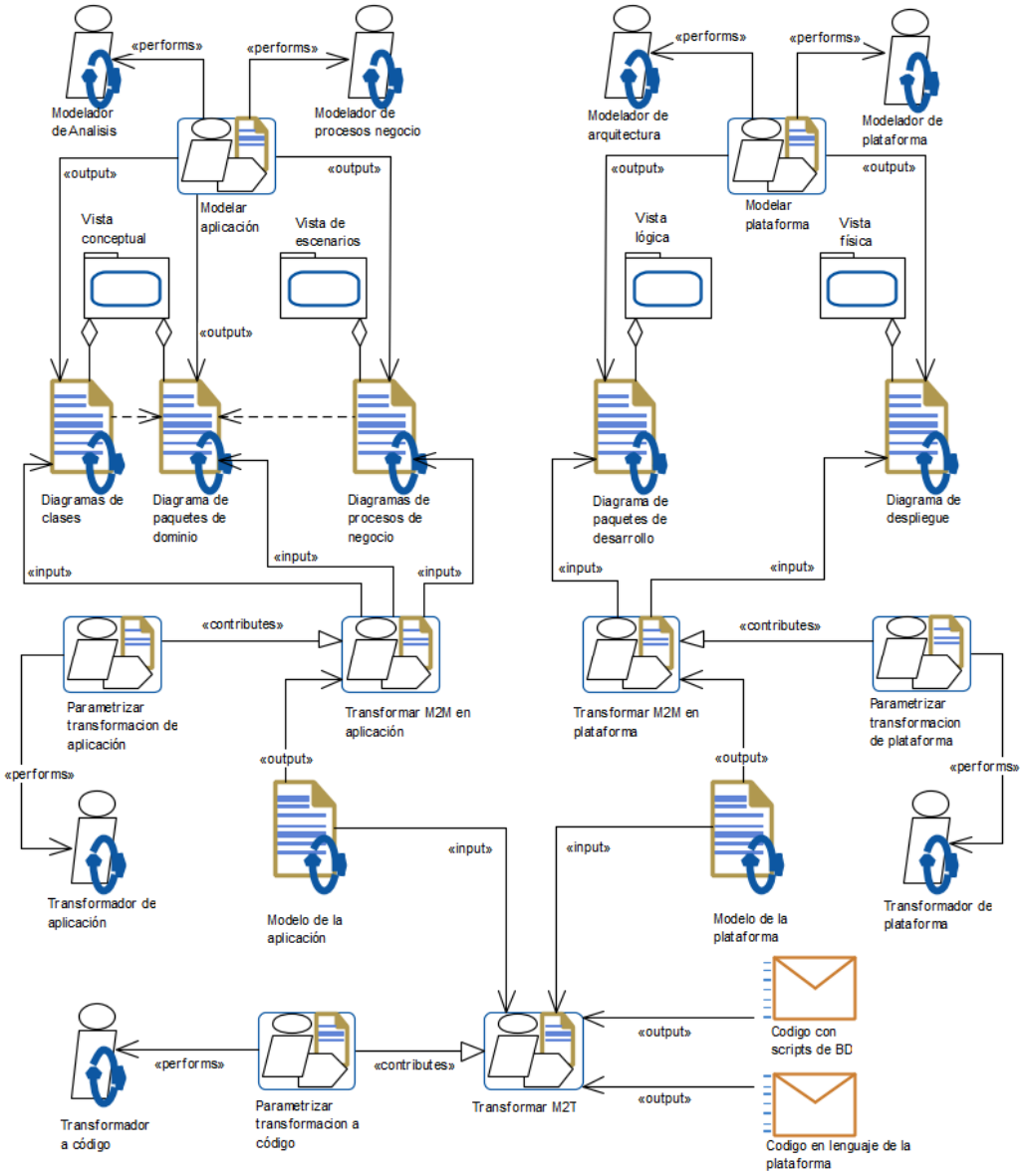


Ilustración 22: Método multi-vistas de Metáfora [29]

Como se aprecia en la Ilustración 22, el método de Metáfora presenta el modelador de análisis como el responsable de la vista conceptual, el modelador de procesos de negocio es el responsable de la vista de escenarios, el modelador de arquitectura es el responsable de la vista lógica y el modelador de plataforma es el responsable de la vista física. Estas 4 vistas se construyen utilizando Modelado y Notación de Procesos de Negocios (BPMN, por sus siglas en inglés) y diagramas UML específicos para cada aspecto del sistema, ya que este es el lenguaje de facto para diseñar software tanto a nivel académico como industrial.

Una vez modeladas las 4 vistas, el siguiente paso es aplicar las transformaciones de modelo a modelo (M2M) para generar los modelos de la aplicación y de la plataforma conforme a sus respectivos meta-modelos. Para fusionar de forma apropiada todos los elementos definidos en dichos meta-modelos, es necesario que las actividades de parametrización de transformación se apoyen en asistentes o modeladores que permitan realizar los mapeos o correspondencias entre todos estos elementos. Para tal propósito el meta-modelo se puede apoyar en propuestas como la planteada en WebSA [30], que define un modelo de subsistemas con capas, un modelo de configuración con el conjunto de componentes web y sus conectores, y un modelo de integración para mezclar los dos modelos con la vista funcional de una aplicación web.

En el caso de Metáfora es requerido también otro mecanismo de mapeo entre el modelo resultante del flujo de la aplicación y el del modelado de plataforma para poder proceder a la transformación modelo a texto (M2T) y la actividad de parametrizar transformación a código que permite intervenir las instancias de ambos modelos de cara a la generación del código fuente en lenguaje de la plataforma específica y los scripts de base de datos dado el caso.

Para implementar el método presentado anteriormente es adecuado aplicar un enfoque *Bottom-Up* [31] en donde se parte de la salida (resultado esperado) para ir escalando hasta las entradas necesarias para producir dicha salida. De esta manera se pueden distinguir 3 nodos, equivalentes a: el meta-modelo de la aplicación, el meta-modelo de la plataforma y la transformación M2T, que representan puntos de quiebre en donde se pueden aislar los flujos de trabajo del método (Ilustración 23):

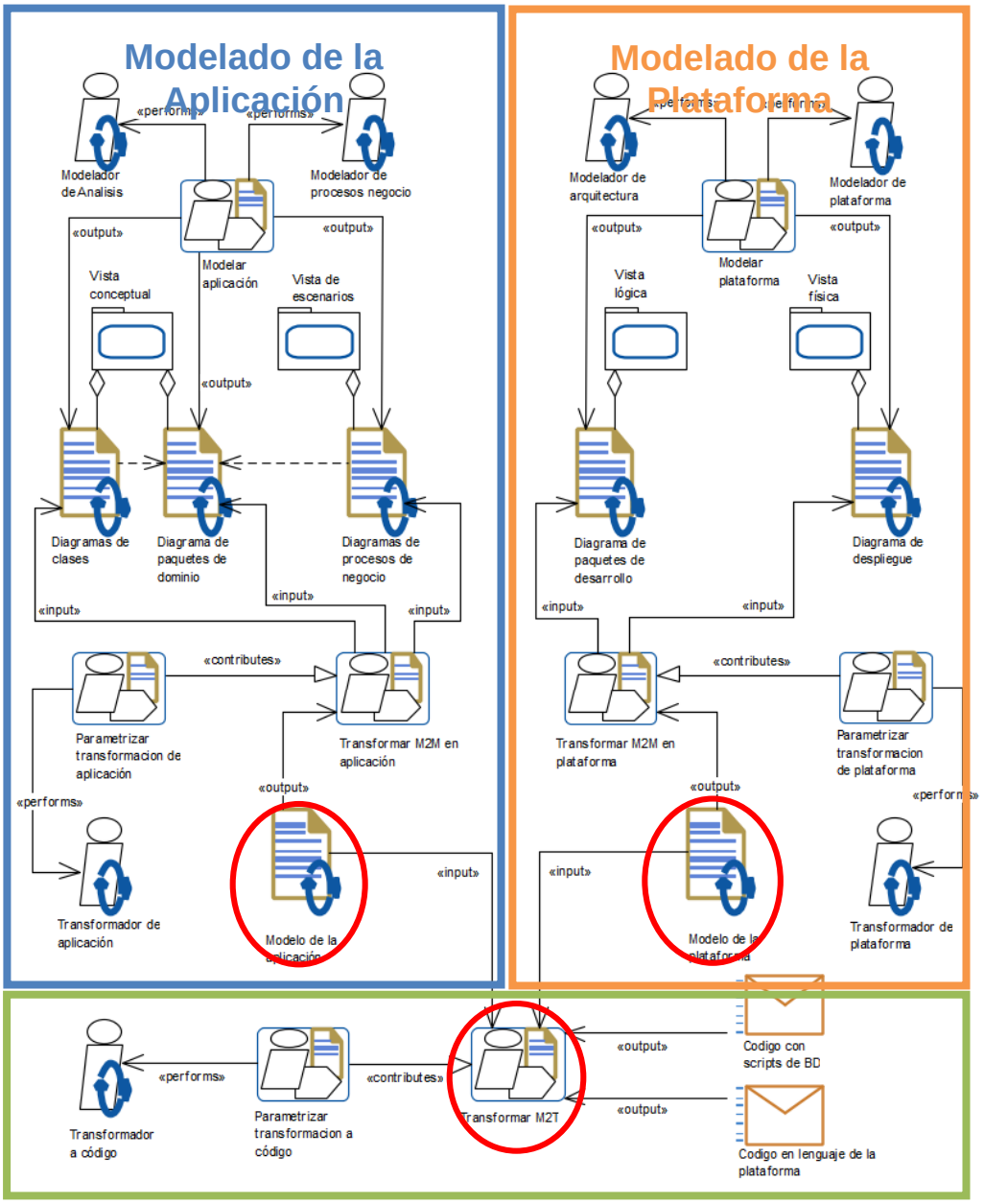


Ilustración 23: Flujos de trabajo del método de Metáfora [29]

De estos 3 nodos se desprenden los 3 flujos de trabajo independientes del proceso de Metáfora, a saber:

Modelado de la aplicación: Este flujo se encarga de los aspectos funcionales del sistema de software y aunque es parte esencial del método, no es objeto del presente esfuerzo ya que hace parte de otra iniciativa también ligada al proyecto Metáfora y además se puede aislar por medio de una instancia del meta-modelo

de aplicación que luego se integra con una instancia del meta-modelo de plataforma para la generación del código fuente. Dentro de este flujo se desarrollan las vistas conceptual y de escenarios con tecnologías y diseños propios e independientes del modelado de la plataforma.

Modelado de la plataforma: Este flujo es el foco principal de la presente iniciativa y es donde se ubica el modelado de las vistas lógica y física. Este frente de trabajo se encarga de aspectos no funcionales apuntando especialmente a la multi-plataformidad del sistema de software.

Generación de código fuente: Este flujo se encarga de combinar los aspectos desarrollados en los otros dos frentes de trabajo a partir de las instancias de los meta-modelos de aplicación y plataforma, concluyendo la transformación M2T donde se genera el producto final de Metáfora (el código fuente).

Dentro de estos 3 flujos de trabajo del método de Metáfora, la presente iniciativa cubre el **modelado de la plataforma** (compuesto por las vistas lógica y física) y **parte de la generación de código fuente**, o sea las siguientes tareas del método de Metáfora:

- Modelar plataforma (UML)
- Parametrizar transformación de plataforma
- Transformar M2M en plataforma
- Transformar M2T (parcialmente)

Una vez expuesto el método propuesto por Metáfora e identificado el foco de la presente iniciativa, podemos enumerar algunas premisas:

- Si bien las vistas lógica y física son independientes en su nivel de abstracción más alto, es conveniente realizar un esfuerzo conjunto para el desarrollo de ambas dada su dependencia en cuanto al diseño arquitectónico se refiere.
- El modelado inicial de las vistas se debe realizar usando la especificación de UML debido a su divulgación y aceptación tanto en la academia como en la industria.

- Se hace necesario un mecanismo para representar estas dos vistas y desarrollarlas dentro del método de Metáfora, en términos de MDSD, es necesario elaborar un DSL que permita la representación de las vistas lógica y física.
- Se requiere un mecanismo para realizar el mapeo entre elementos de diferentes modelos.
- Ya que el modelado de la plataforma es uno de los insumos para la generación de código fuente, es producente que la presente iniciativa desarrolle parte de este último.
- Se debe diseñar un meta-modelo para la plataforma.
- Es preciso identificar los diferentes tipos de artefactos de código fuente de cara a la construcción de las plantillas necesarias para su generación.
- Para demostrar las mejoras aportadas por esta iniciativa y facilitar el desarrollo del DSL, es producente aplicar un estudio de caso en cada una de las etapas del proceso utilizando varias plataformas tecnológicas (tales como: .NET, Java, Android, PHP, etc.).

4. TRABAJOS RELACIONADOS

4.1. Enfoques de Ingeniería Web Dirigida por Modelos

Existen varios enfoques MDWE (Ingeniería Web Dirigida por Modelos) documentados en el estudio de Hincapie y Duitama [32], a saber:

- Object Oriented Web Solutions (OOWS) [33]
- UML-Based Web Engineering (UWE) [34]
- Web Modeling Language (WebML) [35]
- Hypertext Modeling Method of MIDAS [36]
- DSL for the implementation of dynamic web applications (WebDSL) [37]
- DSL for generating Web application (MarTE/Quorra) [38]
- Web Software Architecture (WebSA) [30].

Todas las iniciativas listadas anteriormente tratan de alguna forma aspectos de 3 modelos: estructura, navegación y presentación, en algunos casos se usan algunos modelos extra, así sea de manera implícita como en el caso de MarTE/Quorra. Utilizar los modelos comunes facilita el entendimiento de los diseños de aplicaciones y provee un conjunto de conceptos standard para el modelado de aplicaciones.

Todos los enfoques mencionados, excepto WebML y WebDSL, usan UML junto con perfiles para definir la notación de sus modelos, también hay notaciones gráficas que representan mejor los elementos del dominio de las aplicaciones de software (este es el caso de WebML).

La mayoría de estos enfoques no consideran un mecanismo para describir la especificación de 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 pueda ser reusada para diferentes plataformas. No obstante, en el caso de

WebSA [30], los autores aseguran que su enfoque soporta la modularización de la plataforma y las transformaciones para soportar diferentes arquitecturas. Ellos definen dos transformaciones: una que integra los modelos funcionales y arquitectónicos, y otra que transforma ese modelo integrado a modelos específicos de plataforma. La segunda transformación usa el estándar MOFScript [39] del Grupo de Gestión de Objetos (OMG, por sus siglas en inglés) y define las reglas para transformar el modelo independiente de plataforma en el código específico para cada tecnología (J2EE o .NET en el caso de WebSA).

4.2. Enfoques Ingeniería Móvil Dirigida por Modelos

Existen también algunos enfoques de Ingeniería Móvil Dirigida por Modelos, a saber:

El enfoque de Balagtas-Fernandez [40] que combina los principios de Human Computer Interaction (HCI), más precisamente centrado en el diseño de usuario. El objetivo de este enfoque es el de obtener un sistema que permite a los usuario sin experiencia en programación crear sus propias aplicaciones móviles.

También cabe destacar la iniciativa de Carton [41] que propone un enfoque de desarrollo que combina programación orientada a aspectos y técnicas MDSD. Este enfoque busca afrontar las limitaciones de recursos de las aplicaciones y la diversidad de plataformas de software.

4.3. Enfoques MDSD que involucran Requisitos No Funcionales

Los requisitos no funcionales (NFR) tienen una relación inherente con el modelado de la plataforma en cuanto a que consideran aspectos como: portabilidad, escalabilidad, mantenibilidad y rendimiento, que son expresiones propias de una arquitectura de software en cuanto a la aplicación de patrones y mejores prácticas a la hora de codificar y desplegar la aplicación usando diferentes tecnologías. Es

por eso que en el presente trabajo se considera el estudio de D. Ameller, X. Franch y J. Cabot donde se realizó una revisión sistemática de literatura en la Web of Science (WoS) [4] en la que identificaron 39 referencias de combinaciones de MDSD con NFRs y donde encontraron que los trabajos existentes en este campo se enfocan en un tipo específico de NFR para un dominio en particular como se puede ver en la Tabla 3:

Type of NFR addressed	Domain	Instrument
Modeling NFRs		
Operationalizable NFRs	Independent	NFR Framework + UML annotations
Security, Fault Tolerance	SOA	UML Profile
Any	Independent	UML Profile
Performance	SOA	Own metamodel
Resource Usage	Embedded systems	Own metamodel
Usability	Web IS	Own metamodel
Model Transformation		
Quality of Service (QoS)	Independent	Patterns
Any	Independent	Patterns
Model Analysis		
Quality of Service (QoS)	Independent	Measurable models
Performance, Reliability	Independent	Markov models
Performance, Reliability	SOA	Probabilistic models
Reliability	Independent	LTSA*
Any	Independent	Not specified

Tabla 3: Enfoques MDSD que involucran NFRs [4]

Varios autores proponen el modelado de NFRs usando extensiones UML [42] [43] [44], incluyendo el estándar para los perfiles UML de OMG: MARTE [45] y QoS-Profile [46].

Algunos trabajos han presentado meta-modelos para representar NFRs en MDSD [47] [48] [49].

Existen algunas aproximaciones acerca de la transformación de modelos que proponen una serie de patrones que satisfagan los requisitos de calidad del servicio (QoS) basados en ciertos NRFs [50] [51].

Hay diferentes propuestas que dan cabida al análisis dónde cada NFR equivale a una dimensión completa del software [48] [52] [53] [54] siguiendo la propuesta de S. Röttger y S. Zschaler [55] que analiza la satisfacción de un NFR dado

formalizando un diseño de sistema específico. Otros enfoques analizan los NFR de manera independiente y usan diferentes niveles de abstracción para ellos [52] [56], sin embargo, ninguno de ellos aborda de manera integral un modelado de las arquitecturas o plataformas de desarrollo específicas como si lo pretende hacer la presente iniciativa.

5. ESTUDIO DE CASO Y SELECCIÓN DE TECNOLOGÍAS

La siguiente es una referencia textual del caso de estudio utilizado en el proyecto Metáfora tomada del artículo de Quintero, Hincapie y Anaya [57].

“Para posibilitar la ilustración de las diversas fases de un proceso y las disciplinas de un método, es imperativo tener un estudio de caso que permita ejemplificar las vistas y tipo de modelos a utilizar. Es importante que dicho estudio de caso, permita trabajar las diferentes dimensiones de un sistema de información, de tal forma que se pueda pasar de una vista en la que se consiga implementar una aplicación, hasta múltiples vistas que permitan ver los diferentes frentes de la aplicación construida.

Considerando lo anterior, se tomará como referente de trabajo ITIL (Information Technology Infrastructure Library) [58], marco para la gestión de servicios de tecnologías de la información, el cual propone los tópicos para consolidar el modelo de "ciclo de vida del servicio", separando y ampliando algunos subprocesos hasta convertirlos en procesos especializados. ITIL en su versión 3 consta de los siguientes 5 libros basados en el ciclo de vida del servicio: 1-Estrategia del servicio, 2-Diseño del servicio, 3-Transición del servicio, 4-Operación del servicio y 5-Mejora continua del servicio; esto lo convierte en un referente con un campo de acción muy amplio. Por la naturaleza de este trabajo, el estudio de caso se basará en el libro 4-Operación del servicio, específicamente en el proceso de proceso de gestión de incidentes, frecuentemente llamado IcM por la sigla en inglés de Incident Management.

El estudio de caso se centrará entonces en el desarrollo de un sistema de información para manejar el ciclo de vida del proceso de gestión de incidentes, en este sentido se debe tener el proceso que describe ITIL alrededor de IcM. A continuación se hace una breve compilación de lo que propone el libro 4 en este frente, y se adapta al área de servicios de soporte y mantenimiento de sistemas de información:

- El usuario final detecta un incidente en uno de los sistemas de información que son responsabilidad de la compañía.

- Cuando el analista de la mesa de ayuda recibe el reporte, realiza el registro del incidente, luego lo clasifica y determina si se conoce una solución al posible problema.
- Si existe una solución conocida, procede a resolver y cerrar el incidente.
- Si no existe una solución conocida y la prioridad no es alta, se escala el incidente al experto de la mesa de ayuda, quien investiga, diagnostica y reparar el incidente, luego se resuelve y recupera el sistema, y finalmente se envía al analista de la mesa de ayuda para que cierre el incidente.
- Si no existe una solución conocida y la prioridad es alta, el incidente se envía al experto en la materia en cuestión, quien determina si efectivamente se trata de un incidente mayor.
- Si no se trata de un incidente mayor, se remite el caso al experto de la mesa de ayuda, para que este realice el proceso desde la investigación hasta el envío para el cierre.
- Si por el contrario si se trata de un incidente mayor, el experto en la materia en cuestión procede a encontrar una solución, antes de remitir el caso al experto de la mesa de ayuda, para que este realice el proceso desde la investigación hasta el envío para el cierre."

Con base en el estudio de J. Quintero y R. Anaya en el que se hace una clasificación de herramientas MDSD según su propósito (ver Tabla 4) [59], se selecciona Eclipse como la plataforma de facto para MDSD [6].

Id	Categoría	Herramientas
C1	Herramientas de despliegue de aplicaciones	**H0-** ***OptimalJ***: www.compuware.com/products/optimalj/ **H1-** ***AndroMDA***: www.andromda.org/
C2	Herramientas de Transformación	**H2-** ***UMT/QVT***: umt-qvt.sourceforge.net/ **H3-** ***ATL***: http://www.eclipse.org/gmt/atl/download/
C3	Herramientas clásicas de modelado y desarrollo	**H4-** ***Enterprise Architect***: www.sparxsystems.com/ **H5-** ***JDeveloper***: www.oracle.com/technology/products/
C4	Entornos de verificación de modelos	**H6-** ***OCLE***: lci.cs.ubbcluj.ro/ocle/ **H7-** ***Octopus***: www.klasse.nl/octopus/index.html
C5	Herramientas de Apoyo al desarrollo de herramientas	**H8-** ***EMF***: www.eclipse.org/emf/ **H9-*****Jamda***: jamda.sourceforge.net/

Tabla 4: Principales herramientas de modelado según el propósito [59]

Dentro de Eclipse se encuentra la iniciativa Eclipse Modeling Framework (EMF) que provee un marco de trabajo para el modelado y generación de código fuente.

EMF dispone una gran variedad de plugins para todas las etapas de un proceso MDSD [60] basados en un modelo estructurado y siguiendo la especificación XMI.

Dentro de las herramientas disponibles en EMF y requeridas por el proceso de Metáfora encontramos las siguientes en la Tabla 5:

Categoría	Herramienta
Meta-modelos	Ecore, KM3, Xtext, EMFText, JaMoPP, EcoreTools, MoDisco.
Modelador gráfico	GMF, Papyrus, Borland Together, RSM, Leonardi, Apollo, Rational Software Modeler, Sirius, Graphiti.
Integración de modelos	AMW, Virtual EMF, EMFCompare, Refactory.
Transformaciones de modelo a modelo	ATL, M2M, VIATRA2, Kermeta, Operational QVT.
Transformaciones de modelo a texto	Acceleo, M2T, Merlin Generator, Skyway Builder, EGF.

Tabla 5: Herramientas EMF

Si bien la oferta es amplia, cada tecnología es independiente y con un propósito específico, aumentando la curva de aprendizaje y dificultando la interacción entre las diferentes fases del proceso. No obstante, existe una iniciativa basada en EMF llamada Epsilon, que agrupa una familia de lenguajes y herramientas para la gran mayoría de actividades MDSD, a saber:

Emfatic: lenguaje para representar modelos EMF en forma textual.

Human Usable Textual Notation (HUNT): implementación del estándar OMG para representar modelos en un lenguaje más común para los usuarios.

Epsilon Object Language (EOL): lenguaje imperativo orientado a modelos que combina el estilo procesal de Javascript con las capacidades de consulta de OCL. Este lenguaje es la base de los demás lenguajes de Epsilon.

Epsilon Validation Language (EVL): lenguaje de validación de modelos que soporta la certificación de la consistencia tanto al interior de un modelo como en relación con modelos externos y provee integración con editores EMF y GMF.

Epsilon Transformation Language (ETL): lenguaje de transformación M2M basado en reglas que soporta varios modelos tanto de entrada como de salida y diferentes tipos de reglas y prioridades de ejecución.

Epsilon Comparison Language (ECL): lenguaje para definir reglas de equivalencia entre elementos de diferentes modelos.

Epsilon Merge Language (EML): lenguaje basado en reglas para mezclar múltiples modelos que pueden ser conformes a múltiples meta-modelos de acuerdo a sus correspondencias dictadas por ECL.

Epsilon Generation Language (EGL): un lenguaje de transformación M2T basado en plantillas para generar código fuente u otro tipo de artefactos de software.

Epsilon Wizard Language (EWL): lenguaje adaptado para transformaciones interactivas realizadas por el usuario el mismo editor gráfico.

Epsilon Flock: lenguaje de transformación basado en reglas para actualizar modelos cuando sus meta-modelos cambian.

EuGENia: herramienta *front-end* para GMF que agiliza el proceso de creación de editores gráficos personalizados de modelos.

Exeed: versión mejorada del editor de modelos tipo árbol de EMF que permite personalizar su apariencia.

Modelink: herramienta que permite manipular varios editores de árbol EMF para establecer enlaces entre elementos de diferentes modelos usando *drag-and-drop.*

Workflow: conjunto de tareas ANT para ensamblar flujos de ejecución complejos con lenguajes de Epsilon.

Concordance: herramienta para monitorear y mantener los índices de referencias entre elementos de modelos EMF.

EUnit: framework para realizar pruebas unitarias con lenguajes MDSD.

Después de realizar la evaluación de herramientas con una prueba de concepto sencilla, se concluye que para la elaboración de Morphosys es procedente utilizar el framework Epsilon y sus lenguajes en la mayoría de las etapas del proceso a excepción del modelado gráfico de las vistas, en donde es más conveniente utilizar una herramienta con soporte directo de UML, en este caso Papyrus, dada su facilidad de uso, su integración con eclipse y su compatibilidad con la versión 2.4 de la especificación de UML.

6. VISTAS DE LA PLATAFORMA

Para desarrollar el flujo de modelado de la plataforma de Metáfora, el punto de partida es la construcción de las vistas lógica y física del sistema de software, ya que la intención es usar la documentación UML de la fase de diseño como el principal insumo para el modelado dentro de Morphosys. De esta manera se espera que el resultado del modelado de aplicación (unidades funcionales) se combine con el modelo de plataforma para generar el código fuente.

Se hace necesario entonces analizar estas dos vistas (lógica y física) y definir claramente sus responsabilidades y precisar qué información se puede extraer de ellas antes de realizar las transformaciones MDSD.

Nota: El desarrollo de las vistas en Morphosys está basado en la versión 2.0 de UML detallada en [61].

6.1. Vista Lógica

La vista lógica es la encargada de mostrar los componentes principales 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” [2]. También podemos hablar en esta vista de la distribución de funcionalidades en paquetes o componentes con sentido desde el punto de vista de un desarrollador, aunque típicamente esta vista es diseñada por el arquitecto. Dentro de estos paquetes es donde se ubican los artefactos a nivel físico que van a ser construidos por el equipo de desarrollo [62].

En la vista lógica 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 re-uso o funcionalidades comunes.

La vista lógica también es frecuentemente utilizada para la asignación de bloques de desarrollo a diferentes partes del equipo de trabajo, evaluación de costos, planeación y seguimiento del progreso [2].

6.2. Representación de la Vista Lógica con UML

La vista lógica es representada dentro de Metáfora con el **diagrama de paquetes de desarrollo** de UML que muestra los diferentes componentes del sistema simbolizados por paquetes y las dependencias entre ellos.

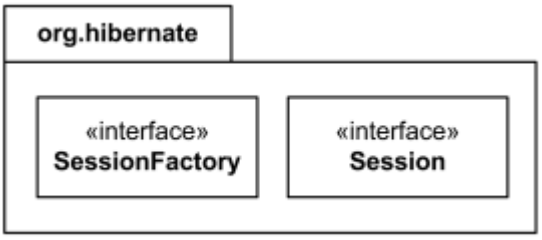


Ilustración 24: Paquetes UML [63]

Los **paquetes** UML (Ilustración 24) sirven para agrupar otros elementos UML o tipificar componentes a un nivel de abstracción más alto para organizar un sistema, en otras palabras, representan un **namespace** único usado para agrupar elementos con similitud semántica que usualmente son modificados de manera conjunta.

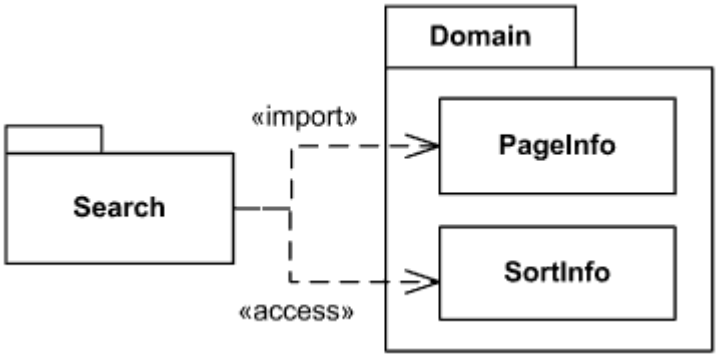


Ilustración 25: Dependencias entre Paquetes [63]

Así como los paquetes pueden contener otros paquetes, también pueden relacionarse entre ellos a través de **dependencias** de diferentes tipos (Ilustración 25). Una dependencia de tipo **import** es una relación que agrega explícitamente los miembros del paquete importado al paquete que importa, quedando fuertemente acoplados. Otra especialización muy usada de las dependencias

import es la de tipo **access**, que también agrega los miembros del paquetes importado pero de manera privada [63]. Por último está la dependencia de tipo **merge** que indica que el contenido de un paquete es extendido por el otro en una relación de tipo herencia similar a la generalización.

UML provee algunos estereotipos que se pueden aplicar a los paquetes (Ilustración 26), tales como: el estereotipo **<<Interface>>** que sirve como contrato para otro paquete o el más recientemente incluido **<<Model>>** que es un paquete especializado que describe un sistema desde cierto punto de vista y que es utilizado comúnmente para representar las capas de una aplicación.

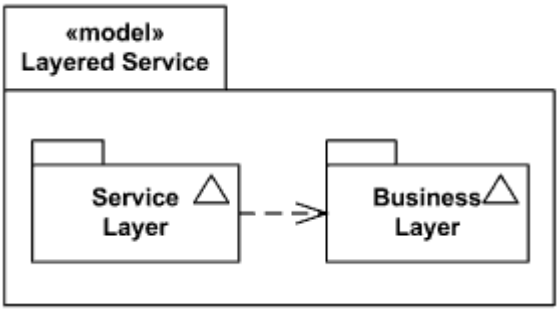


Ilustración 26: Estereotipo Model [63]

6.2.1. Consideraciones para la construcción de la Vista Lógica

Como se mencionó anteriormente, la intención del proyecto Metáfora es importar los diagramas UML directamente de la fase de diseño de software y usar esos artefactos como insumo para el modelado. Por esto cabe anotar que si bien Morphosys soportará la inclusión de cualquier elemento de la versión 2.0 de UML [61], no todos los elementos tienen un valor semántico para la herramienta, por lo cual en este apartado se enfoca especialmente en los elementos que contienen información para alimentar el modelo de la plataforma. Estas son algunas consideraciones a tener en cuenta a la hora de modelar la vista física:

Los diagramas de paquetes son usados típicamente con 2 propósitos diferentes: la separación en módulos funcionales propios de cada dominio y las capas o patrones de desarrollo (Ilustración 27) [16]. Los módulos funcionales pertenecen al dominio del modelado de la aplicación y no deben ser utilizados al diagramar la plataforma en donde solo son relevantes los patrones y capas que puedan

expresar elementos propios de la arquitectura. Sin embargo no se debe abusar del uso de paquetes para generar una granularidad de capas innecesaria.

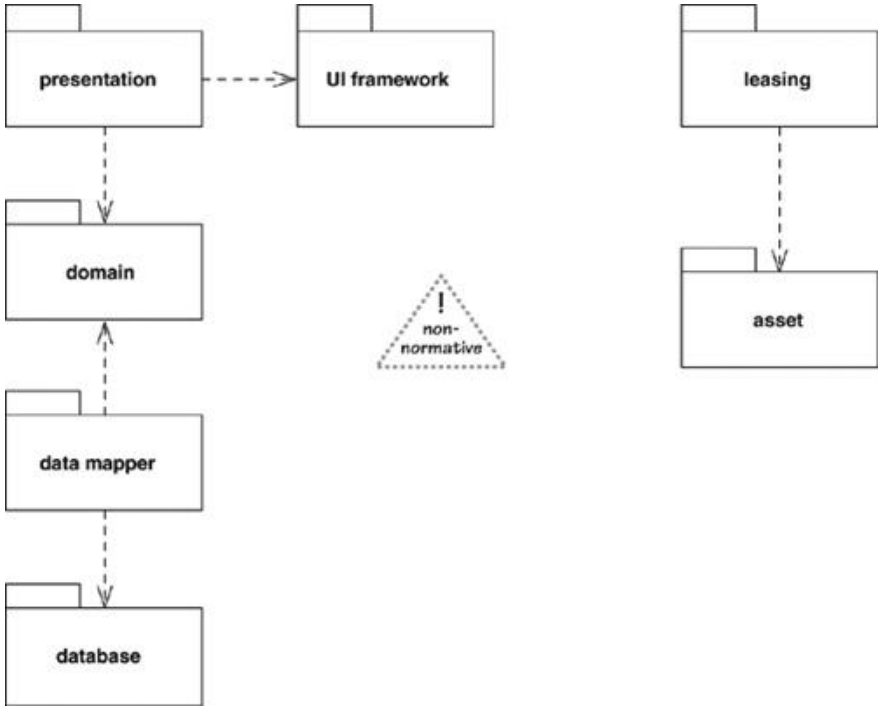


Ilustración 27: Los dos tipos de aspectos comúnmente modelados con paquetes [16]

Cuando se habla de capas en la vista lógica es de suprema importancia realizar la diferencia entre layers y tiers. Los paquetes de la vista lógica representan la distribución lógica, conocida en inglés como **layers**, y no toma en cuenta la distribución física de los componentes, la cual es responsabilidad de las capas físicas (**tiers** en inglés), que normalmente hacen referencia a dispositivos de hardware o con procesamiento propio (tales como: servidores, computadores, ubicaciones remotas, etc.) [64]. Para el diseño de cada una de las capas se recomienda basarse en guías arquitectónicas como las de Microsoft [64] [65].

Se invita también a utilizar el atributo **URI** de los paquetes para indicar el namespace que tendrán los artefactos asociados a cada paquete. Cada paquete debe tener un URI único dentro de su ámbito y se recomienda conservar una consistencia jerárquica en los namespaces con respecto a la agrupación de paquetes, por ejemplo, si un paquete A contiene un paquete B, los URIs deberían

ser namespaceA y namespaceA.namespaceB para facilitar el modelado y posteriormente la lectura del código fuente.

Las dependencias entre paquetes no son de tipo transitivo, lo que permite el correcto aislamiento de las capas y favorece la mantenibilidad y escalabilidad del producto de software.

La elección de los paquetes, su estructura y cuáles elementos deben asociarse a cada uno es un tema subjetivo pero expertos recomiendan el uso del principio de cierre y el principio de re-uso común [66]. El primero indica que los elementos contenidos en un paquete deben necesitar modificaciones por razones similares, mientras que el segundo dice que los elementos deben ser reusados de forma conjunta. De cualquier manera, muchas de las razones para organizar los paquetes están ligadas a las dependencias entre ellos.

Un diagrama de paquetes correctamente elaborado permite ver claramente el flujo de la información desde las capas más exteriores hacia las más internas, teniendo cuidado de no generar referencias circulares que resulten en errores de compilación como lo relata Martin en su principio de la dependencia acíclica [66].

Las dependencias de tipo import deben usarse para indicar referencias físicas a proyectos o librerías (internas o externas), mientras que las de tipo access son más apropiadas para consumo de servicios sin una referencia física, como por ejemplo el llamado a un servicio web.

6.2.2. Aplicación de la Vista Lógica

Para el estudio de caso de IcM se presentan a continuación dos aplicaciones de arquitecturas con la implementación de los patrones de diseño Modelo-Vista-Presentador (MVP) y Modelo-Vista-Controlador (MVC) para la capa de presentación (Ilustración 28).

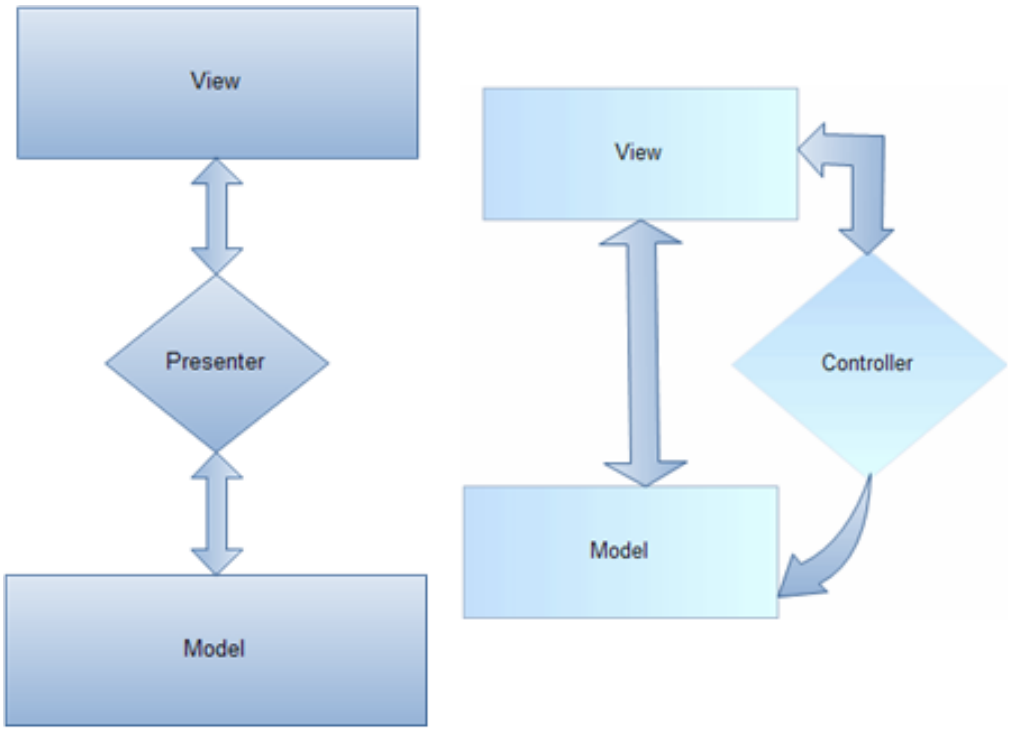


Ilustración 28: Patrones de diseño MVP y MVC [67]

Tanto MVP como MVC buscan un diseño de componentes de presentación de manera desacoplada de modo que se puedan implementar diferentes clientes (web, móvil, embebido, etc) sin afectar la lógica de presentación. Estos enfoques se diferencian en la manera en que sus componentes se comunican, como se puede apreciar en la Ilustración 28 [67].

Para el resto de las capas (a parte de la presentación) se realizan algunas variaciones basadas en las guías arquitectónicas de Microsoft [64] [65].

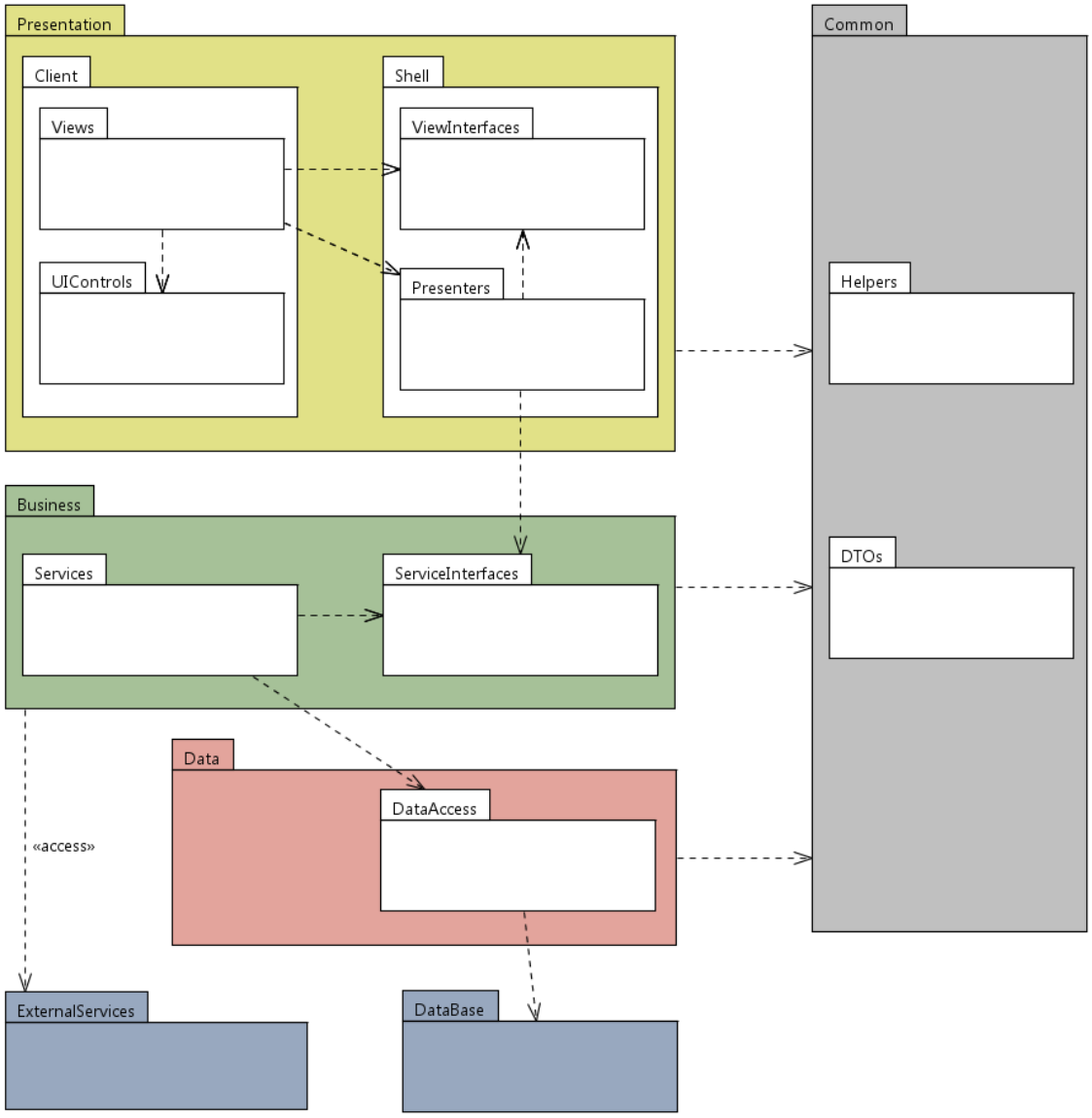


Ilustración 29: Diagrama de Paquetes de IcM implementando el patrón MVP

En el diseño presentado en la Ilustración 29 se diferencian 4 capas: presentación, negocio, datos y común; que a su vez están subdivididas en sub capas (incluyendo servicios externos y fuentes de datos).

La segunda aplicación (ejemplo) de diseño arquitectónico presenta un enfoque MVC basado en componentes y dirigido por eventos. La capa de acceso a datos en este enfoque implementará el patrón Active Record.

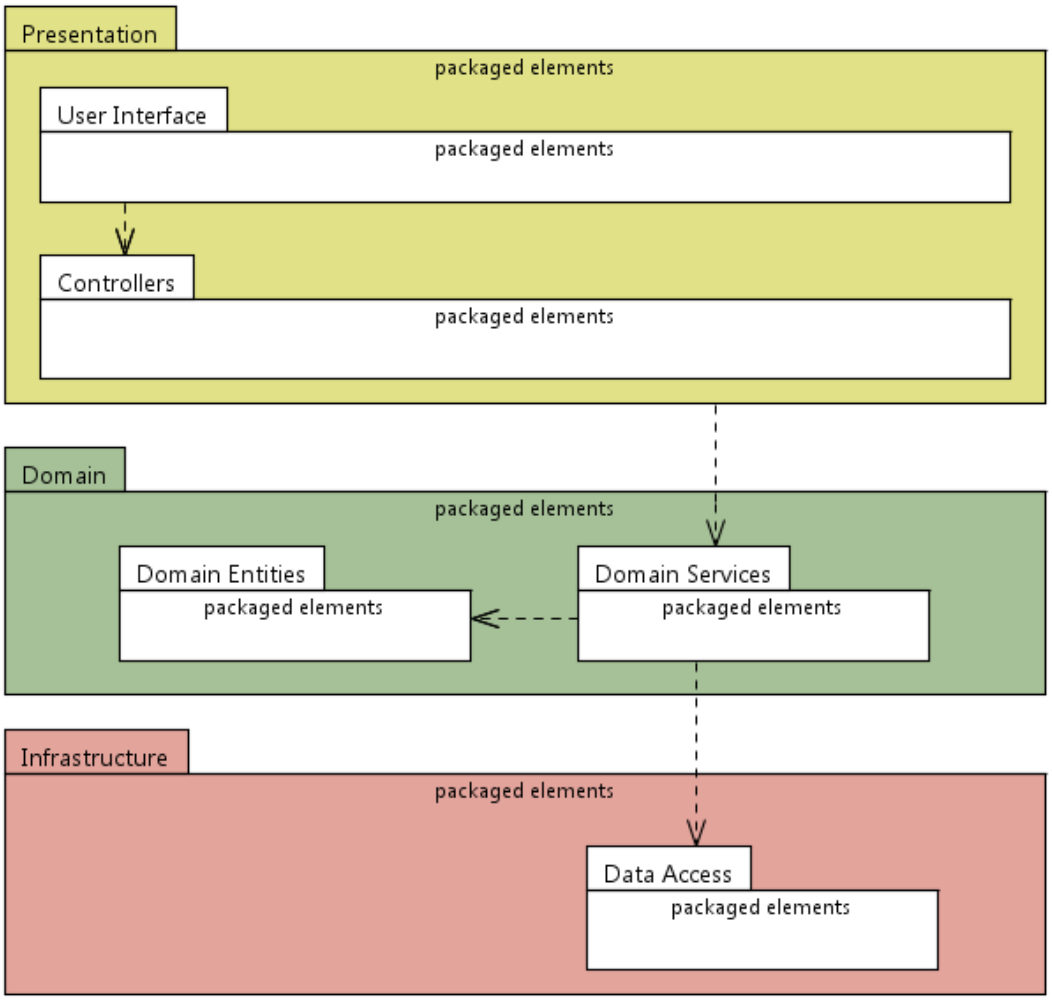


Ilustración 30: Diagrama de Paquetes de IcM implementando el patrón MVC

Cómo se puede apreciar en la Ilustración 30, los patrones de diseño se pueden generalizar y son totalmente independientes del modelado de la aplicación y sus diferentes unidades funcionales, por lo cual se puede aplicar la misma vista lógica para diferentes modelos de aplicación, o aplicar diferentes arquitecturas lógicas al mismo modelo de aplicación, lo que se traduce en ahorro de esfuerzos al modelar con Morphosys.

6.3. Vista Física

La vista física en el diseño arquitectónico describe el mapeo del software en el hardware [2]. Esta vista abarca los nodos que forman la topología de hardware en la que la aplicación se ejecuta. Se enfoca en la distribución, comunicación y aprovisionamiento [62], 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.

El software se ejecuta en una red de computadoras o nodos de procesamiento, los elementos tales como: tareas, procesos y objetos necesitan ser mapeados a los nodos en los cuáles de ejecutan. Estas configuraciones físicas pueden variar según el ambiente (desarrollo, pruebas, producción) o según la tecnología en que se vayan a implementar (.NET, Java, PHP, etc), por eso el software debe ser flexible a la hora de implementar diferentes configuraciones físicas y tener el mínimo impacto posible en el código fuente [62].

En la vista física se deben mostrar también 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.

6.3.1. Representación de la Vista Física con UML

La vista física dentro de Metáfora se representa a través del **diagrama de despliegue** de UML, que muestra la disposición física de los artefactos en un ambiente “real”, en otras palabras, cuáles piezas de software corren o se ejecutan en cuáles piezas de hardware.

El foco principal del diagrama de despliegue en UML 2 son los **artefactos**, que representan una entidad física (o archivo) producido en el proceso de desarrollo de software o en el despliegue de una aplicación. UML provee varios **estereotipos de artefactos** estándar (Ilustración 31) pero frecuentemente cada arquitecto

define sus propios estereotipos o clasificación de acuerdo a la plataforma y el dominio.

Ilustración 31: Estereotipos de Artefactos [63]

Nota: Para el proyecto Metáfora se define una tipificación completa de estos artefactos, que se muestra en el siguiente capítulo, con fines de identificación de requisitos para la construcción de meta-modelo de plataforma.

Los artefactos tienen un atributo que indica su **nombre de archivo** (incluyendo su extensión) y al igual que los paquetes del diagrama de paquetes de desarrollo, también pueden tener **dependencias** y **asociaciones** de composición entre ellos (Ilustración 32, Ilustración 33).

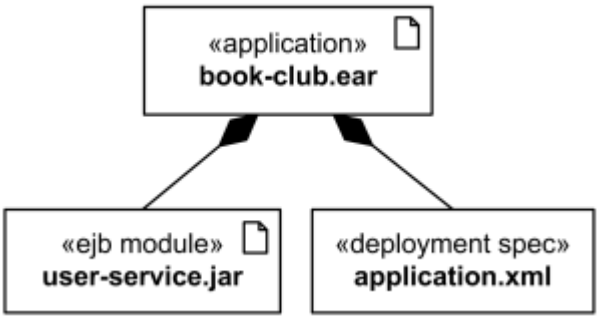


Ilustración 32: Asociación de composición entre Artefactos [63]

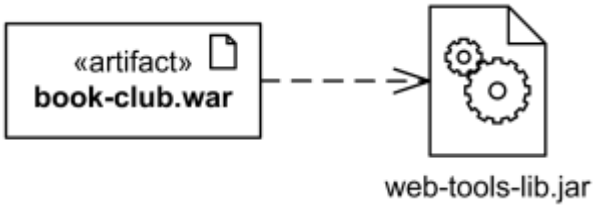


Ilustración 33: Dependencia entre Artefactos [63]

Para indicar la ejecución de los artefactos de software en el hardware, el diagrama de despliegue (Ilustración 34) provee los **nodos** quienes representan un recurso computacional, que puede ser un **dispositivo** físico (tales como: un servidor, un teléfono móvil, etc) o un **ambiente de ejecución** (como un sistema operativo o un contenedor J2EE de Java). Dichos nodos, al igual que los paquetes del diagrama de paquetes de desarrollo, pueden contener otros nodos al igual que artefactos y se comunican también con otros nodos a través de un **camino de comunicación.**

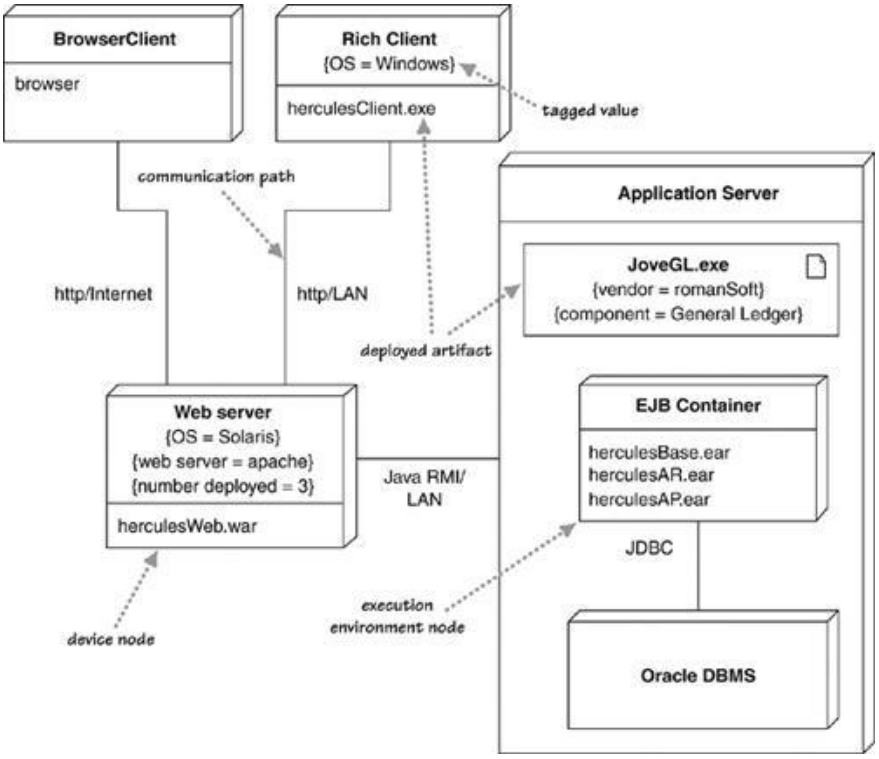


Ilustración 34: Nodos del diagrama de despliegue [16]

Los nodos en el diagrama de despliegue tienen atributos que contienen información importante de la **plataforma** y la **versión** en las que se ejecutan los artefactos.

6.3.2. Consideraciones para la construcción de la Vista Física

Como se mencionó anteriormente, la intención del proyecto es importar los diagramas UML directamente de la fase de diseño de software y usar esos

artefactos como insumo para el modelado. Es por esto que cabe anotar que si bien Morphosys soportará la inclusión de cualquier elemento de la versión 2.0 de UML [61], no todos los elementos tienen un valor semántico para la herramienta, por lo cual este apartado se enfoca especialmente en los elementos que contienen información para alimentar el modelo de la plataforma. Estas son algunas consideraciones a tener en cuenta a la hora de modelar la vista física:

Uno de los principales objetivos del diagrama de despliegue es el reconocimiento de las tecnologías que se van a implementar y las dependencias o caminos de comunicación entre ellas, es por esto que se deben marcar los nodos con las características de cada tecnología o plataforma que van a ejecutar.

En los diagramas de despliegue los artefactos se deben ubicar dentro del nodo en el cual se ejecutan, que en la mayoría de ocasiones es el mismo nodo en el que residen pero no siempre este es así. Este es el caso de los scripts de cliente y las hojas de estilo de una aplicación Web que residen físicamente en el servidor de aplicaciones pero se ejecutan en el navegador que a su vez reside en el nodo del cliente. En la Ilustración 35 se muestra un ejemplo de artefactos asociados a la ejecución y al despliegue.

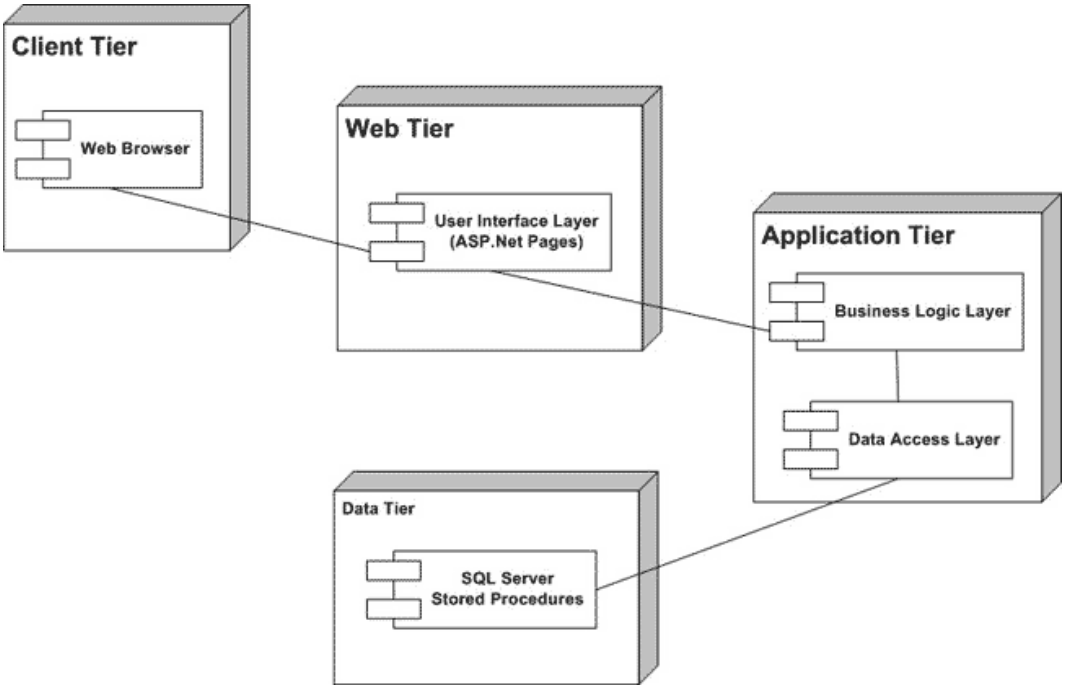


Ilustración 35: Ejecución vs despliegue de Artefactos [68]

Los artefactos también tipifican manifestaciones físicas del software de toda clase de archivos que se despliegan para poner en funcionamiento una aplicación. Entre estos los más comunes son:

- Ejecutables: .exe, binarios, dlls, archivos JAR, assemblies, scripts,...
- Archivos de datos: .dat, .xls, .xml, .msd, .txt,…
- Archivos de configuración: .config, .xml, .ini,...
- Recursos multimedia: imágenes, videos, hojas de estilo (css), iconos, documentos, fuentes …
- Páginas de web: .html, .aspx, .php,…

Para mantener la facilidad de lectura del diagrama, es recomendable incluir sólo aquellos artefactos que representen componentes relevantes y útiles para la generación del código y la documentación del sistema y no repetir el mismo tipo de artefacto innecesariamente. Por ejemplo, si la aplicación usa varias páginas web, basta con diagramar una sola para representar su existencia y peso semántico dentro del flujo de información, pero si hay un tipo de páginas que se comunican con otros componentes diferentes, entonces si es relevante diagramarlas aparte dentro del mismo diagrama. En ocasiones incluso hay archivos que no tienen relevancia semántica dentro del diagrama de despliegue y simplemente se pueden dejar por fuera, este es el caso frecuente de los recursos multimedia.

Es de gran utilidad semántica, en preparación para el modelo de plataforma, que se incluyan las dependencias entre artefactos en el diagrama de modo que el flujo de información sea evidente y esté conectado desde la interacción del usuario hasta la persistencia de datos. No obstante, dada la cantidad de artefactos que resultan normalmente en este tipo de diagramas, se debe tener cuidado de distribuir bien los elementos para que el cruce de líneas no dificulte la facilidad de lectura, esto se logra ubicando cerca los elementos que semánticamente tengan relaciones directas.

6.3.3. Aplicación de la Vista Física

Para el estudio de caso de IcM se presentan a continuación varias aplicaciones de despliegues en diferentes topologías y usando diferentes tecnologías:

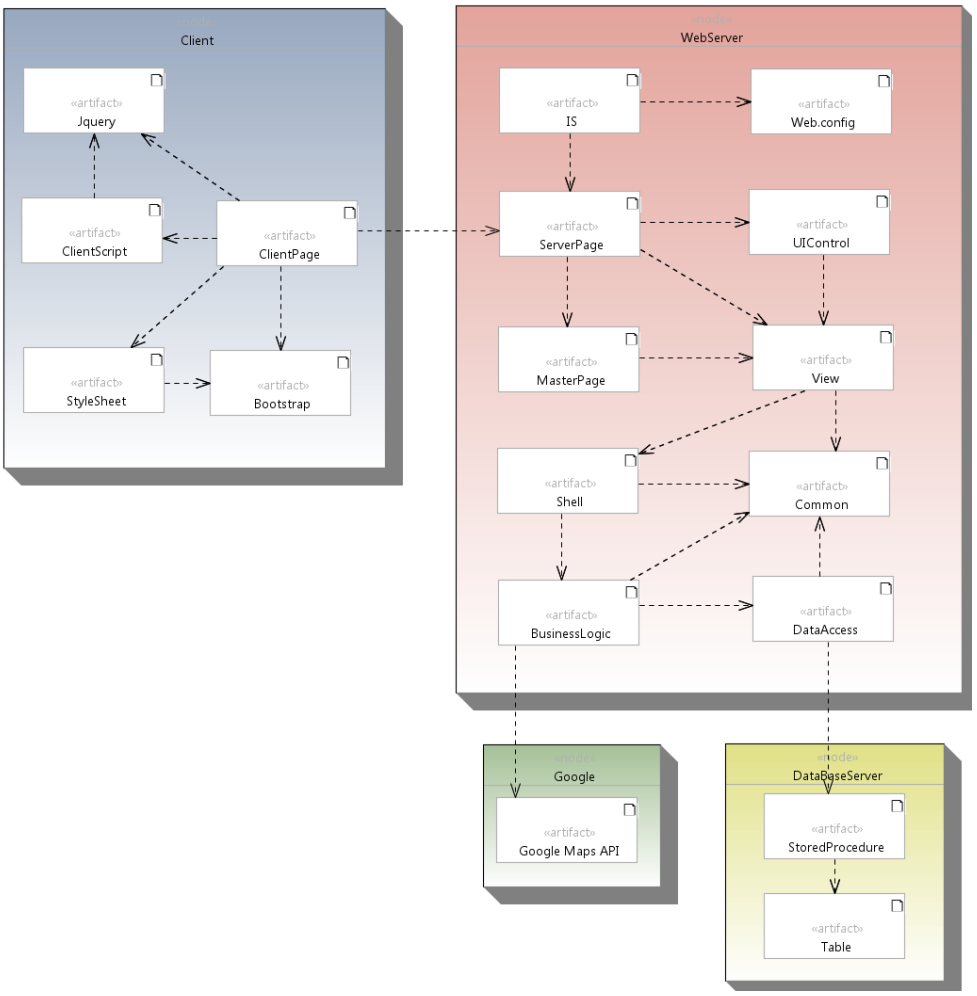


Ilustración 36: Diagrama de despligue de IcM en ASP .NET

La Ilustración 36 muestra el despliegue de la aplicación web en una plataforma .NET (ASP .NET) separando las capas en librerías de clases. La base de datos es Microsoft SQL Server y se usan procedimientos almacenados para la gestión de datos. En el cliente encontramos Jquery con hojas de estilo CSS basadas en Bootstrap y también se utilizan los servicios externos de los API de Google Maps.

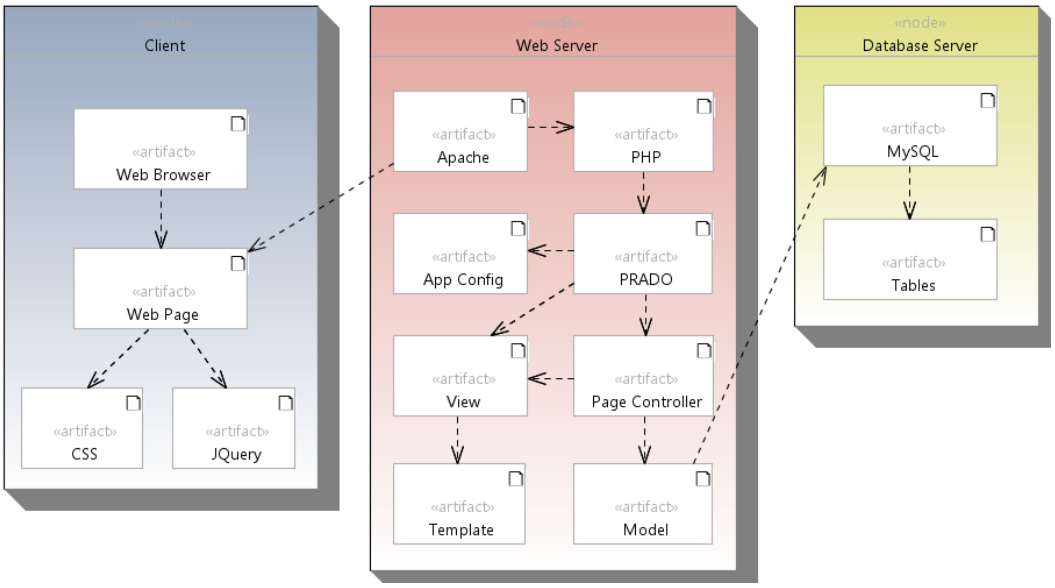


Ilustración 37: Diagrama de despliegue de IcM en PHP con Prado

La Ilustración 37 presenta un despliegue diferente en forma de aplicación web basada en PHP combinado con el framework Prado. En el cliente se usan hojas de estilo CSS y Jquery. La base de datos es MySQL.

6.4. Taxonomía de artefactos de código fuente según su naturaleza

Una vez analizadas las vistas del modelado de plataforma, ya se pueden vislumbrar varias de las características y responsabilidades esenciales de este frente de trabajo dentro del diseño de software. Estos aspectos van a conformar la base del meta-modelo de la plataforma, no obstante, es necesario obtener un mayor detalle sobre las características de los artefactos que se producirán en la generación del código fuente que es el objetivo final de Morphosys.

Muchos son los tipos en los que se pueden agrupar los artefactos según su naturaleza con características y valor semántico para llegar a una generalización independiente de la plataforma. Al no encontrar literatura referente a una tipificación de los artefactos de código fuente, se consultó con varios

desarrolladores y arquitectos con gran experiencia en 8 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 siguiente taxonomía de artefactos según su naturaleza mostrada en la Ilustración 38:

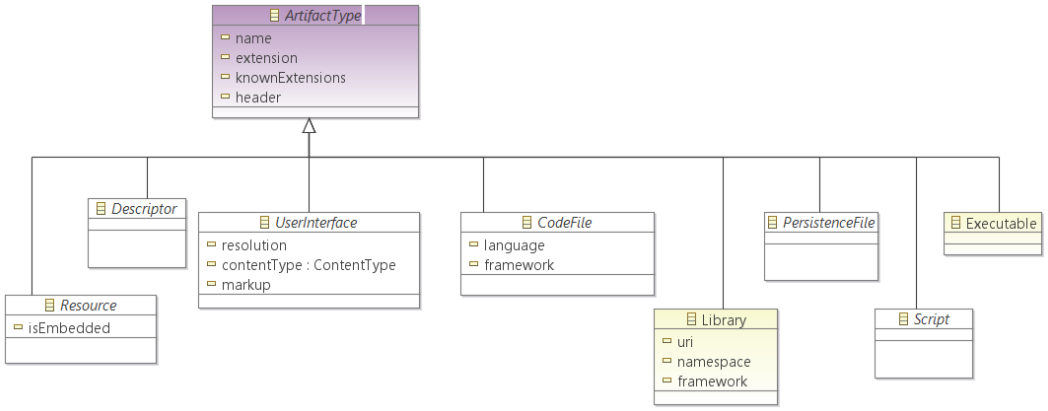


Ilustración 38: Artefactos según su naturaleza

Se identificaron 7 grandes tipos de artefactos que hacen parte de las soluciones en las diferentes tecnologías, a saber:

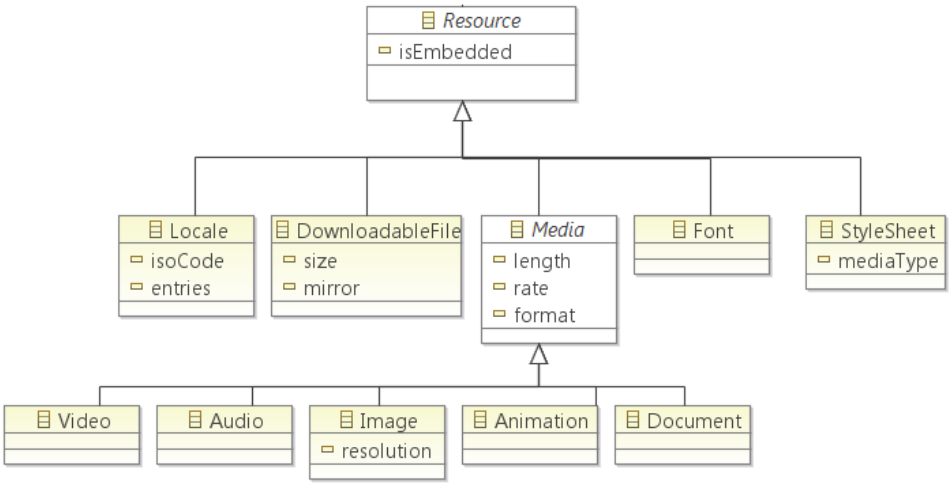


Ilustración 39: Artefactos de tipo recurso

Resource: los recursos o artefactos de apoyo (Ilustración 39) cumplen una función en mayor medida estética (para la interfaz de usuario) y en algunos casos brindan información adicional para los artefactos. Dentro de este tipo encontramos: las fuentes para mejorar la apariencia de la interfaz de usuario, los archivos de localización para presentar la interfaz en varios idiomas y todo tipo de archivos multimedia que se pueden ofrecer en forma de descarga o embeber dentro de la misma interfaz de usuario (tales como imágenes, videos, documentos…). Las extensiones más usadas de este tipo de artefacto son: resx, xml, txt, properties, svg, eot, ttf, woff, otf, zip, msi, exe, rar, css, skin, less, doc, txt, xls, pdf, html, xml, json, rss, mpg, avi, flv, mp4, ogg, web, jpg, gif, png, ico, bmp, cur, svg, mp3, wav, ogg, acc, swf…

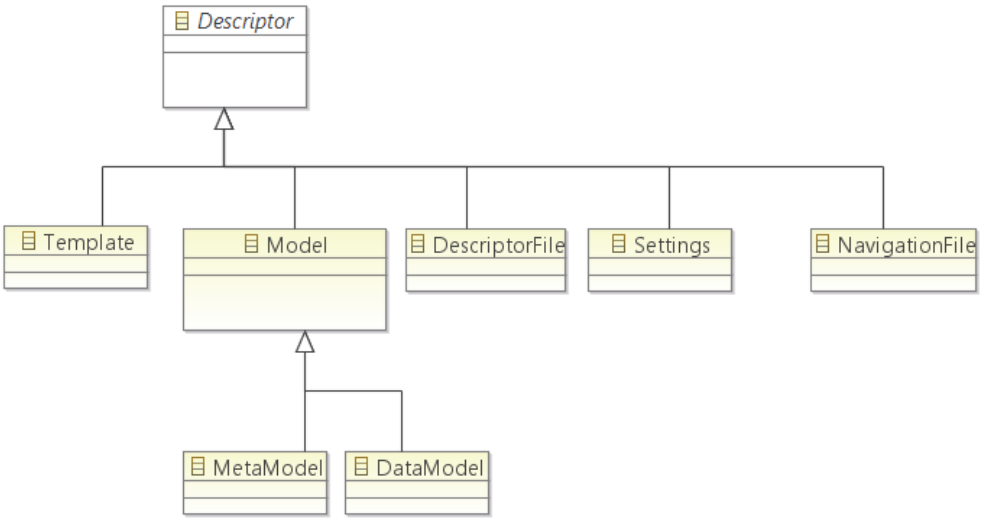


Ilustración 40: Artefactos de tipo descriptor

Descriptor: Los descriptores (Ilustración 40) son artefactos que contienen información acerca de otros artefactos, sus dependencias y la forma en que se agrupan, por ejemplo: los parámetros a usar dentro del código fuente (archivos de configuración), los artefactos contenidos en un paquete (archivos de proyecto o de solución), los archivos que dirigen la ruta de navegación de la interfaz de usuario, los modelos con sus respectivos meta-modelos y las plantillas que describen la estructura de otros artefactos. Algunas de las extensiones conocidas son: rb, tt, dsl, xsd, xslt, dtd, tdl, tpl, template, wsdl, cs, disco, reference, assembly, storyboard, sitemap, config, resx, browser, settings, manifest, ruleset, ini, properties, bs, launch, sh, dtsconfig, csprj, project, sln, ecore, uml, amw, model, modelink, cd, edmx, xsd, dbml, cd…

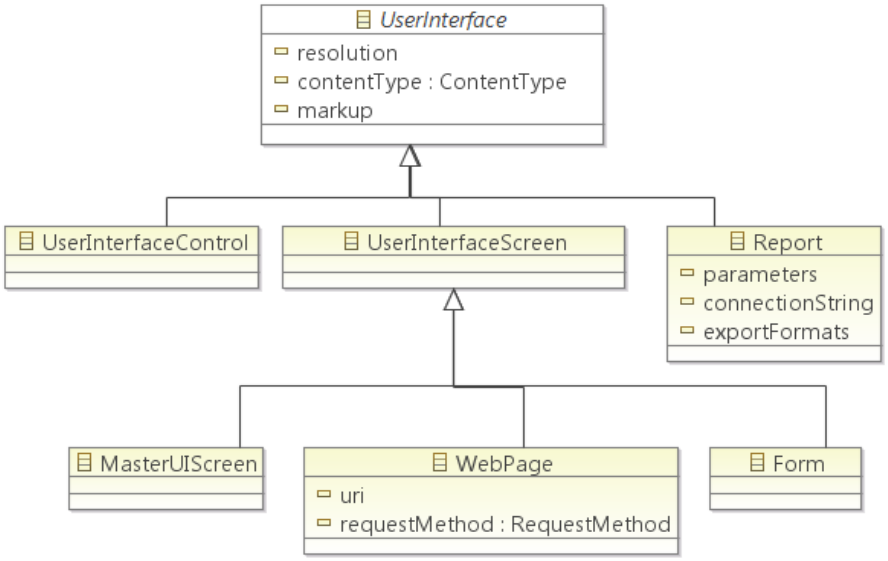


Ilustración 41: Artefactos de tipo interfaz de usuario

UserInterface: Los artefactos de Interfaz de Usuario (UI por sus siglas en inglés) (Ilustración 41) presentan una clara división guiada por las plataformas actuales (web, móvil, embebido…) pero a la vez generalizada en tipos comunes como pantallas de usuario, controles, reportes y páginas web, todas ellos se explican por su mismo nombre. Las extensiones más usadas de este tipo de artefactos son: xaml, ascx, control, vb, cs, xaml, form, rpt, jrxml, master, iframe, mdi, frame, asp, php, aspx, jsp, html, cshtml…

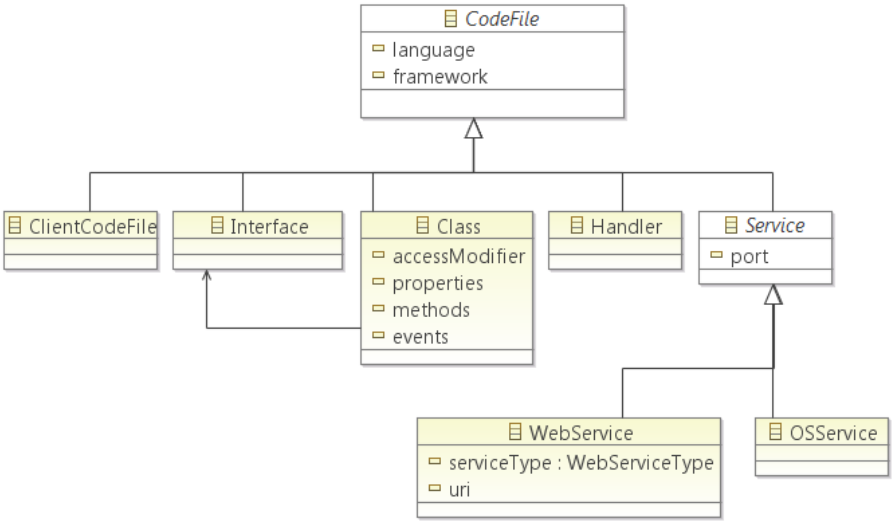


Ilustración 42: Artefacto de tipo archivo de código

CodeFile: Los archivos de este tipo (Ilustración 42) contienen la mayor carga lógica y de procesamiento de la aplicación, son el foco principal de toda solución de código fuente y el elemento esencial para el procesamiento de datos, en otras palabras, contienen el verdadero "código fuente". Entre las extensiones más populares de archivos de código encontramos: cs, vb, java, rb, js, class, ashx, asax, xamlx, asmx, svc, xamlx, vscript, js, ts, coffee, bat, job, wsf, dtsx, sql…

Library: Las librerías son agrupadores de otro tipo de código fuente, típicamente otro archivo de código que comparten una funcionalidad o sentido común. Existen librerías codificadas en el mismo proyecto y también librerías externas conocidas como APIs.

Executable: son artefactos similares a las librerías con la característica agregada de que pueden ser ejecutados sin depender de otros artefactos, este tipo de archivos son el punto de entrada para la ejecución de la mayoría de las aplicaciones, su extensión más conocida es .exe.

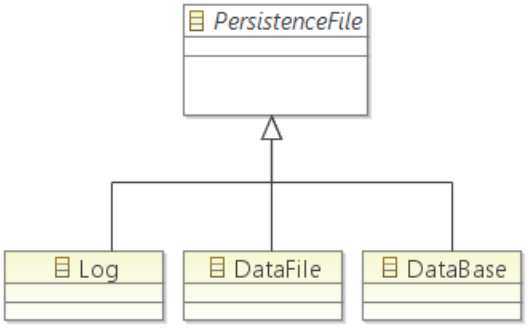


Ilustración 43: Artefactos de tipo archivos de persistencia

PersistenceFile: Los archivos de datos cumplen fines de persistencia (Ilustración 43), tales como el almacenamiento de logs o incluso pueden servir como la base de datos de la aplicación cuando no se cuenta con un motor externo robusto. Algunas de sus extensiones populares son: csv, xls, sdf, mdf, xsd, log.

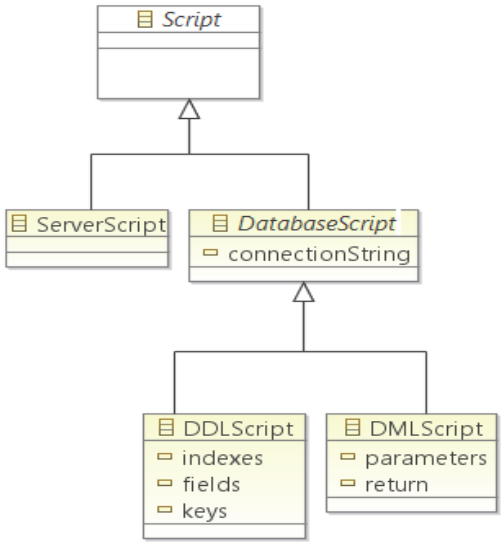


Ilustración 44: Artefacto de tipo script

Scripts: Los Scripts (Ilustración 44) son otro tipo de archivo de código que no hace parte de un framework de ejecución y por lo general su rutina es ejecutada una sola vez o son programadas para cada cierto tiempo. Dentro de estos encontramos los script de servidor (tales como archivos batch o jobs) y los script de base de datos que cumplen una labor fundamental en las capas de persistencia. Los artefactos de este tipo son a menudo separados del resto de artefactos de la solución.

7. CONSTRUCCIÓN DEL DSL

Una vez conocidas las responsabilidades de cada vista, los elementos de los diagramas UML de la plataforma y la taxonomía de artefactos de código fuente según su naturaleza, ya se cuenta con elementos suficientes para estructurar el DSL a partir de los modelos UML (lógico y físico) más un mecanismo de mapeo entre ambos (que se explorará en el siguiente capítulo). De esta manera se inicia la construcción del modelo de plataforma que sintetiza todos aquellos aspectos no funcionales comprendidos por las 2 vistas en cuestión.

Nota: Es importante aclarar que la construcción de los diagramas UML de paquetes de desarrollo (vista lógica) y despliegue (vista física) dentro de Morphosys se realiza con Papyrus, un modelador grafico de EMF, cuya sintaxis abstracta equivale al meta-modelo de UML versión 2.4.1 publicado en http://www.eclipse.org/uml2/4.0.0/UML.

Como se mostró en el marco referencial, todo lenguaje de modelado está compuesto por tres elementos: sintaxis abstracta, sintaxis concreta y semántica.

Nota: Algunos autores mencionan también un mecanismo de serialización como un cuarto componente de los lenguajes de modelado. El mecanismo de serialización usado en Morphosys son los formatos XMI y .model en los que se construyen y se transforman los modelos de Epsilon y EMF.

7.1. Sintaxis Abstracta

La sintaxis abstracta describe la estructura del lenguaje y la forma en que sus elementos se combinan, independientemente de una representación particular [6]. La representación de la sintaxis abstracta de un lenguaje de modelado se realiza con un meta-modelo, que no es más que otro lenguaje para expresar un modelo [12] y que sigue el estándar MOF [69].

En el caso del DSL de plataforma se requiere un meta-modelo propietario. Para este fin EMF provee un meta-meta-modelo llamado Ecore (Ilustración 45) que es una evolución de MOF [70]:

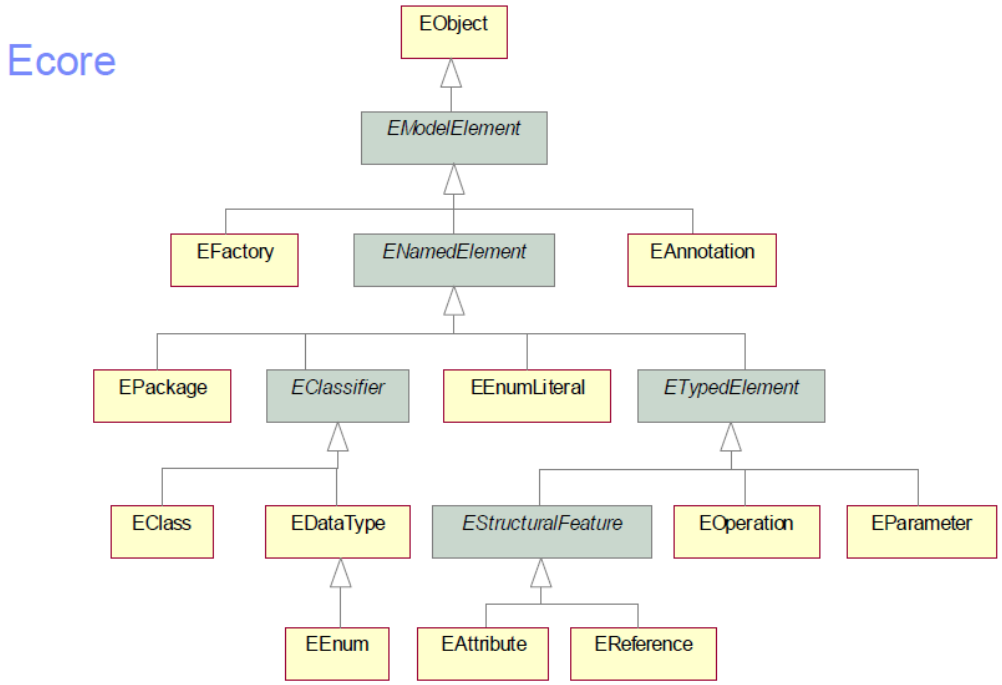


Ilustración 45: Estructura de Ecore (EMF) [70]

El meta-modelo de la plataforma debe sintetizar los aspectos no funcionales del código fuente (patrones y características de despliegue) según la plataforma tecnológica.

Aplicando uno de los patrones más populares en el diseño de software moderno como es el diseño por capas, caracterizamos los contenedores de la vista lógica en **layers** y los de la vista física en **tiers** (Ilustración 46) [64].

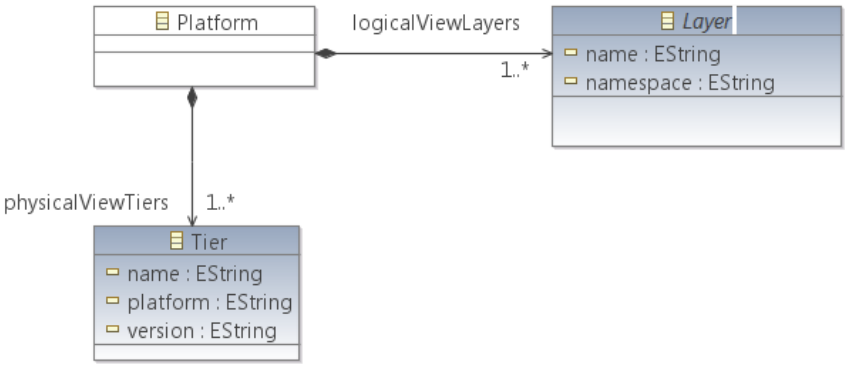


Ilustración 46: Layers y Tiers en el meta-modelo de plataforma

Apoyados en el análisis de las vistas realizado en el capítulo anterior, podemos concluir que de la vista lógica se derivan los siguientes aspectos:

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

De la misma manera podemos distinguir los componentes que pueden ser producto de la vista física:

- 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.

Así podemos extender el meta-modelo en cuestión con layers que representan los paquetes y artefactos contenidos dentro de las tiers (o nodos en el diagrama de despliegue) como se aprecia en la Ilustración 47.

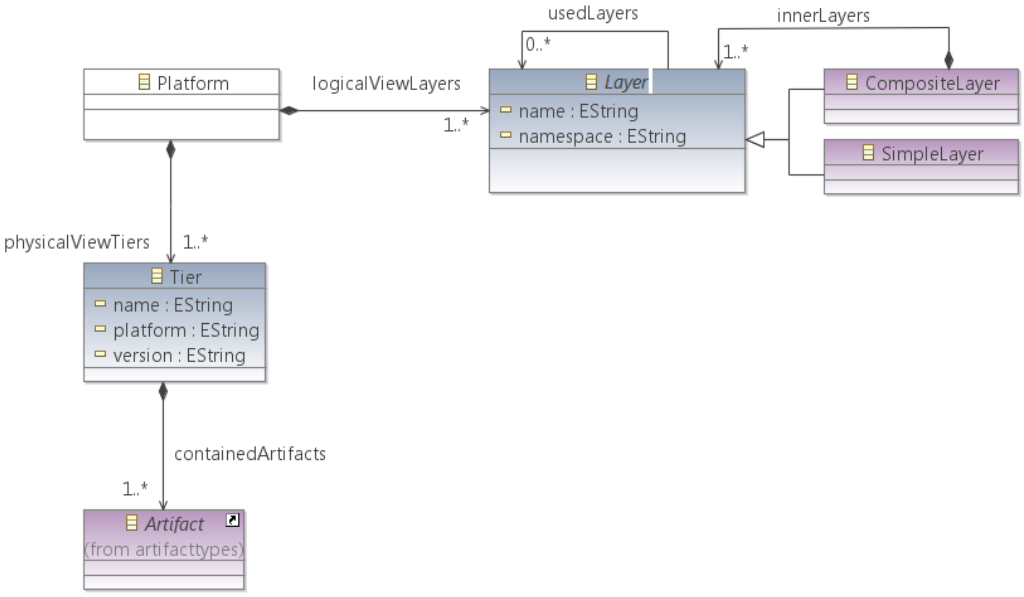


Ilustración 47: Características de Layers y Tiers en el meta-modelo de la Plataforma

Adicionalmente, cabe resaltar que como las layers se pueden anidar formando una jerarquía, es necesario implementar un patrón para soportar este comportamiento. En este caso se implementa el patrón composite, dando como resultado paquetes simples y compuestos (que contienen otros paquetes) [71].

Esto nos deja con un meta-modelo que incluye aspectos tanto físicos como lógicos pero de manera aislada, contrario al enfoque integral que propone Metáfora.

Algunos autores introducen el concepto de manifestación, en el que una layer o capa lógica se realiza en una o varias tiers o capas físicas a través de relaciones de manifestación que implican el uso de artefactos o componentes. Incluso algunos autores [63] mencionan la necesidad de un diagrama intermedio llamado el **diagrama de manifestación** (Ilustración 48)**.**

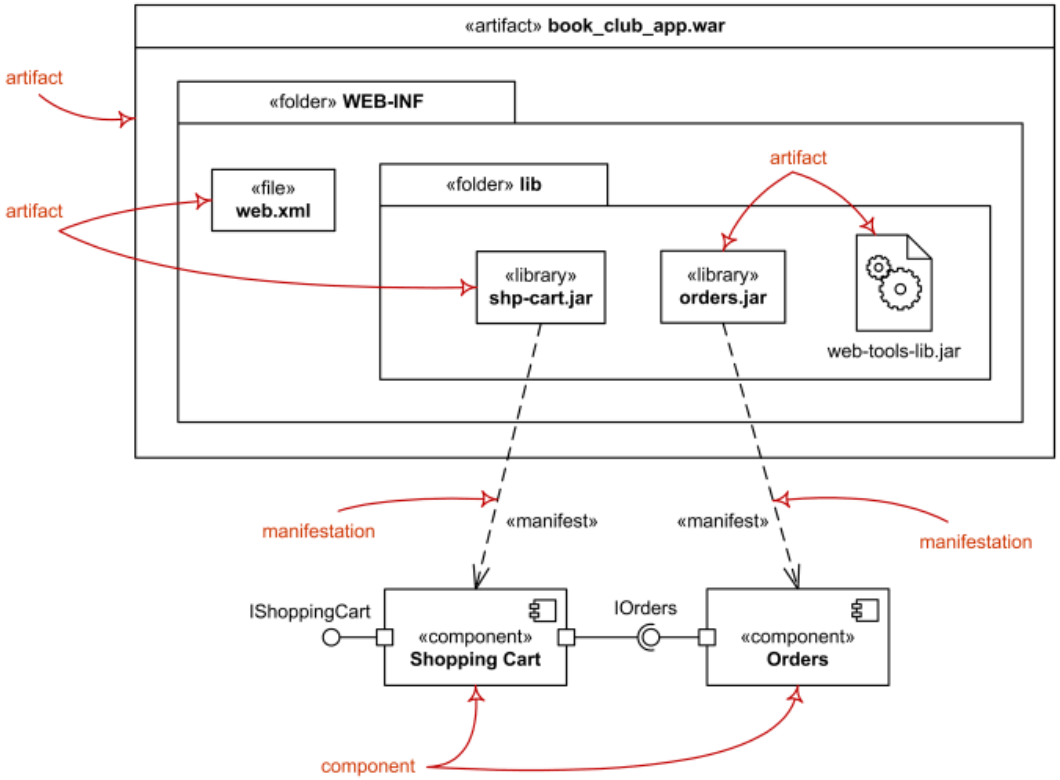


Ilustración 48: Diagrama de Manifestación UML [63]

Dado que este diagrama aún no hace parte de la especificación oficial de UML y que la intención de Metáfora es portar los diagramas tal cual como vienen de la fase de diseño, no se soportará directamente en la primera versión de Morphosys. No obstante, el concepto de manifestación es el candidato perfecto 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 (ver Ilustración 49).

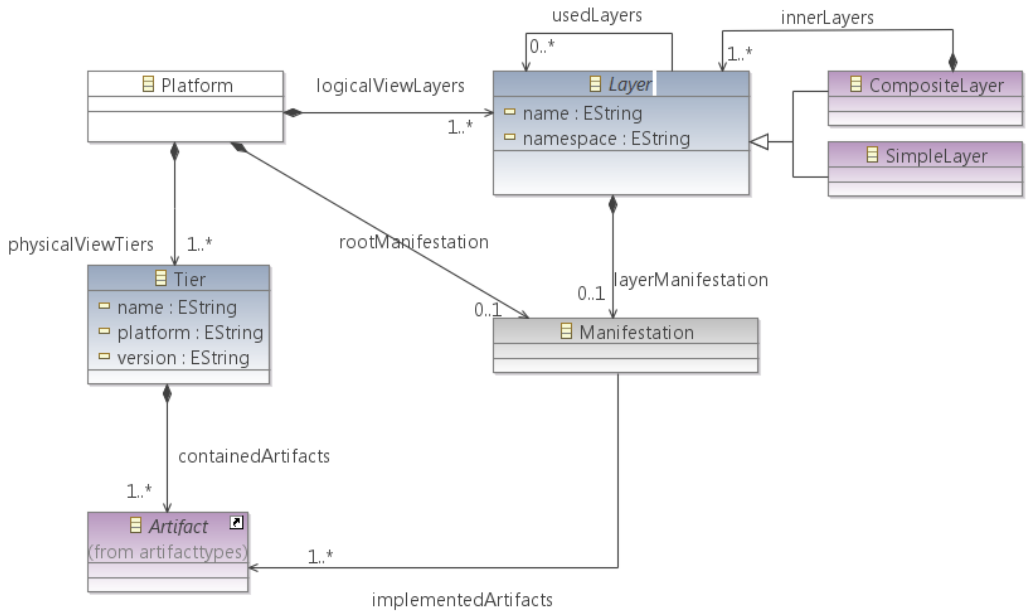


Ilustración 49: Manifestación en el meta-modelo de la plataforma

Estos artefactos poseen diferentes particularidades con un gran valor semántico de cara a la generación de código fuente, es por esto que es necesario analizarlo desde diferentes puntos de vista como se muestra en la Ilustración 50:

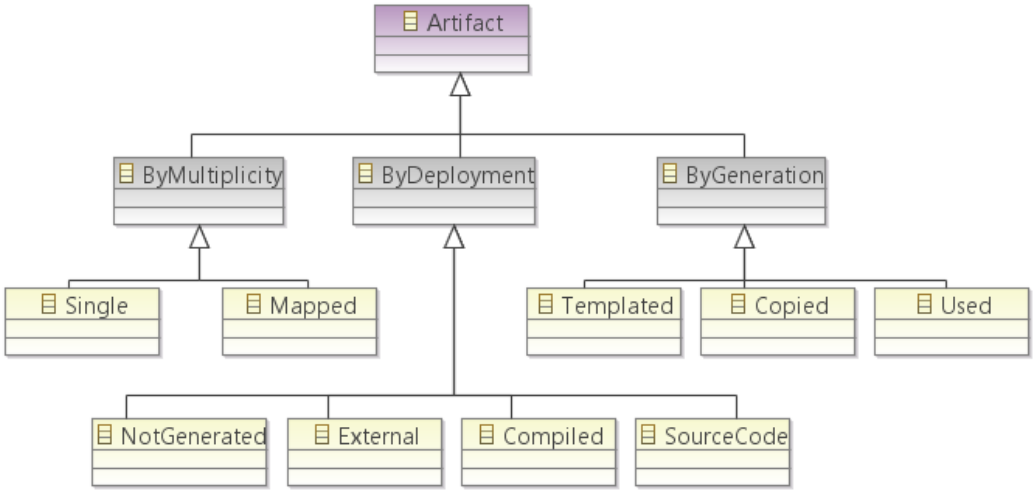


Ilustración 50: Clasificación de artefactos desde diferentes puntos de vista

Según su multiplicidad: un artefacto puede tener una sola instancia, como es el caso de los archivos de configuración, o tener múltiples instancias mapeadas a varios elementos del modelo de la 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 usado en el despliegue.

Según el despliegue: de acuerdo a como se representan los artefactos en el diagrama de despliegue encontramos: artefactos compilados que son compuestos de fuentes, artefactos externos que no están alojados en el repositorio local y los artefactos no generados que se representan a nivel de diseño pero no corresponden a ninguna instancia en el código fuente sino típicamente a entidades en tiempo de ejecución.

De acuerdo a la anterior clasificación y según la manera en la que se despliegan los artefactos, podemos integrar los cuatro tipos principales al meta-modelo de plataforma, como se muestra en la Ilustración 51: fuente, compilado, externo y no generado. Para los artefactos compilados (representados en el diagrama de despliegue) debe existir una correspondencia con varios archivos fuente que estén contenidos dentro del compilado. A su vez los artefactos de tipo fuente pueden ser copiados directamente de otra ubicación o responder a una plantilla de generación.

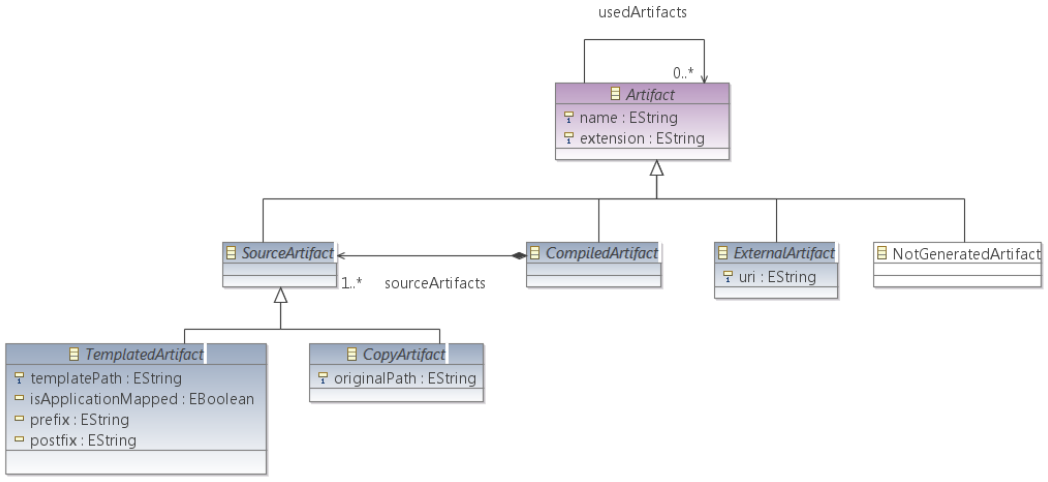


Ilustración 51: Tipos de artefactos en el meta-modelo de plataforma

Para finalizar se cruzan los tipos de artefactos mostrados en la Ilustración 51 con los de la clasificación según la naturaleza que se realizó en el capítulo anterior y que se muestra en la Ilustración 52.

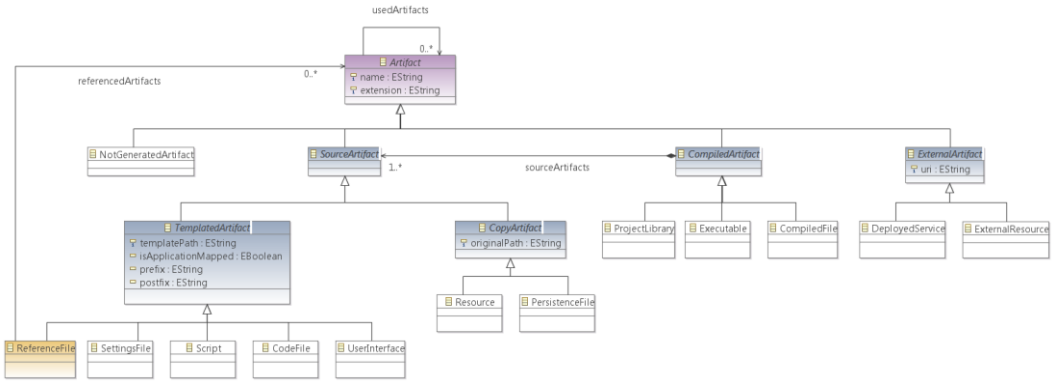


Ilustración 52: Tipos de artefactos específicos del meta-modelo de Plataforma

Nota: Cabe resaltar que la taxonomía de artefactos presentada en la Ilustración 52 hace parte del meta-modelo de plataforma y la herencia es importada de un diagrama externo por facilidad de visualización.

7.2. Sintaxis Concreta

La sintaxis concreta describe la representación específica de un lenguaje de modelado, cubriendo aspectos visuales o de codificación, y está directamente ligada con la expresividad y complejidad de los modelos. A la hora de construir una sintaxis concreta se deben tener en cuenta los siguientes aspectos: capacidad de escritura, capacidad de lectura, facilidad de aprendizaje y efectividad [15].

La sintaxis concreta del modelado de plataforma en Morphosys está dada por tres elementos que dan cumplimiento a las actividades de modelado presentadas en el flujo de trabajo de la plataforma de la

Ilustración 23 (**Modelar plataforma (UML), Parametrizar transformación de plataforma**):

1. El modelado grafico de los diagramas UML de desarrollo y despliegue se realiza utilizando el editor gráfico Papyrus que responde a la sintaxis concreta de UML versión 2.4.

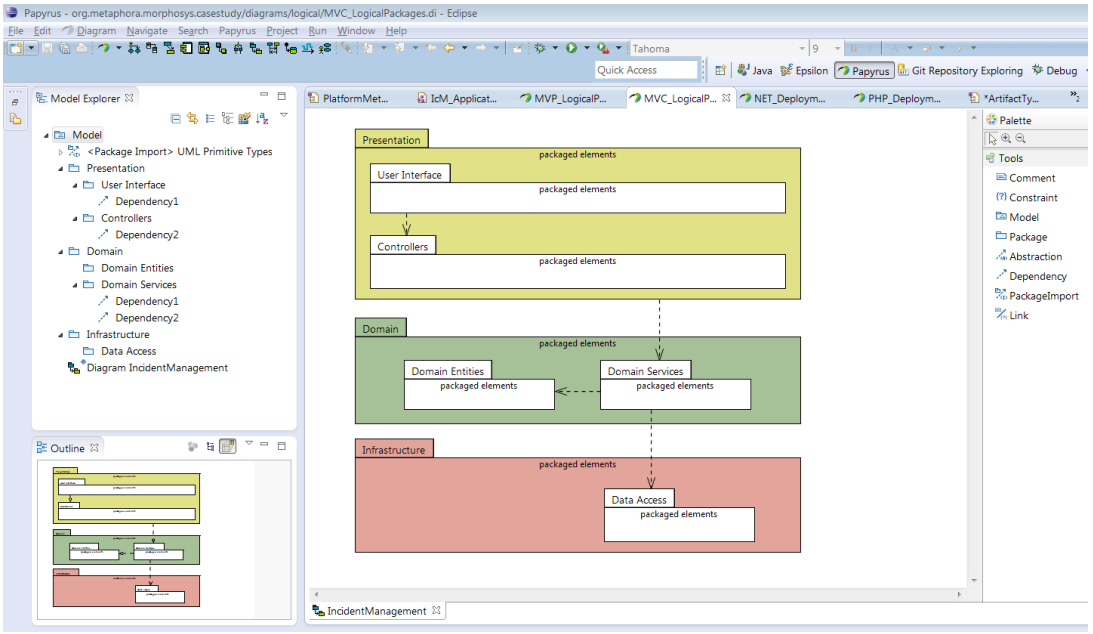


Ilustración 53: Editor Papyrus de eclipse

Con el modelado de las vistas lógica y física en Papyrus (Ilustración 53) podemos empezar a consolidar el proceso de modelado de la plataforma de Morphosys como se muestra en la Ilustración 54

Ilustración 54 con la generación de los diagramas de paquetes de desarrollo y despliegue basados en el meta-modelo de UML versión 2.

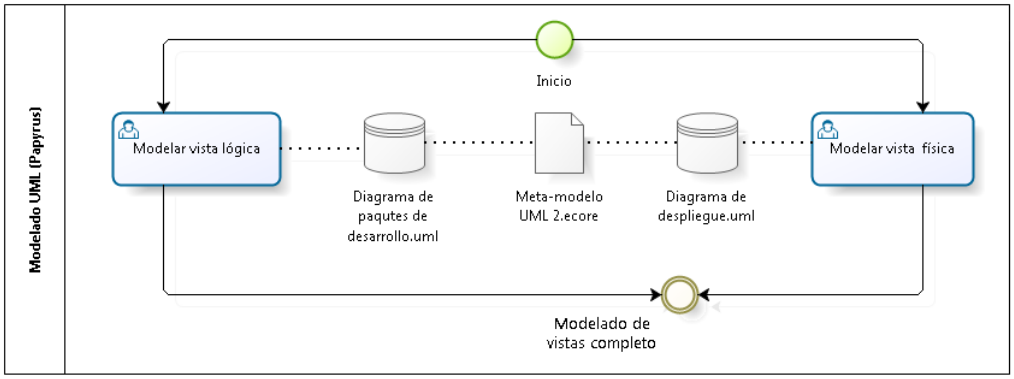


Ilustración 54: Modelado de vistas lógica y física dentro del proceso de modelado de plataforma en Morphosys

2. La representación de modelos (instancia de los meta-modelos), tipifica la sintaxis concreta de los modelos EMF presentada en editores de tipo árbol como **Exeed** (Ilustración 55) en el caso de Epsilon.

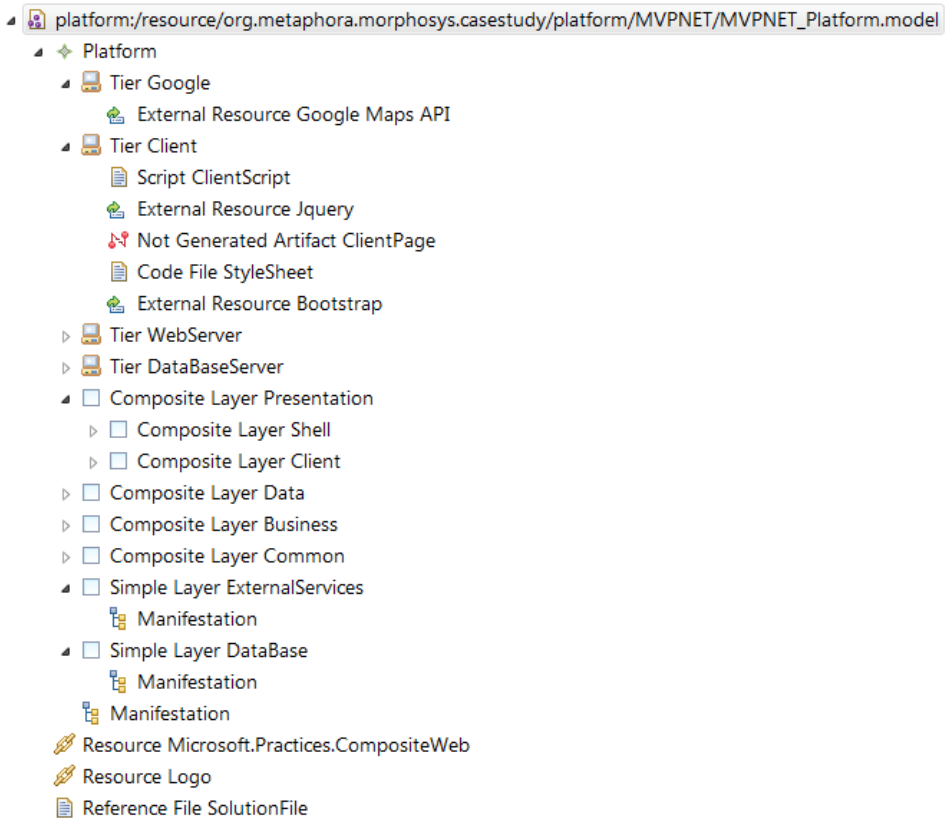


Ilustración 55: Editor de modelos Exeed

3. La tercera forma de sintaxis concreta presente en el modelado de la plataforma es la de editor de Weaving que se detallará en el capítulo siguiente.

7.3. Semántica

La semántica se refiere a la interpretación de un lenguaje, lo que le da sentido a la sintaxis concreta construida a partir de la sintaxis abstracta. Para el modelo de plataforma la semántica está dada por el sentido que tiene cada uno de los elementos del meta-modelo de plataforma y las reglas de validación a las que se conforman. Estas son algunas de las reglas que aplican para dichos elementos:

Para la vista lógica:

- Los paquetes deben tener un nombre.
- Se sugiere que los paquetes tengan un namespace, de lo contrario el namespace se genera de la estructura de carpetas.
- El nombre de un paquete no se puede repetir dentro del mismo paquete contenedor.
- No se pueden crear referencias circulares entre paquetes.
- Los paquetes no pueden tener dependencias duplicadas (mismo origen y destino).
- Las dependencias solo son válidas entre paquetes.
- Sólo se admite un paquete (modelo) raíz en cada modelo, tanto de la vista lógica como de la física.
- Los paquetes deben tener un nombre.

Para la vista física:

- Los nodos deben tener un nombre.
- Como las anidaciones entre nodos no están soportadas en el modelo de plataforma, no se recomienda tener nodos compuestos.
- Los artefactos deben tener nombre o un nombre de archivo.
- Los artefactos deben tener una extensión de archivo.
- Los artefactos deben estar dentro de un nodo.
- El nombre de un artefacto no se puede repetir dentro del mismo nodo.

- El nombre de archivo (incluyendo su extensión) de un artefacto no se puede repetir dentro del mismo nodo.
- Los artefactos no pueden tener dependencias duplicadas (mismo origen y destino).
- Las dependencias solo son válidas entre artefactos.
- Los artefactos deben tener un tipo asignado.
- Un artefacto no puede manifestar más de 1 paquete.
- Los artefactos de primer nivel en una manifestación deben tener un artefacto UML asociado.
- Un paquete no puede tener más de una manifestación.
- Un artefacto compilado debe contener al menos 1 artefacto de código fuente.
- Un artefacto compilado no puede contener otros artefactos compilados.
- Un artefacto de código fuente no puede contener otros artefactos.

8. INTEGRACIÓN DE LAS VISTAS

8.1. Mecanismos para combinar modelos

Con el meta-modelo de plataforma definido y las características de las vistas lógica y física identificadas, aun se requiere integrar los elementos provenientes de las dos vistas mediante un mapeo directo de sus elementos.

Un mecanismo que permita la combinación automática de modelos basado en una serie de correspondencias preestablecidas tiene un número de aplicaciones en el proceso de MDSD [72], por ejemplo se puede usar para unificar dos modelos que sean complementarios pero potencialmente superpuestos y que describan diferentes vistas del mismo sistema, como es el caso del modelado de la plataforma en el proyecto Metáfora. En otro escenario puede ser usado para combinar un modelo *core* con un modelo basado en aspectos (potencialmente conformes a diferentes meta-modelos) como se discute en [73] donde un modelo *core* independiente de plataforma es combinado con otro modelo que aporta los aspectos específicos de plataforma.

El ***model weaving*** [74] es una operación genérica que establece correspondencias directas a un nivel de granularidad detallado entre varios modelos que típicamente responden a diferentes meta-modelos, 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 cuales artefactos manifiestan cada *layer*.

Es necesario entonces definir un meta-modelo intermedio de weaving (Ilustración 56) para la plataforma que contenga los elementos comunes con los que se puedan expresar las manifestaciones. Así mismo, 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 entonces 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.

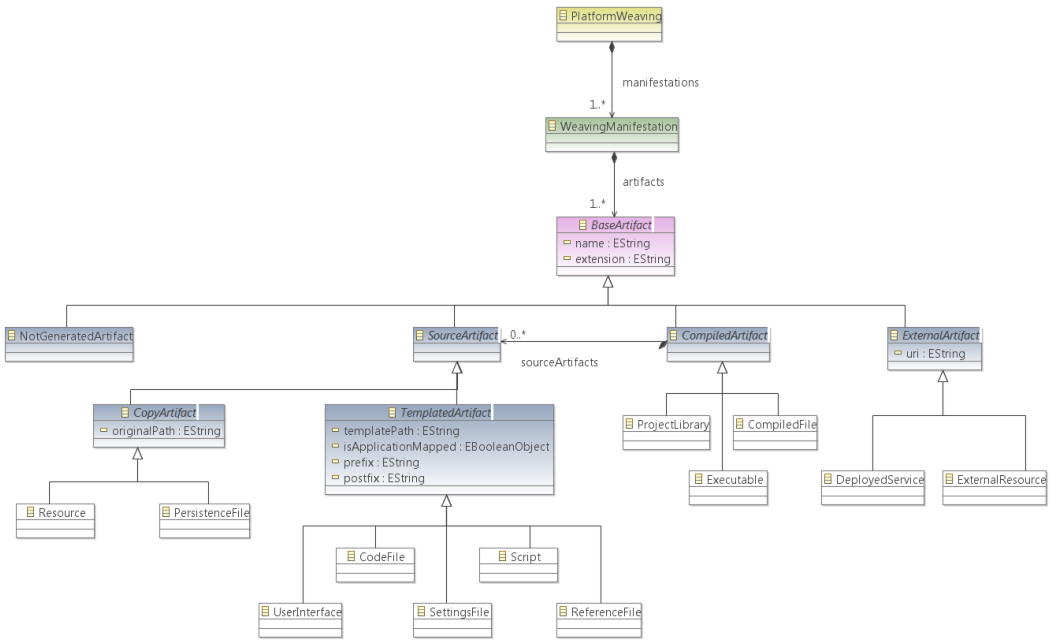


Ilustración 56: Meta-modelo de weaving de plataforma

Es una verdad aceptada que la ejecución repetitiva de tareas manuales es contra producente y susceptible a errores [75]. Por otro lado, el no completar dichas tareas de manera correcta y precisa compromete la consistencia, y por lo tanto la calidad, de los modelos. En MDSD esto cobra una relevancia particular ya que los modelos se usan para producir automáticamente sistemas funcionales o parte de ellos.

Las herramientas de modelado contemporáneas proveen transformaciones integradas (asistentes) para automatizar las tareas repetitivas comunes. Sin embargo, de acuerdo a la arquitectura del sistema diseñado y al dominio específico del problema, aparecen nuevas tareas repetitivas adicionales, que no pueden ser manejadas por los asistentes integrados en la herramienta de modelado. Para abordar el problema de automatización a nivel general, lo

usuarios deben tener la capacidad de definir transformaciones de manera sencilla y que cubra sus necesidades específicas.

Lo anterior puede ser logrado usando la arquitectura extensible que las herramientas de modelado actuales ofrecen, pudiendo agregar funcionalidad mediante el uso de scripts o la implementación de algún lenguaje de modelado. No obstante, como se discute en [76], la mayoría de las herramientas de modelado proveen un API a través del cual expone un modelo editable con una curva de aprendizaje significativa y con características propietarias que impiden su uso en otras herramientas. Por otra parte, los lenguajes de tercera generación como Java y C++ no se prestan para recorrer y modificar modelos [76], y los lenguajes de modelado existentes como QVT, ATL y ETL, no pueden ser usados directamente para este propósito, ya que han sido diseñados para operar en manera de batch sin ninguna intervención del usuario en el proceso. Esto contrasta con la necesidad de intervenir las transformaciones en MDSD y es por esto que se hace necesaria una herramienta para que el usuario pueda construir el modelo de weaving de plataforma.

Epsilon provee una herramienta llamada **ModeLink** (Ilustración 57) 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 y permitir al usuario especificar la información adicional requerida de cara a la generación del modelo de plataforma.

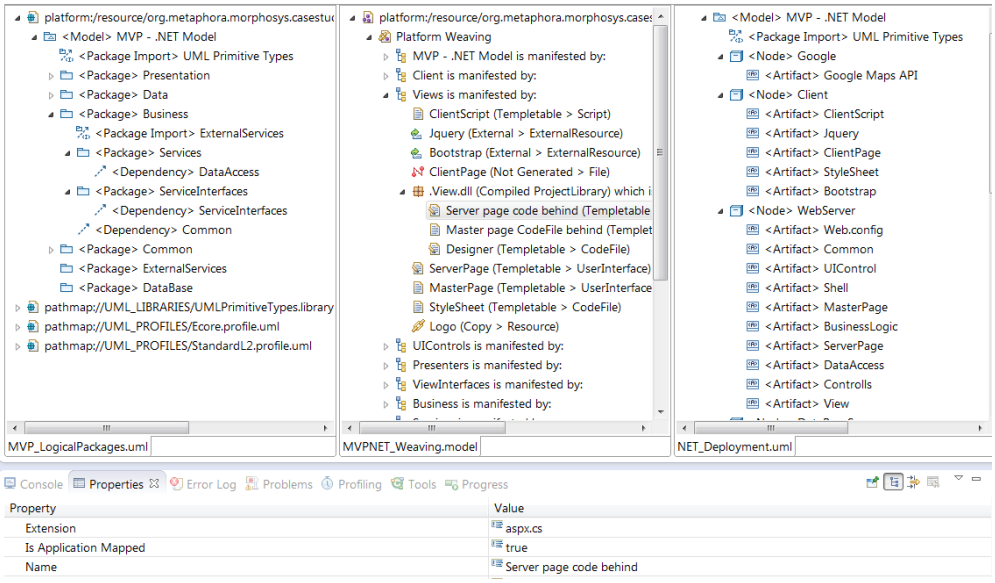


Ilustración 57: Editor ModeLink del weaving de plataforma

Al integrar el modelado de weaving de plataforma dentro del proceso de modelado de la plataforma de Morphosys se amplía el panorama de la presente propuesta, incluyendo el editor de weaving con su respectivo meta-modelo como se aprecia en la Ilustración 58.

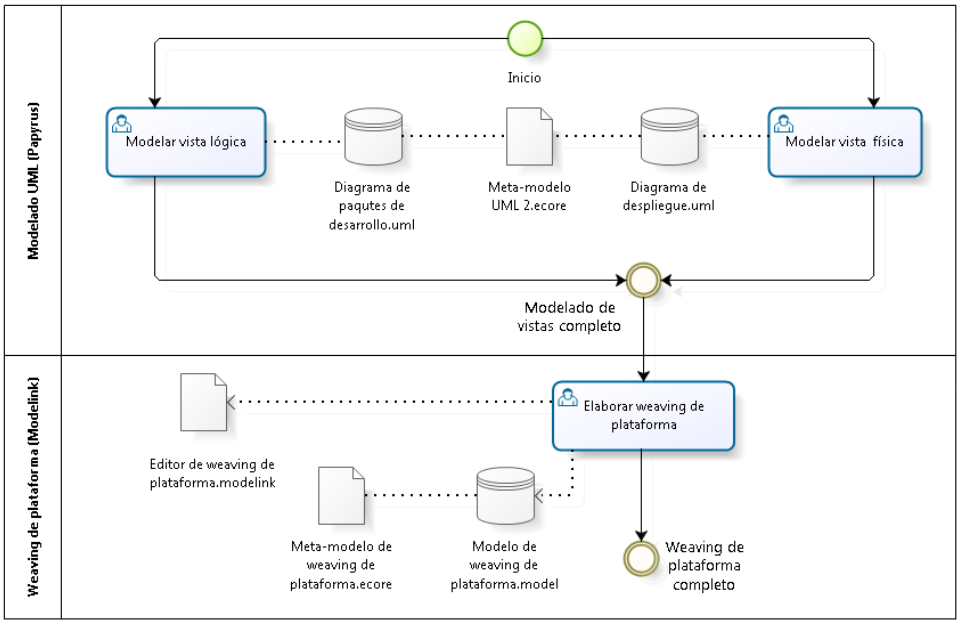


Ilustración 58: Modelado de weaving de plataforma dentro del proceso de modelado de plataforma en Morphosys

8.2. Transformación M2M

De los tipos de transformaciones M2M planteadas en [77] se destacan dos de ellas: mapeo y actualización. Las transformaciones de mapeo típicamente traducen uno o más modelos de origen en un conjunto de modelos de destino expresados en diferentes lenguajes de modelado mediante la creación de cero o más elementos en el modelo de destino por cada elemento en el modelo de origen. Por otro lado, las transformaciones de actualización realizan modificaciones en el mismo modelo de origen.

Para generar el modelo de plataforma en Morphosys se cuenta con tres insumos: los dos modelos UML que representan la vista lógica y física más el modelo de weaving de plataforma que contienen los enlaces entre estas vistas y la información adicional de los artefactos que no se puede expresar en los diagramas UML, por lo tanto la estrategia de transformación M2M se puede definir con la siguiente formula (Ilustración 59):

Ilustración 59: Estrategia de transformación M2M

Para combinar los tres modelos de origen no es suficiente aplicar una transformación de mapeo tradicional sino que se debe aplicar una transformación de modelos basado en los enlaces especificados en el modelo de weaving de plataforma.

Algunas investigaciones [78] [79] han demostrado que la combinación de modelos se puede descomponer en cuatro fases: comparación, conformidad de verificación, mezcla y restructuración (Ilustración 60).

Ilustración 60: Fases del weaving del modelos

Fase de comparación: en la fase de comparación, las correspondencias entre elementos equivalentes de los modelos fuentes son identificados de tal manera que dichos elementos no sean duplicados en el modelo de destino.

Fase de conformidad de verificación: en esta fase, los elementos que han sido identificados como equivalentes en la fase previa son examinados para verificar la conformidad entre ellos. El propósito de esta fase es identificar conflictos potenciales que puedan tornar inviable la combinación de modelos. La mayoría de los enfoques propuestos, tal como [80], abordan la verificación de conformidad de modelos que responden al mismo meta-modelo.

Fase de combinación: se han propuesto varios enfoques para la fase de combinación. En [78] [81] se usan algoritmos basados en grafos para combinar modelos del mismo meta-modelo. En [80] es presentado un proceso interactivo para combinar modelos de UML 2.0. Hay por lo menos dos debilidades en los modelos presentados hasta el momento: por un lado solo se encargan de resolver la necesidad de combinar modelos del mismo meta-modelo y por otra parte el algoritmo de combinación que usan es inflexible y no proveen mecanismos de personalización. En el caso de Morphosys la transformación soporta los diferentes meta-modelos (UML de los diagramas de paquetes y despliegue, y el weaving de plataforma), además de que las reglas de transformación M2M se pueden intervenir y presentan un esquema estructurado que cubre todos los posibles casos de enlaces entre los diferentes modelos.

Fase de restructuración: después de la fase de combinación, el modelo de destino puede contener inconsistencias que necesiten arreglarse. En el paso final del proceso, estas inconsistencias se pueden eliminar y pulir el modelo hasta su forma final. Aunque la necesidad de la restructuración es discutida en [79] [80], en la literatura relacionada este asunto no es tratado específicamente.

Dentro de la familia de lenguajes de Epsilon existe soporte para cada una de las etapas del proceso de combinación de modelos. ECL permite definir los criterios de comparación entre los elementos enlazados, EVL sirve para validar dichos elementos, mientras que EML 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 pueden crear reglas de transformación similares a las creadas EML pero usando ETL de manera que se realizan transformación de mapeo directas mediante la importación en las transformaciones EML (Ilustración 61).

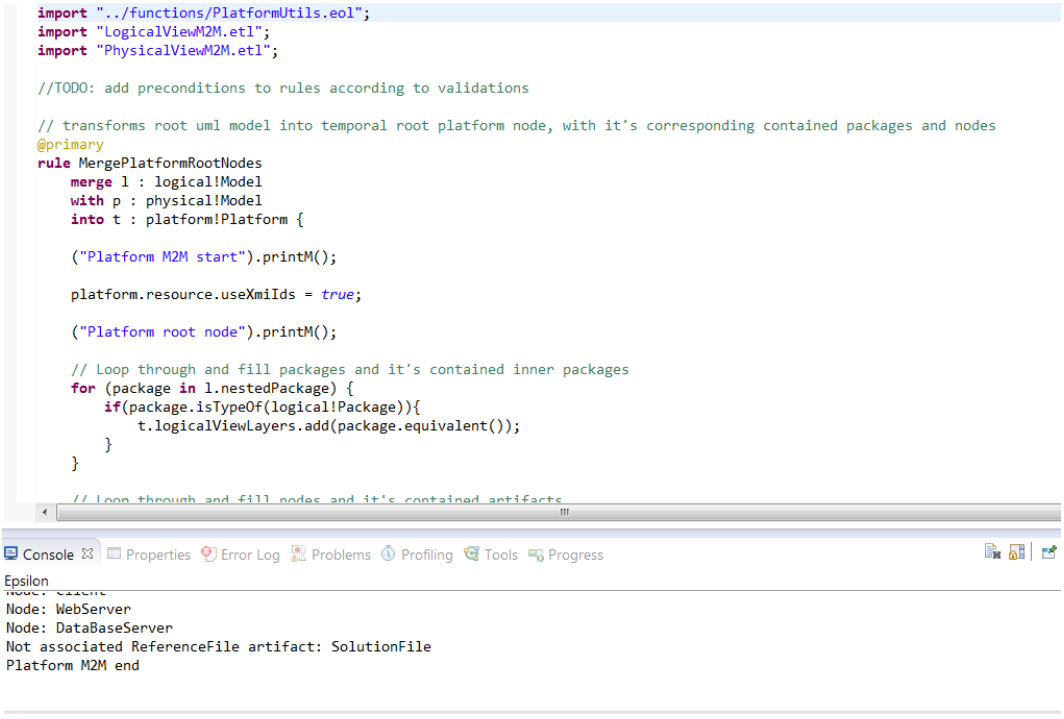


Ilustración 61: Transformación M2M de plataforma en los lenguajes de Epsilon

El proceso de combinación para el modelo de plataforma queda definido así:

- El modelo raíz de la vista lógica se transforma en el nodo raíz del modelo de plataforma, incluyendo referencias a sus paquetes contenidos.
- Los paquetes de la vista lógica que contienen otros paquetes se transforman en paquetes compuestos del modelo de plataforma, incluyendo referencias a sus paquetes contenidos.
- Los paquetes de la vista lógica que no contienen otros paquetes se transforman en paquetes simples del modelo de plataforma.
- Los nodos se transforman en tiers y los artefactos contenidos en los nodos se agregan como artefactos contenidos en los tiers.
- Las anidaciones entre nodos no están soportadas en el modelo de plataforma, por lo cual estos nodos son transformados en nodos simples.
- Si el nombre de un artefacto o su nombre de archivo están vacíos, se llena uno a partir del otro.
- Los artefactos contenidos en el elemento raíz de la vista lógica se transforman en manifestaciones de la raíz de la vista lógica.
- Los paquetes del modelo de plataforma se mezclan con las manifestaciones del weaving en plataforma para generar las manifestaciones del modelo de plataforma.
- Los artefactos contenidos en una manifestación del weaving se agregan a la manifestación de plataforma.
- Las dependencias entre artefactos y paquetes en las vistas físicas y lógica se crean como referencias en el modelo de plataforma.
- El nombre y la extensión de los artefactos en el modelo de weaving toman precedencia sobre los de los diagramas UML de las vistas lógica y física.
- Los tipos de artefactos en el modelo de plataforma son creados en sus correspondencias directas en el modelo de weaving.
- Los atributos adicionales de los artefactos presentes en el modelo de weaving son portados hacia el modelo de plataforma.
- Todos los elementos que no tienen correspondencia (paquetes nodos y artefactos) son igualmente transformados hacia el modelo de plataforma con reglas idénticas a las de los elementos combinados.

El proceso de modelado de la plataforma en Morphosys se amplía entonces con la validación de los modelos de entrada (vista lógica, vista física y weaving de plataforma) que en su validez dan paso a los elementos de la transformación M2M expuestos anteriormente (ver Ilustración 62). Con esto se da cumplimiento a la actividad **Transformar M2M en plataforma** presentada en el flujo de trabajo de la plataforma de la Ilustración 23.

Nota: Si los modelos de entrada no son válidos, el flujo se devuelve a las actividades de modelado hasta que todas las validaciones obligatorias sean exitosas. Morphosys también incluye algunas validaciones a tipo de sugerencia que no son obligatorias para continuar con el flujo.

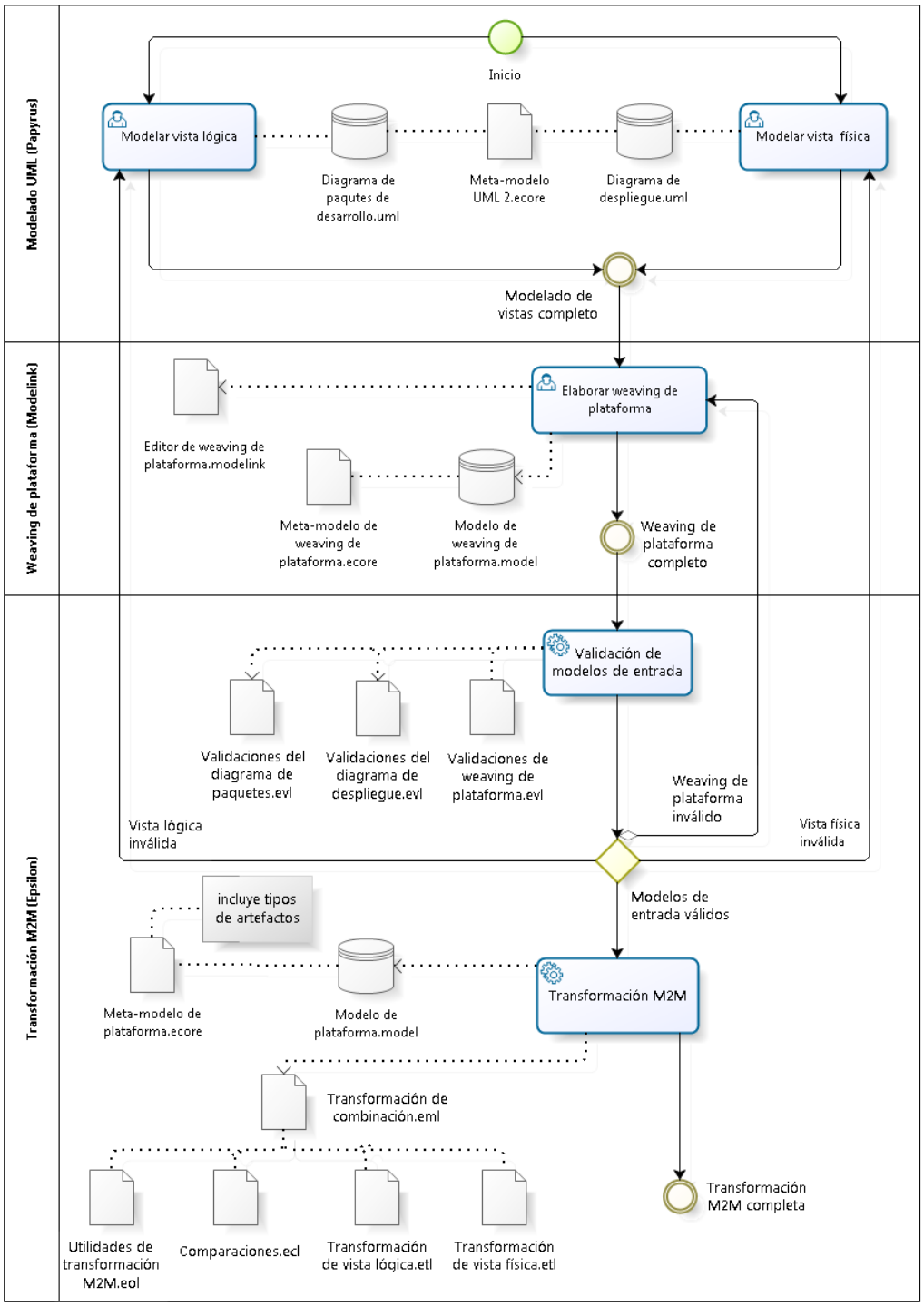


Ilustración 62: Transformación M2M dentro del proceso de modelado de plataforma en Morphosys

Una vez finalizada la transformación M2M ya se puede disponer de una instancia del meta-modelo de la plataforma (ver la Ilustración 63) que sintetiza los aspectos relevantes de las vista física y lógica más el weaving de cara a la generación de código fuente. Este modelo también tiene gran valor arquitectónico y se puede usar como un artefacto de la fase de diseño de software.

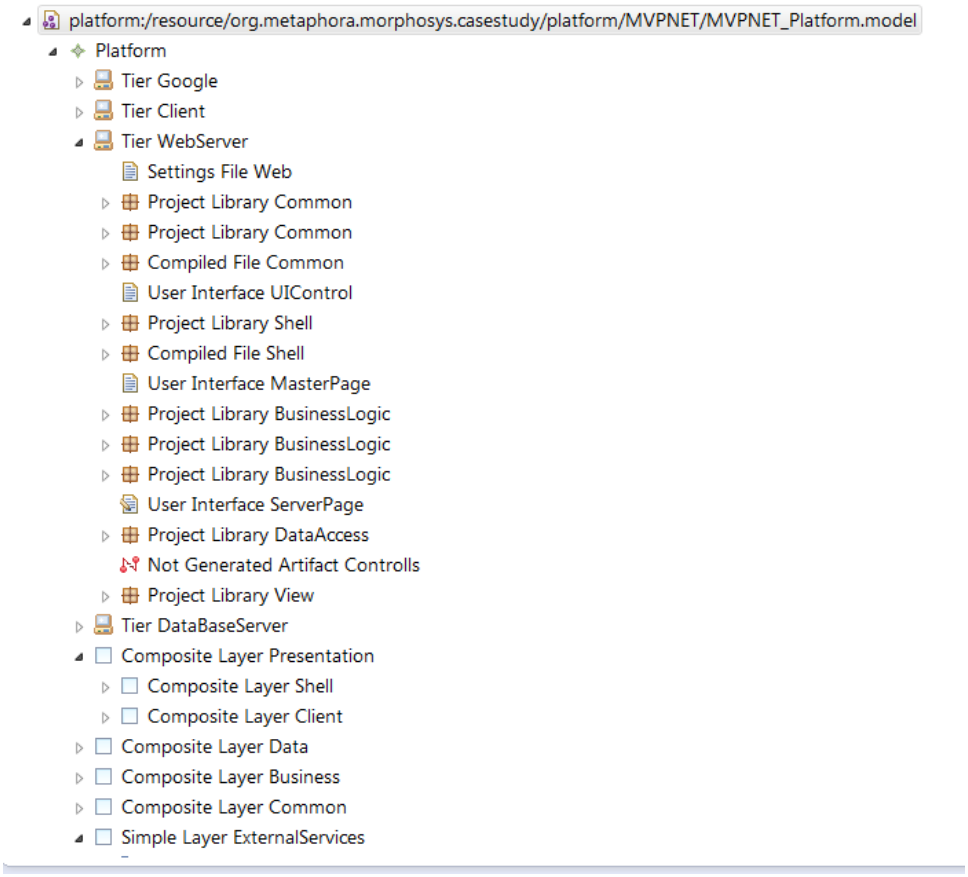


Ilustración 63: Instancia del modelo de plataforma MVP.NET para el caso de ICM

9. GENERACIÓN DE LA APLICACIÓN

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 (transformación M2T) consiste entonces en la combinación del modelo de aplicación (que hace parte de otra iniciativa diferente al presente trabajo) y el modelo de plataforma teniendo en cuenta los siguientes aspectos que hacen parte de la transformación M2T mostrada en la Ilustración 64:

- 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 EGL de Epsilon.
- Los artefactos con plantilla que no están mapeados se generan una sola vez mientras que los que están mapeados se generan una vez 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.

```
[% import "../functions/IntegrationUtils.eol"; %]

[%

("Integration M2T start").printM();


cleanSourceCodeDirectory(sourceCodePath());

TemplateFactory.setTemplateRoot("file:///" + basePath() + "/integration/templates/NET4");
TemplateFactory.setOutputRoot(sourceCodePath());


var root = platform!Platform.all.first();

// generate artifacts on root
root.rootManifestation.generateFiles("");

// loop through layers
for (layer in root.logicalViewLayers) {
    layer.generateFiles("");    // root path
}

("Integration M2T end").printM();

%]

[%

// Global variables
operation basePath() : String
{
    var baseFolder = new Native("java.io.File")("../");
```

Ilustración 64: Transformación M2T

Para cada arquitectura que se implementa en Morphosys se deben generar las respectivas plantillas EGL (ver la Ilustración 65) usando sentencias EOL que contaran 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 especifico en el caso de los artefactos mapeables.

Nota: adicional a las plantillas EGL se pueden incluir archivos utilitarios usando EOL para facilitar la lectura y estructuración.

```
[% import '../../functions/IntegrationUtils.eol'; %]
[% if (appBlock.isTypeOf(application!`List`)) { %]
// ------------------------------------------------------------------------------------------------------------------
// <copyright file="[%= artifact.prefix + appBlock.baseClass.plural + artifact.postfix + "." + artifact.extension %]" company="">
//    Copyright (c) Morphosys 2014. All rights reserved.
// </copyright>
// <summary>
//   The [%= appBlock.baseClass.plural %] code behind class
// </summary>
// ------------------------------------------------------------------------------------------------------------------

namespace IcM
{
    [* TODO: fill usings from platform meta/model used artifacts *]
    using System;
    using System.Collections.Generic;
    using Common;
    using Common.DTOs;
    using Presentation.Shell.ViewInterfaces;

    public partial class Incidents : MvpBasePage<[%= appBlock.baseClass.plural %]Presenter, I[%= appBlock.baseClass.plural %]View>, I[%
    {
        #region Properties

        public List<[%= appBlock.baseClass.name %]DTO> [%= appBlock.baseClass.name %]List [* TODO: fill view name from platform meta/mo
        {
            get
            {
                return ViewState["[%= appBlock.baseClass.name %]List"] as string;
            }

            set
            {
```

Ilustración 65: Ejemplo de plantilla EGL

La intención de Metafora es generar un repositorio que cuente con un conjunto de plantillas para cada una de las arquitecturas más populares de manera que cada modelo de plataforma se pueda reusar para varios modelos de aplicación y de manera recíproca que el mismo modelo de aplicación se pueda generar automáticamente en varias plataformas.

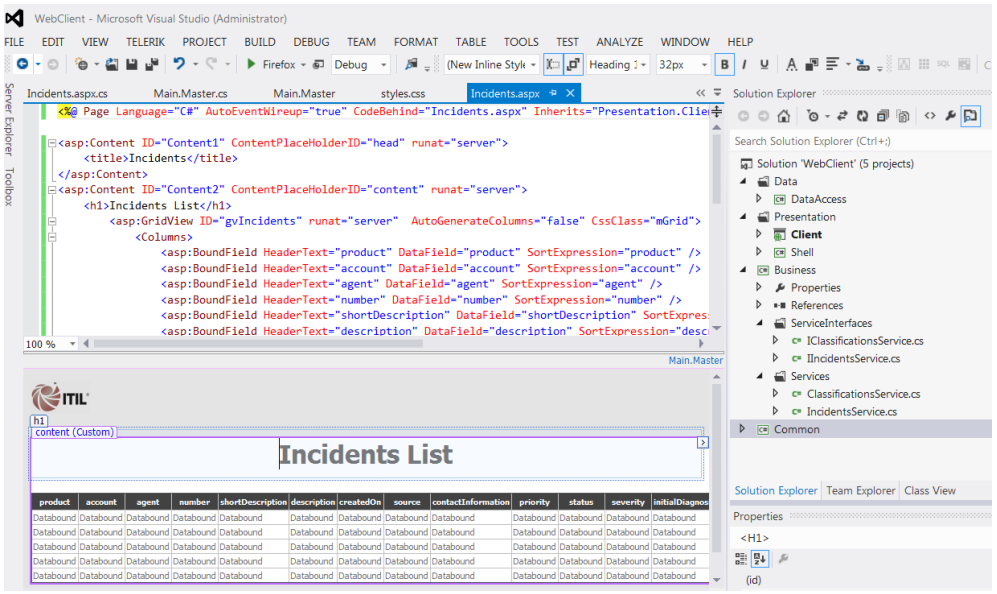
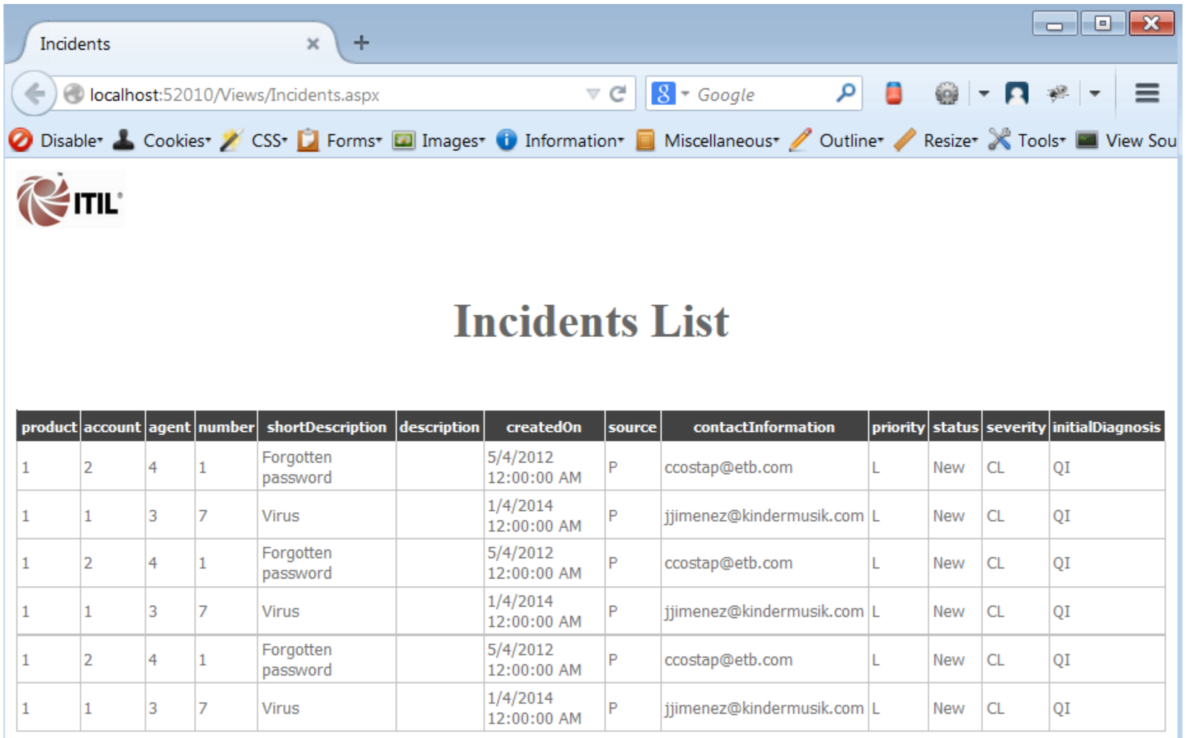


Ilustración 66: Ejemplo de código fuente de la aplicación de IcM generada con Morphosys corriendo en Visual Studio .NET



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

Ilustración 67: Ejemplo de ejecución de la aplicación de IcM generada con Morphosys

Para finalizar el proceso de modelado de plataforma Morphosys incluimos la transformación M2T con las respectivas plantillas y la generación de código fuente como se muestra en la Ilustración 68. Con esto se da cumplimiento a la actividad **Transformar M2T** presentada en el flujo de trabajo de la plataforma de la Ilustración 23. De esta manera y tras implementar el caso de estudio en Morphosys, se puede generar y ejecutar el código fuente como se aprecia en la Ilustración 66 y la Ilustración 67 respectivamente.

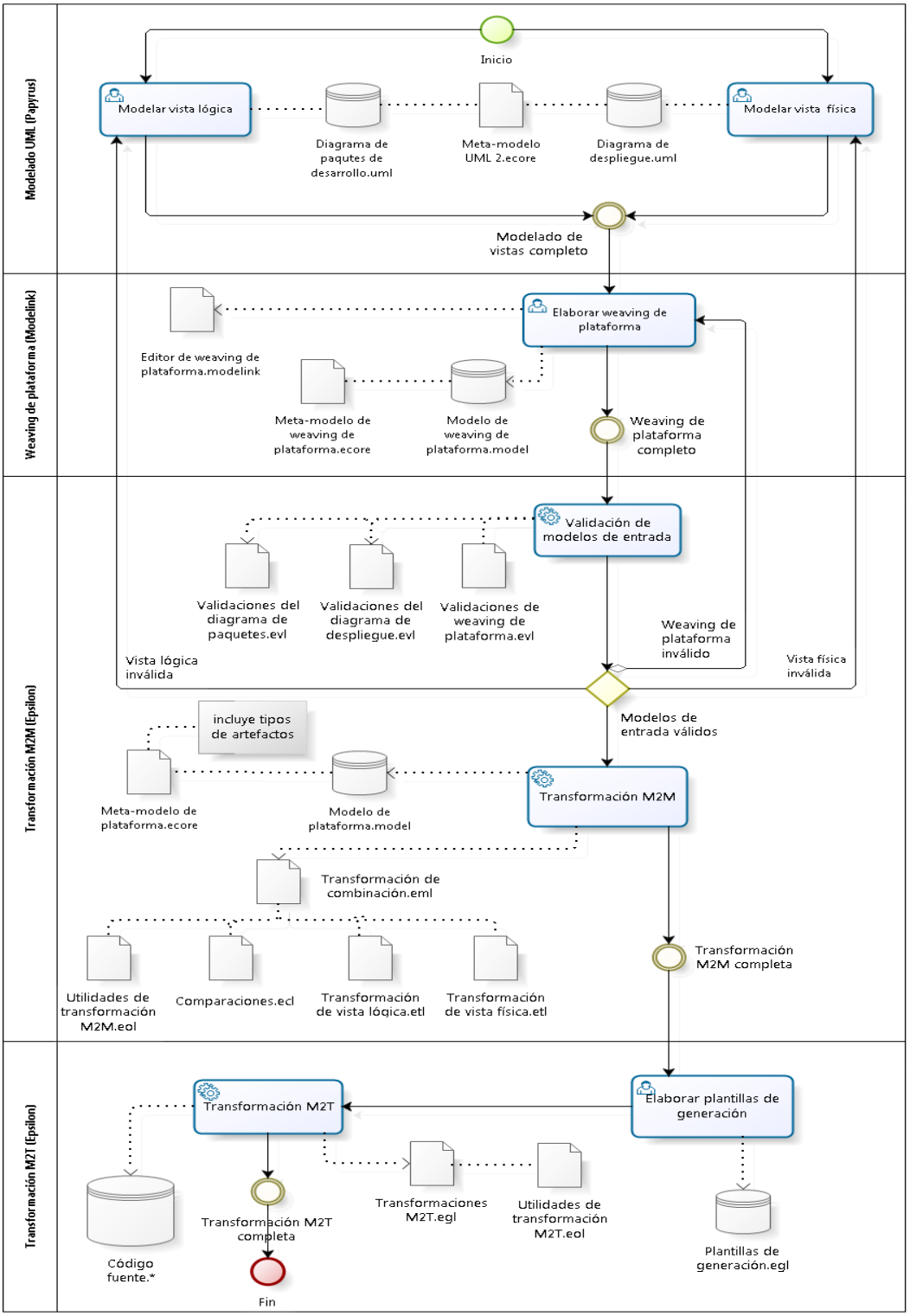


Ilustración 68: Proceso completo de modelado de plataforma en Morphosys

10. CONCLUSIONES Y TRABAJO FUTURO

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 modelar aspectos concernientes a la plataforma de ejecución para enriquecer el poder expresivo de los modelos.

El desarrollo de Morphosys 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 manera generalizada que guía la generación de modelo a texto según su multiplicidad, generación, la manera en que se despliegan o su tipo.

El nivel de detalle utilizado en los diagramas UML facilita la construcción del modelo de weaving de plataforma y por consecuente la transformaciones M2M y M2T. Esto se pudo evidenciar en el caso de los diagramas de MVP y .NET. Así mismo es importante resaltar que gran parte de la información semántica requerida para la ejecución de Morphosys 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 meta-modelo) se desarraigan los elementos de las vistas lógica y física de la herramienta de modelado específica, 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 con Morphosys.

Las siguientes ideas surgieron durante el desarrollo del presente trabajo pero no fueron implementadas dentro del alcance del mismo:

- Incrementar el repositorio de arquitecturas de Morphosys incluyendo modelos de plataforma y plantillas para las configuraciones más populares tales como: desarrollo móvil con middleware en iOS y Android, servicios REST, aplicaciones web de una sola página usando Backbone, etc.
- Configurar una distribución de Eclipse que incluya las herramientas de Epsilon, Papyrus, los meta-modelos de plataforma, aplicación y weaving, el repositorio de arquitecturas y plantillas de diferentes plataformas para facilitar el uso del método de metáfora a los usuarios de Morphosys.
- Reevaluar y someter a mejoramiento la taxonomía de artefactos consultando con grandes casas de desarrollo de software.
- Aprovechar mejor la taxonomía de artefactos para robustecer Morphosys con más reglas y restricciones.
- Incluir ejemplos prácticos de plantillas y código fuente generado por Morphosys.
- Crear un manual de usuario tanto para los usuarios encargados para generar las arquitecturas como para aquellos encargados de modelar la aplicación.
- Permitir la configuración personalizada de carpetas para la generación M2T sin depender de la estructura de layers del modelo de plataforma.
- Permitir la generación de artefactos mapeables en un mismo archivo, como por ejemplo el caso del script de base de datos.
- Habilitar el uso de las referencias entre artefactos y layers dentro de las plantillas
- Habilitar la preservación los bloques de código estáticos insertados por fuera de Morphosys.
- Desarrollar un asistente para la generación y ejecución de todos los pasos de modelado de la plataforma y la generación de código fuente de manera sencilla para el usuario final.

REFERENCIAS BIBLIOGRÁFICAS

[1] J. Quintero, P. Rucinque, R. Anaya y G. Piedrahita, «How face the top MDE adoption problems: An Exploratory Study Case,» *IEEE,* 2012.

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

[3] N. Akhter y N. Tariq, «Comparison of Model Driven Architecture (MDA) based tools,» *Institute of Technology-Karolinska University Hospital,* 2005.

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

[5] J. B. Quintero y J. A. Hincapie, *Proyecto Metáfora,* Universidad de Medellín, 2013.

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

[7] L. Fuentes y A. Vallecillo , «Una Introducción a los Perfiles UML,» *Depto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga,* 2004.

[8] Object Management Group, «Common Warehouse Metamodel (CWM) Specification,» Object Management Group, 2001.

[9] B. Kitchenham, "Procedures for Performing Systematic Reviews," Keele University, 2004.

[10] M. Belaunde, C. Burt and C. Cory, MDA Guide, Joaquin Miller y Jishnu Mukerji, 2003.

[11] T. Stahl and M. Völter, Model-Driven Software Development (Technology, Engineering, Management), John Wiley and Sons, 2006.

[12] Object Management Group, «Meta Object Facility (MOF) 2.0 Core,» Object Management Group, 2006.

[13] S. Sendall y W. Kozaczynski, «Model transformation: the heart and soul of model-driven software development,» *IEEE Software,* 2003.

[14] A. W. Brown, J. Conallen y D. Tropeano, Model-Driven Software Development, Introduction: Models, Modeling, and Model-Driven Architecture (MDA), Springer-Verlag Berlin Heidelberg, 2005.

[15] M. Voelter, S. Benz y C. Dietrich, DSL Engineering: Designing, Implementing and Using Domain-Specific Languages, CreateSpace Independent Publishing Platform , 2013.

[16] M. Fowler y S. Kendall, UML Distilled Third Edition A Brief Guide to the Standard Object Modeling Language, Addison Wesley, 2003.

[17] J. A. Domínguez, «Qué Informática!,» 2012. [En línea]. Available: http://www.que-informatica.com/index.php/software/uml-lenguaje-unificado-de-modelado-2/.

[18] Object Management Group, «MOF XMI Mapping,» Object Management Group, 2011.

[19] Object Management Group, «Object Constraint Language specification (OCL),» Object Management Group, 2012.

[20] J. Den Han, "The Enterprise Architect (Building an agile enterprise)," 2009. [Online]. Available: http://www.theenterprisearchitect.eu/.

[21] J. F. Duitama Muñoz y J. B. Quintero, *Reflexiones acerca de la adopción de enfoques centrados en modelos en el desarrollo de software,* Medellín: Proyecto de Investigación - Universidad de Antioquia, 2011.

[22] L. Balmelli, D. Brown, M. Cantor and M. Mott, "Model-driven systems development," IBM Systems Journal, 2006.

[23] T. Gardner and L. Yusuf, "Explore model-driven development (MDD) and related approaches: A closer look at model-driven development and other industry initiatives," IBM, 14 03 2006. [Online]. Available: http://www.ibm.com/developerworks/library/ar-mdd3/.

[24] R. Hilliard, B. Sherlund, R. Wade y D. Emery, «IEEE 1471 - Architecture of a software-intensive system,» IEEE, 2000.

[25] C. B. Reynoso, «Introducción a la Arquitectura de Software,» Universidad de Buenos Aires, 2004.

[26] Oracle Corp., «Oracle JDeveloper.,» 2013. [En línea]. Available: http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html. [Último acceso: 15 10 2013].

[27] J. Gilbert, «TaylorMDA, Model Driven Architecture on Rails.,» 22 08 2011. [En línea]. Available: http://taylor.sourceforge.net/. [Último acceso: 10 15 2013].

[28] Andro MDA.org, «Generate components quickly with AndroMDA,» 2012. [En línea]. Available: http://www.andromda.org/index.html. [Último acceso: 15 10 2013].

[29] J. B. Quintero, J. A. Hincapié y R. Anaya, *Un enfoque multi-vistas para el Desarrollo Dirigido por Modelos,* Medellín, 2013.

[30] S. Meliá y J. Gómez, «The WebSA approach: Applying model driven engineering to web applications,» *Journal of Web Engineering,* vol. 5, nº 2, pp. 121-149, 2006.

[31] Chethan, «expertz.me,» 8 7 2013. [En línea]. Available: http://expertz.me/top-down-and-bottom-up-design/. [Último acceso: 30 7 2014].

[32] J. Hincapie y F. Duitama, «Model-driven web engineering methods: a literature review,» Grupo de Investigación Ingeniería y Software. Universidad de Antioquia., 2012.

[33] 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.

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

[35] 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.

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

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

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

[39] J. Oldevik, «MOFScript User Guide,» de *Object Management Group*, 2011.

[40] F. Balagtas-Fernandez, «Model-Driven Development of Mobile Applications,» de *23rd IEEE/ACM International Conference on automated Software Engineering*, 2008.

[41] A. C. Carton, «Aspect-Oriented Model-Driven Development for Mobile Context-Aware Computing,» de *First International Workshop on Software Engineering for Pervasive Computing Applications, Systems and Environments*, 2007.

[42] A. Fatwanto and C. Boughton, "Analysis, Specification and Modeling of Non-Functional Requirements for Translative Model-Driven Development," *ICCIS,* 2008.

[43] H. Wada, J. Suzuki and K. Oba, "A Model-Driven Development Framework for Non-functional Aspects in Service Oriented Architecture," in *Int. J. of Web Services Research 5*, 2008.

[44] L. Zhu and Y. Liu, "Model Driven Development with Non-Functional Aspects," in *EA '09 Proceedings of the 2009 ICSE Workshop*, 2009.

[45] Object Management Group, "UML Profile for MARTE, Beta 2," 2008.

[46] Object Management Group, "UML profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, v1.1," 2008.

[47] S. Kugele, W. Haberl and M. Tautschnig, "Optimizing Automatic Deployment Using Non-functional Requirement Annotations," Springer, 2009.

[48] L. Gönczy, Z. Déri and D. Varró, "Model Transformations for Performability Analysis of Service Configurations," in *Models in Software Engineering,*

Workshops and Symposia at MODELS 2008, Toulouse, France, 2008.

[49] F. Molina and A. Toval, "Integrating Usability Requirements that can be Evaluated in Design Time into Model Driven Engineering of Web Information Systems," *Advances in Engineering Software,* 2009.

[50] A. Sterritt y V. Cahill, «Customisable Model Transformations Based on Non-Functional Requirements,» de *SERVICES '08 Proceedings of the 2008 IEEE Congress on Services - Part I*, 2008.

[51] A. Solberg, J. Oldevik and J. Ø. Aagedal, "A Framework for QoS-aware Model Transformation using a Pattern-based Approach," Springer, 2004.

[52] D. Ardagna, C. Ghezzi y R. Mirandola, «Rethinking the use of Models in Software Architecture”.,» *QoSA,* 2008.

[53] S. Gallotti, C. Ghezzi, R. Mirandola y G. Tamburrel, «Quality Prediction of Service Compositions through Probabilistic Model Checking,» *QoSA,* 2008.

[54] G. Rodrigues, D. Rosenblum y S. Uchitel, «Reliability Prediction in Model-driven Development,» *MoDELS,* 2005.

[55] S. Röttger y S. Zschaler, «Model-Driven Development for Nonfunctional Properties: Refinement Through Model Transformation,» *UML,* 2004.

[56] V. Cortellessa, A. Di Marco y P. Inverardi, «Non-Functional Modeling and Validation in Model-Driven Architecture,» *WICSA,* 2007.

[57] J. B. Quintero, J. A. Hincapié y R. Anaya, *Hacia enfoques multi-vistas en el Desarrollo Dirigido por Modelos,* Medellín, 2014.

[58] APM Group Limited, «The Official ITIL® Website,» 10 10 2013. [En línea]. Available: http://www.itil-officialsite.com/.

[59] J. B. Quintero y R. Anaya de Páez, *Marco de referencia para la evaluación de herramientas basadas en MDA,* Medellín: Grupo de Investigación en Ingeniería de Software - Universidad EAFIT, 2007.

[60] «Wikipedia,» 5 9 2013. [En línea]. Available: http://en.wikipedia.org/wiki/List_of_Eclipse_Modeling_Framework_based_software. [Último acceso: 3 7 2014].

[61] J. Rumbaugh, J. Ivar y G. Booch, The Unified Modeling Language Reference Manual, Second Edition, Pearson tech group, 2004.

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

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

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

[65] C. de la Torre, U. Zorrilla, M. Ramos y J. Calvarro, Guía de Arquitectura de N-Capas orientada al Dominio con .NET 4.0, Microsoft, 2010.

[66] R. C. Martin, The Principles, Patterns, and Practices of Agile Software Development, Prentice-Hall,, 2003.

[67] J. Emmatty, «Code Project - Differences between MVC and MVP for Beginners,» 23 11 2011. [En línea]. Available: http://www.codeproject.com/Articles/288928/Differences-between-MVC-and-MVP-for-Beginners. [Último acceso: 18 4 2014].

[68] S. W. Ambler, «Agilemodeling,» [En línea]. Available: http://agilemodeling.com/. [Último acceso: 17 08 2014].

[69] P. Selonen, *Metamodeling and Meta-Object Facility (MOF),* Spring, 2004.

[70] C. Griffin, «Introduction to the Eclipse Modeling Framework,» de *OMG MDA Implementer's Workshop*, 2003.

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

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

[73] J. Mukerji y J. Miller, «Object Management Group MDA guide,» 2001. [En línea]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.. [Último acceso: 2014].

[74] 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.

[75] J. Herrington, Code Generation in Action., Manning, 2003.

[76] D. S. Kolovos, R. F. Paige y F. . A. Polack., «The Epsilon Object Language,» de *Proc. European Conference in Model Driven Architecture (EC-MDA)*, 2006.

[77] K. Czarnecki y S. Helsen, «Classification of Model Transformation Ap-,» de *OOPSLA '03 Workshop on Generative Techniques in the Context of Model*, 2003.

[78] R. A. Pottinger y P. A. Bernstein, «Merging Models Based on Given Correspondences.,» University of Washington,, 2003.

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

[80] K. Letkeman, *Comparing and merging UML models in IBM Rational Software Architect.,* IBM Developerworks, 2005.

[81] S. Melnik, E. Rahm y P. A. Bernstein, *Rondo: A Programming Platform for Generic Model Management,* SIGMOD, 2003.

ANEXO 1: TRANSFORMACIONES M2M

```
import "../functions/PlatformUtils.eol";

// matches models from logical and physical (different from the primitive types import)
rule MatchLayerWithManifestation
  match l : logical!Model
  with p : physical!Model {

  compare: l.name <> "PrimitiveTypes" and p.name <> "PrimitiveTypes"
}

// matches logical packages with weaving manifestations
@greedy
rule MatchUMLPackageWithWeavingManifestation
  match l : logical!Package
  with w : weaving!WeavingManifestation {

  guard: not l.isTypeOf(logical!Model)

  compare: w.umlPackage.isDefined() and l.id = w.umlPackage.id
}

@greedy
rule MatchUMLArtifactWithWeavingArtifact
  match p : physical!Artifact
  with w : weaving!BaseArtifact {

  compare: w.umlArtifact.isDefined() and p.id = w.umlArtifact.id
}

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
----------------------------------------------
```

```
import "../functions/PlatformUtils.eol";
import "LogicalViewM2M.etl";
import "PhysicalViewM2M.etl";

//TODO: add preconditions to rules according to validations

// transforms root uml model into temporal root platform node, with it's corresponding contained packages and nodes
@primary
rule MergePlatformRootNodes
	merge l : logical!Model
	with p : physical!Model
	into t : platform!Platform {

	("Platform M2M start").printM();

	platform.resource.useXmiIds = true;

	("Platform root node").printM();

    // Loop through and fill packages and it's contained inner packages
	for (package in l.nestedPackage) {
		if(package.isTypeOf(logical!Package)){
			t.logicalViewLayers.add(package.equivalent());
		}
	}

	// Loop through and fill nodes and it's contained artifacts
	for (node in p.packagedElement) {
		if(node.isTypeOf(physical!Node)){
			t.physicalViewTiers.add(node.equivalent());
		}
	}

	// artifacts on the root model
```

```
	var weavingRoot = weaving!WeavingManifestation.allInstances.selectOne(m|
m.umlPackage.isDefined() and m.umlPackage.isTypeOf(logical!Model));
	if(weavingRoot.isDefined())
	{
		var manifestation : platform!Manifestation =
platform!Manifestation.createInstance();

		for (artifact in weavingRoot.artifacts) {
			manifestation.implementedArtifacts.add(artifact.equivalent());
		}

		t.rootManifestation = manifestation;
	}

	("Platform M2M end").printM();
}

// Merges uml packages with weaving manifestations into platform layers. Non
correspondence will be handled by ETL
// Links manifestation to packages
// Links artifacts to manifestations
@abstract
@greedy
@lazy
rule MergeManifestation2Layer
	merge l : logical!Package
	with w : weaving!WeavingManifestation
	into t : platform!Layer {

	t.name = l.name.asString();
	t.namespace = l.URI.asString();

	for (dependency in l.clientDependency) {
		if(dependency.supplier.first().isTypeOf(logical!Package)){
			t.usedLayers.add(dependency.supplier.first().equivalent());
		}
	}
```

```
	var manifestation : platform!Manifestation = platform!Manifestation.createInstance();

	for (artifact in w.artifacts) {
		if(artifact.isKindOf(weaving!BaseArtifact)){
			manifestation.implementedArtifacts.add(artifact.equivalent());
		}
	}

	t.layerManifestation = manifestation;
	//var root = platform!Platform.all.first();
	//root.manifestations.add(manifestation);
}

// Merges uml packages with weaving manifestations into platform composite layers. Non correspondence will be handled by ETL
@lazy
rule MergeManifestation2CompositeLayer
	merge l : logical!Package
	with w : weaving!WeavingManifestation
	into t : platform!CompositeLayer
	extends MergeManifestation2Layer {

	guard: not l.nestedPackage.isEmpty()	// only when a package has inner packages

	("Composite layer with Manifestation: " + t.name).printM();

	for (package in l.nestedPackage) {
		if(package.isTypeOf(logical!Package)){
			t.innerLayers.add(package.equivalent());
		}
	}
}

// Merges uml packages with weaving manifestations into platform simple layers. Non correspondence will be handled by ETL
```

```
@lazy
rule MergeManifestation2SimpleLayer
	merge l : logical!Package
	with w : weaving!WeavingManifestation
	into t : platform!SimpleLayer
	extends MergeManifestation2Layer {

	guard: l.nestedPackage.isEmpty() // only when a package has NO inner packages

	("Simple layer with Manifestation: " + t.name).printM();
}

// Merges uml artifacts with weaving artifacts
@abstract
@greedy
@lazy
rule MergeWeavingArtifact2PlatformArtifact
	merge p : physical!Artifact
	with w : weaving!BaseArtifact
	into t : platform!Artifact {

	// weaving name overrides uml artifact name
	if(w.name <> null and w.name <> '') {
		t.name = w.name;
	}
	else {
		t.name = p.displayName();
	}

	// weaving extension overrides uml artifact extension
	if(w.extension <> null and w.extension <> '') {
		t.extension = w.extension;
	}
	else {
		t.extension = p.displayExtension();
	}
```

```
		(t.type().name + " artifact: " + t.name).printM();

		for (dependency in p.clientDependency) {
			if(dependency.supplier.first().isTypeOf(physical!Artifact)){
				t.usedArtifacts.add(dependency.supplier.first().equivalent());
			}
		}
}

// Merges uml artifacts with weaving Compiled artifacts
// Disaggregates compiled artifacts into source code artifacts
@abstract
@lazy
rule MergeWeavingCompiledArtifact2PlatformCompiledArtifact
		merge p : physical!Artifact
		with w : weaving!CompiledArtifact
		into t : platform!CompiledArtifact
		extends MergeWeavingArtifact2PlatformArtifact {

		// Loop through and fill contained source code artifacts
		for (artifact in w.sourceArtifacts) {
			if(artifact.isKindOf(weaving!SourceArtifact)){
				t.sourceArtifacts.add(artifact.equivalent());
			}
		}
}

// Merges uml artifacts with weaving ProjectLibrary artifacts (assigning the specific
type)
@lazy
rule MergeWeavingProjectLibraryArtifact2PlatformProjectLibraryArtifact
		merge p : physical!Artifact
		with w : weaving!ProjectLibrary
		into t : platform!ProjectLibrary
		extends MergeWeavingCompiledArtifact2PlatformCompiledArtifact {
```

```
}

// Merges uml artifacts with weaving Executable artifacts (assigning the specific type)
@lazy
rule MergeWeavingExecutableArtifact2PlatformExecutableArtifact
	merge p : physical!Artifact
	with w : weaving!Executable
	into t : platform!Executable
	extends MergeWeavingCompiledArtifact2PlatformCompiledArtifact {

}

// Merges uml artifacts with weaving CompiledFile artifacts (assigning the specific type)
@lazy
rule MergeWeavingCompiledFileArtifact2PlatformCompiledFileArtifact
	merge p : physical!Artifact
	with w : weaving!CompiledFile
	into t : platform!CompiledFile
	extends MergeWeavingCompiledArtifact2PlatformCompiledArtifact {

}

// Merges uml artifacts with weaving Source Code artifacts
@abstract
@lazy
rule MergeWeavingSourceArtifact2PlatformSourceArtifact
	merge p : physical!Artifact
	with w : weaving!SourceArtifact
	into t : platform!SourceArtifact
	extends MergeWeavingArtifact2PlatformArtifact {

}

// Merges uml artifacts with weaving Templated artifacts
@abstract
```

```
@lazy
rule MergeWeavingTemplatedArtifact2PlatformTemplatedArtifact
	merge p : physical!Artifact
	with w : weaving!TemplatedArtifact
	into t : platform!TemplatedArtifact
	extends MergeWeavingSourceArtifact2PlatformSourceArtifact {

	t.templatePath = w.templatePath;
	t.isApplicationMapped = w.isApplicationMapped;
	t.prefix = w.prefix;
	t.postfix = w.postfix;

}

// Merges uml artifacts with weaving SettingsFile artifacts (assigning the specific
type)
@lazy
rule MergeWeavingSettingsFileArtifact2PlatformSettingsFileArtifact
	merge p : physical!Artifact
	with w : weaving!SettingsFile
	into t : platform!SettingsFile
	extends MergeWeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Merges uml artifacts with weaving Script artifacts (assigning the specific type)
@lazy
rule MergeWeavingScriptArtifact2PlatformScriptArtifact
	merge p : physical!Artifact
	with w : weaving!Script
	into t : platform!Script
	extends MergeWeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Merges uml artifacts with weaving ReferenceFile  artifacts (assigning the specific
type)
```

```
@lazy
rule MergeWeavingReferenceFileArtifact2PlatformReferenceFileArtifact
	merge p : physical!Artifact
	with w : weaving!ReferenceFile
	into t : platform!ReferenceFile
	extends MergeWeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Merges uml artifacts with weaving CodeFile artifacts (assigning the specific type)
@lazy
rule MergeWeavingCodeFileArtifact2PlatformCodeFileArtifact
	merge p : physical!Artifact
	with w : weaving!CodeFile
	into t : platform!CodeFile
	extends MergeWeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Merges uml artifacts with weaving UserInterface artifacts (assigning the specific
type)
@lazy
rule MergeWeavingUserInterfaceArtifact2PlatformUserInterfaceArtifact
	merge p : physical!Artifact
	with w : weaving!UserInterface
	into t : platform!UserInterface
	extends MergeWeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Merges uml artifacts with weaving Copy artifacts
@abstract
@lazy
rule MergeWeavingCopyArtifact2PlatformCopyArtifact
	merge p : physical!Artifact
	with w : weaving!CopyArtifact
	into t : platform!CopyArtifact
```

```
	extends MergeWeavingSourceArtifact2PlatformSourceArtifact {

	t.originalPath = w.originalPath;

}

// Merges uml artifacts with weaving Resource artifacts (assigning the specific type)
@lazy
rule MergeWeavingResourceArtifact2PlatformResourceArtifact
	merge p : physical!Artifact
	with w : weaving!Resource
	into t : platform!Resource
	extends MergeWeavingCopyArtifact2PlatformCopyArtifact {

}

// Merges uml artifacts with weaving PersistenceFile artifacts (assigning the
specific type)
@lazy
rule MergeWeavingPersistenceFileArtifact2PlatformPersistenceFileArtifact
	merge p : physical!Artifact
	with w : weaving!PersistenceFile
	into t : platform!PersistenceFile
	extends MergeWeavingCopyArtifact2PlatformCopyArtifact {

}

// Merges uml artifacts with weaving External artifacts
@abstract
@lazy
rule MergeWeavingExternalArtifact2PlatformExternalArtifact
	merge p : physical!Artifact
	with w : weaving!ExternalArtifact
	into t : platform!ExternalArtifact
	extends MergeWeavingArtifact2PlatformArtifact {

}
```

```
// Merges uml artifacts with weaving ExternalResource artifacts (assigning the specific type)
@lazy
rule MergeWeavingExternalResourceArtifact2PlatformExternalResourceLibraryArtifact
	merge p : physical!Artifact
	with w : weaving!ExternalResource
	into t : platform!ExternalResource
	extends MergeWeavingExternalArtifact2PlatformExternalArtifact {

}

// Merges uml artifacts with weaving DeployedService artifacts (assigning the specific type)
@lazy
rule MergeWeavingDeployedServiceArtifact2PlatformDeployedServiceArtifact
	merge p : physical!Artifact
	with w : weaving!DeployedService
	into t : platform!DeployedService
	extends MergeWeavingExternalArtifact2PlatformExternalArtifact {

}

// Merges uml artifacts with weaving NonGenerated artifacts (assigning the specific type)
@lazy
rule MergeWeavingNotGeneratedArtifact2PlatformNotGeneratedArtifact
	merge p : physical!Artifact
	with w : weaving!NotGeneratedArtifact
	into t : platform!NotGeneratedArtifact
	extends MergeWeavingArtifact2PlatformArtifact {

}

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
----------------------------------------------
```

```
import "../functions/PlatformUtils.eol";

//TODO: add preconditions to rules according to validations

// transform UML package into PMM Layer
@lazy
@abstract
rule UMLPackage2Layer
  transform l : logical!Package
  to t : platform!Layer {

	t.name = l.name.asString();
	t.namespace = l.URI.asString();

	for (dependency in l.clientDependency) {
		if(dependency.supplier.first().isTypeOf(logical!Package)){
			t.usedLayers.add(dependency.supplier.first().equivalent());
		}
	}
}
// transform UML package into PMM Composite Layer, with it's corresponding contained layers
@lazy
rule UMLPackage2CompositeLayer
	transform l : logical!Package
	to t : platform!CompositeLayer
	extends UMLPackage2Layer {

	guard: not l.nestedPackage.isEmpty()    // only when a package has inner packages

	("Composite layer: " + t.name).printM();

	for (package in l.nestedPackage) {
		if(package.isTypeOf(logical!Package)){
			t.innerLayers.add(package.equivalent());
		}
```

```
	}
}

// transform UML package into PMM Simple Layer
@lazy
rule UMLPackage2SimpleLayer
  transform l : logical!Package
  to t : platform!SimpleLayer
  extends UMLPackage2Layer  {

  guard: l.nestedPackage.isEmpty() // only when a package has NO inner packages

  ("Simple layer: " + t.name).printM();
}

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
---------------------------------------------

import "../functions/PlatformUtils.eol";

//TODO: add preconditions to rules according to validations

// transforms UML node into PMM Tier, with it's corresponding contained artifacts
@lazy
rule UMLNode2Tier
  transform p : physical!Node
  to t : platform!Tier {

	t.name = p.name.asString();

	("Node: " + t.name).printM();

	for (artifact in p.nestedClassifier) {
		if(artifact.isTypeOf(physical!Artifact)){
			for (art in artifact.equivalents()) {
				t.containedArtifacts.add(art);
```

```
                    }
                    //var art = platform!Artifact.allInstances.selectOne(a| a.umlId
== artifact.id);
                }
        }
}

// Transforms uml not manifested artifacts into platform NotGenerated artifacts by
default
@greedy
@lazy
rule UMLArtifact2Artifact
  transform p : physical!Artifact
  to t : platform!NotGeneratedArtifact { // converted into Compiled artifact
temporarily because Artifact is abstract

        // set the uml id for reference purposes
        t.name = p.displayName();
        t.extension = p.displayExtension();

        ("Not manifested Artifact: " + t.name).printM();
}

// Transforms weaving artifacts into platform artifacts
@abstract
@lazy
rule WeavingBaseArtifact2PlatformArtifact
  transform w : weaving!BaseArtifact
  to t : platform!Artifact {

        t.name = w.name;
        t.extension = w.extension;

        ("Not associated " + t.type().name + " artifact: " + t.name).printM();
}

// Transforms weaving Compiled artifacts into Compiled platform  Compiled artifacts
```

```
@abstract
@lazy
rule WeavingCompiledArtifact2PlatformCompiledArtifact
  transform w : weaving!CompiledArtifact
  to t : platform!CompiledArtifact
  extends WeavingBaseArtifact2PlatformArtifact {

  // Loop through and fill contained source code artifacts
       for (artifact in w.sourceArtifacts) {
             if(artifact.isKindOf(weaving!SourceArtifact)){
                   t.sourceArtifacts.add(artifact.equivalent());
             }
       }
}

// Transforms weaving ProjectLibrary artifacts into platform ProjectLibrary artifacts
(assigning the specific type)
@lazy
rule WeavingProjectLibraryArtifact2PlatformProjectLibraryArtifact
  transform w : weaving!ProjectLibrary
  to t : platform!ProjectLibrary
  extends WeavingCompiledArtifact2PlatformCompiledArtifact {

}

// Transforms weaving Executable into platform Executable artifacts (assigning the
specific type)
@lazy
rule WeavingExecutableArtifact2PlatformExecutableArtifact
  transform w : weaving!Executable
  to t : platform!Executable
  extends WeavingCompiledArtifact2PlatformCompiledArtifact {

}

// Transforms weaving CompiledFile into platform CompiledFile artifacts (assigning
the specific type)
```

```
@lazy
rule WeavingCompiledFileArtifact2PlatformCompiledFileArtifact
  transform w : weaving!CompiledFile
  to t : platform!CompiledFile
  extends WeavingCompiledArtifact2PlatformCompiledArtifact {

}

// Transforms weaving Source artifacts into platform Source artifacts
@abstract
@lazy
rule WeavingSourceArtifact2PlatformSourceArtifact
  transform w : weaving!SourceArtifact
  to t : platform!SourceArtifact
  extends WeavingBaseArtifact2PlatformArtifact {

}

// Transforms weaving Templated artifacts into platform Templated artifacts
@abstract
@lazy
rule WeavingTemplatedArtifact2PlatformTemplatedArtifact
  transform w : weaving!TemplatedArtifact
  to t : platform!TemplatedArtifact
  extends WeavingSourceArtifact2PlatformSourceArtifact {

      t.templatePath = w.templatePath;
      t.isApplicationMapped = w.isApplicationMapped;
      t.prefix = w.prefix;
      t.postfix = w.postfix;
}

// Transforms weaving SettingsFile artifacts into platform SettingsFile artifacts
@lazy
rule WeavingSettingsFileArtifact2PlatformSettingsFileArtifact
  transform w : weaving!SettingsFile
```

```
  to t : platform!SettingsFile
  extends WeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Transforms weaving Script artifacts into platform Script artifacts
@lazy
rule WeavingScriptArtifact2PlatformScriptArtifact
  transform w : weaving!Script
  to t : platform!Script
  extends WeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Transforms weaving ReferenceFile artifacts into platform ReferenceFile artifacts
@lazy
rule WeavingReferenceFileArtifact2PlatformReferenceFileArtifact
  transform w : weaving!ReferenceFile
  to t : platform!ReferenceFile
  extends WeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Transforms weaving CodeFile artifacts into platform CodeFile artifacts
@lazy
rule WeavingCodeFileArtifact2PlatformCodeFileArtifact
  transform w : weaving!CodeFile
  to t : platform!CodeFile
  extends WeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Transforms weaving UserInterface artifacts into platform UserInterface artifacts
@lazy
rule WeavingUserInterfaceArtifact2PlatformUserInterfaceArtifact
  transform w : weaving!UserInterface
```

```
  to t : platform!UserInterface
  extends WeavingTemplatedArtifact2PlatformTemplatedArtifact {

}

// Transforms weaving Copy artifacts into platform Copy artifacts
@abstract
@lazy
rule WeavingCopyArtifact2PlatformCopyArtifact
  transform w : weaving!CopyArtifact
  to t : platform!CopyArtifact
  extends WeavingSourceArtifact2PlatformSourceArtifact {

      t.originalPath = w.originalPath;

}

// Transforms weaving Resource artifacts into platform Resource artifacts
@lazy
rule WeavingResourceArtifact2PlatformResourceArtifact
  transform w : weaving!Resource
  to t : platform!Resource
  extends WeavingCopyArtifact2PlatformCopyArtifact {

}

// Transforms weaving PersistenceFile artifacts into platform PersistenceFile
artifacts
@lazy
rule WeavingPersistenceFileArtifact2PlatformPersistenceFileArtifact
  transform w : weaving!PersistenceFile
  to t : platform!PersistenceFile
  extends WeavingCopyArtifact2PlatformCopyArtifact {

}

// Transforms weaving External artifacts into platform External artifacts
```

```
@abstract
@lazy
rule WeavingExternalArtifact2PlatformExternalArtifact
  transform w : weaving!ExternalArtifact
  to t : platform!ExternalArtifact
  extends WeavingBaseArtifact2PlatformArtifact {

}

// Transforms weaving ExternalResource artifacts into platform ExternalResource artifacts
@lazy
rule WeavingExternalResourceArtifact2PlatformExternalResourceArtifact
  transform w : weaving!ExternalResource
  to t : platform!ExternalResource
  extends WeavingExternalArtifact2PlatformExternalArtifact {

}


// Transforms weaving DeployedService artifacts into platform DeployedService artifacts
@lazy
rule WeavingDeployedServiceArtifact2PlatformDeployedServiceArtifact
  transform w : weaving!DeployedService
  to t : platform!DeployedService
  extends WeavingExternalArtifact2PlatformExternalArtifact {

}


// Transforms weaving NotGenerated artifacts into platform  NotGenerated artifacts
@lazy
rule WeavingNotGeneratedArtifact2PlatformNotGeneratedArtifact
  transform w : weaving!NotGeneratedArtifact
  to t : platform!NotGeneratedArtifact
  extends WeavingBaseArtifact2PlatformArtifact {
```

```
}

// contained nodes are transformed into simple nodes (no nested nodes support)

--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
/* ------------------------------------------------------------------------------------
---------------
UML Utils
*/

// shows either the artifact name or the file name when the first is empty
operation physical!Artifact displayName() : String {
	return self.Name
	.replaceWhenNull(self.fileName.extractPart('\\.',0));
}

// shows either the display name or the id when the first is empty
operation physical!Artifact displayLabel() : String {
	return self.displayName()
	.replaceWhenNull(self.id);
}

// extracts the artifact extension
operation physical!Artifact displayExtension() : String {
	return self.fileName.extractPart('\\.',1);
}

/* ------------------------------------------------------------------------------------
---------------
String Utils
*/

// extracts part of a string based on a split separator and a position on the
splitted string
```

```
operation String extractPart(regexSeparator : String, position : Integer) : String {
	var retValue : String;
	retValue = "";
	var parts: Sequence;
	parts = self.split(regexSeparator);
	if(parts.size() > position) // the position exists on the parts list
	{
		retValue = parts[position];
	}
	return retValue;
}

// returns an alternative value when the original is empty or null
operation String replaceWhenNull(replaceValue : String) : String {
	var retValue : String;
	retValue = self;
	if(retValue.isUndefined() or retValue == "")
	{
		retValue = replaceValue;
	}
	return retValue;
}

operation String printM()
{
	"Morphosys >>" + self.println();
}
```

ANEXO 2: TRANSFORMACIONES M2T

```
[% import "../functions/IntegrationUtils.eol"; %]

[%

("Integration M2T start").printM();



cleanSourceCodeDirectory(sourceCodePath());

TemplateFactory.setTemplateRoot("file:///" + basePath() +
"/integration/templates/NET4");
TemplateFactory.setOutputRoot(sourceCodePath());



var root = platform!Platform.all.first();

// generate artifacts on root
root.rootManifestation.generateFiles("");

// loop through layers
for (layer in root.logicalViewLayers) {
	layer.generateFiles("");  // root path
}

("Integration M2T end").printM();

%]

[%

// Global variables
operation basePath() : String
{
	var baseFolder = new Native("java.io.File")("../");
```

```
	return (baseFolder.canonicalPath + "/morfosisrepo/org.metaphora.morphosys.casestudy");
}

operation sourceCodePath() : String
{
	return basePath() + "/source_code";
}

// build layer paths and generate artifacts for each inner layer
operation platform!Layer generateFiles(parentPath : String)
{
	var pathAdd : String = "";
	if(parentPath.length() > 0) // no slash on beginning
	{
		pathAdd =  "/";
	}

	parentPath = parentPath + pathAdd + self.name;

	// generate artifacts on inner packages
	if (self.isTypeOf(platform!CompositeLayer))
	{
		for (layer in self.innerLayers) {
			layer.generateFiles(parentPath);
		}
	}

	if(self.layerManifestation.isDefined())
	{
		self.layerManifestation.generateFiles(parentPath);
	}
}

operation platform!Manifestation generateFiles(parentPath : String)
{
	if(self.isDefined())
```

```
	{
		// loop through manifested artifacts and handle according to type
		for (artifact in self.implementedArtifacts) {
			if(artifact.isKindOf(platform!SourceArtifact))
			{
				artifact.generateFile((parentPath));
			}
			else if(artifact.isKindOf(platform!CompiledArtifact))
			{
				// loop through it's source artifacts
				for (sourceArtifact in artifact.sourceArtifacts) {
					sourceArtifact.generateFile((parentPath));

				}
			}
			else if(artifact.isKindOf(platform!ExternalArtifact))
			{
				// send uri to template
			}
			else if(artifact.isKindOf(platform!NotGeneratedArtifact))
			{
				// Do nothing, it's not generated :p
			}
		}
	}
}


// copy artifact from another location
operation platform!CopyArtifact generateFile(parentPath : String)
{
	var originalPath : String = basePath() + self.originalPath;
	var generationDirectory : String = sourceCodePath() + "/" + parentPath;
	var generationPath : String = generationDirectory + "/" + self.name + "." +
self.extension;

	if(not directoryExists(generationDirectory))
```

```
	{
		createDirectory(generationDirectory);
	}

	copyFile(originalPath, generationPath);

	("Generated Copy Artifact on: " + generationPath).printM();
}

// generate artifact from template
operation platform!TemplatedArtifact generateFile(parentPath : String)
{
	var templatePath : String = self.templatePath;

	if(templatePath.isDefined() and templatePath <> "")
	{
		var template : Template = TemplateFactory.load(templatePath);

		var pathAdd : String = "";
		if(parentPath.length() > 0) // no slash on beginning
		{
			pathAdd =  "/";
		}

		var generationPath : String;

		if(self.isKindOf(platform!TemplatedArtifact))
		{
			if(self.isTypeOf(platform!ReferenceFile))
			{
				generationPath =  parentPath + pathAdd + self.name + "." +
self.extension;

				for (class in application!Class.all) {
					//template.populate("class", class);
					//template.populate("artifact", self);
					//template.generate(generationPath);
```

```
                                //("Generated Reference File on: " +
sourceCodePath() + generationPath).printM();
                        }
                }
                else
                {
                        if(self.isApplicationMapped)
                        {
                                //mappable artifacts, each class with each layer
                                for (appBlock in application!`List`.all) {
        //TODO: include other types of blocks
                                        generationPath = parentPath + pathAdd +
self.prefix + appBlock.baseClass.plural + self.postfix + "." + self.extension;


                                        template =
TemplateFactory.load(templatePath); // load it on every loop to populate variables
again
                                        template.populate("app",
application!WebApp.allInstances.first());
                                        template.populate("parentPath", parentPath);
                                        template.populate("appBlock", appBlock);
                                        template.populate("artifact", self);
                                        template.generate(generationPath);


                                        ("Generated Application Mapped Artifact on: "
+ sourceCodePath() + "/" + generationPath).printM();
                                }


                                //TODO: generate classes not associated to appBlocks
                        }
                        else
                        {
                                generationPath =  parentPath + pathAdd + self.prefix
+ self.name + self.postfix + "." + self.extension;


                                template.populate("app",
application!WebApp.allInstances.first());
                                template.populate("parentPath", parentPath);
```

```
                                template.populate("artifact", self);
                                template.generate(generationPath);

                                ("Generated Non Application Mapped Artifact on: " +
sourceCodePath() + generationPath).printM();
                            }
                        }
                }
        }
}

%]
```

```
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------

/* ---------------------------------------------------------------------------
--------------
File Utils
*/

operation cleanSourceCodeDirectory(path : String)
{
	var directory = new Native("java.io.File")(path);
	cleanDirectory(directory);
	("Source code directory cleaned: " + directory.absolutePath).printM();
}

operation cleanDirectory(path : Native("java.io.File"))
{
	if(not path.exists()) return;

	var files = path.listFiles();
	for (file in files) {
		if(file.isDirectory()) {
		cleanDirectory(file);
		}

		file.`delete`; //reserved word
	}
}

operation createDirectory(path : String)
{
	var directory  = new Native("java.io.File")(path);
	directory.mkdirs();
}

operation directoryExists(path : String) : Boolean
```

```
{
	var directory  = new Native("java.io.File")(path);
	return directory.exists();
}

operation copyFile(originalPath : String, copyPath : String)
{
	var originalFile  = new Native("java.io.File")(originalPath);
	var copyFile  = new Native("java.io.File")(copyPath);

	var is = new Native("java.io.FileInputStream")(originalFile);
    var os = new Native("java.io.FileOutputStream")(copyFile);

    var fin =  new Native("java.io.BufferedInputStream")(is);
    var fout = new Native("java.io.BufferedOutputStream")(os);

    var data : Integer;
    data = fin.read();
    while (data <> -1) {
        fout.write(data);
        data = fin.read();
    }

    fin.close();
    fout.close();
}

/* ----------------------------------------------------------------------------
---------------
String Utils
*/

operation String printM()
{
	("Morphosys >> " + self).println();
}
```

ANEXO 3: PLANTILLAS DE GENERACIÓN M2T

```
[% import '../../functions/IntegrationUtils.eol'; %]
[% if (appBlock.isTypeOf(application!`List`)) { %]
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="[%=artifact.prefix + appBlock.baseClass.plural + artifact.postfix + "." + artifact.extension %]" Inherits="IcM.Incidents" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:GridView ID="gv[%=appBlock.baseClass.plural%]s" runat="server">
            <Columns>
             [% for (field in appBlock.fields){%]
                <asp:BoundField HeaderText="[%=field.property.name%]" DataField="[%=field.property.name%]" SortExpression="[%=field.property.name%]" />
                [% } %]
            </Columns>
        </asp:GridView>

    </div>
    </form>
</body>
</html>
[% } %]
```

```
-----------------------------------------------------------------------


[% import '../../functions/IntegrationUtils.eol'; %]
[% if (appBlock.isTypeOf(application!`List`)) { %]
// ------------------------------------------------------------------------------------
----------------------------------
//    <copyright    file="[%=    artifact.prefix    +    appBlock.baseClass.plural    +
artifact.postfix + "." + artifact.extension %]" company="">
//    Copyright (c) Morphosys 2014. All rights reserved.
// </copyright>
// <summary>
//  The [%= appBlock.baseClass.plural %]  Presenter
// </summary>
// ------------------------------------------------------------------------------------
----------------------------------


namespace [%= parentPath.replace("/",".") %]
{
    [* TODO: fill usings from platform meta/model used artifacts *]
    using Presentation.Shell.ViewInterfaces;
    using Microsoft.Practices.CompositeWeb;


    public class [%= artifact.prefix + appBlock.baseClass.plural + artifact.postfix
%]<TView> : Presenter<TView>
        where TView : I[%= appBlock.baseClass.plural %]View [* TODO: fill view name
from platform meta/model used artifacts *]
    {
        #region Events


        public override void OnViewInitialized()
        {
                    View.[%= appBlock.baseClass.name %]List = null;
        }


        #endregion Events
    }
}
[% } %]
```

```
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
[% import '../../functions/IntegrationUtils.eol'; %]
[% if (appBlock.isTypeOf(application!`List`)) { %]
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace [%= parentPath.replace("/",".") %] {


    public   partial   class   [%=artifact.prefix   +   appBlock.baseClass.plural   +
artifact.postfix %] {

        /// <summary>
        /// form1 control.
        /// </summary>
        /// <remarks>
        /// Auto-generated field.
        /// To modify move field declaration from designer file to code-behind file.
        /// </remarks>
        protected global::System.Web.UI.HtmlControls.HtmlForm form1;


        /// <summary>
        /// gv[%= appBlock.baseClass.plural %] control.
        /// </summary>
        /// <remarks>
        /// Auto-generated field.
        /// To modify move field declaration from designer file to code-behind file.
        /// </remarks>
```

```
        protected           global::System.Web.UI.WebControls.GridView           gv[%=
appBlock.baseClass.plural %];
    }
}
[% } %]



------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------


[% import '../../functions/IntegrationUtils.eol'; %]
// ------------------------------------------------------------------------------------
----------------------------------
// <copyright file="[%= artifact.name + "." + artifact.extension %]" company="">
//    Copyright (c) Morphosys 2014. All rights reserved.
// </copyright>
// <author>Morphosys</author>
// <summary>
//   Base Class for all the mvp pages.
// </summary>
// ------------------------------------------------------------------------------------
----------------------------------

namespace [%= parentPath.replace("/",".") %]
{
    using System;
    using Microsoft.Practices.CompositeWeb;

    /// <summary>
    /// Base Class for all the mvp pages.
    /// </summary>
    /// <typeparam name="TPresenter">The type of the presenter.</typeparam>
    /// <typeparam name="TView">The type of the view.</typeparam>
    public class [%= artifact.name %]<TPresenter, TView> : System.Web.UI.Page
        where TPresenter : Presenter<TView>
        where TView : class
```

```
    {
        #region Mvp configuration

        /// <summary>
        /// Generic presenter
        /// </summary>
        private TPresenter presenter;

        /// <summary>
        /// Gets or sets the presenter.
        /// </summary>
        /// <value>The presenter.</value>
        /// <author>Esteban Diaz</author>
        public TPresenter Presenter
        {
            get
            {
                return this.presenter;
            }

            set
            {
                if (value == null)
                {
                    throw new ArgumentNullException("value");
                }

                this.presenter = value;
                this.presenter.View = this as TView;
            }
        }

        #endregion
    }
}
```

```
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------

[% import '../../functions/IntegrationUtils.eol'; %]
// ------------------------------------------------------------------------
// <copyright file="[%= artifact.prefix + appBlock.baseClass.name + artifact.postfix
+ "." + artifact.extension %]" company="PSL">
// Copyright (c) KinderMusik 2012. All rights reserved.
// </copyright>
// <author>Morphosys</author>
// <summary>
//   The [%= appBlock.baseClass.name %] DTO
// </summary>
// ------------------------------------------------------------------------

namespace [%= parentPath.replace("/",".") %]
{
    using System;

    [Serializable]
    public class [%=artifact.prefix + appBlock.baseClass.name + artifact.postfix %]
    {
             [% for (field in appBlock.fields){%]

        public      [%=      field.property.toCSharpType()      +      "      "      +
field.property.name.toPascalCase() %] { get; set; }

        [% } %]
    }
}

------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
```

```
[% import '../../functions/IntegrationUtils.eol'; %]
[% if (appBlock.isTypeOf(application!`List`)) { %]
// ------------------------------------------------------------------------------------
----------------------------------
//    <copyright    file="[%=    artifact.prefix    +    appBlock.baseClass.plural    +
artifact.postfix + "." + artifact.extension %]" company="">
//    Copyright (c) Morphosys 2014. All rights reserved.
// </copyright>
// <author>Morphosys</author>
// <summary>
//    The [%= appBlock.baseClass.plural %] Service
// </summary>
// ------------------------------------------------------------------------------------
----------------------------------

namespace [%= parentPath.replace("/",".") %]
{
      [* TODO: fill usings from platform meta/model used artifacts *]
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using Common.DTOs;
    using Data.DataAccess;

    class [%=artifact.prefix + appBlock.baseClass.plural + artifact.postfix %]
    {
        List<[%=   appBlock.baseClass.name   %]DTO>   Get[%=   appBlock.baseClass.name
%]List() [* TODO: fill view name from platform meta/model used artifacts *]
        {
            return [%= appBlock.baseClass.plural %]DAL.Get[%= appBlock.baseClass.name
%]List(); [* TODO: fill view name from platform meta/model used artifacts *]
        }
    }
}
[% } %]

-------------------------------------------------------------------------
```

```
------------------------------------------------------------------------
------------------------------------------------------------------------

[% import '../../functions/IntegrationUtils.eol'; %]
[% if (appBlock.isTypeOf(application!`List`)) { %]
USE [%= app.name %]
GO

-- ============================================
-- Author:          Morphosys
-- Create date: [%= getSystemDate() %]
-- Description:     Gets the [%= appBlock.baseClass.plural %]
-- ============================================

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE PROCEDURE [dbo].[Get_[%= appBlock.baseClass.plural %]]
AS
BEGIN

SELECT [% var count : Integer = 1;
       for (field in appBlock.fields){
              var fieldText : String = '';
              if(count > 1)
              {
                     fieldText = ',';
              }
              fieldText = fieldText + field.property.name.toPascalCase();
       %]
       [%= fieldText %]
    [%       count = count + 1;
    } %]
```

```
FROM [%= appBlock.baseClass.plural %]

END
GO
[% } %]
```