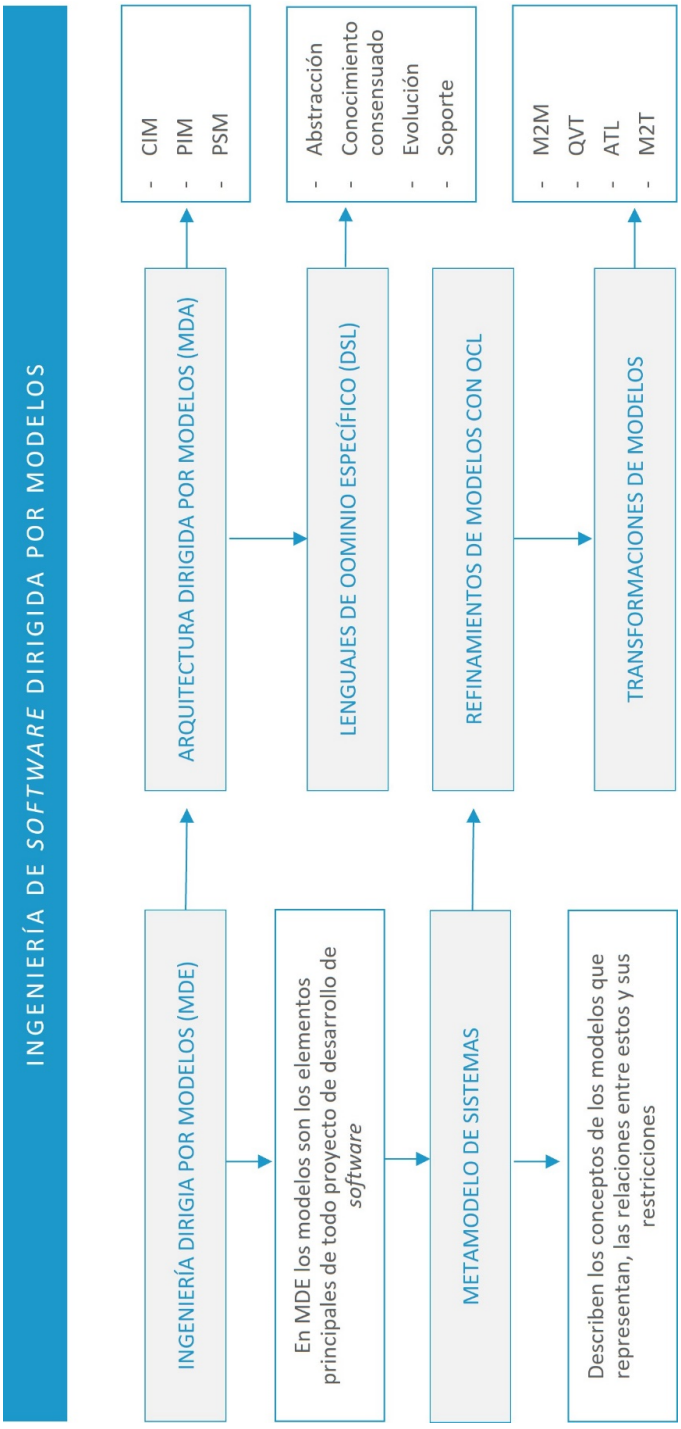
Metodologías, Desarrollo y Calidad en la Ingeniería de Software

Tema 4. Ingeniería de software dirigida por modelos

Índice

Esquema



© Universidad Internacional de La Rioja (UNIR)

4.1. Introducción y objetivos

El **modelado** constituye una herramienta fundamental para el desarrollo de *software.* Con ella el ingeniero de *software* es capaz de simplificar una realidad que es compleja en sí misma, permitiendo su análisis de una forma más conveniente. A través de modelos el ingeniero logra **estructurar** las perspectivas dinámicas y estáticas del futuro sistema.

En este tema se estudian los conceptos asociados a la **ingeniería dirigida por modelos** (*Model-driven engineering*), más conocida y referenciada por sus siglas en inglés, MDE. En este tema se podrá comprobar que desarrollar *software* basado en modelos no es lo mismo que desarrollar *software* dirigido por modelos, ya que, en este último caso, los modelos no son algo complementario y opcional, sino que los modelos constituyen el **eje principal** y la guía para la construcción de sistemas *software* de calidad y reutilizables.

Con el estudio de este tema pretendemos alcanzar los siguientes objetivos:

- Conocer el paradigma de ingeniería dirigida por modelos.
- Comprender los conceptos y fundamentos del desarrollo dirigido por modelos.
- Entender el concepto de metamodelo, ya que juega un papel muy importante en el desarrollo de *software* basado en modelos.
- Conocer el desarrollo dirigido por modelos MDA.
- Estudiar el lenguaje de restricciones de objetos OCL.
- Conocer la transformación de modelos en MDE.

© Universidad Internacional de La Rioja (UNIR)

4.2. Ingeniería dirigida por modelos

Usar modelos para diseñar sistemas, que en la mayoría de los casos **son complejos,** es una cuestión de rigor en cualquier enfoque de Ingeniería. No se puede concebir la construcción de algo tan complejo como, por ejemplo, un edificio o un automóvil, sin realizar una variedad de modelos primero.

Los modelos ayudan en la comprensión y la solución potencial del problema a través de la abstracción.

L a **ingeniería dirigida por modelos** (*Model-driven engineering*, MDE) es un paradigma que pretende ser un elemento clave para la ingeniería de *software,* haciendo que los modelos dejen de utilizarse básicamente para documentar y pasen a convertirse en los **artefactos principales** de todo proyecto de desarrollo de *software* (Bézivin, 2005), reduciendo o eliminando la necesidad de programar.

Con la adopción de MDE para el desarrollo de *software,* los modelos dejan de ser solamente documentación pasiva y complementaria, para pasar a adoptar el rol principal en la descripción, validación, verificación, simulación y generación de código.

La ventaja principal en una primera aproximación es que en un modelo se usan conceptos que serán más próximos al **dominio conceptual del problema** y estarán menos relacionados con elementos de los lenguajes de programación. Esto hará que los lenguajes sean más fáciles de especificar, comprender y mantener.

El término **MDE** fue propuesto inicialmente por Kent (2002), pero este término tiene su origen en una iniciativa del Object Management Group (OMG) conocida como arquitectura dirigida por modelos *(Model Driven Architecture,* MDA) (OMG *Architecture Board* MDA *Drafting Team,* 2001). MDE generaliza conceptos mediante

© Universidad Internacional de La Rioja (UNIR)

el establecimiento de unos principios, donde **MDA** constituye un estándar aplicado de la teoría que presenta.

MDE eleva el nivel de abstracción en el proceso de desarrollo de *software,* lo cual favorece la **gestión de la complejidad** de los sistemas, así como la gestión del cambio inherente de los sistemas *software.*

MDE se fundamenta, básicamente, en dos conceptos: **sistema y modelo;** y en dos relaciones: **representación y conformidad.** La relación de representación es la que se da entre un sistema y un modelo, mientras que la relación de conformidad es la que se da entre un modelo y su metamodelo, donde el metamodelo describe los conceptos que se pueden considerar en un modelo, las relaciones entre estos conceptos y sus restricciones. La Figura 1 ilustra estos fundamentos.

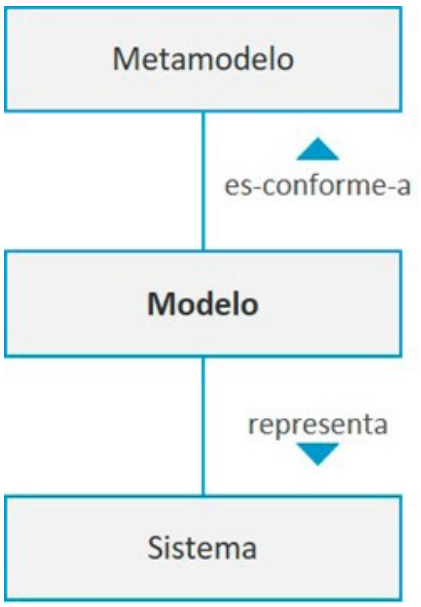


Figura 1. Sistema, modelo y metamodelo en el contexto MDE. Fuente: elaboración propia.

De acuerdo con Bézivin (2005), un **modelo en MDE** queda definido como «una estructura basada en un grafo (dirigido y etiquetado) que representa algunos aspectos de un sistema y es conforme a la definición de otro grafo denominado metamodelo» (p. 43). La Figura 2 transcribe esta definición.

© Universidad Internacional de La Rioja (UNIR)

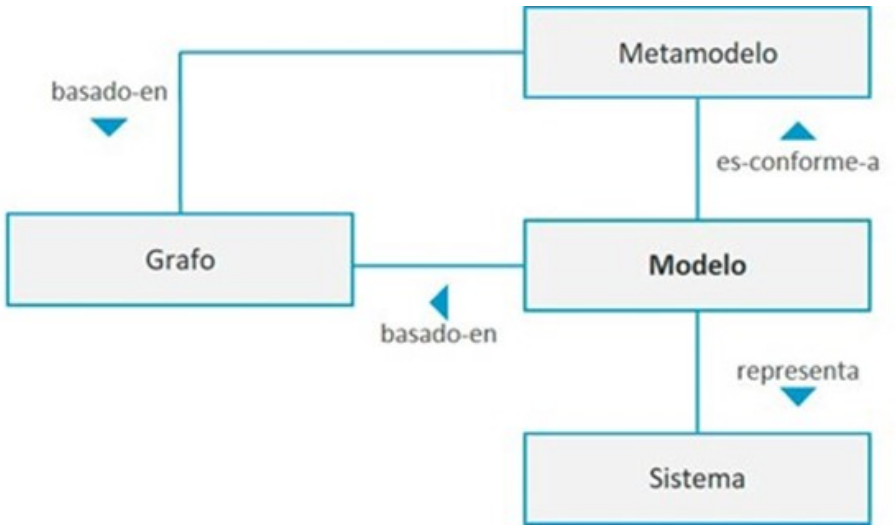


Figura 2. Definición de modelo en el contexto MDE. Fuente: elaboración propia.

Existen otras definiciones de modelo interesantes, como la proporcionada por el OMG en su documento The MDA Foundation Model (OMG, 2010), en la que define un **modelo** como:

> «[…] una representación selectiva de un sistema cuya forma y contenido están basados en un conjunto específico de incumbencias (*concerns*). El modelo se relaciona con el sistema a través de una correspondencia (*mapping*) implícita o explícita» (p. 2).

Ejemplos de modelos que se acogen a esta definición serían: un modelo UML correspondiente a una parte de la aplicación, un modelo de proceso de negocio y un modelo de simulación. La Figura 3 recoge esta definición desde la perspectiva de MDA.

© Universidad Internacional de La Rioja (UNIR)

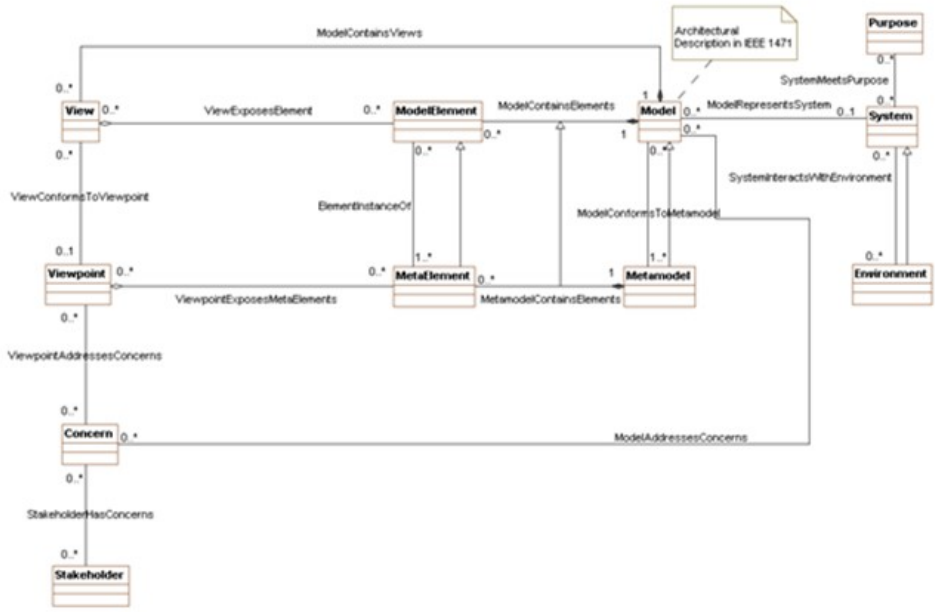


Figura 3. Definición de modelo en el contexto MDA. Fuente: OMG, 2010.

Otra definición de referencia sería la proporcionada por Booch *et al.* (2006), en la que un modelo se ve como «una simplificación de la realidad, siendo esta definida en el contexto del sistema que se está modelando» (p. 468).

En general, un modelo constituye una **representación simplificada** de cierta realidad, de acuerdo a ciertas reglas impuestas por un lenguaje de modelado. La similitud entre modelo y la realidad que representa el modelo será siempre parcial. El modelo solo representará algunas propiedades de la realidad.

El concepto de modelo es demasiado amplio y ambiguo y existen, además, muchos tipos de modelos. Centrado en el área de la ingeniería de *software,* podemos **definir un modelo** de la siguiente forma:

Especificación formal de la función, estructura o comportamiento de un sistema, dentro de un contexto y desde un punto de vista concreto.

© Universidad Internacional de La Rioja (UNIR)

Ahora bien, es importante señalar e insistir en que el modelo y el sistema no son equivalentes. El modelo representa algunas partes del sistema a construir, pero no todas (es decir, **el modelo no es el sistema, sino que lo representa**). Además, el modelo ha de ser conforme a un único metamodelo.

Las características de la relación de un modelo y el sistema real que pretende modelar son:

- **Relación de correspondencia.** Un modelo ofrece una correspondencia con propiedades y relaciones actuales en el sistema original y existente.
- **Relación de simplificación o reducción.** Un modelo solo refleja una selección o subconjunto relevante de las propiedades del sistema real.
- **Relación práctica.** Un modelo deberá ser utilizado en lugar del sistema original con respecto a algún propósito: descriptivo o prescriptivo.

Por tanto, a pesar de que en la práctica los modelos tienden a perder su valor a medida que avanza el proceso de desarrollo del sistema *software* y acaban quedándose obsoletos, para MDE los modelos constituyen **entidades de primera categoría**, artefactos, que siempre representan el sistema.

De acuerdo a Sendall y Kozaczynski (2003), cada modelo debe recoger una **vista diferente del sistema** para que realmente el desarrollo basado en modelos pueda ser útil. Efectivamente, pretender que un único modelo recoja todos los detalles necesarios para especificar un sistema raramente resultará útil y manejable en la práctica.

Además, tal y como destaca Kent (2002), los distintos modelos se podrán expresar en el mismo lenguaje o en lenguajes diferentes que se podrán relacionar a través de correspondencias (*mappings*). Las correspondencias entre modelos descritos en el mismo lenguaje es lo que el Kent denomina **adaptación del modelo** (*model translation*), mientras que las correspondencias entre modelos descritos en lenguajes

© Universidad Internacional de La Rioja (UNIR)

diferentes es lo que se denomina **adaptación del lenguaje** (language translation).

En esta misma línea, Tarr, Ossher, Harrison y Sutton (1999) destacan que, para llegar a cumplir adecuadamente los objetivos del proceso de desarrollo de *software,* en todo proyecto se deben tener en cuenta las **dimensiones del sistema** que se van a contemplar y definir las incumbencias (*concerns*) o vistas necesarias que abarca cada dimensión.

Para Tarr, Ossher, Harrison y Sutton (1999), la separación en vistas va a proporcionar varios beneficios a la hora de implementar una aplicación:

- Reduce la complejidad del sistema.
- Favorece la comprensión del sistema.
- Favorece la reutilización.
- Facilita la evolución del *software.*
- Favorece la trazabilidad entre los distintos artefactos en las distintas fases del proyecto.

El concepto de las dimensiones es importante y merece la pena tenerlo en cuenta. La capacidad para separar y clasificar las diferentes vistas de un sistema constituirá la base para la obtención de un *software* de calidad. El elemento clave lo constituye l a **descomposición simultánea** del sistema a construir en base a las diferentes dimensiones que se hayan establecido.

En resumen, para MDE **«todo es un modelo»** (es decir, todos los artefactos de un proyecto serán creados en base a un lenguaje de modelado) y los modelos son los responsables de **dirigir el proceso de desarrollo** pudiendo automatizar algunas de sus partes mediante el uso de herramientas CASE.

© Universidad Internacional de La Rioja (UNIR)

En este sentido, la **automatización** de tareas es una característica muy importante en todo proceso de desarrollo de *software,* ya que permite la generación de procesos controlados, medibles, auditables y repetibles.

Con lo cual, dado que todos los artefactos generados son modelos, las herramientas CASE adquieren un papel muy importante y, tal y como destaca Kent (2002), además de **minimizar el esfuerzo** para su creación y evolución, permiten **maximizar los beneficios** que se pueden obtener de los modelos, ya que de ellas depende que se puedan realizar simulaciones, transformaciones, comprobaciones, trazabilidad, etc.

Como ejemplos de modelos típicos en la especificación de un sistema *software* están los diagramas UML como:

- Modelos de **casos de uso:** función.
- Modelos de **clases:** estructura.
- Modelos de **componentes:** estructura.
- Modelos de **estados:** comportamiento.

Cada tipo de modelo se basa en **distintas aproximaciones de abstracción.** Por ejemplo, un modelo de flujo de datos se centra en cómo fluyen los datos entre los distintos procesos y las transformaciones que sufren, pero omite o no considera la estructura que tienen estos datos. Sin embargo, en un modelo de datos, el modelo se centra en las estructuras de datos omitiendo su funcionalidad o transformación en un proceso.

© Universidad Internacional de La Rioja (UNIR)

4.3. Metamodelado de sistemas

Al igual que en las definiciones de procesos de desarrollo de *software,* o modelos de proceso, los **metamodelos** juegan un papel muy importante en el desarrollo de *software* basado en modelos, ya que definen lenguajes que permiten expresar modelos del sistema.

Los metamodelos describen las entidades conceptuales de los modelos que representan, las relaciones entre estos conceptos y sus restricciones o reglas.

De acuerdo con Bézivin (2005), al igual que los modelos, los metamodelos se componen de elementos. Los elementos del metamodelo proporcionan un esquema de «tipos» para los elementos del modelo. El tipado de los elementos del modelo se expresa mediante la **metarrelación** entre el elemento del modelo y su metaelemento (es decir, elemento del metamodelo). Por tanto, un elemento del modelo es del tipo de su metaelemento. Entonces, un modelo es conforme a un metamodelo si y solo si cada elemento del modelo tiene su metaelemento definido en el metamodelo.

De esta manera, tal y como se indica en Bézivin (2005), para definir un lenguaje de modelado como, por ejemplo, UML, hay que proporcionar distintos tipos de información (es decir, la información se encuentra en varias partes separadas del metamodelo), tales como: conocimiento de la estructura, de la ejecución, de la visualización, etc. Esta disposición de la información en un metamodelo contribuye precisamente a una clara separación de las vistas o incumbencias. Además, como las vistas se encuentran en partes separadas del metamodelo que se combinan entre sí, se está favoreciendo de manera importante la aplicación de aspectos como la **modularidad y la reutilización.**

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

Una especificación basada en modelos es formal cuando se basa en un lenguaje que tiene una sintaxis y semántica bien definidas y asociadas a cada concepto que representa el lenguaje.

Algunos lenguajes de modelado son, por ejemplo: UML, BPMN, E/R, OWL y XML Schema. Como otro ejemplo, aparte de estos lenguajes de modelado, se podría tomar el **formalismo de las redes de Petri** explicado por Bézivin (2005). En este caso, una red de Petri se puede definir con cuatro tipos de conocimientos:

- **Conocimiento de la estructura.** El conocimiento de la estructura se puede proporcionar con un diagrama de clases con los siguientes conceptos: `Pnet`(el diagrama de visión global), `Place, Transition, Token, relaciones basicRelation` y `numberOfTokens`.

- **Conocimiento de las aserciones.** El conocimiento de las aserciones se puede proporcionar con un lenguaje que permita definir restricciones como *Object Constraint Language* (OCL), que se analizará más adelante, que declaran que el valor del atributo token no puede ser nunca negativo y que una `basicRelation` puede enlazar un `Placecon` una `Transitiono` una `Transition` con un `Place`, pero nunca un `Placecon` un `Place` o una `Transition` con una `Transition`

- **Conocimiento de la ejecución.** El conocimiento de la ejecución se puede proporcionar con la siguiente descripción:

```
function fireable (t:Transition)
{Devuelve verdadero si cada Place de entrada de t tiene al menos un Token,
en caso contrario devuelve falso}
context Pnet action;
repeat
seleccionar de Pnet una Transition t cualquiera tal que fireable(t);
decrementar el número de tokens de cada Place de entrada de t;
incrementar el número de tokens de cada Place de salida de t;
until ninguna Transition t en pNet cumpla fireable(t);
```


© Universidad Internacional de La Rioja (UNIR)

- **Conocimiento de la visualización.** El conocimiento de la visualización se puede proporcionar con la siguiente descripción:
  - Representar una `Transition` con un rectángulo.
  - Representar un `Place`con un círculo.
  - Representar un `Arc`con una flecha.

Con la distribución del metamodelo en distintos tipos de conocimientos se está favoreciendo la **reutilización** en caso de necesitar ampliar el metamodelo con otros elementos.

Por otro lado, el metamodelo también se puede ver como un repositorio. El metamodelo almacena conceptos adoptados mediante acuerdos y el conjunto de reglas del metamodelo. De este modo, los usuarios del repositorio utilizan una **terminología común** para los conceptos clave del desarrollo del sistema *software.* El repositorio evita la malinterpretación de los modelos como consecuencia de un entendimiento incompleto del significado y utilización práctica de los modelos. Con lo cual, las herramientas CASE basadas en un metamodelo determinado podrán utilizar este repositorio para especificar y manipular los diferentes modelos que serán conformes al metamodelo considerado.

Lógicamente, debido al creciente número de metamodelos, se ha reconocido la necesidad de disponer de un *framework* de integración para todos los metamodelos: el metametamodelo. Se tiene pues, que un metamodelo, llamémoslo A, define un modelo, llamémosle B. Pero si este modelo B constituye un metamodelo, entonces el metamodelo A es en realidad un metametamodelo. Es decir, **un metametamodelo define un metamodelo.** Con lo cual, el metametamodelo contempla la definición de los metamodelos.

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

De la misma manera que los modelos se definen para ser conformes a su metamodelo, los metamodelos se definen en términos del metametamodelo.

El metamodelo ha de ser conforme a su metametamodelo. *MetaObject Facility* (MOF) (OMG-MOF, 2014), el estándar del OMG, constituye un ejemplo metametamodelo y el metamodelo de UML (OMG, 2011) es conforme a él. Al igual que los modelos y los metamodelos, el metametamodelo se compone a su vez de elementos. Así, un metamodelo es conforme al metametamodelo si y solo si cada uno de sus metaelementos tiene su metametaelemento definido en el metametamodelo (Bézivin, 2005).

La Figura 4 y la Figura 5, ambas basadas en la explicación anterior, resumen la relación **modelo-metamodelo-metametamodelo** desde diferentes perspectivas. En la Figura 4 se muestran de forma piramidal de los distintos niveles de modelado de la relación. La Figura 6 ilustra el extracto de un ejemplo concreto de un sistema *software* a través de los distintos niveles de modelado.

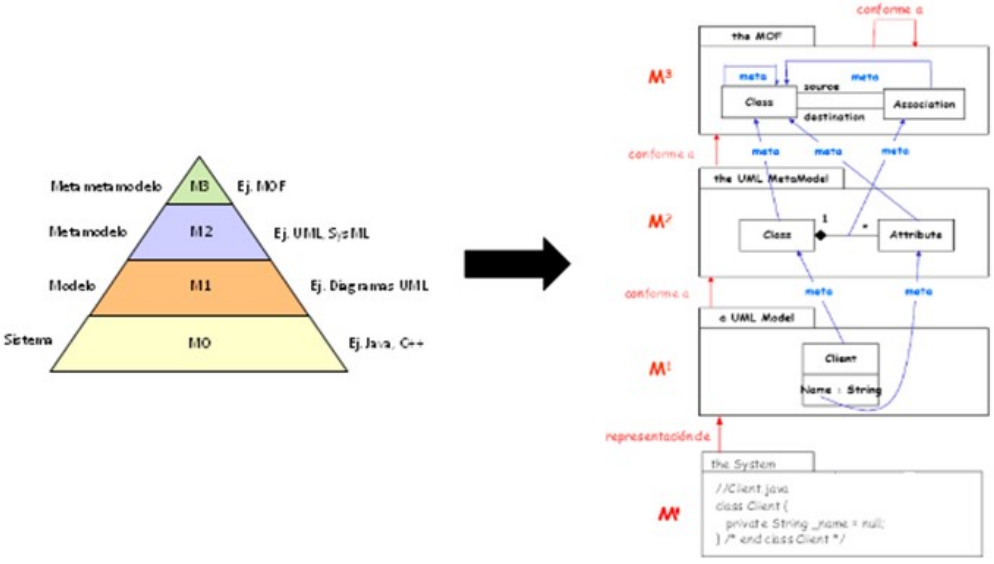


Figura 4. Niveles en la relación modelo-metamodelo-metametamodelo. Fuente: Vicente-Chicote y Alonso, 2007.

© Universidad Internacional de La Rioja (UNIR)

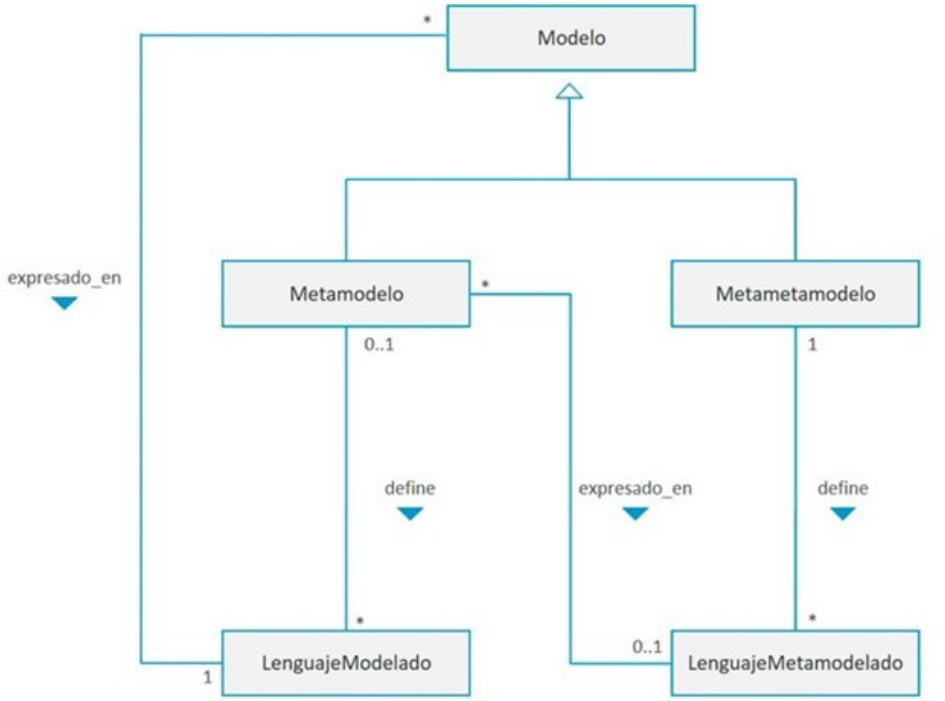


Figura 5. Relación modelo-metamodelo-metametamodelo. Fuente: elaboración propia.

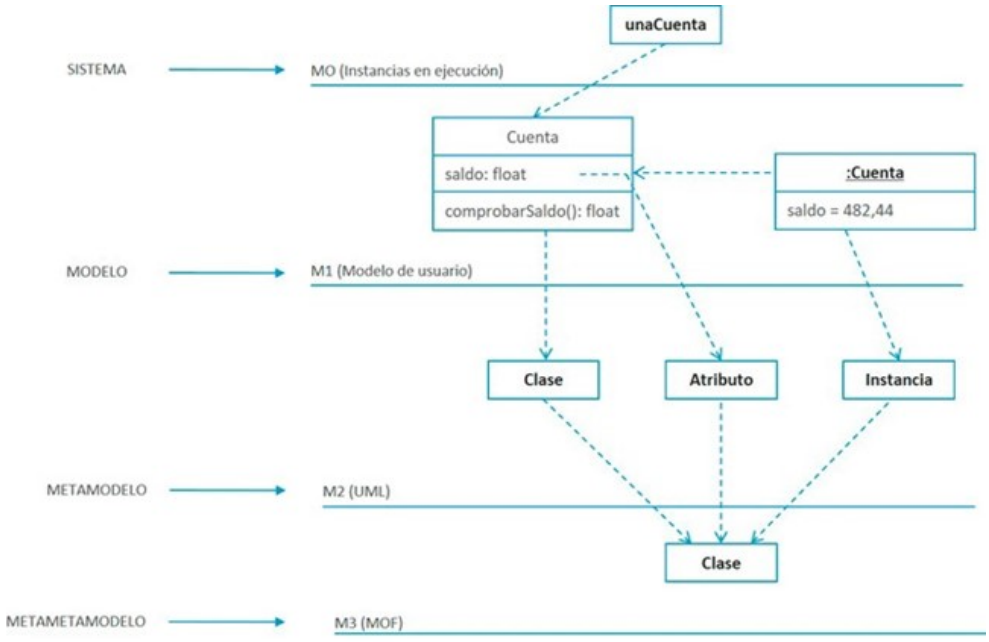


Figura 6. Caso práctico sobre la relación modelo-metamodelo-metametamodelo. Fuente: elaboración propia.

© Universidad Internacional de La Rioja (UNIR)

4.4. Arquitectura dirigida por modelos

La **arquitectura dirigida por modelos** (*Model Driven Architecture*, MDA) es una iniciativa del OMG que propone utilizar modelos durante todas las etapas del proceso de desarrollo de *software* (OMG-MDA, 2014). En este sentido, los modelos se podrían considerar un nuevo lenguaje de «programación» (es decir, se alcanza un nuevo nivel de abstracción con respecto a lenguajes de alto nivel como Java o C++).

La idea principal que subyace en MDA es la separación de la especificación de la funcionalidad de un sistema *software* respecto de los detalles sobre cómo se lleva a cabo esta funcionalidad en una determinada plataforma.

De este modo, se pueden encontrar compiladores capaces de traducir modelos de datos y aplicaciones definidos en MOF (OMG-MOF, 2014) y UML (OMG, 2011) a un lenguaje de alto nivel y a plataformas que implementan sistemas vigentes (OMG-MDA, 2005). En MDA, **MOF es el metametamodelo** que permite definir lenguajes de metamodelado, mientras que **UML es un metamodelo** conforme a MOF que define un lenguaje de modelado.

De acuerdo con Guttman y Parodi (2007), desde una perspectiva holística, MDA se puede considerar como una mejora del ciclo de vida del *software* (es decir, especificación, arquitectura, diseño, desarrollo, despliegue, mantenimiento, e integración) basada en un **modelado formal.**

Se ve así, como los modelos adquieren un papel fundamental en los proyectos, los cuales permanecen siempre actualizados y representan en todo momento el sistema. Los modelos en el **contexto de MDA** quedan definidos como «instancias de metamodelos de MOF compuestos por elementos del modelo y de enlaces entre ellos» (OMG-MDA, 2005). ¿Pero qué es MDA exactamente?, ¿qué abarca MDA?

MDA va a permitir separar la especificación de una funcionalidad de su

© Universidad Internacional de La Rioja (UNIR)

implementación en una plataforma tecnológica concreta. Claramente, esto supone un gran avance ya que se va a favorecer la reutilización de todos los modelos que sean independientes de una implementación concreta, porque, aunque cambie la tecnología, los modelos de más alto nivel, es decir, los que son independientes de la tecnología aplicada, **permanecerán inalterados.** Solo habrá que modificar aquellos modelos que representen una tecnología concreta, para que se adapten a las nuevas propiedades.

MDA no se trata simplemente, como mucha gente cree debido a las herramientas disponibles en el mercado, de una nueva técnica basada en UML para generar código. MDA va mucho más allá. Si bien es cierto que MDA en su origen estaba basado en UML, también es cierto que un uso muy común de MDA está orientado a la **generación automática de código** y de otros **artefactos** de ciclo de vida del *software.*

Sin embargo, tal y como se destacan Guttman y Parodi (2007), MDA se perfila como un **gran potencial** para la recolección de requisitos, establecimiento de estándares arquitectónicos, mantenimiento de la trazabilidad y para la comunicación efectiva entre la parte de negocio y la parte tecnológica, es decir, de las tecnologías de la información en las organizaciones.

Así pues, MDA tiene que ver con todo aquello que sea lenguajes de modelado, entornos de desarrollo, herramientas que permitan especificar lenguajes de modelado de dominio específico, tecnología para el intercambio de modelos a través de un conjunto de herramientas, transformaciones que usan reglas de negocio para el desarrollo de modelos *software,* traducción de modelos a otros tipos de lenguajes, utilización de modelos en la ingeniería de sistemas, redirección de modelos y *software* de una plataforma a otra y, finalmente, por supuesto, generación de sistemas *software* a partir de modelos y especificaciones de las arquitecturas de las plataformas (OMG-MDA, 2005).

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

En realidad, el concepto MDA no es algo nuevo. Aunque el término MDA fue propuesto inicialmente en el año 2001 (OMG *Architecture Board MDA Drafting Team,* 2001), la técnica que MDA propone para el desarrollo de sistemas ya se había utilizado **tiempo atrás** para generar sistemas integrados y sistemas de tiempo real, tal y como señala Frankel (2003). MDA debe su origen a la gran variedad de tecnologías, paradigmas, etc., que existen en la actualidad y que van apareciendo.

Como ya se ha dicho anteriormente, MDA parte de la idea de elevar el nivel de abstracción en el desarrollo de sistemas *software* a los modelos y propone utilizar modelos para dirigir las distintas actividades que comprende todo desarrollo de un sistema *software:* análisis, diseño, construcción, despliegue, operación, mantenimiento y modificación o evolución. A través de **transformaciones de modelos** se podrá convertir un modelo que especifica el sistema en otro modelo del mismo sistema, pero para una plataforma concreta.

MDA define **tres capas** de arquitectura para los sistemas en función de su nivel de abstracción (OMG-MDA, 2014):

- *Computation Independent Model* (CIM). Vista del sistema desde la perspectiva del **sistema de información** y de la **implementación** (plataforma tecnológica). CIM es lo que se conoce habitualmente como **modelo de negocio** o **modelo de dominio.**

Con lo cual, las instancias de un modelo **CIM** se corresponden con «cosas reales», pertenecientes al mundo real y que son independientes del sistema de información (ya existen y se dan en el mundo sin la aplicación a desarrollar), es decir, que las instancias de un modelo CIM no representan «cosas» del sistema de información (OMG-MDA, 2014).

Por tanto, aunque puede ofrecer mucho detalle del dominio del problema, desde el punto de vista de desarrollo del sistema (es decir, para implementación del sistema) se trata del **modelo más abstracto** en MDA.

© Universidad Internacional de La Rioja (UNIR)

Dicho de otra manera, este modelo se correspondería con un modelo expresado en el lenguaje propio de los expertos del dominio, que es independiente (es decir, no aporta ningún tipo de información) de la arquitectura e implementación del sistema.

- *Platform Independent Model* (PIM). Vista del sistema desde la perspectiva **independiente de la plataforma.** PIM es lo que se conoce en MDA como **modelo lógico de diseño** (OMG-MDA, 2014), es decir, es un modelo del sistema con un alto nivel de abstracción, independiente de tecnologías de implementación concretas, y que se puede expresar en un lenguaje de propósito general o específico del dominio en el que se va a utilizar el sistema.

De acuerdo con Arlow y Neustadt (2005), un PIM se debería construir teniendo en cuenta los conceptos del CIM. Ahora bien, tal y como se señala en OMG-MDA (2005), Arlow y Neustadt (2005) y en Brown (2004), es muy importante tener en cuenta que el concepto «independiente de la plataforma» es un **término relativo,** es decir, un modelo PIM puede ser considerado independiente de ciertas tecnologías (puede mostrar características comunes a un conjunto de plataformas similares), pero a la vez ser específico de otras lo que le haría ser PSM, ya que contiene detalles concretos para el despliegue en una plataforma concreta.

Así, lo que para una persona es un **PIM,** para otra persona podría ser un **PSM.** Por ejemplo, tal y como explica Brown (2004), un modelo *Common Object Request Broker Architecture* (CORBA) (Pope, 1998) puede ser considerado PIM por ser independiente del sistema operativo, pero, en cambio, también se podría considerar PSM al ser específico con respecto al *middleware* de comunicación que se utiliza. Con lo cual, cuando se denote un modelo como PIM sería conveniente definir de qué plataformas es independiente.

- *Platform Specific Model* (PSM). Vista del sistema desde la perspectiva de la **plataforma específica.** PSM es lo que se conoce en MDA como **modelo de implementación** (OMG-MDA, 2014), es decir, es un modelo adaptado que

© Universidad Internacional de La Rioja (UNIR)

especifica el mismo sistema que se especifica en el PIM, pero en términos de una implementación concreta.

Un PSM combina las especificaciones que contiene el PIM con información más o menos detallada que indica cómo se va a utilizar un tipo de plataforma concreta en el sistema. De este modo, cada PSM se transformará en código utilizando un lenguaje de programación de **alto nivel** (transformación modelo-a-texto).

Sin embargo, hay que señalar que, en algunos casos, el código de la aplicación podría considerarse también un modelo (por ejemplo, un programa escrito en Java podría considerarse un modelo Java que constituye una abstracción del código máquina que genera el compilador), por lo que también podría clasificarse como PSM (transformación modelo-a-modelo). En este caso el PSM se correspondería con la especificación (o descripción) del sistema en forma de código fuente.

En resumen, en MDA se utilizan modelos que pueden estar expresados en distintos lenguajes para representar distintas vistas (CIM, PIM y PSM), que constituyen las dimensiones de modelado para el sistema *software.*

La transformación entre las diferentes vistas permite pasar, a través de correspondencias de los modelos (ver Figura 7), de niveles más altos de abstracción a niveles más concretos que permitirán obtener el **sistema implementado,** es decir, el código (ver Figura 8). Sería como una evolución del modelo a través de las distintas vistas que representa (ver Figura 9).

Figura 7. Desarrollo de *software* visto como transformaciones entre diferentes vistas con el enfoque MDA. Fuente: elaboración propia.

© Universidad Internacional de La Rioja (UNIR)

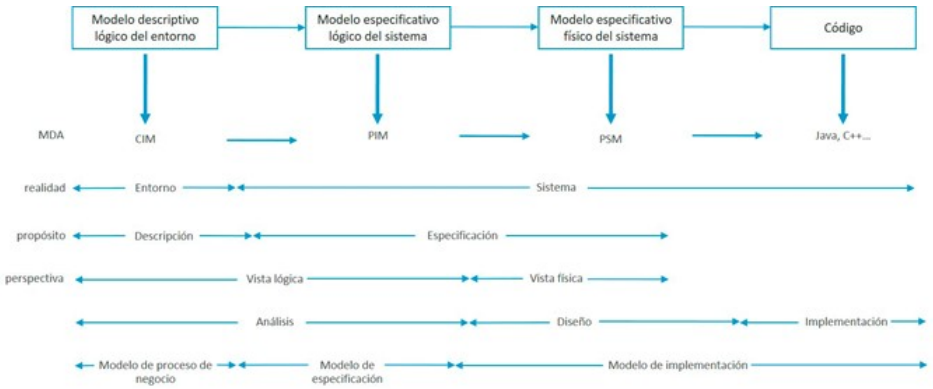


Figura 8. Distribución de los modelos a lo largo del proceso de desarrollo de software con el enfoque MDA. Fuente: elaboración propia.

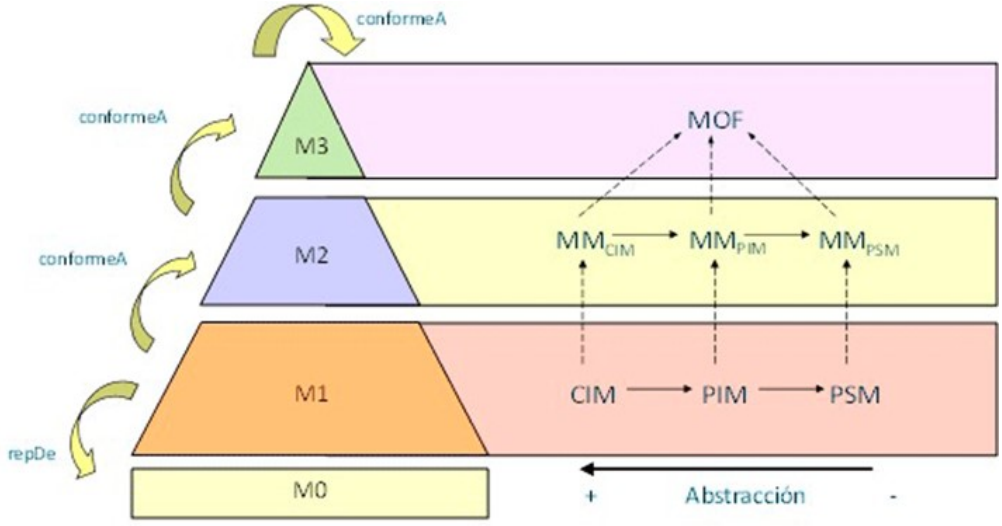


Figura 9. MDA y la pirámide de las cuatro capas de metamodelado del OMG. Fuente: Vicente-Chicote y Alonso, 2007.

© Universidad Internacional de La Rioja (UNIR)

4.5. Lenguajes de dominio específico

Los lenguajes de dominio específico (*Domain-Specific Languages,* DSL) son lenguajes que han sido diseñados específicamente para un dominio, contexto o compañía concreta de cara a facilitar la tarea de tener que describir cosas que se dan en dicho dominio, contexto o compañía (Brambilla, Cabot y Wimmer, 2012, p. 13).

> Serían ejemplos de DSL: HTML para el desarrollo de páginas web, VHDL para lenguajes de descripción de *hardware* y SQL para las bases de datos.

Si el lenguaje es de modelado, entonces se le conoce como **lenguaje de modelado de dominio específico** (*Domain-Specific Modeling Language,* DSML), aunque normalmente se les suele identificar también como **DSL.** Desde este punto de vista, un DSL se puede ver como un metamodelo que incluye conceptos propios de un determinado dominio.

Por ejemplo, un DSL para domótica incluiría conceptos como sensor de temperatura, calefacción automática, sensor de luz de persiana automática, etc. En este caso, cada metamodelo puede tener asociado un DSL concreto que describe los distintos elementos del modelo, su disposición, sus relaciones y sus restricciones. La ventaja de utilizar DSL es que así se consigue **recoger las necesidades** de los *stakeholders* de un sistema para un dominio concreto. La Figura 10 muestra un ejemplo de DSL para modelado.

A pesar de que UML se trata de un lenguaje de propósito general y podría aplicarse a varios dominios, hay dominios para los que este lenguaje no dispone de los elementos adecuados y es por ello por lo que se ha de hacer uso de los perfiles (*profiles*) que define UML si se desea definir un DSL concreto. El problema es que

© Universidad Internacional de La Rioja (UNIR)

con los perfiles de UML, aunque permite crear DSL, y aunque solamente se están utilizando un conjunto de elementos de modelado del metamodelo de UML, se está «arrastrando» todo el metamodelo completo con el perfil, no solo con la parte que interesa para el DSL.

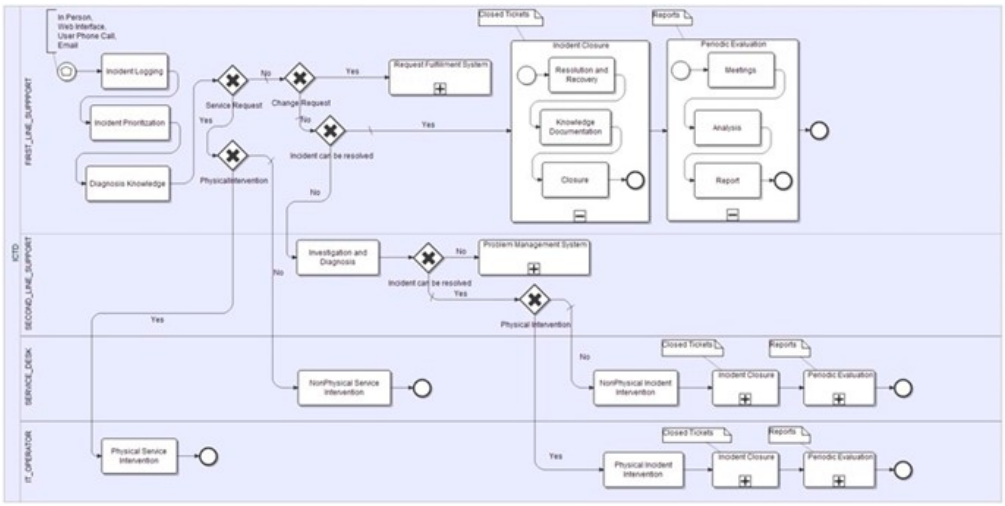


Figura 10. Modelo BPMN del proceso de Gestión de incidencias de ITIL v3. Fuente: OMG-BPMN, 2011.

Para que un DSL sea realmente útil, Brambilla, Cabot y Wimmer (2012, p. 70) definen una serie de **principios** que todo DSL debe cumplir:

- El DSL ha de proporcionar un nivel de abstracción adecuado para el desarrollador, debe ser intuitivo y facilitarle el trabajo, no complicárselo más.
- El DSL no debe depender del conocimiento de una única persona, sino que su definición ha de estar consensuada por varios expertos del dominio y haber recibido algún tipo de evaluación.
- El DSL debe evolucionar y mantenerse actualizado de acuerdo a las necesidades del dominio en estudio, porque en caso contrario, con el tiempo, se quedará obsoleto y no se utilizará.
- El DSL debe venir acompañado de herramientas y métodos que le den soporte, porque, aunque los expertos del dominio se preocupan de maximizar su

© Universidad Internacional de La Rioja (UNIR)

productividad trabando en la definición de su dominio, no están dispuestos a perder tiempo definiendo métodos y herramientas asociados a dicho dominio.

© Universidad Internacional de La Rioja (UNIR)

4.6. Refinamientos de modelos con OCL

Object Contraint Language (OCL) es un lenguaje que tuvo su origen en un intento de resolver las limitaciones de UML a la hora de especificar determinados aspectos en los modelos de diseño del *software.* OCL es un **lenguaje formal de propósito general** adoptado como un estándar por el OMG (OMG-OCL, 2014).

> Las expresiones escritas en OCL añaden información vital de una manera clara y sin ningún tipo de ambigüedad en los modelos orientados a objetos y en otros artefactos de modelado de objetos (Warmer y Kleppe, 2003).

En UML 1.1 la información añadida por OCL era exclusivamente para **especificar restricciones** sobre uno o más valores de un (o parte de un) modelo o sistema orientado a objetos (Warmer y Kleppe, 2003). Sin embargo, en UML 2, OCL no se utiliza exclusivamente para definir restricciones, sino que también se puede utilizar para **especificar reglas** definidas de manera formal en el diseño de DSL y para especificar **otro tipo de operaciones** que actúen sobre los modelos, como, por ejemplo, a la hora de establecer patrones de correspondencia en las transformaciones modelo-a-modelo y modelo-a-texto (Cabot, 2012, 2020).

OCL es un lenguaje de especificación con las siguientes características (Bambrilla, Cabot y Wimmer, 2012, p. 74):

- **Tipado.** Cada expresión OCL tiene un tipo, evalúa a un valor de ese tipo y debe ser conforme a las reglas y operaciones de dicho tipo.
- **Sin efectos laterales.** Las expresiones OCL pueden hacer consultas o imponer restricciones en el estado del sistema, pero en ningún caso lo va a modificar.
- **Declarativo.** OCL no incluye construcciones imperativas.
- **Especificación.** OCL es un lenguaje de especificación que no incluye ningún detalle

© Universidad Internacional de La Rioja (UNIR)

de implementación, ni ofrece guías para la implementación del sistema.

OCL se suele utilizar también como complemento en la definición de los metamodelos, proporcionando un **conjunto de reglas textuales** que todo modelo, que sea conforme a dicho metamodelo, ha de cumplir.

Las restricciones en OCL se presentan como invariantes definidas en el contexto de un tipo específico, conocido como **tipo de contexto.** Como cuerpo de la restricción se define una expresión booleana que ha de satisfacerse en todas las instancias del tipo de contexto.

La siguiente expresión define un ejemplo de invariante donde en cada instancia de `Convocatoria`el atributo fin debe ser **mayor** que el atributo inicio (Bambrilla, Cabot y Wimmer, 2012, p. 75):

```
Context Convocatoria
inv: self.end > self.start
```

Otro ejemplo sería la siguiente expresión, donde al atributo `numParticipantes` de una instancia de la clase `Convocatoria`se le asigna el total del número de participantes que se dan en todos los programas que están incluidos dentro de dicha convocatoria a través de la colección `programas.participantes`, sin que cuente más de una vez al mismo objeto participante, que podría asistir a distintos programas dentro de una misma `Convocatoria` (Warmer y Kleppe, 2003, p. 32):

```
Context Convocatoria
inv: numParticipantes:
numParticipantes = programas.participantes -> asSet() -> size()
```


© Universidad Internacional de La Rioja (UNIR)

4.7. Transformaciones de modelos

Para MDE, un proceso de desarrollo de *software* se puede modelar como un conjunto de **transformaciones de modelos** que toma modelos de entrada y genera modelos de salida a través de una **serie de correspondencias** (*mappings*).

La transformación en sí misma es vista también como un modelo en MDE, por lo que constituye un artefacto más del proyecto. Por tanto, los modelos contemplarán las distintas dimensiones del sistema y los modelos se relacionarán unos con otros a través de las correspondencias que se definan.

Por su parte, en MDA, el metametamodelo MOF (OMG-MOF, 2014) constituye el lenguaje de metamodelado que permite **definir transformaciones** entre distintos lenguajes de modelado, como, por ejemplo, UML (OMG, 2011). En MDA, las transformaciones aceptan elementos de entrada y producen elementos de salida (OMG, 2010). *Meta Object Facility (MOF) Query/View/Transformation* (QVT) es el estándar del OMG para MDA que pretende facilitar la definición y automatización de transformaciones de modelos que sean instancias de MOF (OMG, 2015). La Figura 11 muestra, de manera muy simplificada, el esquema de la transformación MDA de un modelo Ma a un modelo Mb a través de la aplicación de un modelo de transformación Mt.

© Universidad Internacional de La Rioja (UNIR)

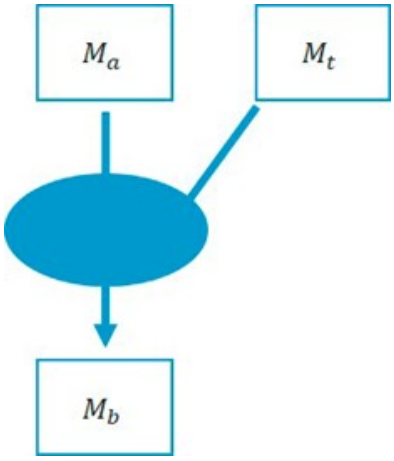


Figura 11. Transformación de modelos en MDA. Fuente: elaboración propia.

Según Lopes, Hammoudi, Bézivin y Jouault (2005), en el **proceso de transformar** un PIM en un PSM se consideran los siguientes conceptos: un metametamodelo (es decir, MOF en el caso de MDA), un modelo origen, un modelo destino, los correspondientes metamodelos de los modelos origen y destino (por ejemplo, el metamodelo de UML), un modelo de correspondencia, y un modelo de transformación.

El modelo de correspondencia especifica las correspondencias que se dan entre el metamodelo origen y el metamodelo destino. Por su parte, el modelo de transformación se genera a partir del modelo de correspondencia utilizando un lenguaje de transformación (por ejemplo, en el caso de MDA, se utilizaría el lenguaje QVT). Por tanto, antes de poder definir una transformación será necesario disponer de la **especificación de las correspondencias.**

Por su parte, Sendall y Kozaczynski (2003) definen **tres técnicas arquitectónicas** para la definición de transformaciones:

- **Manipulación directa del modelo.** Acceso a una representación interna del modelo y capacidad para manipular esa representación a través de un conjunto de API. Esta técnica tiene la ventaja de que el lenguaje utilizado para acceder y manipular las API suele ser un lenguaje de propósito general como Java o Visual Basic, es decir,

© Universidad Internacional de La Rioja (UNIR)

lenguajes con los que los desarrolladores están muy familiarizados.

El problema de esta técnica es que las API suelen **restringir** el tipo de transformaciones que se pueden realizar y, además, al tratarse de lenguajes de propósito general están bastante **limitados** a la hora de especificar ciertos tipos de transformaciones e impiden alcanzar ciertos niveles de abstracción. Aplicado a MDA, la semántica de acción de UML (*Action Semantics for UML,* AS) permite a los desarrolladores definir la especificación del comportamiento a mayor nivel de abstracción. AS permite describir acciones con un alto nivel de abstracción que se pueden analizar y ejecutar de manera automática.

La **semántica de acción** más el **lenguaje de acción** permiten una especificación detallada de los elementos de comportamiento de los modelos UML (por ejemplo, las operaciones de las clases) directamente en el modelo UML. Esta característica es la que hace posible que la especificación UML sea computacionalmente completa y que se puedan obtener modelos UML ejecutables. Así, el conjunto de acciones formales hace que el modelo PIM en UML sea ejecutable.

- **Representación intermedia.** Exportación del modelo a un formato estándar como, por ejemplo, XML. Este formato podrá ser procesado por una herramienta adecuada que realizará la transformación. Aplicado a MDA, como formato de intercambio para modelos que sean conformes a MOF (por ejemplo, UML), se dispone del estándar XML *Metadata Interchange* (XMI) (OMG-XMI, 2014).

-
  - **Soporte para el lenguaje de transformación.** Disponibilidad de un lenguaje que proporciona un conjunto de construcciones que permiten, ya de manera explícita, expresar, componer y aplicar transformaciones. Como ejemplo de esta técnica, para aplicación en MDA, la especificación QVT (OMG, 2015) representa un lenguaje específico para describir transformaciones de modelos que sean conformes al estándar MOF (OMG-MOF, 2014).

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

La interrelación entre los modelos y las transformaciones es mucho más profunda de lo que a primera vista puede parecer, ya que va a representar la arquitectura fundamental de los artefactos que se construyen durante un proyecto *software:* en cualquier proyecto habrá un conjunto de modelos que proporcionan distintas vistas del sistema a construir y las correspondencias que relacionan esos modelos.

Transformaciones modelo-a-modelo

En general, una **transformación modelo-a-modelo** (M2M) se define como «un programa que toma uno o más modelos como entrada y produce un o más modelos como salida» (Brambilla, Cabot y Wimmer, 2012, p. 107). En la mayoría de los casos, las transformaciones de un modelo se transformarán en otro modelo **(transformación uno-a-uno)**, como sería el caso de transformar un diagrama de clases UML en un modelo relacional de bases de datos; pero en algunas situaciones será necesario transformar un modelo en varios modelos **(transformación uno-a-muchos)** o varios modelos en un único modelo **(transformación muchos-a-uno)**.

Un ejemplo de muchos-a-uno se daría, por ejemplo, cuando el objetivo sea unificar varios diagramas de clases UML en una única vista integrada (Brambilla, Cabot y Wimmer, 2012, p. 107). La Figura 12 muestra el patrón de una transformación modelo-a-modelo haciendo uso de un lenguaje de transformación.

© Universidad Internacional de La Rioja (UNIR)

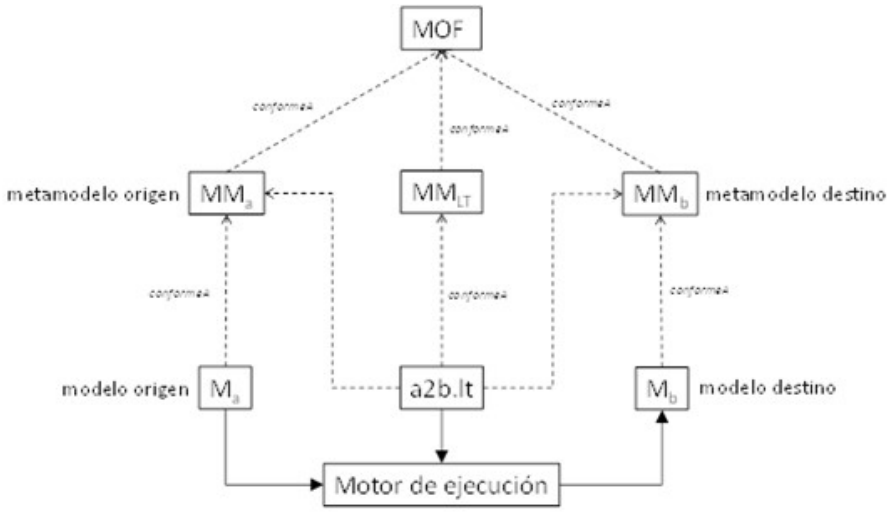


Figura 12. Patrón de transformación modelo-a-modelo. Fuente: Brambilla, Cabot y Wimmer, 2012.

Dentro de la categoría de las transformaciones modelo-a-modelo se puede distinguir entre (Czarnecki y Helsen, 2006):

- **Transformaciones endógenas**, donde el modelo origen (entrada) y el modelo destino (salida) son conformes a un mismo metamodelo.
- **Transformaciones exógenas,** donde el modelo origen y el modelo destino son conformes a metamodelos distintos.

La Figura 13 ilustra los dos tipos de transformaciones.

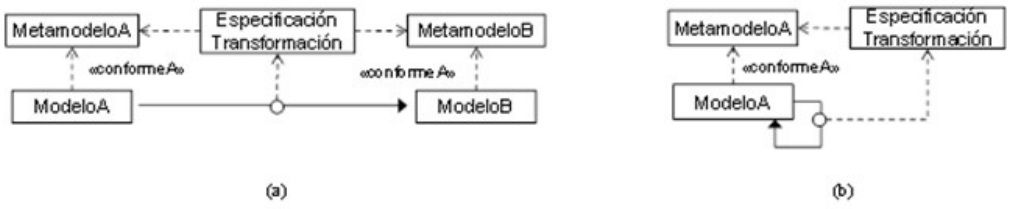


Figura 13. Tipos de transformaciones de modelo-a-modelo: (a) exógenas y (b) endógenas. Fuente: Brambilla, Cabot y Wimmer, 2012.

QVT

QVT sería un ejemplo de lenguaje para **transformaciones modelo-a-modelo** de

© Universidad Internacional de La Rioja (UNIR)

tipo **exógeno** que, como ya ha mencionado anteriormente, utiliza el enfoque MDA. QVT define tres lenguajes:

- ***Relations.*** En el lenguaje *Relations* una transformación entre modelos se especifica con un conjunto de relaciones que se deben tener para que la transformación se lleve a cabo con éxito. Una relación va a ser la unidad básica de especificación de una transformación. Una transformación se define como un conjunto de modelos que se transforma en otro conjunto de modelos. El lenguaje *Relations* presenta una notación tanto gráfica como textual. Además, el lenguaje *Relations* crea de manera implícita trazas entre clases y objetos para poder registrar lo que ha ocurrido durante la ejecución de una transformación.

- ***Core.*** El lenguaje *Core* es tan potente como el lenguaje *Relations,* pero más simple, aunque las descripciones de transformaciones utilizando este lenguaje resultan mucho más complicadas que con el lenguaje *Relations.* La trazabilidad en el lenguaje Core se realiza de manera explícita.

- ***Operational Mappings.*** El lenguaje *Operational Mappings* permite definir transformaciones utilizando una técnica imperativa completa (transformaciones operacionales). Además, el lenguaje también se puede utilizar para complementar las transformaciones relacionales con operaciones imperativas que implementan las relaciones.

Por tanto, QVT presenta dos partes bien diferenciadas: una **parte declarativa** que determina las relaciones entre modelos MOF (lenguaje *Core* y lenguaje *Relations*) y una **parte imperativa** que proporciona la implementación operacional (lenguaje *Operational Mappings*).

ATL

ATL (Jouault, Allilaire, Bézivin y Kurtev, 2008) sería otro ejemplo de lenguaje para **transformaciones modelo-a-modelo** de tipo **exógeno** desarrollado por el grupo de investigación AtlanMod (Inria y Lina), que surgió como respuesta a la

© Universidad Internacional de La Rioja (UNIR)

propuesta OMG MOF/QVT *Request For Proposals* (RFP) y que forma parte de la plataforma *Atlas Model Management Architecture* (AMMA).

ATL define su sintaxis abstracta a través de un metamodelo. Esto significa que cada transformación ATL será a su vez un modelo.

Al igual que QVT, el lenguaje ATL es un **lenguaje híbrido** ya que presenta una parte declarativa y una parte imperativa. En ATL las transformaciones son unidireccionales (para transformaciones bidireccionales sería necesarios definir dos transformaciones, una por cada dirección), operan sobre modelos origen de solo-lectura y generan modelos destino de solo-escritura. Además, ATL también da soporte a la trazabilidad de manera automática.

ATL se acompaña de un conjunto de herramientas: el motor de transformaciones ATL, el entorno de desarrollo integrado (*Integrated Development Environment,* IDE) ATL basado en Eclipse, y el depurador ATL.

Transformaciones modelo-a-texto

La **transformación modelo-a-texto** (M2T) es la que se asocia a la generación de código. Las transformaciones M2T vinculadas al área de MDE están enfocadas a la generación de código con el objetivo de conseguir la transición (semi) automática modelo a código fuente en un lenguaje de programación. La Figura 14 ilustra la generación de código en MDA de acuerdo al proceso unificado y la Figura 15 muestra el esquema de generación de código de acuerdo al enfoque MDE.

© Universidad Internacional de La Rioja (UNIR)

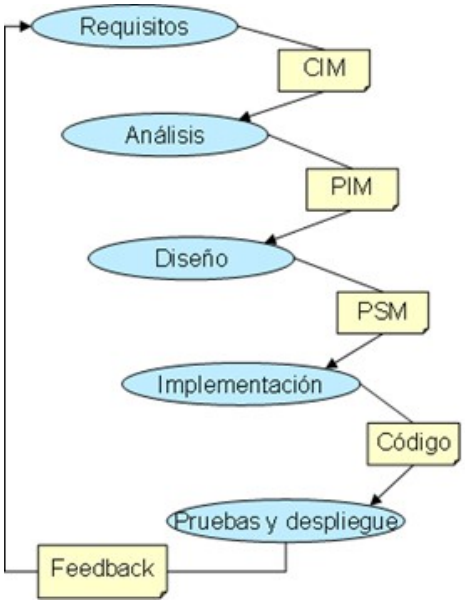


Figura 14. Generación de código en MDA siguiendo el proceso unificado. Fuente: Jouault *et al*, 2008.

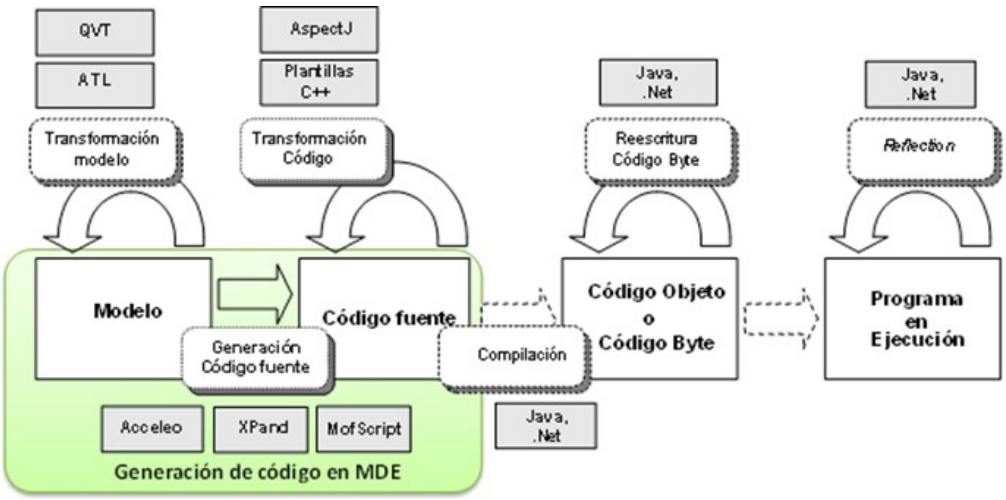


Figura 15. Generación de código en MDE. Fuente: Jouault *et al,* 2008.

Las transformaciones M2T sirven para establecer un enlace entre las herramientas de análisis y las plataformas de ejecución (Jouault *et al*., 2008). La Figura 16 ilustra un ejemplo de transformación de un extracto de un modelo UML al código fuente en Java correspondiente.

© Universidad Internacional de La Rioja (UNIR)

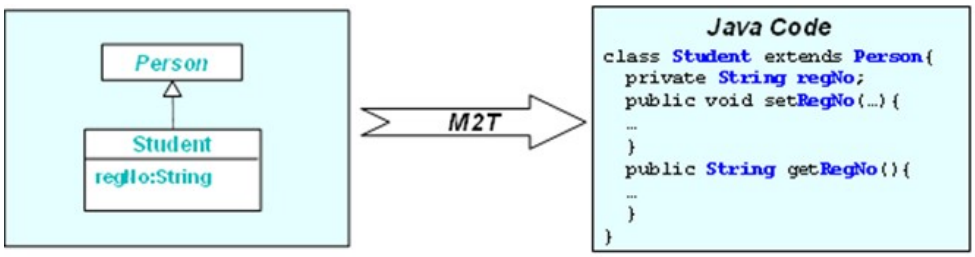


Figura 16. Extracto de un modelo UML y el código fuente correspondiente. Fuente: Jouault *et al*, 2008

© Universidad Internacional de La Rioja (UNIR)

4.8. Referencias bibliográficas

Arlow, J. y Neustadt, I. (2005). *UML 2 and the Unified Process. Practical Object-Oriented Analysis and Design* (2ª Ed).Addison-Wesley.

Bézivin, J. (2005). Model Driven Engineering: An Emerging Technical Space. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering* (GTTSE 2005).

Booch, G., Rumbaugh, J. y Jacobson, I. (2006). *El Lenguaje Unificado de Modelado. UML 2.0* (2ª ed.). Pearson Addison-Wesley.

Brambilla, M., Cabot, J. y Wimmer, M. (2012*). Model-Driven Software Engineering in Practice.* Synthesis Lectures on Software Engineering.

Brown, A. W. (2004). Model driven architecture: Principles and practice. *Journal of Software and System Modeling (SoSym), 3*(4), pp. 314-327.

Cabot, J. (2012). *Why you need to learn OCL*. MOdelinng LAnguages. https://modeling-languages.com/why-you-need-to-learn-ocl/.

Cabot, J. (2020). *The Ultimate Object Constraint Language (OCL) tutorial.* https://modeling-languages.com/ocl-tutorial/.

Czarnecki, K. y Helsen, S. (2006). Feature-based survey of model transformation approaches. *Journal of IBM Systems Journal, 45*(3), pp. 621-645.

Frankel, D. S. (2003). *Model Driven Architecture. Applying MDA to Enterprise Computing*. Wiley.

Guttman, M. y Parodi, J. (2007). *Real-Life MDA. Solving Business Problems with Model Driven Architecture.* Morgan Kaufmann Publishers.

© Universidad Internacional de La Rioja (UNIR)

Jouault, F., Allilaire, F. Bézivin, J. y Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming, 72*(12), 31-39.

Kent, S. (2002). Model Driven Engineering. En *Actas de Third International Conference on Integrated Formal Methods* (pp. 286-298).

Lopes, D., Hammoudi, S., Bézivin, J. y Jouault, F. (2005). Mapping Specification in MDA: From Theory to Practice. En *Actas de First International Conference on Interoperability of Enterprise Software and Applications.* (pp. 253-264). Springer-Verlag.

OMG (2010). *The MDA Foundation Model.* ORMSC DRAFT. http://www.omg.org/cgi-bin/doc?ormsc/10-09-06.

OMG (2011). *OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.4.1.* Document formal. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

OMG-BPMN (2011). *Business Process Model and Notation (BPMN).* https://www.omg.org/spec/BPMN/2.0/PDF/.

OMG (2015). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.2.* Document formal. http://www.omg.org/spec/QVT/1.2/PDF/.

OMG Architecture Board MDA Drafting Team (2001). *Model Driven Architecture.* Document ormsc. http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.

OMG-MDA (2005). *MDA Guide Revision Draft, Version 00.03.* Document ormsc.

OMG-MDA (2014). *Model Driven Architecture (MDA) MDA Guide rev. 2.0.* Document ormsc. http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

OMG-MOF (2014). *Meta Object Facility (MOF) Core Specification, Version 2.4.2.* Document formal. http://www.omg.org/spec/MOF/2.4.2/PDF/.

© Universidad Internacional de La Rioja (UNIR)

OMG-OCL (2014). *Object Constraint Language (OCL), Version 2.4.* Document formal. http://www.omg.org/spec/OCL/2.4/PDF/.

OMG-XMI (2014). *XML Metadata Interchange (XMI) Specification, Version 2.4.2.* Document formal. http://www.omg.org/spec/XMI/2.4.2/PDF/.

Pope, A. (1998). *The Corba reference guide: understanding the common object request broker architecture.* Addison-Wesley.

Sendall, S. y Kozaczynski, W. (2003). Model Transformation: The Heart and Soul of Model-Driven Software Development. *Journal of IEEE Software,* pp. 42-51.

Tarr, P., Ossher, H., Harrison, W. y Sutton, S.M. (1999). N degrees of separation: Multidimensional separation of concerns. En *Actas de 21st International Conference on Software Engineering (ICSE'99)* (pp. 107-119).

Vicente-Chicote, C. y Alonso D. (2007). Tutorial: Herramientas Eclipse para Desarrollo de Software Dirigido por Modelos. XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2007). *Actas de Talleres y Tutoriales de las Jornadas de Ingeniería del Software y Bases de Datos, 1*(8).

Warmer, J. y Kleppe, A. (2003). *The Object Constraint Language. Getting Your Models Ready for MDA* (2ª ed.). Pearson Addison-Wesley.

© Universidad Internacional de La Rioja (UNIR)

Ingeniería de software

Página de Ingeniería de Software.

Web creada por Jordi Cabot, experto y apasionado de la ingeniería de *software,* en la que se puede encontrar información muy útil sobre el modelado de *software,* la ingeniería de *software* dirigida por modelos (MDE) o el lenguaje de modelado unificado (UML).

© Universidad Internacional de La Rioja (UNIR)

Desarrollo de software dirigido por modelos: Conceptos, métodos y herramientas

Molina, J., Rubio, F. O., Pelechano, V., Vallecillo, A., Vara, J. M. y Vicente-Chicote, C. (2013). *Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas.* Ra-Ma.

Este libro ofrece una explicación de los conceptos básicos relacionados con la ingeniería dirigida por modelos. Además de ejemplos y presentación de las herramientas más utilizadas, este libro combina rigor científico y experiencia práctica, ofreciendo una panorámica actual y completa sobre el desarrollo *software* dirigido por modelos.

© Universidad Internacional de La Rioja (UNIR)

Model-Driven Software Engineering in Practice

Mambrilla, M., Cabot, J. y Wimmer, M. (2017). *Model-Driven Software Engineering in Practice* (2ª ed.). Morgan & Claypool Publishers.

Este libro analiza cómo los enfoques basados en modelos pueden mejorar la práctica diaria de los profesionales del desarrollo de *software,* aumentando la eficiencia y la eficacia en el desarrollo. El objetivo de este libro es ayudar a comprender rápidamente los principios y técnicas básicos de MDE.

© Universidad Internacional de La Rioja (UNIR)

MOdeling LAnguajes

Página de MOdeling LAnguages.

En esta web encontrarás artículos y noticias actualizadas sobre el modelado de *software*. Entre otras cosas, se pueden ver las tendencias futuras de modelado, técnicas y herramientas más utilizadas en MDE, incluidas herramientas OCL, libros o librerías JavaScript para crear diagramas UML.

© Universidad Internacional de La Rioja (UNIR)

Test

1. ¿Qué diferencia hay entre un enfoque basado en modelos y un enfoque dirigido por modelos a la hora de desarrollar *software*?

A. No hay diferencia, son términos sinónimos.

B. El enfoque dirigido por modelos utiliza los modelos básicamente como documentación o para entender mejor un determinado problema, mientras que, en el enfoque basado en modelos, los modelos constituyen los artefactos del *software.*

C. El enfoque basado en modelos utiliza los modelos básicamente como documentación o para entender mejor un determinado problema, mientras que, en el enfoque dirigido por modelos, los modelos constituyen los artefactos del *software.*

D. El enfoque basado en modelos solo utiliza modelos en la extracción de requisitos, mientras que el enfoque dirigido por modelos utiliza los modelos en todas las fases del proceso de desarrollo de *software.*

2. ¿Cuáles son los fundamentos de MDE?

A. MDE se fundamenta en tres conceptos y en dos relaciones.

B. MDE se fundamenta en que los modelos sirven como documentación del sistema.

C. MDE se fundamenta en los siguientes elementos: sistema, modelo, representación y conformidad.

D. MDE se fundamenta en los principios básicos de la OO.

© Universidad Internacional de La Rioja (UNIR)

3. ¿Qué es un modelo?

A. Una representación gráfica del sistema.

B. Una ilustración simplificada del sistema.

C. Una estructura basada en un grafo (dirigido y etiquetado) que representa todo un sistema.

D. Una estructura basada en un grafo (dirigido y etiquetado) que representa un conjunto específico de incumbencias.

4. ¿Qué es un metamodelo?

A. Un modelo que cubre todas las vistas del sistema.

B. Un modelo que proporciona un esquema de «tipos» para los elementos del modelo.

C. Un lenguaje de modelado.

D. Un lenguaje de dominio específico.

5. ¿Cuáles son las tres dimensiones de modelado definidas en MDA?

A. PSG, PIM y PSM.

B. CIM, PSG y PSM.

C. CIM, PIM y PSM.

D. CIM, PMM y PSM.

6. De acuerdo al enfoque de ingeniería dirigida por modelos:

A. Un modelo es conforme a un metamodelo.

B. Un modelo es instancia de un metamodelo.

C. Un modelo es conforme a varios metamodelos.

D. Un modelo y el sistema son equivalentes.

© Universidad Internacional de La Rioja (UNIR)

7. La separación en vistas del sistema:

A. Aumenta la complejidad del sistema y favorece la comprensión del sistema.

B. Reduce la complejidad del sistema, pero no favorece la reutilización.

C. No permite cambios en el *software,* ni facilita la evolución del *software.*

D. Favorece la reutilización y la trazabilidad entre los distintos artefactos del sistema.

8. ¿Qué es OCL?

A. Un lenguaje informal que permite añadir información adicional a los modelos UML.

B. Un lenguaje gráfico que permite añadir restricciones a los modelos UML.

C. Un lenguaje textual que permite añadir restricciones a los modelos UML.

D. Un lenguaje formal y textual que solo permite especificar restricciones sobre el sistema modelado.

9. ¿Qué se entiende por M2M?

A. Un programa que toma uno o más modelos como entrada y produce uno o más modelos como salida.

B. Transformar modelos a un mismo nivel de abstracción.

C. Una técnica que se da en el enfoque MDA, pero no en el enfoque MDE.

D. Una técnica que consiste en código fuente a partir de un modelo de entrada.

© Universidad Internacional de La Rioja (UNIR)

10. ¿Qué se entiende por M2T?

A. Generar la documentación del sistema a partir de los modelos generados para implementar el sistema.

B. Establecer un enlace de comunicación entre herramientas de análisis, pero no con plataformas de ejecución.

C. Generar un código fuente a partir de los modelos generados para implementar el sistema.

D. Definir casos de prueba a partir de los modelos generados para implementar el sistema.

© Universidad Internacional de La Rioja (UNIR)