Gonzalo Génova Fuster[1], María C. Valiente Blázquez[1], Jaime Nubiola Aguilar[2]

[1] Depto. de Informática, Universidad Carlos III de Madrid; [2] Depto. de Filosofía, Universidad de Navarra

<{ggenova, mcvalien}@inf.uc3m.es>, <jnubiola@unav.es>

Modelos en UML: un enfoque semiótico

Este artículo fue seleccionado para su publicación en **Novática** entre las ponencias presentadas a Phise'05 (*1st International Workshop on Philosophical Foundations of Information Systems Engineering*), evento celebrado en Oporto (Portugal) y del que ATI fue entidad colaboradora.

Resumen: *en este artículo vamos a intentar clarificar, con la ayuda de algunas nociones de semiótica, las confusiones que encontramos en torno a los términos "modelo de análisis" y "modelo de diseño", ampliamente usados en Ingeniería del Software. En nuestra experiencia, estas confusiones son la raíz de algunas dificultades que los profesionales encuentran al modelar, y que en ocasiones conducen a malas prácticas de ingeniería. Cuando los ingenieros de software dicen "análisis", pueden referirse principalmente a dos tipos distintos de actividades de modelado, o incluso pueden mezclarlas descuidadamente: construir modelos de software o construir modelos del "mundo real". La primera de ellas es en realidad el primer paso en el diseño, en el que se especifica la vista externa del sistema. El mayor peligro de confundir estos dos tipos de modelos sería construir un modelo del mundo real y usarlo luego como especificación del sistema software, produciendo un sistema que imita sin necesidad la estructura del mundo real.*

Palabras clave: *desarrollo de software, modelos de análisis, modelos de diseño, semiótica, UML.*

1. Introducción

Desde el comienzo de la era de los sistemas informáticos se identificó la gestión del conocimiento como uno de los problemas claves en el proceso de producción de software [2]. Los problemas de comunicación entre diversas comunidades (usuarios, propietarios, analistas, arquitectos, diseñadores, programadores...), e incluso dentro de las comunidades mismas, conducían con frecuencia al fracaso de los proyectos. Pronto se reconoció la necesidad de disponer de una buena documentación que contuviera los modelos apropiados, pero no todo el mundo actuó en consecuencia. Los detractores de la documentación creían que documentar es una tarea aburrida que hay que hacer cuando el proyecto está prácticamente terminado y que debe ser realizada por trabajadores de bajo nivel en la organización. No comprendían que documentar y modelar es tal vez una de las tareas más difíciles dentro del proceso de desarrollo de software, que requiere habilidades de muy alto nivel, y que debe acompañar a todo el proceso desde su comienzo. En consecuencia, se generaban documentos y modelos inútiles, mal escritos y aburridos, lo que autoalimentaba esta concepción errónea.

Afortunadamente, las cosas han ido cambiando con el paso de los años, y hoy día nadie discute el papel crucial de los modelos en el proceso de desarrollo de software. La iniciativa MDA (*Model Driven Architecture* – Arquitectura Dirigida por Modelos) [20], que es resultado de las fuerzas liberadas por UML (*Unified Modeling Language* – Lenguaje Unificado de Modelado) [21], parece ser el último episodio de esta historia. El mensaje central de MDA es que debemos mover el núcleo del desarrollo de software desde el código hacia los modelos, hasta el punto de construir modelos que puedan ser directamente compilados y ejecutados [15][18]. Supuestamente, los modelos están más cerca del entendimiento humano que el código, de modo que trabajar con modelos será menos propenso al error que trabajar con lenguajes de programación.

Existen diferentes clases de modelos: modelos de análisis y diseño, modelos de estructura y de comportamiento, etc., de modo que entender el significado de cada tipo de modelo, cómo se relacionan entre sí, y cómo evolucionan, es de la mayor importancia. Los documentos del OMG (*Object Management Group*) que definen UML [21], y gran parte de la literatura científica, tratan con mucho detalle la *notación* de UML (la sintaxis), pero apenas proporcionan otra cosa que comentarios informales, si acaso, acerca del *significado* del lenguaje (la semántica) [9].

Nuestro objetivo en este trabajo consiste específicamente en investigar los diferentes *modos de significar* de los modelos de análisis y diseño. En una primera aproximación, "análisis" designa la comprensión de un *problema* o situación, mientras que "diseño" se relaciona con la creación de una *solución* para el problema analizado; un "modelo" es una simplificación utilizada para comprender mejor el problema ("modelo de análisis") o la solución ("modelo de diseño") [24][1]. Pero tomemos estos términos un poco más en serio.

Análisis es una palabra griega que significa lo mismo que *descomposición*, de raíz latina: "*romper un todo en sus partes componentes (con el fin de comprenderlo mejor)*". Se opone a *síntesis* (en griego), o *composición* (en latín): "*construir un todo a partir de sus partes componentes*". La palabra *diseño*, en cambio, está relacionada con la tarea de dibujar los planos de algo antes de construirlo. El diseño anticipa y guía el proceso de producción, la "síntesis". Así pues, análisis y diseño no son propiamente términos opuestos o complementarios: el complemento del análisis es la síntesis, no el diseño. En todo caso, como el diseño es el primer paso en la síntesis, el análisis y el diseño a menudo se tratan como si fueran complementarios. En Ingeniería del Software, un *modelo* es una abstracción, es decir, una simplificación de la realidad. Un modelo simplifica la realidad que representa, suprimiendo detalles irrelevantes y reteniendo los aspectos esenciales. Un modelo *representa* cierta porción de la realidad, es decir, es un *signo* o *significa* esa realidad. Como la semiótica es la ciencia de los signos y la significación (del griego *semeion*, signo), consideraremos algunas nociones de semiótica en la siguiente sección, con el fin de entender mejor *qué es un modelo*.

2. Semiótica, diagramas y modelos

La semiótica es una disciplina moderna, cofundada por Charles S. Peirce (1839-1914), en Norteamérica, y Ferdinand de Saussure (1857-1913), en Europa. De acuerdo con la definición de Peirce, "*un signo es algo que está para alguien en lugar de algo según cierto aspecto o capacidad*" (*"a sign is something which stands to somebody for something in some respect or capacity"* [23]). Nótese la naturaleza triádica de los signos: "*un signo es algo que está* ***para alguien*** *en lugar de algo*". La relación de significación requiere siempre tres elementos para tener lugar: el *signo* (o significante), el objeto (o significado), y un tercer elemento denominado *interpretante* por Peirce. Esta relación triádica (**figura 1**) asocia el signo S con el objeto O al que sustituye ("está en lugar de") y, por otra parte, el vínculo S-O está él mismo asociado al interpretante I ("está para"). En otras palabras, no hay una conexión automática desde el significante hacia el significado: es el interpretante el que

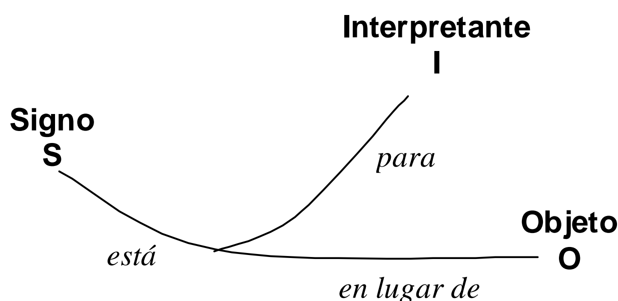

Figura 1. La relación triádica de significación según Peirce.

conecta el signo a su objeto. Esto tiene, al menos, dos consecuencias importantes. Primero, un signo siempre requiere interpretación, no hay una conexión absolutamente fija entre el signo y su objeto. Segundo, el signo es significativo sólo para un ser inteligente que debe interpretarlo.

Según Peirce, un signo puede estar en lugar de su objeto de tres maneras no excluyentes, respectivamente denominadas icono, índice y símbolo [23] (ver **tabla 1**).
§ **icono**: el objeto es significado según cierto tipo de *semejanza*;
§ **índice**: el objeto es significado según cierto tipo de *conexión* física o causal;
§ **símbolo**: el objeto es significado según una pura convención o *ley*.

No hace falta decir que los objetos que pueden ser significados por signos no son exclusivamente entidades físicas. Un signo puede representar cualquier realidad: una cosa, una acción, un concepto... En particular, un signo puede representar a otro signo y, así, de modo transitivo, al objeto de este segundo signo. Por ejemplo, una palabra escrita es un signo de una palabra hablada, que a su vez es un signo de un concepto, que a su vez puede ser él mismo signo de alguna otra cosa. Esto es importante para los diagramas, que a menudo contienen signos de palabras del lenguaje natural.

Más que tres tipos de signo completamente separados, Peirce considera que todo signo participa en mayor o menor medida de la naturaleza de icono, índice y símbolo, aun cuando uno de los tres aspectos pueda estar subrayado en un signo en particular. Por ejemplo, las señales de tráfico, cuyo significado se determina por pura convención o ley (símbolo), se escogen de modo que tengan algún tipo de parecido icónico con la acción que prohíben u ordenan; la imagen de una persona en su tarjeta de identidad se parece (icono) por supuesto a esa persona, pero también está conectada causalmente (índice) con ella a través del proceso físico de la fotografía, y más aún, es una convención social (símbolo) que la imagen de la cara represente a la persona entera.

Así pues, desde este punto de vista semiótico, podemos considerar que un diagrama UML es *un signo hecho de signos*, y podemos observar en él la articulación de estos tres modos de significar [19]. Por ejemplo, en un diagrama de clases (**figura 2**), que está formado por diferentes tipos de signos gráficos, el uso de un rectángulo para representar una clase de objetos es puramente convencional (símbolo); las propiedades (*features*) de una clase se ordenan en dos grupos, para indicar (índice) su diferente naturaleza de atributos u operaciones; y los nombres de las clases, atributos y operaciones se escogen de modo que se parezcan (icono) a los nombres naturales de esos conceptos.

Un modelo UML, sin embargo, no es meramente un diagrama, o una colección de diagramas. Un diagrama UML es la representación gráfica de un conjunto lógico de elementos interconectados que pertenecen a un modelo [1][21], es decir, *un diagrama es una vista parcial de un modelo*. En otras palabras, un modelo UML puede representarse gráficamente mediante diagramas UML, siguiendo las reglas de la notación UML, pero también puede representarse gráficamente en una estructura en árbol (como habitualmente ocurre en el típico panel izquierdo en las herramientas CASE), o incluso en forma puramente textual (informes generados por herramientas, serialización XMI, etc.): es decir, los modelos UML son independientes de la notación UML.

Así pues, un modelo UML es *un conjunto lógico de elementos* que representan algo, que están definidos de acuerdo a la sintaxis abstracta del lenguaje (el metamodelo), y que pueden representarse de acuerdo a su sintaxis concreta (la notación). El modelo como un todo significa una cierta realidad, y cada elemento significa una parte pequeña de esa realidad.

En resumen, un diagrama UML es un signo de un modelo UML; la realidad significada por el diagrama es la parte del modelo que representa gráficamente. La siguiente y obvia pregunta es, ¿de qué es signo un modelo UML, cuál es la realidad significada por él, cuál es su significado? Trataremos de dar una respuesta a esta pregunta en las secciones siguientes.

3. El significado de un modelo UML: modelos de software

Existe un tipo de modelos para los que la respuesta a esta pregunta es relativamente sencilla. Se denominan "modelos de software" porque el objeto significado por el modelo es un artefacto software: un fragmento de código, un componente ejecutable, todo un sistema o subsistema, etc. Supongamos un ejemplo más bien trivial, en el que un fragmento de código escrito en el lenguaje de programación Java es representado usando la notación UML con diferentes niveles de detalle (**figura 3**). Como podemos observar, la notación UML para una clase es básicamente una representación simplificada del código de la clase, con la ventaja de que podemos omitir incómodos detalles de programación. Esto responde perfectamente a la definición de modelo como "vista simplificada" o "representación abstracta" de cierta porción de la realidad, siendo la "realidad", en este caso, el código mismo.

Un artefacto software puede modelarse en *diferentes niveles de abstracción*: cuanto más abstracto sea el modelo, menos detalles representa, y más alejado está del artefacto significado. Por ejemplo, en la **figura 3** se representa una clase con tres niveles de detalle diferentes: el rectángulo inferior es el que muestra más detalles, el rectángulo de en medio omite los tipos de datos, los parámetros de las operaciones y las visibilidades, y el rectángulo superior omite incluso los atributos y operaciones. Las tres representaciones son signos (o abstracciones)

Signo	Modo	Ejemplo de signo	Ejemplo de objeto
icono	semejanza	(figura humana)	persona
índice	conexión	humo	fuego
símbolo	ley	+	operación de sumar

Tabla 1. Los tres tipos de signo según Peirce.

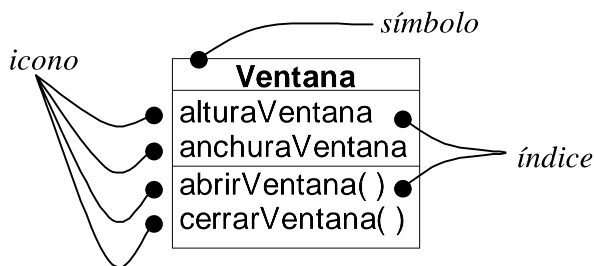

Figura 2. Icono, índice y símbolo en un diagrama de clases UML.

válidos del fragmento de código de la derecha. Es más, las representaciones de nivel superior pueden usarse como signos de las representaciones de nivel inferior; podemos considerarlas como tres representaciones diversas de un único modelo en tres niveles diferentes de detalle, o incluso como tres modelos diferentes que son unos abstracción de los otros. En realidad, el propio código en Java es un signo o abstracción de un artefacto software de nivel inferior, el código ensamblador, que a su vez es signo o abstracción de los bits en la memoria del ordenador, que a su vez son signos de niveles de voltaje y transiciones en los circuitos... La historia de los ordenadores y de la ingeniería del software puede resumirse como el esfuerzo continuo por elevar el nivel de abstracción del procesamiento de información, para acercarlo al entendimiento humano.

Un lenguaje de modelado tal como UML puede ser más o menos adecuado para representar los artefactos software creados con determinados lenguajes de programación, como por ejemplo Java, C++, Visual Basic, Pascal, C, Ada, etc. De hecho, los lenguajes de modelado tienen su origen en los lenguajes de programación, de modo que, dependiendo del paradigma de software de que se trate, a veces el ajuste es bueno, y a veces es malo (incluso muy malo).

Por ejemplo, UML es apropiado para representar Java o C++ (en general, lenguajes de programación orientados a objetos); pero, en cambio, a duras penas se puede producir con UML una buena abstracción de un programa escrito en C o en Ada, que son buenos arquetipos de lenguajes de programación estructurados, en los que no existen clases, objetos ni mensajes. En otras palabras, un lenguaje de modelado es bueno para representar un lenguaje de programación si comparten un conjunto común de conceptos básicos, lo que facilita la traducción entre modelo e implementación, y a la inversa; en caso contrario la traducción resultará antinatural.

Además de los clásicos lenguajes de programación orientados a objetos, los orígenes de UML están ligados a otras fuentes, en particular a las técnicas de modelado de datos del campo de las bases de datos. Esto explica por qué algunas construcciones de UML son tan fáciles de implementar en los lenguajes de programación orientados a objetos (clase, atributo, operación, mensaje...), mientras que otras son mucho más difíciles y requieren estructuras muy complejas para implementar correctamente la semántica definida en UML [7].

Este es el caso, por ejemplo, de las asociaciones bidireccionales y otros tipos de asociación (exceptuando la asociación unidireccional simple, que es equivalente a un atributo). Resumiendo, UML puede ser muy inadecuado para representar un lenguaje de implementación que provenga de un paradigma distinto a la orientación a objetos y, a su vez, un lenguaje orientado a objetos clásico puede verse muy limitado para implementar aquellas construcciones de UML que no tienen su origen en la orientación a objetos.

Los modelos de software pueden usarse principalmente de dos maneras distintas, tradicionalmente conocidas como ingeniería directa e inversa. En la *ingeniería directa*, el modelo es utilizado como anticipación del sistema software que se desea construir; el modelo puede usarse como plantilla para guiar la construcción del sistema, como plataforma para simular el comportamiento del sistema antes de construirlo de verdad, etc. En la *ingeniería inversa*, el modelo es utilizado como herramienta conceptual para entender un sistema existente que hay que mantener o mejorar. Como podemos observar, estos dos usos de los "modelos a escala" en la ingeniería del software son muy similares a los que ocurren en otras ramas de la ingeniería, tales como arquitectura, electrónica, mecánica, etc. Además, estos dos usos están relacionados con la útil distinción entre modelo-como-original y modelo-como-copia [17]: en la ingeniería directa el modelo a escala es usado como *original* a partir del cual se construye un artefacto software; en la ingeniería inversa el modelo a escala es una *copia* simplificada del artefacto software que representa, usada para entenderlo mejor. Finalmente, podemos poner todas estar nociones en relación con otro par bien conocido que hemos mencionado antes: la ingeniería directa es un proceso de *síntesis* en el que un sistema es *construido* a partir de un modelo, mientras que la ingeniería inversa es un proceso de *análisis* en el que un sistema es *entendido* por medio de un modelo (ver **tabla 2**).

No obstante, y tal vez de modo paradójico, este sentido perfectamente legítimo del término "análisis", muy cercano al significado original de la palabra, no es el más habitual entre los ingenieros de software, como vamos a mostrar en la siguiente sección.

4. Modelos de análisis como modelos de software

Cuando los ingenieros de software dicen "análisis", pueden referirse principalmente a dos tipos distintos de actividades de modelado, o incluso pueden mezclarlas descuidadamente: construcción de modelos de software (ingeniería directa) o construcción de modelos del "mundo real" (ingeniería inversa). Exploraremos estos dos sentidos de la palabra en esta sección y la siguiente.

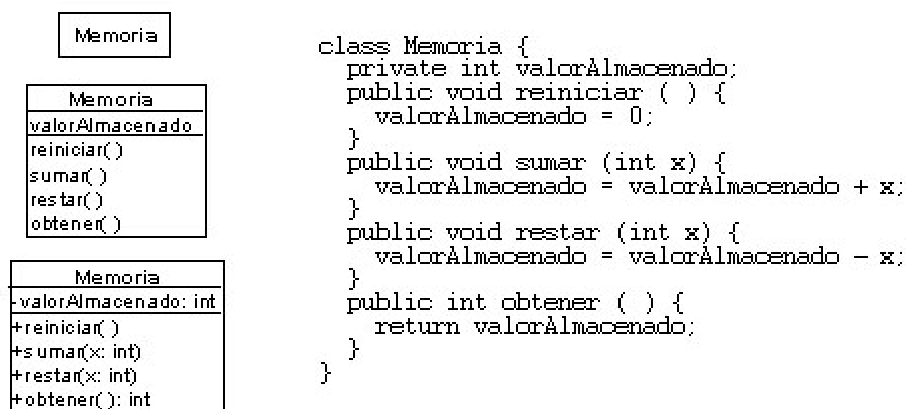

Figura 3. Representaciones abstractas de código en Java, con distintos niveles de detalle.

Modelo → Sistema	Sistema → Modelo
ingeniería directa	ingeniería inversa
modelo-como-original	modelo-como-copia
síntesis	análisis

Tabla 2. Dos usos diferentes de los modelos de software en la ingeniería del software: la ingeniería directa *usa* un modelo-como-original en el proceso de síntesis de un sistema nuevo; la ingeniería inversa *produce* un modelo-como-copia en el proceso de análisis de un sistema existente.

Es muy habitual caracterizar el análisis como la tarea de especificar el *qué*, mientras que el diseño es especificar el *cómo*: qué se supone que el sistema debe hacer para sus usuarios, cómo va a hacerlo (en realidad, esto no expresa otra cosa que un principio clásico de la ingeniería del software: separar la especificación de la implementación).

El *modelo de análisis* debe pues *capturar los requisitos de usuario* sin adoptar prematuramente decisiones de implementación, es decir, omitiendo detalles dependientes de la tecnología [14], y utilizando conceptos extraídos únicamente del dominio del problema. Por el contrario, el *modelo de diseño* debe *definir una solución software* que satisfaga de modo efectivo y eficiente los requisitos especificados en el análisis, y al hacer esto el modelo incorporará nuevos artefactos (nuevas clases, nuevos atributos y operaciones para las clases, etc.) y tendrá en cuenta la plataforma tecnológica concreta sobre la que debe construirse el sistema software. Como puede fácilmente observarse, esta distinción entre análisis y diseño es paralela a la que existe en MDA entre PIM (*Platform Independent Model* – Modelo Independiente de la Plataforma) y PSM (*Platform Specific Model* – Modelo Específico de la Plataforma) [15][18].

El punto clave es que ambos tipos de modelos, análisis y diseño, o PIM y PSM, representan el mismo sistema, *ambos son modelos de software* utilizados en el contexto de un proceso de ingeniería directa, aun cuando tengan una diferencia importante en cuanto a propósito y perspectiva [12], y por tanto en *la forma en que significan* el sistema que se desea construir. Recordemos que "un signo es algo que está para alguien en lugar de algo según cierto aspecto o capacidad" [23]. Este *aspecto o capacidad* es diferente en cada caso: el modelo de análisis representa la *vista externa* del sistema (la especificación), mientras que el modelo de diseño representa la *vista interna* (la implementación).

En cada uno se ejerce un tipo diferente de abstracción, que produce un tipo diferente de modelo del mismo sistema, de modo que éste es significado también de forma distinta: el diseño omite más o menos detalles de implementación, mientras que el análisis omite la implementación misma; el análisis especifica lo que el diseño realiza. El efecto visible puede ser un *diferente nivel de complejidad* en los modelos, pero el punto clave no es ése, sino el *diferente propósito* que tienen. (Por tanto, no estamos pretendiendo que el análisis sea meramente un primer borrador del diseño: esto sería errar el tiro. Nótese, también, que la expresión que se usa habitualmente para distinguirlos, "diferente *nivel* de abstracción", aun cuando no sea esencialmente incorrecta, puede resultar engañosa.)

Esta noción de análisis, que *en realidad equivale a un primer paso en la síntesis*, es fácilmente reconocible en multitud de prácticas de ingeniería y libros de texto, aun cuando raramente sea reconocida como tal de modo explícito (hay excepciones, por supuesto [13]).

El *modelo de casos de uso* es un buen ejemplo de esto. Debido a su carácter de "alto nivel", más cercano al usuario que a la implementación, los casos de uso habitualmente se definen dentro de la fase de "análisis" de un proyecto. Sin embargo, un caso de uso es la especificación de la funcionalidad esperada que un sistema software debe proporcionar, descrita mediante interacciones típicas usuario-sistema [8]: debería ser obvio para cualquiera que un caso de uso está modelando el sistema software que se desea construir, y no el "mundo real" tal como es antes de que exista el sistema, o tal como será después [10].

Así pues, cualquier actividad de modelado que se derive del modelo de casos de uso estará referida a este mismo sistema que se desea construir, y por tanto también debe considerarse que el modelo que produzca será un modelo de software: por ejemplo, el modelo estructural de los principales conceptos presentes en el modelo de casos de uso (a menudo conocido como *modelo conceptual*), tales como datos intercambiados entre el usuario y el sistema, datos almacenados dentro del sistema, etc.

Lo mismo se aplica a los *requisitos de usuario* en general (expresados o no mediante casos de uso): son especificaciones del sistema informático deseado, no describen el mundo fuera del ordenador. En efecto, los requisitos de usuario, en los que habitualmente predomina la forma textual, pueden considerarse con toda propiedad como un signo, es decir, un modelo, del sistema requerido (un *modelo textual*, ciertamente, ¿quién dijo que un modelo debe ser gráfico?). Dado que muchos conciben el análisis como una actividad en la que se averiguan y escriben claramente los requisitos de usuario [5][14], está claro que estos mismos están considerando el análisis como parte de un proceso de ingeniería directa, es decir, de síntesis.

Vistas así las cosas, la transición desde el modelo de análisis al modelo de diseño puede ser difícil o no, pero no ofrece dificultad conceptual desde el punto de vista semiótico: ambos son modelos del mismo sistema software, si bien desde una perspectiva diferente y con un propósito diferente. Nuestro énfasis aquí no es que la transición sea fácil o difícil. De hecho, el diseño debe proporcionar una solución creativa para el problema especificado en el análisis, y esto raramente será fácil [12][14].

Más aún, un buen modelo de diseño que tenga en cuenta requisitos no funcionales tales como rendimiento, reutilización, mantenibilidad, etc., puede producir un sistema que apenas se parezca al sistema especificado en el modelo de análisis [11][12][22].

Pero esto no niega que ambos modelos sean signos del mismo sistema software, que es el punto que queremos subrayar: tanto el modelo de análisis como el modelo de diseño son modelos de software. Los modelos de análisis y diseño no tienen por qué parecerse, pero significan el mismo sistema. No obstante, si el análisis es entendido como en la siguiente sección (de modo radicalmente distinto), surge un nuevo problema conceptual acerca de la transición entre análisis y diseño.

5. Modelos de análisis como modelos de dominio

La otra forma habitual de explicar el uso de un modelo de análisis en un proyecto software es entenderlo como un modelo del dominio, o del contexto, del problema que se debe resolver: un modelo del "mundo real" fuera del sistema informático [4][16][24]. Éste también se denomina a veces *modelo de negocio*, o, en MDA, Modelo Independiente de la Computación (CIM – *Computation Independent Model*).

La diferencia entre análisis y diseño ya no es que se ejercite un tipo distinto de abstracción, sino que *se significa una realidad diferente*: las clases de análisis representan conceptos del mundo real, mientras que las clases de diseño representan fragmentos de código [6, 12, 14]. El nivel de abstracción puede ser alto o bajo en ambos casos.

Esta noción de modelo de análisis (modelo-como-copia del dominio o negocio, obtenido mediante ingeniería inversa), mucho más cercano al significado original de la palabra, plantea sin embargo varias dificultades.

En primer lugar, el hecho de que usemos una *notación uniforme* para representar *cosas de muy diferente tipo* puede llevar a engaño, como sucede a menudo [14]. El error más común ocurre cuando se supone que el modelo de análisis representa el mundo real (modelo de dominio), pero entonces es utilizado como especificación del sistema que se desea construir (modelo de software). Aun cuando algunos métodos recomienden comenzar con un modelo del "mundo real", para luego incluir este modelo dentro de un modelo de diseño que es completado con otros artefactos requeridos para proporcionar la solución [4][24], esto no puede ser una regla general válida.

El sistema software puede incluir un modelo del mundo exterior para *simularlo*, pero la solución que se espera que el sistema proporcione está probablemente mucho más allá de la simulación de la estructura o del comportamiento del mundo exterior, que incluso podría no requerir en absoluto. Los objetos de dentro del sistema, que proporcionan una adecuada solución al problema, pueden ser o no ser copias de los objetos del mundo exterior.

Así pues, como el sistema no es generalmente una copia o simulación del mundo exterior, *un modelo de ingeniería inversa del mundo no puede servir como modelo de ingeniería directa del sistema.*

En otras palabras, el análisis del "mundo real" en el que el sistema software todavía no existe no puede ser usado para modelar lo que el sistema debe hacer en el futuro. Para conseguir esto, lo que necesitamos no es un modelo del mundo real, sino un modelo del sistema requerido, es decir, un modelo de software, tal como se ha explicado en la sección precedente.

De hecho, una especificación de requisitos de usuario para el sistema no debería ser considerada como un modelo del mundo real, sino más bien como un modelo del sistema que se desea construir [10].

Más aún, un modelo del mundo real imaginado *después* de que el nuevo sistema ya exista, por oposición a un modelo del mundo real *antes* de que el nuevo sistema exista, no sirve tampoco como modelo de lo que el sistema debe hacer: es siempre un modelo del mundo, no un modelo del sistema. Tomando una analogía de la ingeniería civil, un modelo de coches y ríos no pude ser un modelo del puente que los coches necesitan para cruzar por encima del río.

No obstante, no puede negarse que *entender el mundo real*, es decir, analizarlo (en el sentido original de la palabra), es utilísimo para obtener una buena especificación de requisitos de usuario, y por tanto un sistema software práctico que resuelva las necesidades de los usuarios.

En efecto, afirmábamos en la sección precedente que el análisis-como-modelo-de-software debería utilizar los conceptos encontrados en el análisis-como-modelo-de-dominio. La diferencia es sutil, pero real: utilizando el mismo vocabulario, estos dos tipos de modelos representan dos cosas distintas. Tal vez sea éste el origen de la confusión que tratamos de evitar.

Elaborar un modelo-como-copia del mundo real para entenderlo mejor es una práctica perfectamente legítima y útil, aun cuando deba distinguirse cuidadosamente de elaborar un modelo-como-original del sistema futuro.

Ahora bien, ¿es UML un lenguaje adecuado para producir modelos del mundo real? Más aún, ¿es la orientación a objetos verdaderamente un buen paradigma para entender el mundo real?

Muchos libros sobre orientación a objetos transmiten una visión totalmente ingenua sobre este tema: el mundo real está hecho de las cosas físicas que podemos ver y tocar, tales como personas, libros, aviones, etc.; el mundo real está hecho de objetos, de modo que la orientación a objetos es la forma más natural de entender el mundo, y de desarrollar sistemas informáticos que resuelvan las necesidades humanas [24]. Tal vez las raíces de la orientación a objetos puedan conectarse con los orígenes de la filosofía y cultura occidentales.

Un ejemplo sería la filosofía aristotélica, en la que "entidad" es la noción central. Pero las similitudes terminan aquí: la *entidad* arquetípica en la filosofía de Aristóteles es el *ser vivo*, que es esencialmente "uno", es decir, posee una fuerte unidad en sí mismo que no puede ser descompuesta en partes sin sacrificarlo. En cambio, la noción de *objeto* en software está inspirada más bien en el *dispositivo mecánico*, que es un ensamblaje de partes fruto de la técnica humana.

Por ejemplo, en el mundo orientado a objetos, las operaciones son ejecutadas *en* los objetos, mientras que en el mundo aristotélico las operaciones son ejecutadas *por* las entidades, lo que hace que la traducción entre los dos mundos sea bastante difícil, y desmiente la pretendida naturalidad de la orientación a objetos. Más aún, existen otras concepciones filosóficas del mundo opuestas a la aristotélica, en las que la noción central no es la de objeto o entidad, sino más bien el proceso (véase por ejemplo la filosofía de Alfred North Whitehead).

No es éste el lugar para intentar dar respuesta a la cuestión sobre cuál sea la verdadera estructura del mundo. Pero tampoco tenemos por qué hacerlo: si sólo queremos construir sistemas de información, todo lo que necesitamos entender es *el mundo del que hablamos*, el mundo de la información. Así pues, en lugar de la expresión *mundo real*, parece mucho más adecuado usar la menos comprometedora *universo del discurso*, ya utilizada en otras ramas de las ciencias de la computación y tecnologías de la información.

La orientación a objetos es una forma particular de concebir y construir sistemas software. ¿Es adecuada para modelar el universo del discurso, el mundo del que hablamos? Bien, probablemente no hay mucho peligro en ello, siempre que seamos conscientes de las *limitaciones del método*, y del lenguaje de modelado en particular.

El peligro aparece cuando se acepta acríticamente que "todo es un objeto"; en efecto, más que expresar una propiedad del "mundo real", esto expresa una propiedad (en este caso, una limitación) del modelador. Como es sabido, para quien sostiene un martillo en la mano, todo son clavos ...

Ya hemos mencionado la falta de adecuación de UML para modelar artefactos software diseñados con lenguajes de programación no orientados a objetos, como C o Ada, porque no comparten un conjunto básico de conceptos. ¿Qué ocurre si los conceptos en el universo de discurso se conciben de modo natural en una forma no orientada a objetos? Bien, la orientación a objetos, en este caso, puede enriquecer nuestra comprensión del universo del discurso, pero puede también empobrecerla. En realidad, cuando tratamos de "objetificar" un mundo que no es orientado a objetos, lo que realmente hacemos es crear un nuevo mundo: *un universo del discurso orientado a objetos.*

Una inadecuada ontología en el lenguaje de modelado [3] puede explicar algunas de las limitaciones que se perciben en UML cuando se usa específicamente para modelar objetos del dominio en lugar de objetos del software. Para la típica mente occidental, puede ser más o menos natural identificar objetos, y clases de objetos, en el universo del discurso; incluso los atributos no son habitualmente difíciles de identificar; sin embargo, el concepto de "mensaje" como invocación de operación, que es absolutamente central en la orientación a objetos, resulta mucho más antinatural en muchos dominios.

Un modelo de comportamiento basado en intercambios de mensajes, con parámetros, valores devueltos, invocaciones polimórficas, etc., puede conducir a modelos realmente retorcidos que complican, más que simplifican, la comprensión del dominio.

6. Conclusiones

Hemos examinado los *dos significados diferentes de los modelos de análisis* que podemos encontrar entre los ingenieros software: un modelo que especifica la vista externa del sistema software, o bien un modelo del "mundo real" que constituye el contexto del sistema software deseado. Es bastante frecuente confundir estas dos concepciones. La práctica real corresponde habitualmente al uso del análisis-como-modelo-de-software, aun cuando las explicaciones teóricas apuntan con frecuencia hacia el análisis-como-modelo-de-dominio.

Un *peligro moderado* de la confusión es pensar que estamos modelando el mundo real, cuando en realidad estamos elaborando una especificación o diseño de alto nivel del sistema software. Un peligro mucho más serio es construir un sistema que imite sin necesidad la estructura del mundo real; esto puede empeorar si usamos un lenguaje de modelado que haga violencia a la forma natural de hablar del dominio, de modo que el modelo obtenido sea mucho más complicado de lo necesario, o no tenga en cuenta detalles esenciales que no pueden expresarse adecuadamente en el lenguaje utilizado.

Si tiene que utilizar la palabra "análisis" en el contexto de la ingeniería del software, escoja el sentido que prefiera, pero sea consciente de su elección y de los posibles malentendidos con su audiencia. Esperemos que los nuevos términos acuñados en la iniciativa MDA, a saber, Modelo Independiente de la Computación, Modelo Independiente de la Plataforma, y Modelo Específico de la Plataforma, ayuden a evitar las confusiones, y a reservar el significado original de la palabra *análisis*.

Referencias

[1] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
[2] F. P. Brooks. *The Mytical Man-Month.* Addison-Wesley, 1975.
[3] J. M. Cañete, F. J. Galán, M. Toro. "Some Problems of Current Modelling Languages that Obstruct to Obtain Models as Instruments". *Jornadas de Ingeniería del Software y Bases de Datos*, 10-12 November 2004, Málaga, Spain, pp. 13-24.
[4] P. Coad, E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, 1991.
[5] European Space Agency. Board for Software Standardisation and Control. *Guide to the Software Requirements Definition Phase*. PSS-05-03, March 1995.
[6] M. Fowler, K. Scott. *UML Distilled.* Addison-Wesley, 2004.
[7] G. Génova, C. Ruiz del Castillo, J. Llorens. "Mapping UML Associations into Java Code", *Journal of Object Technology*, 2(5):135-162, September-October 2003, <http://www.jot.fm/issues/issue_2003_09/article4>.
[8] G. Génova, J. Llorens, J. Nubiola. "Métodos abductivos en ingeniería del software". *2º Workshop en Métodos de Investigación y Fundamentos Filosóficos en Ingeniería del Software y Sistemas de Información-MIFISIS'04*. November 5-6, 2004, Valadolid, Spain.
[9] D. Harel, B. Rumpe. "Meaningful Modeling: What's the Semantics of 'Semantics'?". *IEEE Computer*, October 2004:64-72.
[10] D. Hay. "There Is No Object-Oriented Analysis". *Data to Knowledge Newsletter*, 27(1), January-February 1999.
[11] W. Haythorn. "What Is Object-Oriented Design?". *Journal of Object-Oriented Programming* 7(1):67-78, 1994.
[12] G. M. Høydalsvik, G. Sindre. "On the Purpose of Object-Oriented Analysis". *VIII Conf. on Object-Oriented Prog., Syst., Lang., and Appl. (OOPSLA-93)*, September 26 – October 1 1993, Washington, DC, USA. *ACM SIGPLAN Notices* 28(10):240-255.
[13] I. Jacobson. "A Confused World of OOA and OOD". *Journal of Object-Oriented Programming* 8(5):15-20, 1995.
[14] H. Kaindl. "Difficulties in the Transition from OO Analysis to Design". *IEEE Software*, 16(5):94-102, 1999.
[15] A. Kleppe, J. Warmer, W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
[16] C. Larman. *Applying UML and Patterns*. Prentice-Hall, 1998.
[17] E. Marcos, A. Marcos. "A Philosophical Approach to the Concept of Data Model: Is a Data Model, in Fact, a Model?". *Information Systems Frontiers*, 3(2):267-274, 2001.
[18] S. J. Mellor, K. Scott, A. Uhl, D. Weise. *MDA Distilled. Principles of Model-Driven Architecture*. Addison-Wesley, 2004.
[19] B. Morand. "Modeling: Is it Turning Informal into Formal?". *The First International Workshop on the Unified Modeling Language*, June 3-4, 1998, Mulhouse, France, Springer LNCS 1618, pp. 37-48.
[20] Object Management Group. *MDA Guide Version 1.0.1*. (http://www.omg.org/).
[21] Object Management Group. *Unified Modeling Language Specification*. Version 1.5, March 2003, <http://www.omg.org/>.
[22] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". *Communications of the ACM*, 15(12):1053-1058, December 1972.
[23] C. S. Peirce. "Ground, Object and Interpretant" (*CP* 2.228, 1897), and "A Second Trichothomy of Signs" (*CP* 2.247-249, 1903). In C. Hartshorne, P. Weiss y A. W. Burks (eds.), *The Collected Papers of Charles Sanders Peirce*, vols. 1-8. Harvard University Press, Cambridge (Massachusetts), 1931-1958.
[24] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.