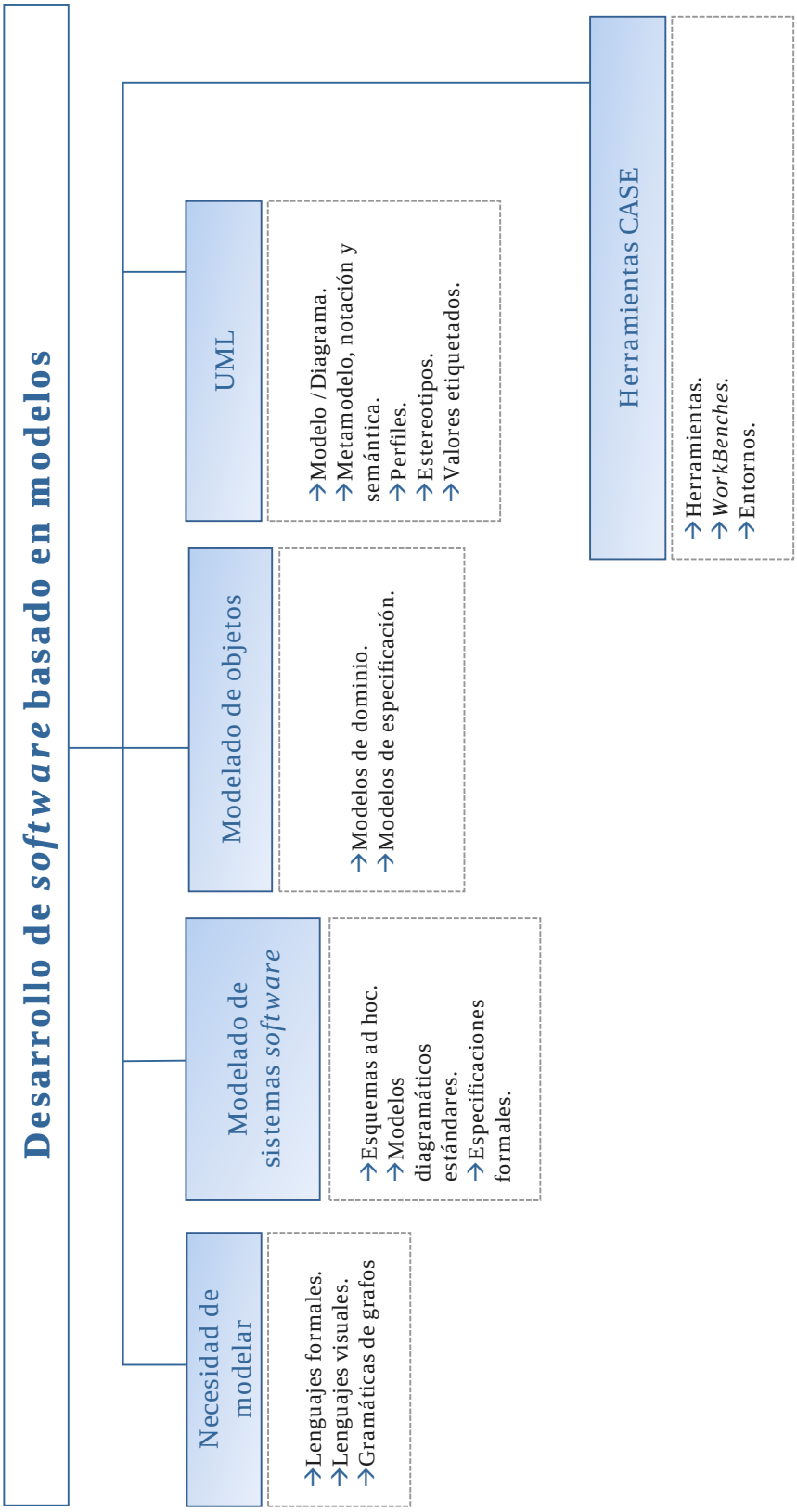
Desarrollo de *software* basado en modelos

TEMA 4

Esquema



Ideas clave

4.1. ¿Cómo estudiar este tema?

Para estudiar este tema lee las **Ideas clave** que encontrarás a continuación.

En este tema se estudian los conceptos básicos sobre modelado de *software* orientado a objetos. Se revisa el lenguaje de modelado por excelencia, UML y se analiza el papel que juegan las herramientas CASE a lo largo del proceso de desarrollo de *software*.

4.2. La necesidad de modelar

En su origen, en los años 60, la **ingeniería de *software*** tuvo que enfrentarse a la **complejidad inherente al *software***. En este periodo, este problema se gestionó proporcionando un significado a los programas: se formularon conceptos básicos para los lenguajes de programación y sistemas operativos (Habermann, 1986). Fortran introdujo el concepto de abstracción a nivel de procedimiento, y posteriormente Algol60 presentó un conjunto de conceptos nuevos: *tipos de datos* (del inglés data *types*), procedimientos recursivos, las perspectivas estática y dinámica, la descripción formal de la sintaxis del lenguaje, etc.

Simula 67 también apareció en este periodo y fue el primer lenguaje que utilizó el concepto de **Orientación a Objetos** (OO) a través de las clases y las subclases. Algol68 y Pascal introdujeron los tipos de datos definidos por el usuario, variables por referencia y estructuras de tipos disjuntos. Además, en este periodo, la ingeniería de *software* estuvo enfocada a la optimización del procesamiento (del inglés *parsing*) y a la generación automática de código, así como a la utilización eficiente de los recursos *hardware* en sistemas operativos de tiempo compartido.

En los años 70, la ingeniería de *software* tuvo que enfrentarse a grandes proyectos, difíciles de controlar en un único programa. Este problema se gestionó con la modularidad y la estructura a través de la programación estructurada. Además, en este periodo apareció el principio de ocultación o encapsulamiento de Parnas (1972), y surgieron mecanismos para el diseño del control de versiones y la gestión de configuración de los sistemas *software*.

En los años 80, la ingeniería de *software* tuvo que enfrentarse a la variabilidad de los requisitos. Este problema se gestionó con el paradigma OO y el modelado. En este momento todo era un objeto y el lenguaje unificado de modelado (del inglés *Unified Modeling Language*, UML) constituía la pieza clave para la transición de la programación orientada a objetos a la ingeniería de *software* basada en modelos. En este periodo, Xerox decidió comercializar Smalltalk. Smalltalk es el segundo lenguaje de programación orientado a objetos más antiguo, por detrás de Simula, en el que de hecho está inspirado.

En los años 90, la ingeniería de *softwar*e tuvo que enfrentarse a sistemas distribuidos y al despliegue masivo. Este problema se gestionó con la utilización de, por ejemplo, componentes y líneas de producto (del inglés *product lines*). Por su parte, la ingeniería de *software* Basada en Componentes (del inglés *Component-Based Software Engineering*, CBSE) surgió a finales de los 90 como una técnica basada en reutilización para el desarrollo de sistemas *software*.

Los componentes, aunque puedan estar compuestos de una colección de objetos, son más abstractos que los objetos en sí y pueden alcanzar mayor nivel de reutilización del que se puede conseguir a través de los objetos. Tal y como explica Sommerville (2005), CBSE comprende el proceso de definir, implementar e integrar o ensamblar componentes independientes en el desarrollo de aplicaciones, con un bajo nivel de acoplamiento.

Con el objetivo de resolver el problema de desarrollo de *software* complejo, costoso, con poca fiabilidad y que no satisface los requisitos especificados, en los últimos años han ido apareciendo diversos lenguajes o formalismos que se podrían clasificar en las siguientes categorías: lenguajes formales, lenguajes visuales y gramáticas de grafos.

Lenguajes formales

Los lenguajes formales han tenido un gran efecto en la práctica del desarrollo de *software*: compiladores, tecnología ingeniería de *software* asistida por ordenador (del inglés *Computer-Aided Software Engineering*, CASE), gestión de configuración, y sistemas de interconexión de módulos. De acuerdo a Salomaa (1973), la teoría de los lenguajes formales considera a los lenguajes como conjuntos de cadenas basados en algún **alfabeto**, y especifica lenguajes potencialmente infinitos con conjuntos de reglas concisas denominadas gramáticas (del inglés *grammars*). Un alfabeto es un conjunto

finito no vacío. Los elementos de un alfabeto se denominan **letras**. Una palabra de un alfabeto en particular es una cadena finita formada por cero o más letras de dicho alfabeto, donde la misma letra puede aparecer varias veces. La cadena que contiene cero letras se denomina **palabra vacía** y se denota como λ.

Representada de manera formal, una gramática es un cuádruplo ordenado G= (V_N, V_T, X_0, F) V_N y V_T son alfabetos disjuntos. Los elementos de V_N se denominan **no-terminales**, y los elementos de V_T se denominan **terminales**; $X_0 \in V_N$ se conoce como **letra inicial**; F es un conjunto finito de pares ordenados (P, Q), donde Q es una palabra del alfabeto$V = V_N \cup V_T$, y P es una palabra de V que contiene al menos una letra de V_N. Los elementos (P, Q) de F se denominan **reglas de reescritura** (del inglés *rewriting rules*) y se escriben P → Q.

En general, las gramáticas se pueden considerar **descripciones estructurales de los sistemas *software***. Tal y como explica Klint, Lämmel & Verhoef (2005), el término gramática se utiliza para designar todos los formalismos de gramáticas y notaciones de gramáticas establecidos. Por tanto, incluye: (i) **gramáticas de contexto libre** (del inglés *context-free grammars*), que permiten la definición de la sintaxis concreta de los lenguajes de programación, por ejemplo, *Extended Backus-Naur Form*; (ii) **diccionarios de clases** (paradigma OO); y (iii) *XML Schemas*, que permiten la definición de formatos de intercambio las aplicaciones; así como otras formas de gramáticas de grafos y árboles (estructuras de grafos).

Los antiguos lenguajes formales aparecieron como lenguajes dedicados a resolver problemas en ciertas áreas específicas: **COBOL** para procesos de negocio, **FORTRAN** para cálculos numéricos, y **Lisp** para el procesamiento de símbolos. Sin embargo, de manera gradual, estos lenguajes han evolucionado hasta convertirse en lenguajes de propósito general (del inglés *General Purpose Languages*, GPL). Por tanto, ha surgido de nuevo la necesidad de soporte por parte de lenguajes más especializados para **resolver problemas en dominios de aplicación concretos**. En este contexto, se encuentran los lenguajes de dominio específico (del inglés *Domain Specific Languages*, DSL).

En DSM Forum se define DSL como «un lenguaje personalizado que está dirigido a un dominio pequeño». Por su parte, **van Deursen, Klint y Visser (2000)** definen un DSL como «un lenguaje de programación o un lenguaje de especificación ejecutable que proporciona, mediante notaciones y abstracciones adecuadas, alto poder expresivo

restringido a un dominio en particular». **Otra definición interesante** se encuentra en **(Cook, 2004)**, donde se definen los DSL como «lenguajes que en vez de enfocarse en un problema tecnológico particular como sería la programación, el intercambio de datos o la configuración, están diseñados para que puedan representar de manera más directa el dominio del problema para el que están dirigidos».

Por tanto, se puede concluir diciendo que los DSL son normalmente pequeños y se utilizarán para describir problemas específicos. Así, lenguajes como **UML, COBOL, FORTRAN y Lisp no serán considerados DSL**, ya que no son suficientemente pequeños y, además, su poder expresivo no está restringido exclusivamente a determinados dominios, sino que **permiten describir problemas de distinta índole**. No obstante, por ejemplo, el lenguaje de modelado UML se puede utilizar para definir otros lenguajes de modelado con una representación gráfica específica para un determinado dominio.

Lenguajes visuales

Los lenguajes visuales no se encuentran definidos de manera muy precisa, pero al menos, según Bardohl et al. (1999), se puede afirmar que **un lenguaje es clasificado como visual** «si permite una representación multi-dimensional de los elementos del lenguaje, es decir, constituye una representación gráfica». Los lenguajes visuales se utilizan para diferentes propósitos, como por ejemplo, modelado del conocimiento, diseño de bases de datos, análisis y diseño de sistemas *software* orientado a objetos, etc.

Los lenguajes visuales surgieron con los primeros prototipos de ordenadores con disponibilidad gráfica: Ivan Sutherland fue el primero en utilizar **un lápiz óptico** que permitía crear gráficos directamente sobre la pantalla (los patrones gráficos se podían almacenar en memoria, y posteriormente se podían recuperar y manipular como si fueran datos). Desde entonces, han ido apareciendo un gran número de lenguajes visuales. Los diagramas de UML y los diagramas de flujos de datos (del inglés *Data Flow Diagrams*, DFD) son ejemplos de lenguajes visuales muy utilizados en el desarrollo de *software*.

De acuerdo a Bardohl et al. (1999) los lenguajes visuales se pueden clasificar en las siguientes categorías: (i) **lenguajes visuales de procesamiento de la información**: aunque estos lenguajes tienen una representación textual, en realidad han sido diseñados para la manipulación de objetos visuales. Por ejemplo, **GRAIN**, tiene

una interfaz de usuario en modo texto, pero soporta el almacenamiento y recuperación de imágenes a partir de un sistema de base de datos relacional; (ii) **lenguajes visuales de interacción:** gestionan objetos que tienen impuesta una representación gráfica. Por ejemplo, lenguajes de programación de interfaz de usuario como Visual Basic; y (iii) **lenguajes visuales de programación**: proporcionan construcciones gráficas orientadas a la programación. Por ejemplo, entre estos lenguajes se encuentran *Pygmalion*, probablemente el primer lenguaje de programación a través de ejemplos basado en iconos, donde cada icono es un objeto con la representación dual de una parte lógica, «el significado», y una parte física, «la imagen»; y UML, como un lenguaje basado especialmente en diagramas y orientado a modelar las distintas vistas de un sistema, y simplificar el complejo proceso de desarrollo de sistemas.

Gramáticas de grafos (del inglés *Graph Grammars*)

Descrito de manera formal, Franck (1976) define un **grafo** como un par G= (K, R) donde K es un conjunto finito de *nodos* y $R \subset K \times K$ un conjunto de *arcos*. El grafo será *dirigido* si R está ordenado (es decir, es un subconjunto de pares ordenado de nodos), y *no dirigido* en caso contrario. Un **grafo etiquetado** es un cuádruplo MG= (K, R, k, r) donde K es un conjunto finito de nodos; $R \subset K \times K$ un conjunto de arcos; k: K → V una correspondencia para etiquetar los nodos; y r: R → M una correspondencia para etiquetar los arcos.

Una gramática de grafos es un cuádruplo GG= (V, T, M, P, S) donde V es un conjunto finito de símbolos; $T \subset V$ un conjunto de símbolos terminales; M un conjunto finito de etiquetas de arcos; S un grafo etiquetado, el grafo de inicio; y P es un conjunto finito de producciones $p = (G_1, G_r, H^f, m^f, H^t, m^t)$ donde G1 y Gr son grafos etiquetados con el conjunto de nodos K1 y Kr, respectivamente, $H^f \subset K_r \times K_1$, $H^t \subset K_1 \times K_r$, $m^f : H^f \to M$ y $m^t : H^t \to M$.

En una palabra, las gramáticas de grafos se utilizan para la generación de grafos. Tal y como se define en (Bardohl et al., 1999), el lenguaje de una gramática de grafos es «el conjunto de todos los grafos que se pueden derivar en una secuencia finita de pasos de reescritura del grafo de inicio». Por tanto, las gramáticas de grafos se utilizan mucho en determinadas aplicaciones, como por ejemplo, en la especificación del *software*, bases de datos y reconocimiento de patrones.

Una gramática de grafos es parecida a una gramática de caracteres, en el sentido de que la gramática consiste en un conjunto de etiquetas de nodos, un conjunto de etiquetas de arcos, un axioma (es decir, un grafo de inicio), y un conjunto finito de producciones. Cada nodo y cada arco tienen su etiqueta en el conjunto correspondiente. Estos elementos se pueden utilizar como tipos o como símbolos terminales o no terminales. Cada producción consiste en un grafo *left-hand-side* (LHS) y un grafo *right-hand-side* RHS, y describe cómo una ocurrencia o aparición del LHS en un grafo se puede sustituir por el RHS; el grafo resultante se deriva de un grafo anterior mediante un paso de reescritura.

Los **editores dirigidos por sintaxis** (del inglés *syntax-directed editors*) constituyen un ejemplo de la aplicación de las gramáticas de grafos. Tal y como se explica en Arefi, Hughes y Workman (1990), dado que los programas son una colección de objetos con un sentido sintáctico y computacional: identificadores, procedimientos, bucles, tipos de datos... los editores dirigidos por sintaxis los gestionan como colecciones jerárquicas de construcciones de lenguajes de programación y permiten que los programadores creen y manipulen sus programas en términos de esas construcciones del lenguaje (es decir, los editores dirigidos por sintaxis tienen conocimiento de las construcciones subyacentes al lenguaje de programación).

Se llega así a que los editores dirigidos por sintaxis, además de facilitar la edición de operaciones que tienen sentido dentro de las construcciones del lenguaje de programación, ayuda a los desarrolladores a construir y manipular sus programas. Por cada construcción de programación, los editores dirigidos por sintaxis proporcionan al desarrollador una **plantilla** (del inglés *template*). Las plantillas en este contexto describen la estructura de la construcción de programación que representan e indican donde se permiten las inserciones por parte del usuario. De esta manera, al generar programas que son sintácticamente correctos se reducen los errores de introducción del código.

4.3. Modelado de sistemas *software*

La utilización de modelos durante el desarrollo de sistemas es una técnica muy consolidada que permite reducir su complejidad y realizar una gestión más óptima del proceso. Resulta obvio que para que el modelo resulte útil, este ha de ser más fácil que manejar que el sistema que representa. Pero, ¿por qué modelar?, ¿qué beneficios aporta realmente?

De acuerdo a Petre (2005), existen diversas técnicas a la hora de modelar en el contexto de la ingeniería de *software*:

- **Esquemas *ad hoc***. Según esta técnica los modelos son construidos *ad hoc*, sin ningún lenguaje común que se pueda compartir fácilmente con otras organizaciones. Sin embargo, los modelos serán comprendidos por todo el equipo de desarrolladores que forman parte del proyecto. En este caso, construir el sistema consiste básicamente en codificar, sin apenas planificar ni estructurar el proyecto.

- **Modelos diagramáticos estándares**. La utilización de modelos diagramáticos se debe fundamentalmente a la aceptación de UML en la ingeniería de *software*. Los modelos UML OMG-UML2_SS (2011) se pueden mostrar en forma de diagramas. Tal y como se define en (Booch, Rumbaugh y Jacobson, 2006), un diagrama UML es «la representación gráfica de un conjunto lógico de elementos interconectados que pertenecen a un modelo, de tal forma que los elementos del sistema pueden aparecer en algunos o en todos los diagramas». Por ejemplo, los diagramas de casos de uso describen la vista externa, funcional, de un sistema.

  Debido a su semántica un tanto ambigua, algunos autores consideran informales los modelos UML. Sin embargo, clasificar los modelos UML como modelos informales puede resultar demasiado radical. En cualquier caso, resulta posible aplicar una semántica formal a UML a través de correspondencias (del inglés *mappings*) entre modelos UML y métodos formales reconocidos para el desarrollo de *software*, como por ejemplo, Object-Z (Smith, 2000) y las redes de Petri (Silva, 1985), (Jensen, 1996). La gran ventaja de este enfoque (es decir, el uso de correspondencias) es que se puede reutilizar todo el trabajo realizado hasta la fecha sobre métodos formales.

- **Especificaciones formales.** Las especificaciones formales son modelos que se utilizan en métodos formales. Un método formal es una técnica de prevención de

errores utilizada para construir sistemas fiables. Un método formal se corresponde con el área de la Informática que tiene que ver con la aplicación de técnicas matemáticas que permiten el diseño e implementación de *hardware*, y sobre todo de sistemas *software*. Un método formal consiste en un lenguaje lógico de especificación (del inglés *logic-based language*), que también se utiliza como técnica de desarrollo. **El lenguaje de especificación se define matemáticamente** (es decir, conforme a reglas matemáticas) y se utiliza para recoger modelos del sistema a diferentes niveles de abstracción. La técnica de desarrollo puede establecer, por ejemplo, qué es lo que se necesita para que la especificación del sistema recoja de manera correcta los requisitos del sistema, o para que una especificación del sistema más detallada desarrolle de manera correcta una especificación más abstracta del mismo sistema. CSP (Hoare, 1978), Z (Spivey, 1992) y, los ya mencionados, Object-Z (Smith, 2000) y las redes de Petri (Silva, 1985) y (Jensen, 1996), son algunos ejemplos de métodos formales.

Sin embargo, el problema de los métodos formales consiste precisamente en su base matemática, lo que los hace más difíciles de tratar. Además, existe una gran multitud de métodos formales estándares, cada uno con su propia notación y técnica de desarrollo. Por tanto, la comunicación entre estos modelos resulta mucho más complicada que la comunicación, por ejemplo, entre modelos UML.

Al examinar estas técnicas se puede concluir que la formalidad de los modelos es una característica a tener en cuenta cuando se utilicen modelos en el desarrollo de *software*, y que la formalidad se medirá entonces en función de su semántica (es decir, irá en función de sus correspondencias con otros tipos de modelos tal y como se ha explicado anteriormente). Así, los modelos pueden tener, por ejemplo, una semántica operacional, si se realiza una correspondencia de las operaciones a su interpretación computacional. De este modo, la semántica operacional se va a poder utilizar como base para la construcción de herramientas de generación de código, así como simulación y verificación de modelos.

4.4. Modelado de objetos

Como ya se ha dicho anteriormente, enfrentarse a la construcción de un sistema *software* completo basándose en un único modelo resulta inviable. La solución está en desarrollar un conjunto de modelos interconectados que ofrecerán distintas vistas del sistema a construir. A tal fin, se puede decir que la tecnología de objetos permite simplificar considerablemente el proceso de construcción de modelos durante el desarrollo de *software*. El modelado de objetos es una forma tradicional de representar conocimiento, donde tanto el dominio del problema como su solución se pueden modelar utilizando objetos y desde diferentes perspectivas: objetos que se relacionan con otros objetos y con su entorno.

Lo que hay que tener claro, tal y como destaca Graham, Henderson-Sellers y Younessi (1997), es que con esto no se quiere decir que el mundo real se compone de los objetos incluidos en nuestro modelo, pero sí que se puede modelar el dominio como si estuviera compuesto por ellos. Un buen ejemplo de esta afirmación se encuentra en (Cook y Daniels, 1994), donde se realiza una clara distinción entre los **modelos de dominio** y los **modelos de especificación**: los modelos del mundo real (modelos de dominio) son distintos a los modelos del sistema *software* (modelos de especificación), aunque como se indica en (Génova, Valiente y Nubiola, 2005), dentro del *software* habrá un modelo que imita ciertas cosas del mundo real.

Cook y Daniels (1994) defienden que el mundo real no está compuesto de objetos que se envían mensajes entre sí. Sin embargo, por su utilidad, merece la pena considerar la posibilidad de modelar el mundo real como si se tuvieran objetos que envían mensajes. Además, este modelo del mundo real será el precursor del modelo del sistema *software*. Los autores ponen como ejemplo el hecho de que el Sol no despierta cada mañana a los pájaros para que canten enviándoles un mensaje. Efectivamente, esta situación no es la que se da en el mundo real. Sin embargo, en el modelado de cierto dominio puede resultar aceptable modelar la salida del Sol como un evento que le llegará a cada pájaro y hará que canten.

De este modo, para modelar el mundo real se van a utilizar dos conceptos: **objetos** (o mejor, los tipos de objetos, es decir, **clases**) y **eventos**; mientras que para modelar el *software* se utilizarán los conceptos: **objetos** (o mejor, los tipos de objetos, es decir, **clases**) **y mensajes**. Los objetos en el modelo del mundo real representan **cosas** y los eventos modelan **hechos** (del inglés *occurrences*) que se dan en el mundo real. Los

objetos del modelo pueden ser concretos o abstractos, temporales o persistentes, reales o imaginarios (por ejemplo, Pájaro, Sol, Posición, etc.). Los hechos representados por los eventos en el modelo serán nombres que se corresponderán con los cambios de estado de las cosas que se están modelando (por ejemplo, que salga el Sol en una posición, que canten los pájaros, etc.).

En cambio, los objetos del modelo del sistema *software* hacen referencia al **encapsulamiento de datos (estructura)** con sus **operaciones asociadas (comportamiento)**, donde los **mensajes** hacen referencia a las invocaciones de esas operaciones.

Por tanto, tal y como se indica en (Jacobson, Ericsson y Jacobson, 1995), un modelo de objetos consistirá en un conjunto de **clases** (es decir, las clases del sistema que definen los objetos que se crean a partir de ellas), y un conjunto de **asociaciones**. La **asociación entre clases** indicará que un objeto de una clase puede utilizar otro objeto de la clase con la que se comunica a través de la asociación, mediante el envío de mensajes (es decir, la invocación de operaciones).

Indudablemente, de lo anterior se puede deducir que habrá casos en los que la OO resultará totalmente inapropiada. Por ejemplo, la solución de una ecuación diferencial difícilmente se podría modelar de manera óptima a través de objetos. Por tanto, la OO tendrá interés cuando los objetos y, especialmente, las relaciones (es decir, **colaboraciones**) entre ellos, tienen mucha más importancia que los algoritmos y las funciones (Cook y Daniels, 1994). No hay que olvidar que, como ya se estudió en el tema anterior, la OO solamente tiene sentido cuando existe una colaboración entre diferentes objetos.

4.5. UML

El **Lenguaje Unificado de Modelado** (del inglés *Unified Modeling Language*, UML) (OMG-UML2_IS, 2011) y (OMG-UML2_SS, 2011) es un lenguaje visual de propósito general para el modelado de sistemas. Aunque UML se encuentra muy vinculado al modelado de sistemas *software* OO, este lenguaje puede abarcar un campo más amplio de aplicación gracias a sus mecanismos de extensibilidad (Arlow y Neustadt, 2005).

UML fue diseñado para incorporar las mejores prácticas en técnicas de modelado e ingeniería del *software*. Es importante no confundir UML con una metodología de modelado como puede ser por ejemplo, el **Proceso Unificado** (del inglés *Unified Process*, UP), que utiliza UML como la notación de su modelado visual. UML se puede utilizar con todas las metodologías o ciclos de vida existentes. UML lo que hace es proporcionar una sintaxis visual (notación) que permite la construcción de modelos.

Los lenguajes de modelado OO comenzaron a aparecer entre a mediados de los 70 y finales de los 80. En este periodo diversos expertos en modelado experimentaron con diferentes técnicas de análisis y diseño OO. A pesar de la variedad de métodos OO disponibles en esta época, los usuarios no acababan de encontrar el más adecuado a sus intereses, lo que desencadenó la conocida **«guerra de los métodos»**.

A mediados de los 90 surgieron nuevas versiones de los métodos existentes que incorporaban las técnicas de los otros métodos. Así, unos pocos métodos empezaron a adquirir una importancia especial y destacar sobre los demás.

El desarrollo de UML comenzó a finales de 1994 cuando Grady Booch y Jim Rumbaugh de Rational Software Corporation unieron sus respectivos trabajos: el método *Booch* (también conocido como *Object Oriented Design*, OOD) y *Object Modeling Technique* (OMT). En 1995, Ivar Jacobson y Objectory Company con el método *Object-Oriented Software Engineering* (OOSE) se unieron a Rational. Como autores principales de los métodos Booch, OMT y OOSE, Grady Booch, Jim Rumbaugh e Ivar Jacobson se animaron a crear un lenguaje unificado de modelado, motivados principalmente por tres razones:

1. Sus métodos evolucionaban de manera independiente hacia los otros dos métodos, lo que hacía que tuviera sentido continuar esa evolución juntos como un único método y no como tres métodos distintos con muchos aspectos en común.

2. Si se alcanzaba una unificación en lo que se refería a semántica y notación, se podría alcanzar cierta estabilidad en el mercado de la OO, gracias a la existencia de un único lenguaje de modelado suficientemente maduro y a la creación de herramientas que se centrarían en los aspectos más útiles del lenguaje.
3. Los tres autores (o los tres amigos, como también se les conoce) esperaban que con la colaboración conjunta se podrían alcanzar importantes mejoras en los métodos individuales, ayudando a determinar las lecciones aprendidas y a contemplar problemas que ninguno de los métodos había sabido manejar correctamente.

El trabajo en equipo de Booch, Rumbaugh y Jacobson dio como resultado las versiones UML 0.9 y UML 0.91 en junio y octubre de 1996, respectivamente. Hasta poco antes todavía lo denominaban **Método Unificado** (del inglés *Unified Method*), pero desistieron de unificar sus métodos y se conformaron con unificar sus notaciones. A lo largo de 1996 los autores de UML solicitaron y recibieron *feedback* de toda la comunidad, que fueron introduciendo en el lenguaje. De este modo, varias organizaciones empezaron a ver la importancia estratégica de UML para su negocio e hicieron sus contribuciones para obtener una versión UML 1.0 mucho más robusta.

En enero de 1997 surge UML 1.1, como resultado de la unión de nuevas compañías y en un intento de clarificar la semántica de UML 1.0. Esta versión final es la que fue aprobada y aceptada por el OMG.

En el año 2000, UML 1.4 introduce una ampliación muy significativa a UML mediante la semántica de acción. La semántica de acción describe el comportamiento de un conjunto de acciones primitivas que se pueden implementar mediante determinados lenguajes de acción. 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, es decir, modelos que contienen toda la información necesaria para generar, transformando a código utilizando un lenguaje de programación, la funcionalidad requerida para el sistema.

Los primeros borradores de UML 2.x aparecen en el año 2003, introduciendo bastante sintaxis visual nueva, y sustituyendo e intentando clarificar sintaxis de las versiones anteriores 1.x. La Figura 1 resume la historia de los lenguajes de modelado hasta la versión actual más estable de UML (2.4.1).

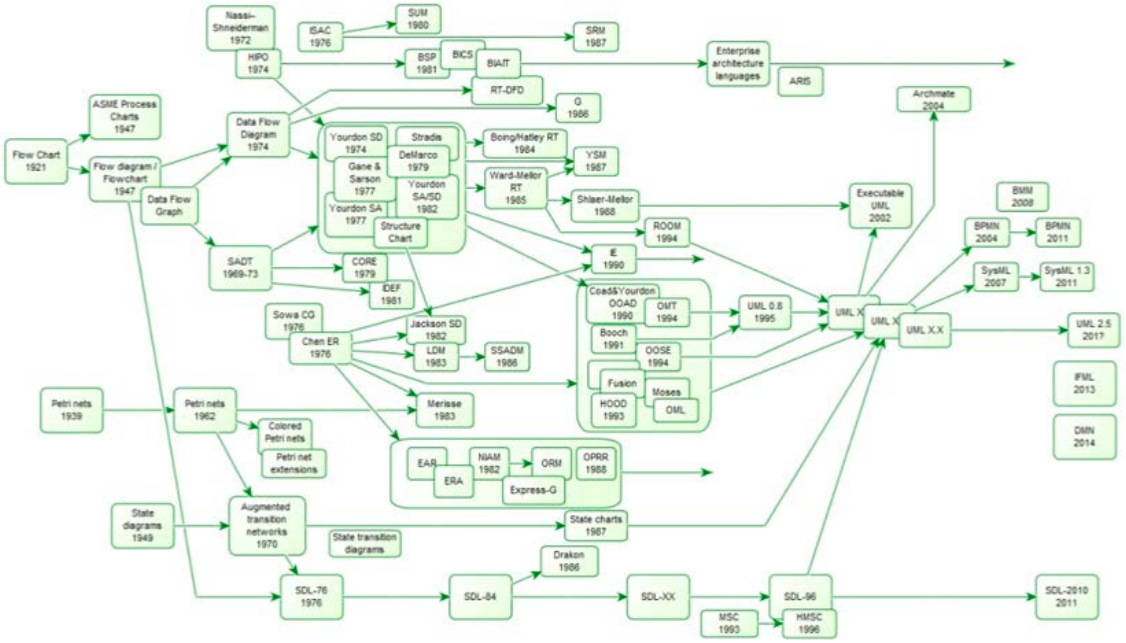


Figura 1. Evolución de los lenguajes de modelado. Fuente: http://modeling-languages.com/wp-content/uploads/2014/04/historyModelingLanguages.jpg

Como ya se ha dicho anteriormente, los modelos UML se pueden representar gráficamente mediante diagramas UML, siguiendo las reglas de la notación UML. De esta manera, resulta posible representar los modelos UML como **modelos diagramáticos** (del inglés *diagrammatic models*) (Petre, 2005). En cualquier caso, es importante tener en cuenta que un modelo UML no es meramente un diagrama, o una colección de diagramas, sino que se podría representar, por ejemplo, de manera textual mediante serialización XMI (Génova, Valiente y Nubiola, 2005).

De acuerdo a Fowler (2003), los diagramas UML pueden tener tres finalidades bien diferenciadas:

- ***Sketch.*** Aquí se clasificarían los diagramas informales e incompletos que se crean para analizar y entender las partes más complejas en el dominio del problema o en el dominio de la solución.
- ***Blueprint.*** Aquí se clasificarían los diagramas de diseño más o menos detallados utilizados para (ver Figura 2): (i) **ingeniería inversa**, que permitirá visualizar y comprender mejor el código ya existente; o (ii) generación de código (**ingeniería directa**).
- **Lenguaje de programación.** Aquí se clasificarían especificaciones completas y ejecutables de un sistema **software** en UML. El código ejecutable se podrá generar automáticamente y normalmente ni será visto ni modificado por los desarrolladores,

que solamente trabajarán con UML como si de un lenguaje de programación se tratase.

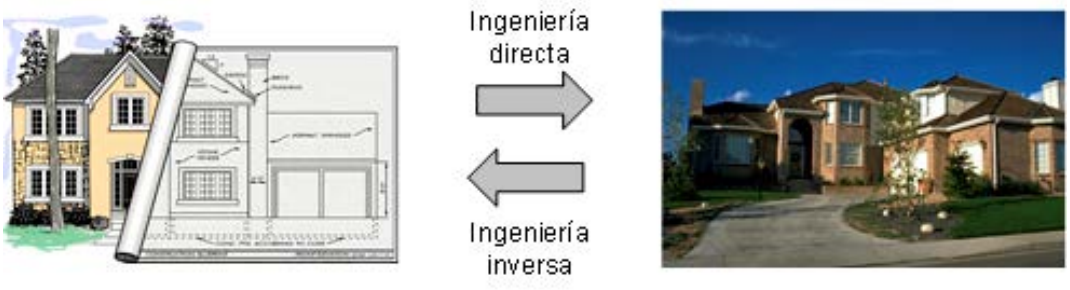


Figura 2. Ingeniería directa vs Ingeniería inversa. Fuente: Curso de Diseño de *Software* Avanzado disponible en el *OpenCourseWare* de la Universidad Carlos III de Madrid.

Por tanto, si se toma UML como lenguaje diagramático, un modelo UML estará formado normalmente por diferentes diagramas, donde cada diagrama representa un aspecto parcial del modelo. Algunos diagramas UML muestran la estructura estática de un sistema *software*. Por ejemplo, un diagrama de clases muestra un conjunto de clases, tipos e interfaces, y sus asociaciones. Un diagrama de clases puede mostrar las operaciones de cada clase pero no describe el comportamiento real de cada operación.

Por el contrario, los diagramas de comportamiento, como por ejemplo, los diagramas de secuencia y los diagramas de actividad, se pueden utilizar para describir la dinámica de un modelo UML. Tal y como indica Shlaer y Mellor (1992), «[...] los diagramas de comportamiento pueden describir qué es lo que ocurre cuando se invoca una operación o pueden describir toda la vida de un objeto».

Así, viendo los modelos UML como modelos diagramáticos, un modelo UML completo siempre va a contener diagramas de estructura y de comportamiento, que describen, respectivamente, los aspectos estáticos y dinámicos de un sistema.

En (Génova, Valiente y Nubiola, 2005), un modelo UML se define como «un conjunto lógico de elementos que representan algo (semántica), 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».

Con lo cual, se puede decir que UML va a estar definido por tres elementos: (i) **sintaxis abstracta (metamodelo)**; (ii) **sintaxis concreta (notación)**; y (iii) **semántica**.

El **metamodelo** de UML (OMG-UML2_SS, 2011) es un modelo UML que define un lenguaje de modelado para explicar la generación de modelos UML válidos. La Figura 3 muestra un ejemplo de la **sintaxis abstracta** para algunos elementos del modelo de actividad. Así, podemos ver, cómo ha de estar definido un modelo de actividad para que sea conforme a UML.

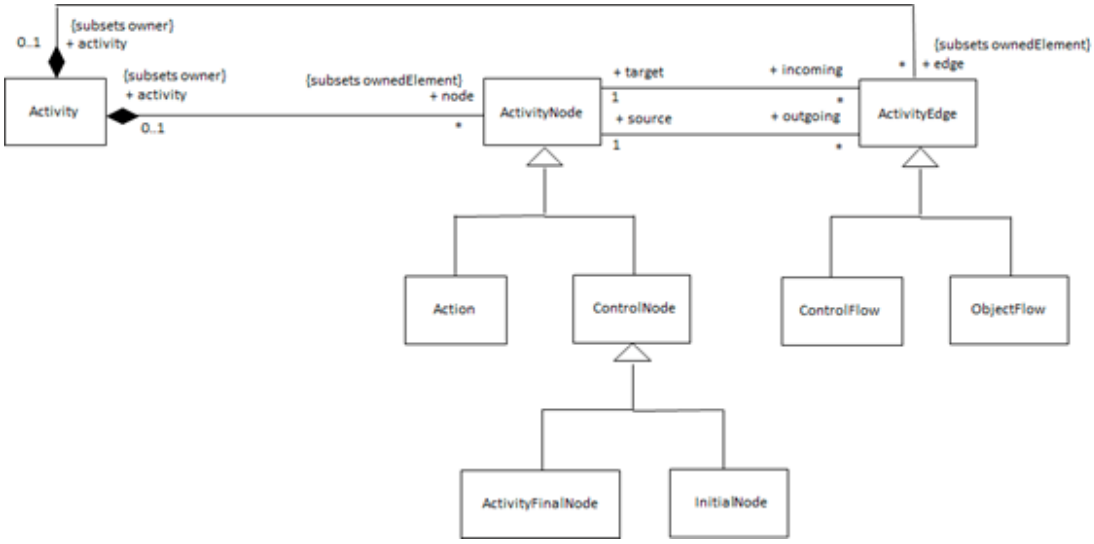


Figura 3. Ejemplo de sintaxis abstracta para el modelo de actividad

Por su parte, la **notación** describe la representación gráfica que se utiliza para representar un **concepto** (es decir, **un elemento del modelo**) en los diagramas (solo los conceptos que puedan aparecer en los diagramas tendrán la notación definida). La Figura 4 muestra un ejemplo de sintaxis concreta para algunos elementos que pueden aparecer en los diagramas de actividad.

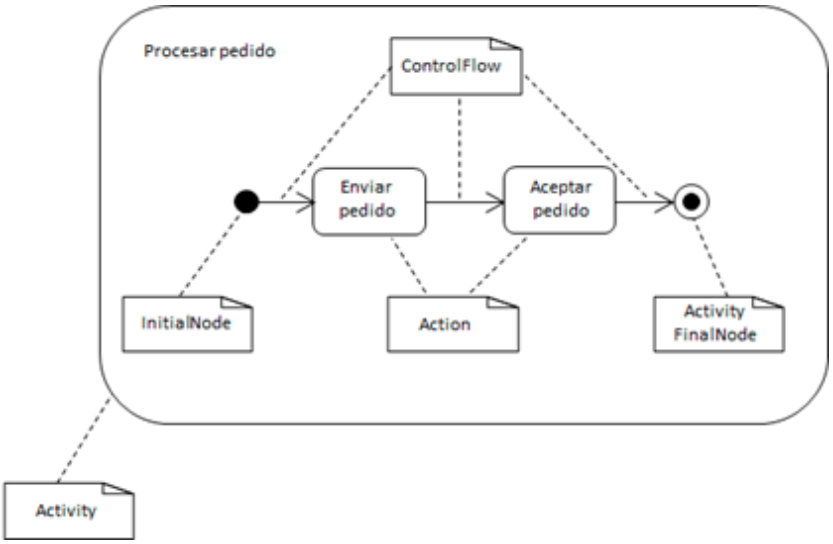


Figura 4. Ejemplo de sintaxis concreta para un diagrama de actividad

Por último, la **semántica** se encuentra descrita en lenguaje natural y describe el significado de un concepto: **lo que representa y su comportamiento**. UML contempla un mecanismo que permite añadir nueva semántica: **las restricciones**. Una restricción indica las condiciones que deben cumplirse para que el modelo esté bien formado. Una restricción se puede escribir en texto en claro, o de manera más formal con el lenguaje *Object Constraint Language* (OCL) (OMG-OCL, 2014). Una restricción se presenta como una cadena de caracteres entre llaves junto al elemento asociado. La Figura 5 muestra un ejemplo de restricción, que indica, en este caso, que los aeropuertos en los que hace escala un determinado vuelo han de estar ordenados.

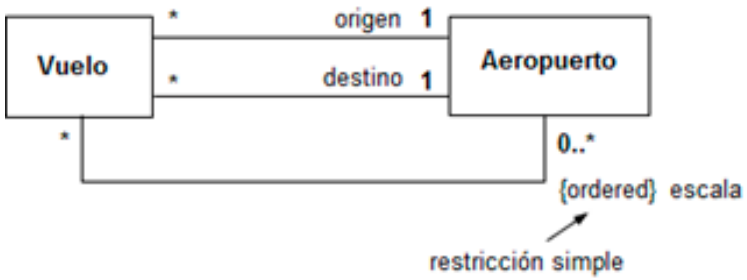


Figura 5. Restricciones en UML

Además, a partir de la versión UML 2.0 se han incluido lo que se ha denominado **puntos de variación semántica** (del inglés *semantic variation points*) que identifican áreas donde se da libertad de interpretación para dominios concretos. Para representar los puntos de variación semántica se pueden utilizar lo que se conoce, por ejemplo, como **perfiles** (del inglés *profiles*).

Un perfil de UML amplía el metamodelo de UML para adaptarlo a dominios o plataformas concretas. Es decir, un perfil proporciona un lenguaje de modelado que se ha de utilizar exclusivamente cuando se utilice un dominio o plataforma determinada. Por ejemplo, UML *Profile for Schedulability, Performance, and Time* (OMG-SPT, 2005), es el perfil de UML que se ha de utilizar cuando se desee modelar, por ejemplo, un sistema de tiempo real con planificación de tareas. En (OMG-UML2_SS, 2011) se define un perfil como «una extensión limitada de un metamodelo de referencia con el objetivo de adaptar el metamodelo a una plataforma específica o dominio específico». Un perfil es una técnica de metamodelado específica que contiene estereotipos (considerados como **metaclases** específicas) y valores etiquetados (considerados como **metaatributos** estándares) adaptados para un dominio o plataforma concreta. Los ***profiles*** en UML 2.x son considerados **paquetes específicos**.

De acuerdo a Booch, Rumbaugh y Jacobson (2006), un estereotipo es «una extensión del vocabulario de UML que permite crear nuevos bloques de construcción que se derivan a partir de los existentes en el metamodelo de UML, pero específicos a un problema concreto que se está modelando». En su forma más sencilla, un estereotipo se representa como un nombre entre comillas tipográficas (<<nombre>>) y se coloca sobre el nombre de otro elemento del modelo. Como señal visual, se puede definir un icono para el estereotipo y mostrar ese icono a la derecha del nombre del elemento del modelo (si se utiliza la notación básica para el elemento) o utilizar ese icono como símbolo básico para el elemento del modelo que se ha estereotipado. La Figura 6 muestra ejemplos de estereotipos.

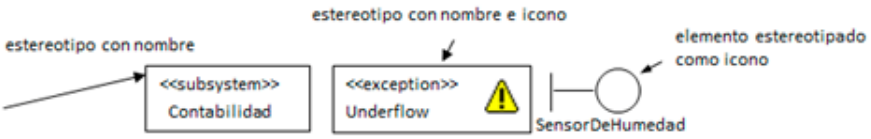


Figura 6. Estereotipos en UML

Por su parte, un valor etiquetado se define en (Booch, Rumbaugh y Jacobson, 2006) como «una extensión de las propiedades de un estereotipo de UML que permite añadir más información en el elemento que lleva ese estereotipo». Gráficamente, un valor etiquetado se representa como una cadena de caracteres. Esta cadena incluye un **nombre** (**etiqueta**), un **separador** (el símbolo =), y un **valor** (de la etiqueta) dentro de una **nota** asociada al objeto. Además, se pueden definir etiquetas para elementos existentes de UML, o bien, se pueden definir etiquetas que se apliquen a estereotipos individuales de forma que cualquier elemento con ese estereotipo tenga ese valor etiquetado. La Figura 7 muestra un ejemplo de uso de un valor etiquetado.

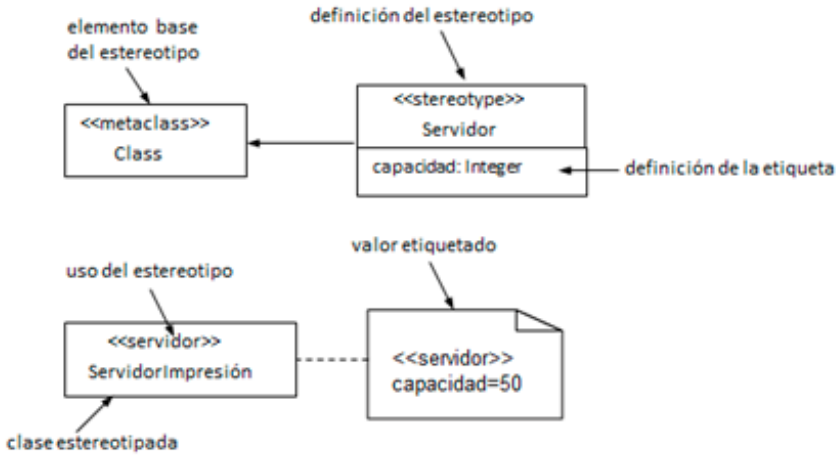


Figura 7. Valores etiquetados en UML

En definitiva, la utilización de UML en la ingeniería de *software* aporta un gran número de beneficios. Debido a su aceptación como estándar en la industria del *software*, se puede utilizar como medio de comunicación común entre los *stakeholders* del proyecto. Por otro lado, su lenguaje diagramático facilita su utilización y promueve la creación de herramientas que soporten modelos UML. Además, gracias a la variedad de diagramas UML adecuados a las distintas fases del proyecto, UML se puede utilizar durante todo el proceso de desarrollo de *software*.

El inconveniente principal de utilizar UML se debe a que el lenguaje no se encuentra definido con total precisión (la semántica no es totalmente formal), lo que puede causar ambigüedad en su interpretación, es decir, cada *stakeholder* podría interpretar, algún aspecto de un modelo UML de manera diferente. Si un programador interpreta el modelo de manera distinta al diseñador, el programador implementará un sistema que no se corresponderá al diseño establecido. Visto así, se perdería cualquier beneficio que se pudiera obtener por la utilización de un lenguaje estandarizado. Ahora bien, lo que se puede hacer en un proyecto real es dotar a UML de una semántica formal que elimine toda posible malinterpretación de los modelos realizando una correspondencia con un modelo formal.

4.6. Herramientas CASE

La tecnología **ingeniería de *software* asistida por ordenador** (del inglés *Computer-Aided Software Engineering*, CASE) proporciona soporte automatizado para gestionar las distintas actividades de los procesos *software*, entre las que estarían, entre otras, ingeniería de requisitos, diseño, desarrollo y pruebas. Así, las herramientas CASE incluyen editores de diseño, diccionarios de datos, compiladores, depuradores (del inglés *debuggers*), herramientas que facilitan la construcción del sistema *software*, etc.

De acuerdo a Sommerville (2007, 79-82) Las herramientas CASE se pueden clasificar en distintas categorías (ver Figura 8):

- **Herramientas.** Las herramientas CASE dan soporte a tareas individuales del proceso *software* (por ejemplo, comprobación de la consistencia de un diseño, compilación de un programa, comparación de los resultados de las pruebas, etc.). Las herramientas pueden ser de propósito general, herramientas *stand-alone* (por

ejemplo, un procesador de palabras) o se pueden agrupar en **bancos de trabajo** (del inglés *workbenches*).

- ***Workbenches.*** Los *workbenches* CASE consisten en un conjunto más o menos integrado de herramientas que dan soporte a la automatización del proceso completo de desarrollo del sistema *software* (por ejemplo, *workbenches* para diseño suelen dar soporte a la parte de programación y pruebas del sistema). El producto final generado por este tipo de herramientas es un sistema en código ejecutable junto con su documentación. Agrupar un conjunto de herramientas CASE en un *workbench* ofrece la ventaja de tener un soporte más uniforme que el que puede ofrecer una herramienta de manera individual.
- **Entornos.** Los entornos CASE también dan soporte a todo, o al menos a una parte sustancial, del proceso *software*. Los entornos normalmente comprenden varios *workbenches* integrados.

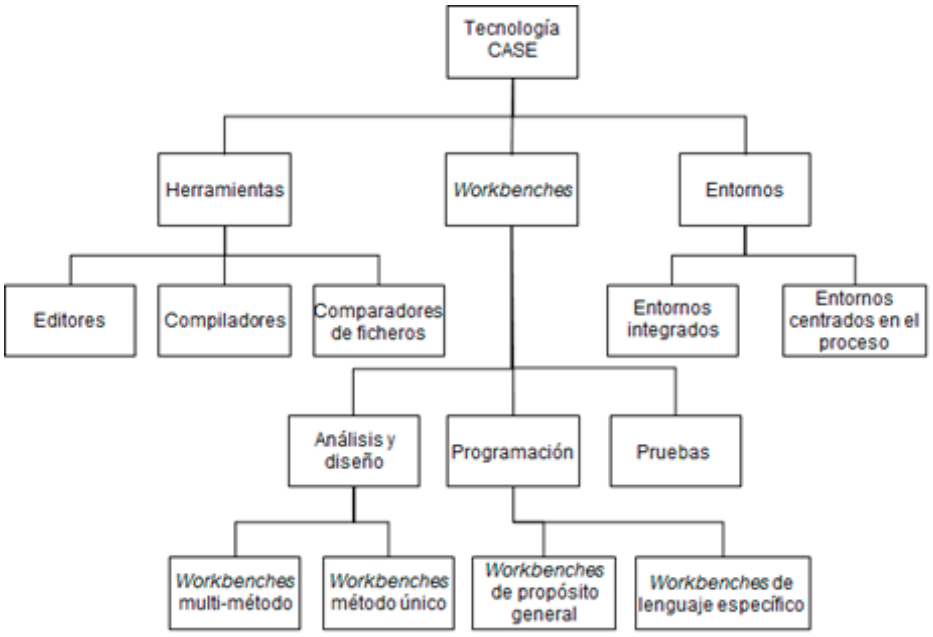


Figura 8. Tecnología CASE: herramientas, *workbenches* y entornos.

Para el modelado de sistemas *software* con UML, las herramientas CASE pueden ofrecer, entre otras, las siguientes características:

- Representación gráfica.
- Corrección sintáctica.
- Coherencia entre distintos diagramas.
- Integración con otras aplicaciones de modelado.
- Trabajo multiusuario.

- Reutilización.
- Generación de código.

En el contexto del desarrollo de *software* dirigido por modelos (DSDM) que se estudiará más adelante, donde los modelos son los que guían la construcción del *software* (no es lo mismo que el desarrollo basado en modelos, donde se utilizan modelos para el desarrollo del sistema pero no constituyen el eje fundamental), las herramientas CASE van a permitir entre otras funcionalidades (OMG-MDA, 2005):

- Especificar un sistema *software* independientemente de la plataforma sobre el que se va a soportar.
- Especificar plataformas o seleccionar especificaciones de plataformas existentes.
- Seleccionar una plataforma determinada para el sistema a desarrollar.
- Transformar la especificación del sistema *software* en una especificación específica para la plataforma de desarrollo seleccionada.

4.7. Referencias bibliográficas

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

Arefi, F., Hughes, C.E. y Workman, D.A. (1990). Automatically Generating Visual Syntax-Directed Editors. *Journal of Communication of the ACM*, 33(3), 349-360.

Bardohl, R., Taentzer, G., Minas, M. y Schurr, A. (1999). Application of Graph Transformation to Visual Languages. *Handbook of Graph Grammars and Computing by Graph Transformation, volume II: Applications, Languages and Tools*, 105-180. World Scientific.

Booch, G., Rumbaugh, J. y Jacobson, I. (2006). *El Lenguaje Unificado de Modelado. Guía del usuario. Segunda edición*. Addison-Wesley.

Steve Cook, S. y Daniels, J. (1994). *Designing object systems: object-oriented modelling with syntropy*. Prentice-Hall.

Franck, R. (1976). PLAN2D- Syntactic Analysis of Precedence Graph Grammars. *Actas de 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, 134-139. Atlanta, Georgia.

Fowler, M. (2003). *UML Distilled, 3rd edition*. Reading, MA.: Addison-Wesley.

Génova, G., Valiente, M.-C. y Nubiola, J. (2005). A Semiotic Approach to UML Models. *Actas de 1st International Workshop on Philosophical Foundations of Information Systems Engineering (PHISE'05). June 13, 2005. Porto, Portugal. Esperanza Marcos, Roel Wieringa (eds.)*, 547-557.

Graham, I., Henderson-Sellers, B. y Younessi, H. (1997). *The OPEN Process Specification*. Addison Wesley.

Habermann, A.N. (1986). Technological Advances in Software Engineering. ACM Annual Computer Science Conference. *Actas de 1986 ACM fourteenth annual conference on Computer Science. Cincinnati, Ohio, United States*, 29-37.

Hoare, C.A.R. (1978). Communicating Sequential Processes. *Journal of Communications of the ACM*, 21(8), 666-677.

Jacobson, I., Ericsson, M. y Jacobson, A. (1995). *The Object Advantage. Business Process Reengineering With Object Technology*. Addison-Wesley.

Jensen, K. (1996). *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1 (Second Edition)*. Springer.

Klint, P., Lämmel, R. y Verhoef, C. (2005). Toward an Engineering Discipline for GRAMMARWARE. *Journal of ACM Transactions on Software Engineering Methodology (TOSEM)*, 14(3), 331-380.

OMG (2005). MDA Guide Revision Draft, Version 00.03. *Document ormsc/05-11-03*.

OMG (2005). UML Profile for Schedulability, Performance, and Time Specification. Version 1.1. *Document formal/05-01-02*. Recuperado de http://doc.omg.org/formal/2005-01-02.pdf

OMG (2011). OMG Unified Modeling Language (OMG UML), Infrastructure. Version 2.4.1. *Document formal/2011-08-05*. Recuperado de http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/

OMG (2011). OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.4.1. *Document formal/2011-08-06*. Recuperado de http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

OMG (2014). Object Constraint Language. Version 2.4. *Document formal/2014-02-03*. Disponible en: http://www.omg.org/

Parnas, D.L. (1972). On the Criteria to Be Used in Decomposing Systems into Modules. *Journal of Communications of the ACM*, 15 (12), 1053-1058.

Petre, L. (2005). Modeling with Action Systems. *Turku Centre for Computer Science, TUCS Dissertations*, 69.

Salomaa, A. (1973). *Formal Languages*. Academic Press.

Silva, M. (1985). *Las Redes de Petri: en la Automática y la Informática*. Editorial AC.

Smith, G. (2000). *The Object-Z Specification Language. Advances in Formal Methods*. Kluwer Academic Publishers.

Sommerville, I. (2005). *Ingeniería del Software. Séptima edición*. España: Pearson Addison-Wesley.

Spivey, J.M. (1992). *The Z Notation: A Reference Manual (Second Edition)*. Prentice Hall International Series in Computer Science.

van Deursen, A., Klint, P. y Visser, J. (2000). Domain Specific Languages: An Annotated Bibliography. *Journal of ACM SIGPLAN Notice*, 35(6), 26-36.

Lo + recomendado

No dejes de leer...

El Lenguaje unificado de modelado. Manual de referencia

Rumbaugh, J., Jacobson, I. y Booch, G. (2007). *El Lenguaje Unificado de Modelado. Manual de Referencia. UML 2.0 2ª Edición*. España: Pearson Addison-Wesley.

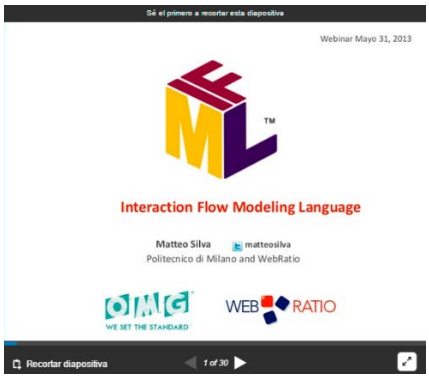
Libro que proporciona una referencia completa sobre los conceptos y elementos que forman parte de UML, incluyendo su semántica, notación y propósito. El libro está organizado para ser una referencia práctica, a la vez que minuciosa, para el desarrollador de *software* profesional. Para profundizar en lo estudiado en el tema se recomienda la lectura de la parte 1: Antecedentes (páginas 1-21) y el capítulo 12: Perfiles (páginas 103-106) de este libro.

Disponible en el aula virtual bajo licencia CEDRO

No dejes de ver...

Webinar IFM en español

Transparencias del webinar sobre el lenguaje IFML que ayudan a entender este nuevo estándar del OMG para el modelado de la interfaz de usuario de una aplicación.



Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
http://es.slideshare.net/silvamatteo/webinar-ifml-en-espaol

Estandarización del lenguaje de modelado

Este vídeo muestra una interesante entrevista realizada al presidente del Object Management Group (OMG), Richard Soley, sobre la estandarización del lenguaje de modelado de interfaces de usuario *Interaction Flow Modeling Language* (IFML).

Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://www.youtube.com/watch?v=ZT1Z0zOrOc4

+ Información

A fondo

Diferencia entre análisis y diseño, ¿por qué es relevante?

Génova, G., Valiente, M.C. y Marrero, M. (2006). Sobre la Diferencia entre análisis y diseño, y por qué es relevante para la transformación de modelos. *III Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones (DSDM'06). En el marco de las XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006). Sitges, 3 de octubre de 2006.*

En este artículo se intenta clarificar las confusiones que encontramos en torno a los términos «modelo de análisis» y «modelo de diseño», ampliamente usados en la ingeniería de *software*.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://ceur-ws.org/Vol-227/paper01.pdf

Enlaces relacionados

MODELS

Model Driven Engineering Languages and Systems (MODELS) es el Congreso anual de referencia sobre modelado de sistemas y *software* que inició su andadura en el año 1998 y que cubre todo lo relacionado en este ámbito, desde lenguajes y métodos hasta herramientas y aplicaciones.

Accede a la página desde el aula virtual o a través de la siguiente dirección web:

http://www.modelsconference.org/

OMG

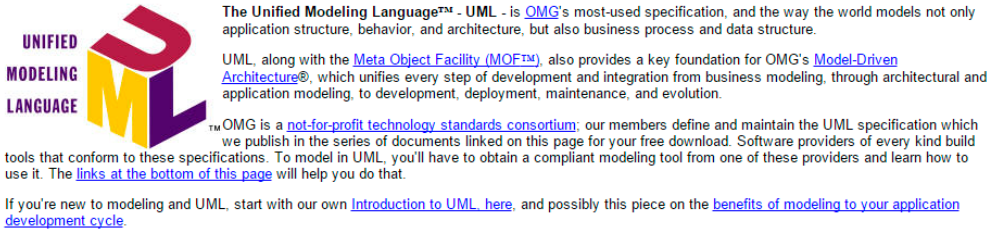
El *Object Management Group* (OMG) es una organización internacional sin ánimo de lucro que se dedica al establecimiento de diversos estándares de tecnologías OO y de modelado, como por ejemplo, *Unified Modeling Language* (UML), *Model Driven Architecture* (MDA) y *Business Process Model and Notation* (BPMN), favoreciendo el diseño visual, así como la ejecución y mantenimiento de *software* y de otros procesos.

Accede a la página desde el aula virtual o a través de la siguiente dirección web:

http://www.omg.org

UML

El enlace *Unified Modeling Language* (UML) *Resource Page* forma parte del OMG y contiene toda la información asociada con el lenguaje de modelado UML.



Accede a la página desde el aula virtual o a través de la siguiente dirección web:

http://www.uml.org/

Bibliografía

Booch, G., Rumbaugh, J. y Jacobson, I. (2006). *El Lenguaje Unificado de Modelado. Guía del usuario. Segunda edición*. Addison-Wesley.

Booch, G., Rumbaugh, J. y Jacobson, I. (2006). *UML 2.0 2ª Edición*. Pearson Addison-Wesley.

Booch, G., Rumbaugh, J. y Jacobson, I. (2007). *El Lenguaje Unificado de Modelado. Guía de Referencia. UML 2.0 2ª Edición*. Pearson Addison-Wesley.

Fowler, M. (2003). *UML Distilled. A Brief Guide to the Standard Object Modeling Language*. Pearson Addison-Wesley.

Graham, I., Henderson-Sellers, B. y Younessi, H. (1997). *The OPEN Process Specification*. Addison Wesley.

Jacobson, I., Booch, G. y Rumbaugh, J. (2000). *El Proceso Unificado de Desarrollo de Software*. Addison Wesley.

Jensen, K. (1996). *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1 (Second Edition)*. Springer.

Pressman, R. S. (2010). *Ingeniería del software. Un enfoque práctico. Séptima edición*. McGraw-Hill.

OMG. (2011). OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.4.1. *Document formal/2011-08-06*. Recuperado en: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

Test

1. ¿Qué comprende el proceso CBSE?

A. Agrupar objetos independientes en el desarrollo de aplicaciones, con un bajo nivel de acoplamiento.

B. Definir, implementar e integrar o ensamblar componentes independientes en el desarrollo de aplicaciones, con un bajo nivel de acoplamiento.

C. Definir, implementar e integrar o ensamblar componentes independientes en el desarrollo de aplicaciones.

D. Reutilizar componentes a la hora de implementar aplicaciones.

2. Un lenguaje formal:

A. No presenta ambigüedad semántica.

B. No admite la OO.

C. Considera a los lenguajes como conjuntos de cadenas basados en algún alfabeto.

D. No permiten definir lenguajes de dominio específico (DSL).

3. ¿Qué define un DSL?

A. Un lenguaje de modelado de propósito general.

B. Un lenguaje personalizado que está dirigido a un dominio pequeño.

C. Un lenguaje de modelado dirigido a un dominio pequeño.

D. Ninguna de las anteriores definiciones es correcta.

4. El lenguaje UML:

A. Solo se puede utilizar para la OO.

B. Es un lenguaje formal.

C. Es un lenguaje que no presenta ambigüedad semántica.

D. Es un lenguaje de modelado que permite modelar sistemas y *software*.

5. Entre las características de los lenguajes visuales se encuentran:

A. La representación gráfica.

B. Son lenguajes formales.

C. Proporcionan construcciones gráficas orientadas a la programación.

D. Son los lenguajes de modelado para la interfaz de usuario.

6. Marcar cuál(es) de las siguientes afirmaciones es correcta:

A. UML es un lenguaje visual.

B. UML modela las vistas estática y dinámica de los sistemas.

C. UML es un lenguaje sólo válido para un tipo de dominios.

D. UML es un lenguaje de propósito general.

7. Modelar el *software* es útil para:

A. No resulta útil, es una pérdida de tiempo, lo más eficiente es programar directamente la solución.

B. Entender el sistema que se desea implementar.

C. Entender el negocio en el que va a funcionar el sistema a implementar.

D. Evitar la ambigüedad semántica.

8. ¿Qué es el modelo de dominio?

A. El modelo que especifica lo que se ha de implementar.

B. El modelo que representa el mundo real.

C. El modelo que define la plataforma de desarrollo.

D. Es el modelo que representa el entorno en el que va a funcionar el sistema a implementar.

9. ¿Qué es el modelo de especificación?

A. El modelo que define la estructura y el comportamiento de los objetos del *software*.

B. El modelo que representa el mundo real.

C. El modelo que generan los expertos del negocio del sistema a implementar.

D. El modelo que especifica cómo se ha de integrar el sistema en el dominio del que va a formar parte.

10. Las herramientas CASE:

A. Solo son válidas para introducir código.

B. Facilitan la extracción de requisitos.

C. Facilitan la ingeniería de requisitos.

D. Transforman la especificación del sistema *software* en una especificación específica para la plataforma de desarrollo seleccionada.

