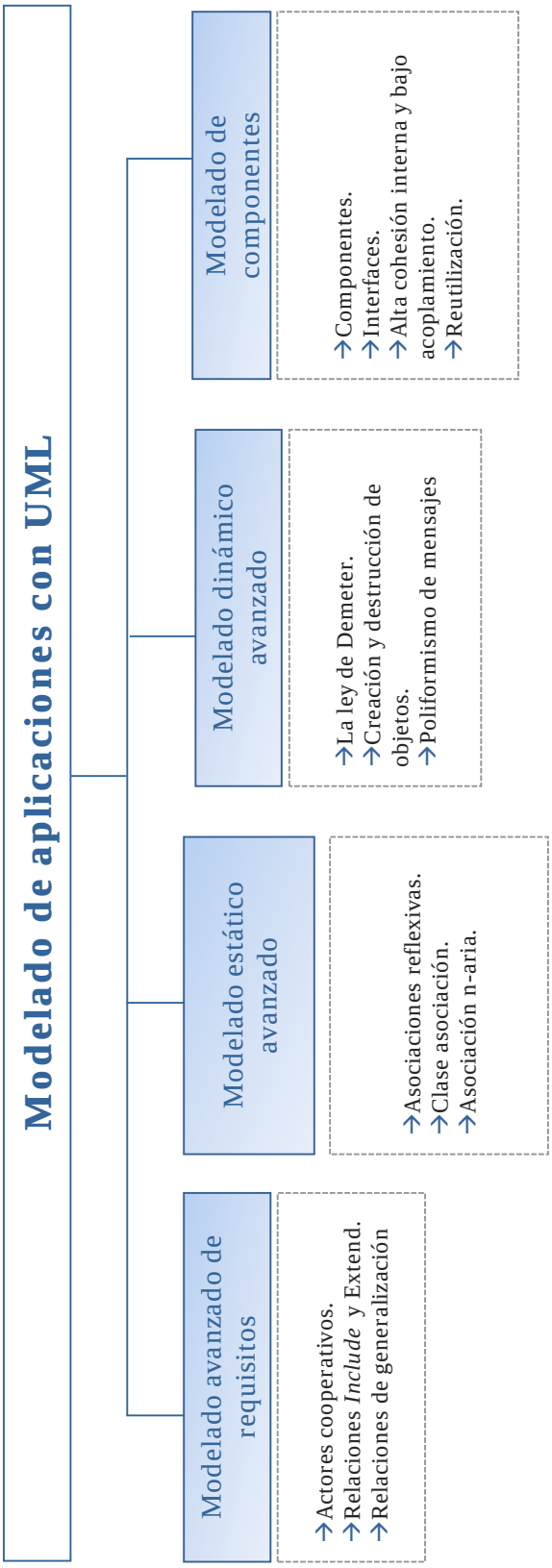
Modelado de aplicaciones con UML

TEMA 5

Esquema



Ideas clave

5.1. ¿Cómo estudiar este tema?

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

En este tema se revisan conceptos y se estudian aspectos concretos más avanzados en el modelado de *software* con UML 2. En particular, se analizarán más en detalle ciertos elementos vinculados con la ingeniería de requisitos (la disciplina que trata de determinar y documentar los requisitos de un sistema), la semántica vinculada a las asociaciones y el modelado de componentes.

5.2. Modelado avanzado de requisitos

La extracción de requisitos de un sistema *software* es la primera y se podría decir que la más crítica de las tareas a realizar en todo proyecto, y a la que, curiosamente, demasiadas veces no se gestiona adecuadamente implementando, en consecuencia, aplicaciones de baja calidad con las que el cliente no se siente satisfecho.

A pesar de que en la actualidad se dispone de una gran cantidad de herramientas formales que facilitan la tarea a la hora de modelar sistemas *software* y que incluso dan soporte automatizando algunas de las tareas de análisis, el número de proyectos que fracasan porque el sistema no hace lo que el cliente necesitaba sigue siendo demasiado elevado.

Es muy normal que el cliente y el contratista (es decir, la organización proveedora del *software* o del servicio) sufran problemas de comunicación, tal y como se ilustra de manera muy acertada en la Figura 1. Problema que hay que saber abordar y donde los modelos del sistema que puedan ser entendidos por los expertos del dominio del negocio (que lo normal es que no tengan un perfil técnico) juegan un papel fundamental en esta etapa.

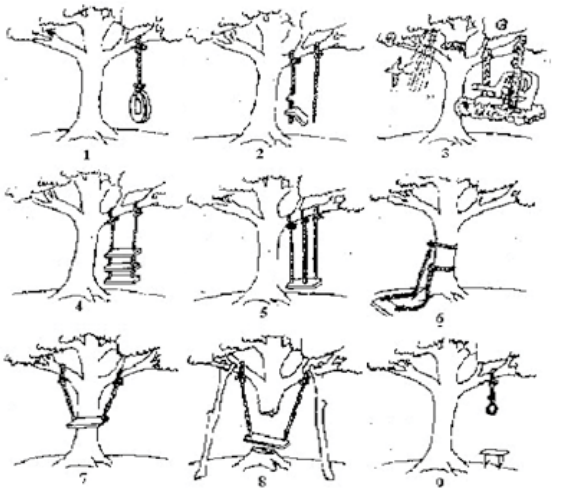


Figura 1. El típico problema de incomunicación. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

Los modelos deben convertirse en una herramienta efectiva de comunicación entre clientes y contratistas de tal forma que, el contratista ha de ser capaz de **especificar los requisitos** de la manera más **formal** posible, entendibles por los clientes, **sin que existan ambigüedades ni contradicciones**, y en caso de darse, se detecten en las fases más tempranas del proyecto.

La **gestión de los requisitos** es, por tanto, una de las actividades más importantes a la que debe enfrentarse todo proyecto si quiere tener éxito. La gestión de requisitos se define en (Leffingwell y Widrig, 2003, 16) como «el enfoque sistemático para la elicitación, organización, y documentación de los requisitos del sistema, y el proceso que establece y mantiene acuerdos entre el cliente y el equipo del proyecto para tratar los cambios en los requisitos».

En la fase de **elicitación de requisitos**, el cliente y el contratista analizan los problemas que se dan en el dominio, así como las necesidades que se identifican y sus características. En base a este análisis, se decide qué cambios se han de introducir en el dominio y la funcionalidad que se ha de llevar a cabo en el nuevo sistema para cubrir las necesidades del cliente que permitirán mejorar su negocio (Olivé, 2007, 27).

Dentro de este contexto, un **requisito *software*** se define como una responsabilidad del *software* que se ha de cumplir para poder alcanzar los objetivos del sistema definidos

en un contrato, especificación o cualquier otro tipo de documentación proporcionada (Dorfman y Thayer, 1990).

Un aspecto fundamental en la actividad de especificación de requisitos *software* es saber distinguir los **requisitos falsos** de los **requisitos verdaderos** tal y como los definen McMenamin y Palmer (1984). Un requisito falso sería aquel que sin que se implemente, la aplicación sigue cumpliendo sus objetivos. Cualquier característica irrelevante para el proyecto actual, así como una actividad que simplemente se ha de llevar a cabo para adaptarse a la tecnología que se va a utilizar para implementar el sistema serían ejemplos de requisitos falsos del sistema a implementar de acuerdo a McMenamin y Palmer (1984).

Los requisitos falsos son clasificados por McMenamin y Palmer (1984) en dos categorías: **requisitos tecnológicos** y **requisitos arbitrarios**. Los **requisitos falsos de tipo tecnológico** pueden ser introducidos por los analistas cuando, por ejemplo, anticipan las características técnicas del nuevo sistema y las introducen en la especificación de requisitos. Por otro lado, **los requisitos falsos de tipo arbitrario** son los que introducen los analistas, por ejemplo, al especificar más funcionalidad de la que necesita el sistema, sin que la haya solicitado el cliente. Los requisitos falsos suponen, aunque a veces no se vea así, una gran amenaza al éxito del proyecto, ya que pueden derivar en especificaciones incorrectas y que no se entienden, proyectos que se van de tiempo y presupuesto, sistemas difíciles de mantener, etc.

Si se adopta la técnica de los casos de uso para modelar la funcionalidad del sistema hay que ser conscientes de la ambigüedad presente en UML y definirlo de manera concisa con la documentación adicional que sea necesaria. Por ejemplo, de acuerdo a la documentación de UML (OMG, 2011) más de una actor podrá estar conectado a un caso de uso (ver Figura 2), pero no hay forma de indicar con elementos de UML si se necesita que todos los actores actúen a la vez para que se realice el caso de uso, o el caso de uso es para cada actor que lo ejecutará de manera independiente, o se refiere a que deben seguir un orden determinado.

Esta ambigüedad solo es posible solventarla con información adicional. Como ocurre con este ejemplo, claramente, en un caso de uso se pueden dar un montón de situaciones que con la **descripción (especificación) textual**, **precondiciones y postcondiciones** con la que se suele acompañar, no resulta suficiente. Es por ello, que se podrán

complementar con **diagramas de interacción, diagramas de actividad y diagramas de esta**dos que se relacionen y favorezcan su comprensión.

Figura 2. Actores cooperativos

Por otro lado, resulta conveniente limitar, o mejor eliminar, las relaciones ***Include*** (<<*include*>>) y ***Extend*** (<<*extend*>>) que se suelen utilizar en los diagramas de casos de uso (ver Figura 3), ya que, por un lado, no está clara su diferencia (reutilización vs inserción), por otro lado, confunde con la definición de caso de uso que dice que **un caso de uso representa una unidad coherente de funcionalidad**, y además puede llevar a la malinterpretación, observando el diagrama, de que se da un procesamiento secuencial, cuando en realidad, ese comportamiento adicional podría estar localizado en cualquier parte de la funcionalidad que representa el caso de uso.

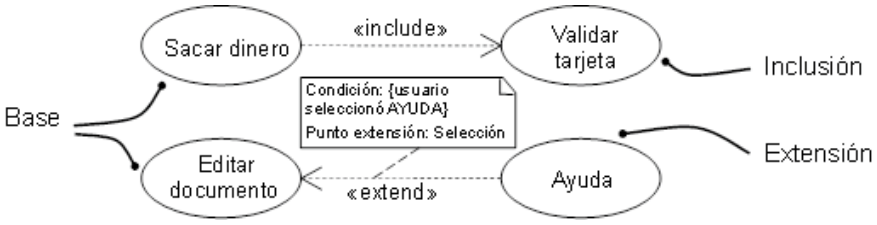


Figura 3. Relaciones *Include* y *Extend*. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

Para modelar las situaciones que tienen en común una secuencia de acciones previa a la ejecución del resto de casos de uso, como sería por ejemplo, el acceso al sistema (identificación, inicio y cierre de sesión, etc.), la solución menos problemática sería definir dichas acciones como una subactividad dentro del diagrama de actividad correspondiente a cada caso de uso que se vea afectado por esta situación.

Por último, comentar también que, a la hora de modelar casos de usos también hay que tener en cuenta, por un lado, **las relaciones de generalización** que se dan entre actores, cuando se identifican distintos roles que han de realizar un mismo caso de uso y que son subtipos unos de otros (ver Figura 4); y por otro lado, las relaciones de

generalización que se da entre casos de uso, cuando se detectan especializaciones para un caso de uso concreto (ver Figura 5). Al igual que ocurría con las relaciones de *Include* y *Extend*, las generalizaciones entre casos de uso es una solución que se debería evitar en la medida de lo posible y realizar otras técnicas de modelado que favorezcan su comprensión.

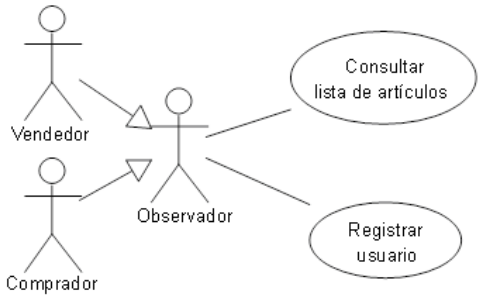


Figura 4. Relaciones de generalización entre actores. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

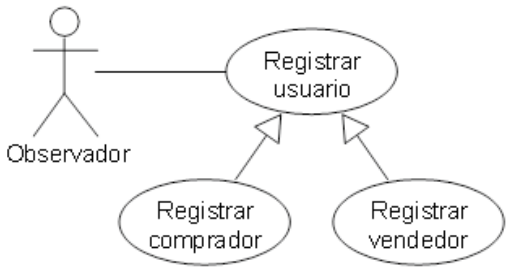


Figura 5. Relaciones de generalización entre casos de uso. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

5.3. Modelado estático avanzado

Cuando se desarrolla una aplicación el modelo de clases constituye una pieza muy importante de comunicación entre analistas, diseñadores y programadores para entender qué entidades se consideran, como están relacionadas unas con otras y de qué manera se van a comunicar. Dentro del modelado de la vista estática (es decir, estructura) del sistema con UML, se dan una serie de aspectos que hay que tener en cuenta y se han de comprender correctamente para que los modelos realmente constituyan una fuente fiable de cómo se encuentra definido el sistema.

En este punto, también es interesante recordar la diferencia entre el **modelo conceptual (análisis)** que es el que identifica las clases, atributos y operaciones corresponden a conceptos del dominio del negocio del sistema; y **el modelo de *software* (diseño)** que se correspondería con las clases, atributos y métodos que se corresponden con los conceptos de la plataforma de desarrollo escogida para su implementación.

Además, merece la pena comentar que se podría hacer un modelo previo al modelo conceptual que sería lo que se puede denominar como **modelo del entorno** que «define esa parte de la realidad que rodea al sistema informático y con la que este interactúa para proporcionarle determinados servicios» (Génova, Valiente y Marrero, 2006). El modelo del entorno podría ser también de ayuda a la hora de entender y extraer los requisitos del sistema *software* a implementar.

A continuación, se describen una serie de elementos que forman parte de la vista estática del sistema y que requieren especial atención.

Asociaciones reflexivas

Una **asociación reflexiva** o recursiva es una asociación que enlaza una clase consigo misma, tal y como se ilustra en la Figura 6. Una asociación reflexiva presenta las siguientes características:

- Los enlaces pueden conectar dos instancias diferentes de la misma clase, pero también puede conectar una instancia consigo misma.
- De cara a poder distinguir los extremos de la asociación, los nombres de rol son obligatorios.

- No es simétrica.

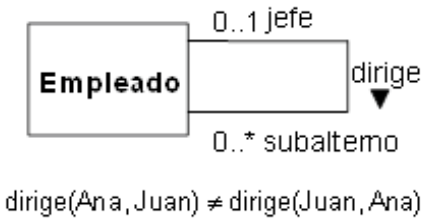


Figura 6. Asociación reflexiva. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

Clase asociación

Una **clase asociación** se corresponde con una clase que tiene las propiedades propias de una clase y una asociación, tal y como se ilustra en la Figura 7. Una clase asociación presenta las siguientes características:

- Como se trata de un elemento único, ha de tener un único nombre.
- Como cualquier otra asociación, en principio, no puede contener *tuplas* repetidas (es decir, no puede conectar dos mismos objetos más de una vez), aunque los valores de los atributos sean distintos. Si se deseara representar el registro histórico (es decir, que permita *tuplas* repetidas), se tendría que poner la restricción {*nonunique*} en cada extremo de asociación.

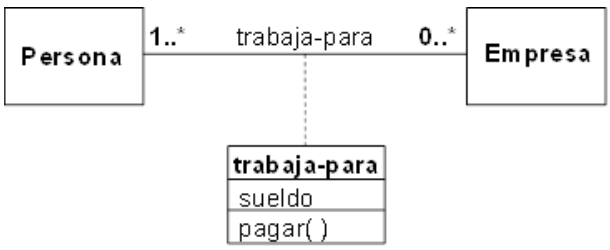


Figura 7. Clase asociación. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

A la hora de programarla en una plataforma concreta (Java, .NET...), en diseño, la clase asociación se tendrá que transformar en una clase intermedia haciendo un cruce de las cardinalidades, tal y como se muestra en la Figura 8.

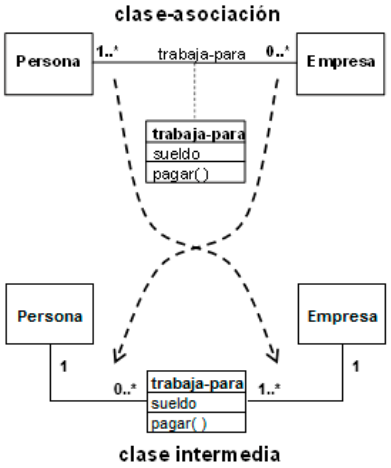


Figura 8. Transformación de una clase asociación en clase intermedia. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

Asociación n-aria

Una **asociación n-aria** se corresponde con una asociación que enlaza n clases, tal y como se ilustra en la Figura 9. Una asociación n-aria presenta las siguientes características:

- No permite especificar la dirección del nombre, la navegabilidad ni las asociaciones de agregación.
- Admite clases asociación.
- La multiplicidad especificada en un extremo de una clase representa el número permitido de instancias para cualquier posible combinación de instancias de las otras n-1 clases.
- La multiplicidad mínima suele ser 0.
- Efecto **«rebote del uno»**. Este efecto indica que cuando la multiplicidad mínima especificada en un extremo de una clase sea 1 (o superior) implica que debe existir un enlace (o más) para cualquier posible combinación de instancias de las otras n-1 clases.

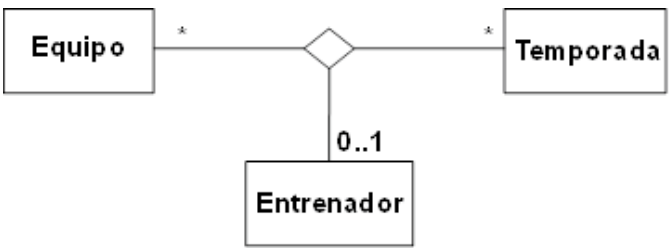


Figura 9. Asociación n-aria. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

5.4. Modelado dinámico avanzado

Como ya se ha comentado en el apartado anterior, con la vista estática del sistema el equipo del proyecto determina la estructura completa del sistema a implementar. Sin embargo, esta vista no es suficiente para implementar la aplicación y se hace necesario modelar el comportamiento en aquellas funcionalidades que requieran un tratamiento especial o sea difícil de comprender si no se visualiza gráficamente. UML ofrece varios modelos dinámicos que ayudarán a especificar el comportamiento del sistema como son los diagramas de interacción, los diagramas de actividad y los diagramas de estado.

A continuación, se describen una serie de elementos que forman parte de la vista dinámica del sistema y que requieren especial atención.

La ley de Demeter

La **ley de Demeter** sirve de referencia clave a la hora de entender cómo se realiza la comunicación vía mensajes entre los objetos que constituyen el sistema. La ley de Demeter define a qué instancias puede enviar mensajes una instancia (objeto) determinada de una clase (ver Figura 10):

- Una instancia que esté conectada mediante un enlace navegable (es decir, una instancia de asociación de la clase).
- Una instancia recibida como parámetro en la activación.
- Una instancia creada localmente en la ejecución, o bien, una variable local.
- A sí misma, es decir, la instancia en sí es el emisor del mensaje (no confundir con la asociación reflexiva).

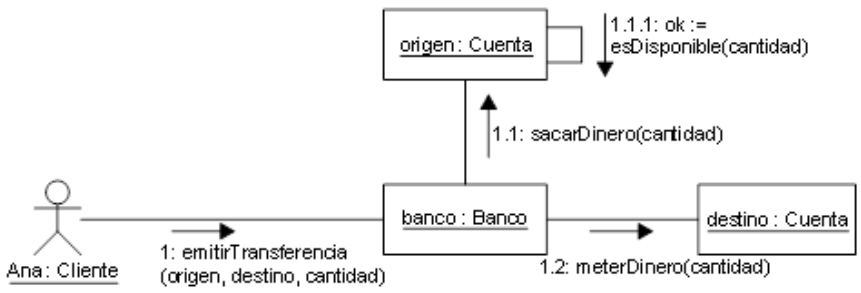


Figura 10. Diagrama de comunicación que ilustra la ley de Demeter. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

Creación y destrucción de objetos

La Figura 11 ilustra cómo se ha de modelar con UML la creación y la destrucción de objetos en un diagrama de interacción. En este caso concreto, se muestra como se cancela una cuenta y se transfiere el saldo a una nueva de otro tipo.

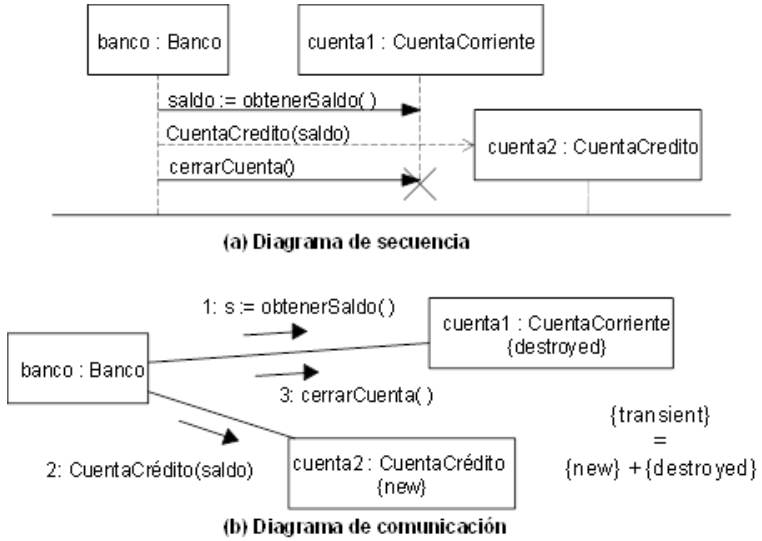


Figura 11. Diagramas de interacción que modelan la creación y destrucción de objetos. Fuente: Curso de Diseño de Software Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

Polimorfismo de mensajes

Como se ha comentado con anterioridad, los objetos se comunican entre sí a través del envío de mensajes. El envío de mensajes es particularmente expresivo cuando la operación invocada se ejecuta en forma polimórfica en los receptores, es decir, que aunque varios objetos compartan la misma interfaz por ser de un mismo tipo (de manera directa o indirecta) cada uno puede interpretar el mensaje de manera distinta.

El uso correcto de la propiedad de polimorfismo va a facilitar la evolución del sistema en el caso de que se tenga necesidad de añadir nuevas clases a su estructura, sin que haya necesidad de hacer uso de instrucciones de ramificación múltiple ya que la ramificación es implícita, es decir, según sea la clase del objeto así interpretará el mensaje. La Figura 12 ilustra esta situación.

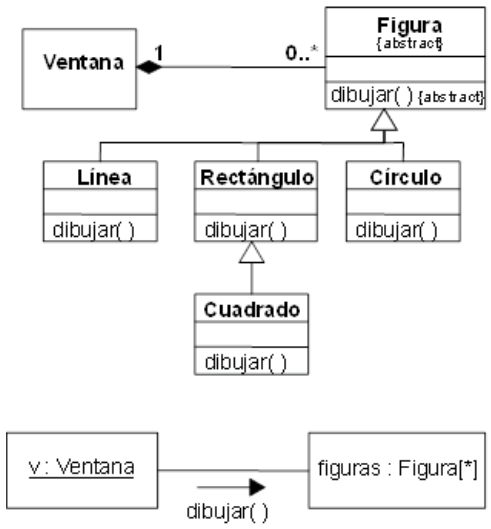


Figura 12. Polimorfismo de mensajes. Fuente: Curso de Diseño de *Software* Avanzado disponible en el OpenCourseWare de la Universidad Carlos III de Madrid.

5.5. Modelado de componentes

Con el objetivo de favorecer la **reutilización** de objetos y del *software* en general, de una manera más uniforme, el **diseño basado en componentes** ha ido adquiriendo mayor importancia dentro la ingeniería de *software* como técnica arquitectónica. Los componentes van a permitir la creación de nuevos elementos a partir de otras «piezas» ya creadas con la particularidad de ofrecer, además, un alto nivel de abstracción, ya que un componente ha de ser desarrollado de manera mucho más genérica que cualquier

otro artefacto *software*, pero primando los principios de **alta cohesión interna** y **bajo acoplamiento externo**. A pesar de que un componente va a estar formado fundamentalmente por objetos, es importante no confundir ambos conceptos. Un componente se define como «una unidad de 'despliegue' independiente» y presenta las siguientes características (Szyperski, 1998):

- Un componente podría ser utilizado por cualquier organización sin necesidad de ningún conocimiento previo.
- Un componente no tiene estado persistente en sí mismo, lo tienen (opcionalmente) sus objetos.
- Implementa una serie de interfaces y utiliza otras.
- Encierran otros elementos de modelado, como por ejemplo clases, permitiendo a otros componentes acceder a ellos.
- Un componente, además de clases, también puede contener otros módulos de código.

Otras definiciones bastante descriptivas se encuentran en (Stevens y Pooley, 2000) y en (Booch, Rumbaugh y Jacobson, 2006). Stevens y Pooley (2000, 11, 213) definen un componente como «un elemento reutilizable y reemplazable, lo que requiere una interfaz bien definida, abstracción cohesiva y un bajo acoplamiento con otras interfaces». Por su parte, de manera un poco más simplificada, Booch, Rumbaugh y Jacobson (2006, 208) definen un componente como «una parte reemplazable de un sistema que conforma y proporciona la implementación de un conjunto de interfaces» y donde la interfaz viene definida como «una colección de operaciones que especifican un servicio proporcionado o solicitado por una clase o componente». En este punto, conviene aclarar que las interfaces implementadas o utilizadas por un componente son realmente utilizadas y/o implementadas por el correspondiente contenido.

Para alcanzar sus objetivos, los sistemas *software* estarán basados en componentes que hacen uso de otros componentes, que en UML se denotará a través de una relación de dependencia. Para poder conseguir el bajo acoplamiento, no se hace que un componente dependa de otro, sino que al final dependa solamente de una o varias de sus interfaces. En UML, el diagrama de componentes da soporte tanto a componentes lógicos (componentes de negocio, componentes de proceso...), como a componentes físicos (EJB, CORBA, COM+, JavaBeans, .NET...), (OMG, 2011, 145). La Figura 13 muestra un ejemplo de un diagrama de componentes modelado con UML. En la Figura 14 se muestra también un diagrama de componentes, pero en este caso se puede observar, como un componente está constituido a su vez por otros componentes.

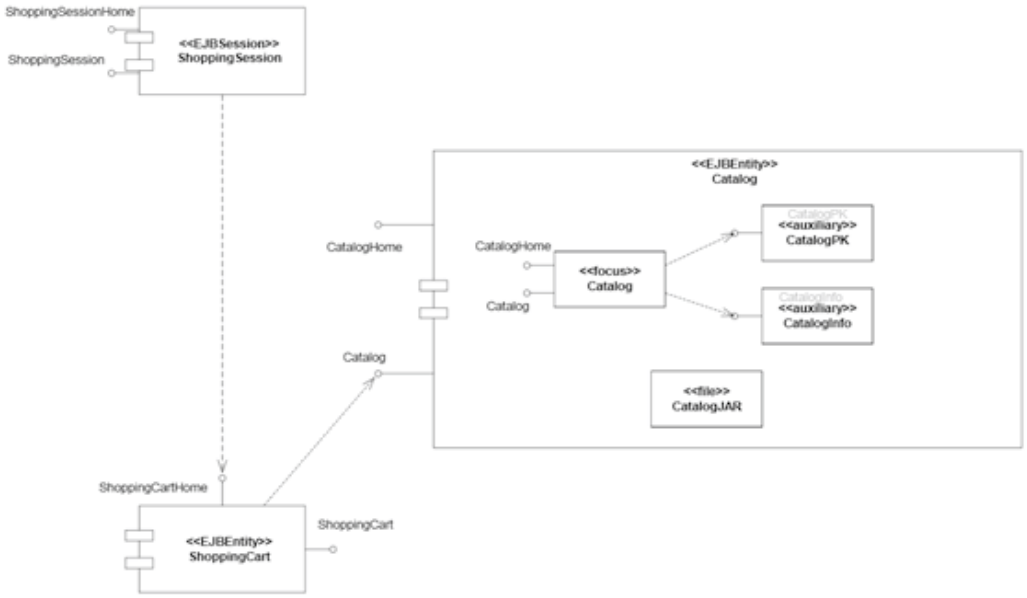


Figura 13. Diagrama de componentes con notación UML. Fuente: http://www.omg.org/news/meetings/workshops/presentations/embedded-rt2002/02-1_Kobryn_UML_Tutorial.pdf

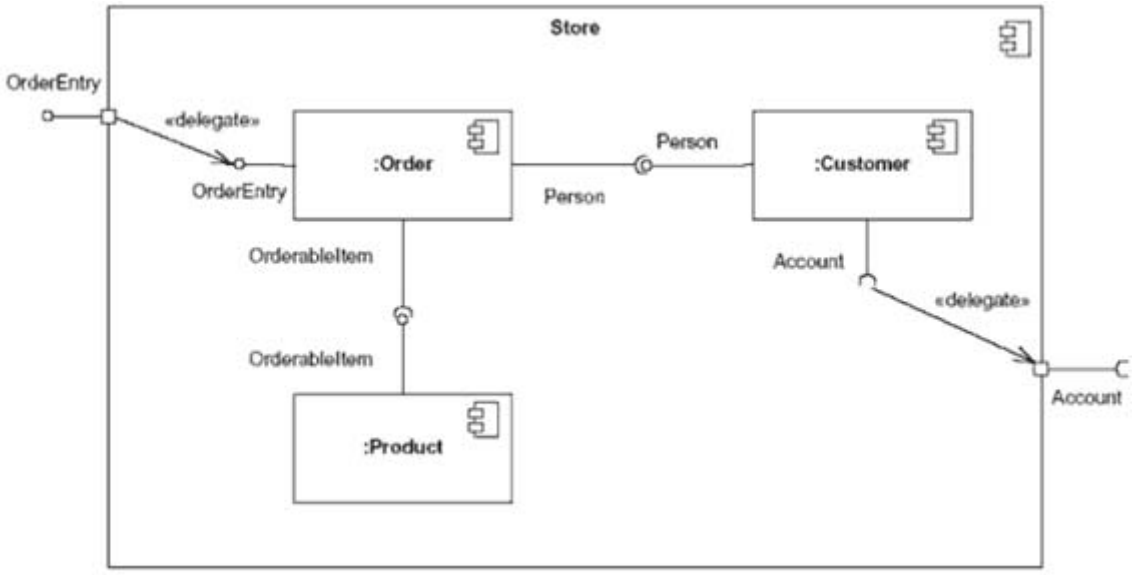


Figura 14. Componente formado a partir de otros componentes. Fuente: http://www.ibm.com/developerworks/rational/library/dec04/bell/

Cuando una organización opta por el ensamblado de componentes a la hora de especificar el sistema *software* que necesita para su negocio, deberá plantearse si va a adquirir todos los componentes necesarios para el sistema o, por el contrario, prefiere desarrollarlos de manera interna. La Tabla 1 y la Tabla 2 resumen las ventajas e inconvenientes de cada aproximación.

Tabla 1: Ventajas e inconvenientes en la adquisición de componentes

Ventajas	Inconvenientes
Es coste y plazo de entrega se fija con antelación, por lo que se reduce el riesgo del proyecto.	Puede obligar a que se haga reingeniería de procesos internos de la organización.
El contratista se encargará de su mantenimiento.	Al ser adquirido, está también disponible para los competidores, por lo que puede que no suponga entonces una ventaja competitiva con respecto al resto de compañías que se encuentran en el mismo sector.
Se asegura que el software cumplirá las normas y estándares establecidos.	Habrá que realizar una búsqueda exhaustiva para tratar de encontrar la mejor relación calidad/precio.
Seguramente ya tengan incorporadas las mejores prácticas de desarrollo de software que van surgiendo.	

Tabla 2: Ventajas e inconvenientes en la fabricación de componentes

Ventajas	Inconvenientes
Se lleva más control de lo que se está desarrollando y se va adecuando a las necesidades del negocio.	El desarrollo del software suele resultar más costoso.
Se controla el know-how de la organización y se favorece la posibilidad de ser más competitivo por valores añadidos al software que no disponen los competidores en el sector.	Posibles lagunas conceptuales cuando el desarrollador no sea un experto del dominio del negocio.
Posibilidad de venta de componentes a otras compañías.	La adecuación a normas y estándares puede ser larga y tediosa.

5.6. Referencias bibliográficas

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

Dorfman, M. y Thayer, R.H. (1990). Standards, Guidelines, and Examples of System and Software Requirements Engineering. *Los Alamitos, CA: IEEE Computer Society Press*.

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.*

Leffingwell, D. y Widrig, D. (2003). *Managing Software Requirements. Second Edition. A Use Case Approach*. Pearson Education.

McMenamin, S.M. y Palmer, J.F. (1984). *Essential Systems Analysis*. Prentice-Hall.

Olivé, A. (2007). *Conceptual Modeling of Information Systems*. Springer-Verlag.

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/

Stevens, P. y Pooley, R. (2000). Using UML. *Software Engineering with Objects and Components. Updated edition*. England: Pearson Addison-Wesley

Szyperski, C. (1998). *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley.

Lo + recomendado

No dejes de leer...

Algunos problemas de la generalización en el metamodelo de UML

Génova, G. y Llorens, J. (2009). Algunos problemas de la generalización en el metamodelo de UML. *Actas de los Talleres de las Jornadas de Ing. del Software y BBDD, 3(2)*.

En este artículo se intenta resolver los inconvenientes que presenta el lenguaje UML a la hora de modelar las relaciones de generalización, para favorecer la manipulación de los modelos por parte de las herramientas CASE que tratan de seguir fielmente el estándar.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web: https://www.researchgate.net/profile/Juan_Llorens/publication/228629788_Algunos_problemas_de_la_generalizacin_en_el_metamodelo_de_UML/links/02e7e525c16dfdceaf000000.pdf

El lenguaje unificado de modelado. Manual de referencia

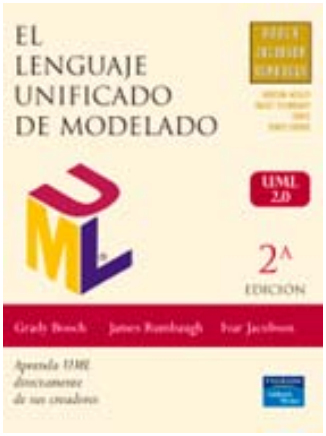
Rumbaugh, J., Jacobson, I. y Booch, G. (2007). *El Lenguaje Unificado de Modelado. Manual de Referencia. UML 2.0 2ª Edición*. España: Pearson Addison-Wesley.

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 del capítulo 3: Un paseo por UML (páginas 25-42) de este libro.

Disponible en el aula virtual bajo licencia CEDRO

El lenguaje unificado de modelado. UML 2.0 2ª Edición

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



Libro que proporciona una referencia del uso de características específicas de UML, pero sin proporcionar una referencia completa de este lenguaje (la referencia completa se encuentra en el libro *El Lenguaje Unificado de Modelado. Manual de Referencia. UML 2.0 2ª Edición*). Para profundizar en lo estudiado en el tema se recomienda la lectura del capítulo 12: *Paquetes* (páginas 173-185); capítulo 15: *Componentes* (páginas 207-222); capítulo 23: *Procesos e hilos* (páginas 351-363) y capítulo 24: *Tiempo y espacio* (páginas 365-372) de este libro.

Disponible en el aula virtual bajo licencia CEDRO

Guía de estilo de los elementos de un esquema conceptual en UML/OCL

Aguilera, D., Gómez, C. y Olivé, A. (2013). Guía de estilo completa para nombrar los elementos de un esquema conceptual en UML/OCL. *Novática: Revista de la Asociación de Técnicos de Informática*, 39(226), 52-58.

En este documento se encuentra una guía de estilo que establece las normas que hay que aplicar a la hora de nombrar los elementos de un esquema conceptual. Se trata de un guía completa, ya que ofrece una regla de nominación para cada uno de los elementos del lenguaje UML a los que se le puede dar nombre.

Naturaleza de las relaciones entre actores y casos de uso

Génova, G. & Llorens, J. (2004). Naturaleza de las relaciones entre actores y casos de uso. *Novática: Revista de la Asociación de Técnicos de Informática*, 30(169), 56-61.

En este artículo se tratan temas referentes a las relaciones en las que pueden participar actores y casos de uso, actualmente definidas en UML como asociaciones: la multiplicidad de estas asociaciones, las ambigüedades que aparecen al asociar un caso de uso con varios actores y finalmente la naturaleza misma de estas relaciones, para las cuales se propone un enfoque diferente.

Accede al documento desde el aula virtual o a través de la siguiente dirección web:
http://www.ie.inf.uc3m.es/ggenova/pub-novatica2004.pdf

No dejes de ver...

UML, un resumen ágil en 39 diapositivas

En este artículo de Javier Garzás se incluye una sesión de transparencias que resume muy bien los diagramas UML a utilizar como herramienta de comunicación cuando se desarrolla un sistema *software*.

Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
http://www.slideshare.net/JavierGarzas/uml-un-resumen-gil-de-uml

+ Información

A fondo

Semantics of the minimum multiplicity in ternary associations in UML

Génova, G., Llorens, J. y Martínez, P. (2001). Semantics of the Minimum Multiplicity in Ternary Associations in UML. *Actas de 4th International Conference UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools.*

En este artículo se intenta clarificar las ambigüedades que se encuentran en el lenguaje UML cuando se modelan asociaciones ternarias.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web: https://www.researchgate.net/profile/Paloma_Martinez3/publication/220868325_Semantics_of_the_Minimum_Multiplicity_in_Ternary_Associations_in_UML/links/0912f5087a0f21b08e000000.pdf

The meaning of multiplicity of n-ary associations in UML

Génova, G., Llorens, J. y Martínez, P. (2002). The meaning of multiplicity of n-ary associations in UML. Software and Systems Modeling, 1(2): 86-97, 2002. *A preliminary version in: Gonzalo Génova, Juan Llorens, Paloma Martínez. [Semantics of the Minimum Multiplicity in Ternary Associations in UML].*

En este artículo se profundiza en más detalle sobre las ambigüedades que se encuentran en el lenguaje UML cuando se modelan asociaciones n-arias en general.

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

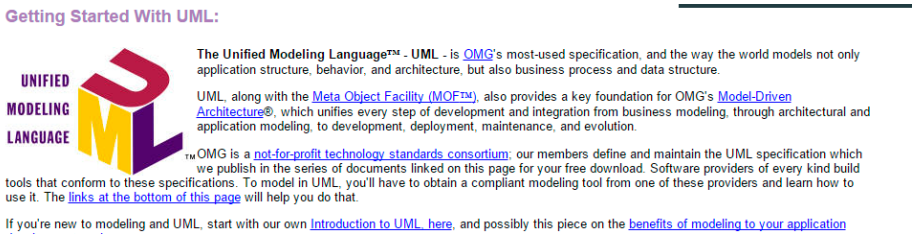
El *Object Management Group* (OMG) es una organización internacional sin ánimo de lucro que se dedica al establecimiento de diversos estándares de tecnologías OO y de modelado, como por ejemplo, *Unified Modeling Language* (UML), *Model Driven Architecture* (MDA) y *Business Process Model and Notation* (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.


Getting Started With UML:

UNIFIED MODELING LANGUAGE

The Unified Modeling Language™ - UML - is OMG's most-used specification, and the way the world models not only application structure, behavior, and architecture, but also business process and data structure.

UML, along with the Meta Object Facility (MOF™), also provides a key foundation for OMG's Model-Driven Architecture®, which unifies every step of development and integration from business modeling, through architectural and application modeling, to development, deployment, maintenance, and evolution.

™OMG is a not-for-profit technology standards consortium; our members define and maintain the UML specification which we publish in the series of documents linked on this page for your free download. Software providers of every kind build tools that conform to these specifications. To model in UML, you'll have to obtain a compliant modeling tool from one of these providers and learn how to use it. The links at the bottom of this page will help you do that.

If you're new to modeling and UML, start with our own Introduction to UML, here, and possibly this piece on the benefits of modeling to your application

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

http://www.uml.org/

Bibliografía

Cockburn, A. (2001). *Writing Effective Use Cases*. Addison-Wesley.

Leffingwell, D. y Widrig, D. (2003). *Managing Software Requirements. Second Edition. A Use Case Approach*. Pearson Education.

Martin, R.C. (2004). *UML para programadores Java*. Pearson Prentice-Hall.

McMenamin, S.M. y Palmer, J.F. (1984). *Essential Systems Analysis*. Prentice-Hall.

Olivé, A. (2007). *Conceptual Modeling of Information Systems*. Springer-Verlag.

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

Actividades

Trabajo: Comparativa entre herramientas CASE para modelado con UML

Para este trabajo, de cara a poder modelar el caso práctico expuesto en el tema 3, el alumno deberá escoger un par de herramientas gratuitas disponibles en la web que permitan modelar con UML (no necesariamente tiene que dar soporte a UML2, con UML1.x sería suficiente). Después debe hacer una comparativa en lo que se refiere a la capacidad para modelar distintos diagramas, no hace falta probar con todos, se puede limitar a los más comunes, como serían los diagramas de casos de uso, diagramas de clases y alguno de interacción. El alumno debe tener en cuenta los aspectos avanzados como los que se han visto en clase. Comentar qué herramienta es más fácil de utilizar y respeta más la especificación de UML definida en su documento *OMG Unified Modeling Language (OMG UML), Superstructure*.

El trabajo solicitado se entregará en un documento en formato Word que contenga la información descrita anteriormente.

Extensión máxima: 10 páginas (Georgia 11 e interlineado 1,5).

Competencias

CB6. Poseer y comprender conocimientos que aporten una base u oportunidad de ser originales en el desarrollo y/o aplicación de ideas, a menudo en un contexto de investigación.
CB8. Que los estudiantes sean capaces de integrar conocimientos y enfrentarse a la complejidad de formular juicios a partir de una información que, siendo incompleta o limitada, incluya reflexiones sobre las responsabilidades sociales y éticas vinculadas a la aplicación de sus conocimientos y juicios.
CB9. Que los estudiantes sepan comunicar sus conclusiones y los conocimientos y razones últimas que las sustentan a públicos especializados y no especializados de un modo claro y sin ambigüedades.
CB10. Que los estudiantes posean las habilidades de aprendizaje que les permitan continuar estudiando de un modo que habrá de ser en gran medida autodirigido o autónomo.

CG5. Capacidad para la puesta en marcha, dirección y gestión de procesos de diseño y desarrollo de sistemas informáticos, con garantía de la seguridad para las personas y bienes, la calidad final de los productos y su homologación.

CE1. Capacidad para modelar, diseñar, definir la arquitectura, implantar, gestionar, operar, administrar y mantener aplicaciones, sistemas, servicios y contenidos informáticos.

CE2. Capacidad para utilizar y desarrollar metodologías, métodos, técnicas, programas de uso específico, normas y estándares de Ingeniería de Software.

CE5. Capacidad para evaluar y utilizar entornos de Ingeniería de Software avanzados, métodos de diseño, plataformas de desarrollo y lenguajes de programación.

CT1. Analizar de forma reflexiva y crítica las cuestiones más relevantes de la sociedad actual para una toma de decisiones coherente.

CT2. Identificar las nuevas tecnologías como herramientas didácticas para el intercambio comunicacional en el desarrollo de procesos de indagación y de aprendizaje grupal.

CT4. Adquirir la capacidad de trabajo independiente, impulsando la organización y favoreciendo el aprendizaje autónomo.

Test

1. ¿Qué se considera requisito falso?

A. Un requisito imposible de implementar.

B. Un requisito que si se implementa el sistema no cumplirá con las especificaciones.

C. Un requisito que aunque no se implemente el sistema seguirá cumpliendo con las especificaciones.

D. Un requisito dado por el cliente que luego decide cambiar.

2. Las relaciones *Include* y *Extend*:

A. Son relaciones que se dan en los diagramas de actividad que describen un caso de uso.

B. Debería limitarse su uso o mejor eliminarse ya que generan confusión.

C. Indican la secuencialidad del caso de uso.

D. Indican especializaciones del caso de uso.

3. Las relaciones de generalización que aparecen en un diagrama de casos de uso:

A. Indican jerarquía entre los elementos relacionados.

B. Indican funcionalidad adicional del caso de uso.

C. Indican el orden en el que se deben realizar los casos de uso.

D. Ninguna de las anteriores definiciones es correcta.

4. Si dos actores aparecen conectados a un mismo caso de uso significa:

A. Que los dos actores colaboran para realizar el caso de uso.

B. Que los dos actores han de realizar una acción de manera simultánea para que se ejecute el caso de uso.

C. Que cada actor realiza el caso de uso de manera independiente.

D. Presenta ambigüedad y habría que consultar información adicional para determinar de qué manera los actores participan en el caso de uso.

5. El modelo conceptual:

A. Identifica las clases, atributos y operaciones del sistema a implementar en términos del dominio del negocio.

B. Identifica las clases, atributos y métodos del sistema a implementar en términos del dominio del negocio.

C. Modela el entorno del sistema tal y como está antes de implementar el sistema.

D. Modela el sistema en términos del lenguaje de programación.

6. El modelo del sistema:

A. Incluye el modelo de análisis y el modelo de diseño del sistema.

B. Modela el entorno del sistema tal y como queda una vez implementado el sistema.

C. Identifica las clases, atributos y métodos del sistema a implementar en términos de la plataforma de desarrollo escogida.

D. Identifica las clases, atributos y métodos del sistema a implementar en términos del dominio del negocio.

7. El modelo del entorno:

A. Modela el negocio tal y como va a quedar una vez se haya implementado el sistema.

B. Identifica las clases, atributos y operaciones del sistema a implementar en términos del dominio.

C. Identifica las clases, atributos y métodos del sistema a implementar en términos de la plataforma de desarrollo escogida.

D. Modela la parte de la realidad que rodea al sistema informático y con la que éste interactúa para proporcionarle determinados servicios.

8. El polimorfismo de mensajes significa:

A. Que distintos objetos pueden actuar de manera distinta ante el mismo mensaje.

B. Que distintos objetos pueden invocar una operación con el mismo nombre pero con diferente número de parámetros definidos en cada operación.

C. Que un objeto puede enviar varios mensajes a la vez.

D. Que distintos objetos de distintas clases comparten una misma interfaz.

9. ¿Qué define la ley de Demeter?

A. La ley de Demeter define el conjunto de operaciones que se dan entre las clases en un sistema concreto.

B. La ley de Demeter define las restricciones de comunicación entre objetos que se dan en el sistema.

C. La ley de Demeter define que un objeto no se puede mandar mensajes a sí mismo, si no tiene definida una asociación reflexiva.

D. La ley de Demeter define a qué instancias puede enviar mensajes un objeto de una clase.

10. ¿Qué se considera componente desde el punto de vista de modelado de *software*?

A. Una unidad reutilizable persistente.

B. Una unidad compuesta de objetos.

C. Una unidad de 'despliegue' independiente que se puede reutilizar y reemplazar.

D. Una unidad que proporciona alta cohesión y un alto acoplamiento.