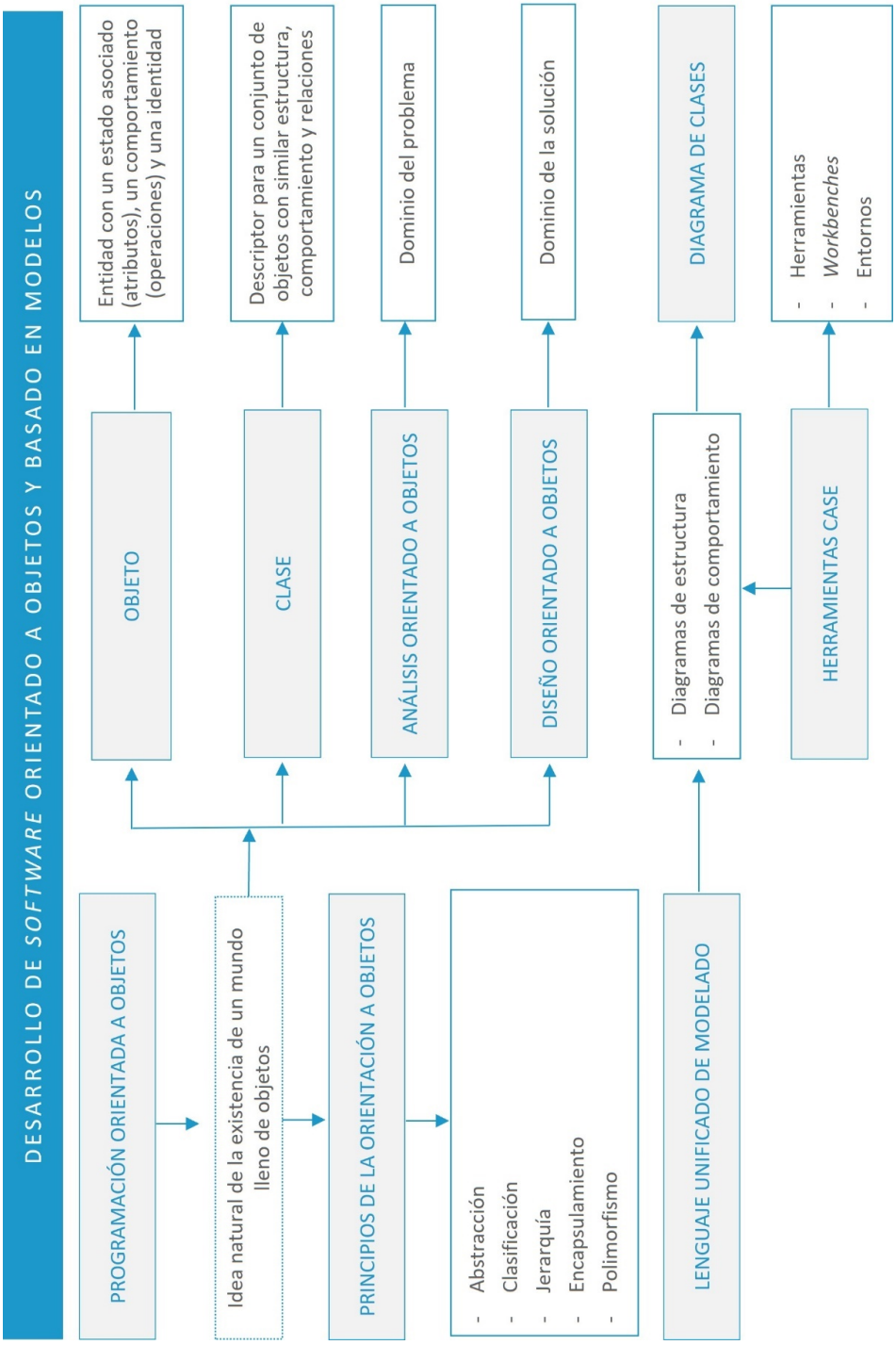
Metodologías, Desarrollo y Calidad en la Ingeniería de Software

Tema 3. Desarrollo de software orientado a objetos y basado en modelos

Índice

Esquema



© Universidad Internacional de La Rioja (UNIR)

3.1. Introducción y objetivos

En este tema se estudian y consolidan conceptos asociados a la **orientación a objetos,** con sus **principios fundamentales:** abstracción, jerarquía, clasificación, encapsulamiento y polimorfismo. Además, se analiza la necesidad de diferenciar entre análisis orientado a objetos y diseño orientado a objetos, y la dificultad que ello supone.

Junto a esto, se estudia el lenguaje de modelado por excelencia, **UML,** con sus características principales. Se realiza un estudio de los diagramas de clases con sus tipos de relaciones más comunes y se analiza el papel que juegan las **herramientas CASE** a lo largo del proceso de desarrollo de *software.*

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

- Consolidar el concepto del paradigma de programación orientado a objetos.
- Estudiar los principios fundamentales de la orientación a objetos.
- Comprender los conceptos de clase y objeto.
- Estudiar y diferenciar entre el análisis y el diseño de la orientación a objetos.
- Estudiar el lenguaje de modelado unificado.
- Afianzar los conceptos principales de los diagramas de clases. Conocer la utilidad de trabajar con herramientas CASE durante el desarrollo de *software.*

© Universidad Internacional de La Rioja (UNIR)

3.2. Programación orientada a objetos

Aunque el concepto de programación orientada a objetos (POO) apareció por primera vez con **Simula** (Dahl y Nygaard, 1967), fue realmente con el lenguaje de programación **Smalltalk** (Goldberg y Kay, 1976) cuando se consolidó de manera definitiva.

POO se basa en la idea natural de la existencia de un mundo lleno de objetos, con características que los diferencian (atributos) y con un conjunto de acciones propias que pueden realizarse sobre ellos (operaciones).

Los atributos permiten representar y almacenar datos, y las operaciones permiten manipular los atributos. De esta manera, tal y como se afirman Shlaer y Mellor (1992) «[...] los **programas orientados a objetos** se construyen en base a un conjunto de objetos que colaboran entre sí mediante el intercambio de mensajes».

Booch *et al.* (2007), definen la programación orientada a objetos como:

> «[…] un método de implementación en el que los programas se organizan como una colección de objetos que colaboran entre sí, donde cada objeto representa una instancia de una clase determinada, y donde las clases forman parte, todas ellas, de una jerarquía de clases relacionadas a través de la herencia» (p. 41).

La gran diferencia entre la programación orientada a objetos frente a la programación estructurada es el hecho de que la programación orientada a objetos hace de los objetos, y no de los algoritmos (basados en procedimientos y funciones), los **elementos principales** para el desarrollo del *software,* donde cada objeto ha de ser instancia de alguna clase. Con la programación orientada a objetos se consigue agrupar los elementos del código que tienen funcionalidades similares en un concepto unificado que permitirá acceder y modificar esos elementos.

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

Los lenguajes de programación orientados a objetos (LPOO) tuvieron su punto álgido durante los años ochenta, con la aparición de **lenguajes** como *C++*, *Objective C*, *Self*, y *Eiffel*. En las últimas décadas, **Java** se ha constituido como uno de los lenguajes de programación orientado a objetos más extendido. De acuerdo a Booch *et al.* (2007), un lenguaje de programación será orientado a objetos si cumple las siguientes condiciones:

- Los objetos son abstracciones de datos con una interfaz que contiene los nombres de las operaciones disponibles y tienen el estado oculto.
- Los objetos tienen un tipo asociado (clases).
- Las clases pueden heredar atributos de las superclases.

Fue precisamente en los años ochenta cuando los conceptos introducidos por los LPOO fueron generalizados dentro de los principios de la ingeniería del *software.* El término conocido como **diseño orientado a objetos** fue utilizado por primera vez por Booch (Booch, 1981; Booch, 1982) y, posteriormente, derivó en el paradigma conocido como **desarrollo orientado a objetos** (Booch, 1986).

El desarrollo de *software* orientado a objetos aboga por el uso de objetos durante todo el **ciclo de vida** del proyecto *software.* De este modo, pero con ciertos matices, los objetos identificados en el dominio del problema (análisis) jugarán un papel fundamental en el dominio de la solución (diseño). Así, el análisis orientado a objetos (AOO) se utiliza para **definir los objetos de análisis** que representan el dominio del problema.

De manera simplificada, se puede decir que el **dominio del problema** es una descripción reutilizable del problema a resolver y su contexto. Por su parte, el diseño orientado a objetos (DOO) propone una solución al problema a través de la creación y combinación de objetos de diseño.

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

L a **orientación a objetos** (OO) parece ser un buen enfoque cuando hay que representar características del mundo real, ya que se puede asumir que el mundo real está formado por un conjunto de entidades conceptuales cuyas instancias colaboran entre sí y tienen unas características estructurales y de comportamiento que las definen.

Si se fuese capaz de identificar objetos, pero no una colaboración entre ellos, el enfoque de la orientación a objetos no tendría sentido. Solo cuando haya **colaboración** es cuando la orientación a objetos puede ser útil. La orientación a objetos no tiene por qué ser siempre la mejor forma de enfrentarse a un problema que requiere una solución *software.*

Entre las ventajas que puede ofrecer el enfoque orientado a objetos se encuentran las que se indican a continuación:

- **Análisis.** Favorece el análisis del dominio (es decir, análisis del problema y su solución) a un mayor nivel de abstracción.
- **Comunicación.** Favorece la comunicación entre los desarrolladores y los usuarios del *software* a implementar, ya que se realizan abstracciones del mundo real más fáciles de entender.
- **Tolerancia.** La orientación a objetos tiene mejor tolerancia a cambios que el enfoque estructurado, ya que una modificación en alguna funcionalidad no implica el tener que recorrer un gran número de funciones o procedimientos que no estarán relacionados por un nexo común.
- **Reutilización.** Favorece la reutilización de las clases definidas. Las clases tienen entidad propia, por lo que un cambio en una clase no debería, en principio, si está bien pensada, afectar al resto de clases, por lo que serán más fáciles de reutilizar.

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

- **Verificación.** Favorece el proceso de comprobación de su conformidad con los requisitos definidos.
- **Extensibilidad.** Favorece la extensión de funcionalidad en el alcance del proyecto.

© Universidad Internacional de La Rioja (UNIR)

3.3. Principios de la orientación a objetos

Dentro de la orientación a objetos se definen una serie de **principios** que constituyen la base en la que se fundamenta. Los principios fundamentales para OO son los que se describen a continuación (Booch *et al.,* 2007):

Abstracción

La abstracción constituye el mecanismo por el que las personas se enfrentan a la complejidad. En el contexto de la OO, Booch *et al.* (2007) definen la abstracción como (ver Figura 1):

> «Aquello que denota las características esenciales de un objeto, que le distingue de otros objetos ignorando los detalles que no son importantes desde un determinado punto de vista» (p. 45).

Figura 1. El nivel de abstracción varía dependiendo de cada punto de vista. Fuente: Booch, Maksimchuk, Engle, Young, Conallen y Houston, 2007.

Para un desarrollador, se podría decir que la abstracción es una forma de **simplificar la realidad** (dominio del problema y de su solución), en la que se queda con los aspectos relevantes y prescinde de todos aquellos que no son de interés para el

© Universidad Internacional de La Rioja (UNIR)

sistema a implementar. De acuerdo a Booch et al. (2007, p. 45), existen varios tipos de abstracciones a la hora de desarrollar un sistema (ordenados de mayor a menor relevancia):

- **Abstracción de entidad.** Un objeto que representa un modelo útil de una entidad del dominio del problema o del dominio de la solución.
- **Abstracción de acción.** Un objeto que proporciona un conjunto generalizado de operaciones con un objetivo común.
- **Abstracción de máquina virtual.** Un objeto que agrupa operaciones que son utilizadas por algún nivel de control superior u operaciones que utilizan algún conjunto de operaciones auxiliares.
- **Abstracción casual.** Un objeto que engloba un conjunto de operaciones que no tienen relación entre sí.

Clasificación

La clasificación es el principio de OO que **determina la relación entre las clases,** donde los objetos estarán clasificados en clases, es decir, la clase será su tipo, tal y como se ilustra en la Figura 2. Booch *et al.* (2007), definen la clasificación como:

> «[…] la forma en la que a un objeto se le asigna un tipo, de tal forma que, objetos de diferente tipo no se podrán intercambiar, o en caso de que se puedan intercambiar, lo harán de manera muy restrictiva» (p. 65).

© Universidad Internacional de La Rioja (UNIR)

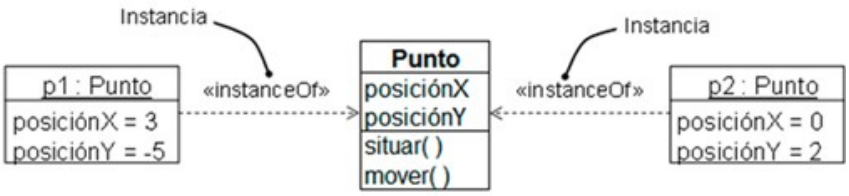


Figura 2. Principio de clasificación.

En el dominio de la clasificación se reconocen dos formas diferentes de clasificar objetos:

- **Fuerte vs débil.** Esta clasificación se refiere a la consistencia del tipo, es decir, la clasificación será fuerte cuando no se deje invocar la operación de un objeto, a menos que la signatura de esa operación esté definida en la clase o en alguna superclase de dicho objeto.

- **Estática vs dinámica.** Esta clasificación se refiere al momento en el que los nombres son ligados a los tipos. La **clasificación estática** (conocida también en inglés como *static binding* o *early binding*) significa que los tipos de todas las variables y expresiones se fijan en tiempo de compilación, mientras que la **clasificación dinámica** (conocida en inglés como *late binding*) permite que los tipos de las variables y expresiones no se conozcan hasta el tiempo de ejecución.

> Un ejemplo de lenguaje de programación estático y fuertemente tipado sería Ada, mientras que C++ y Java entrarían en la categoría de lenguajes de programación fuertemente tipados, pero dinámicos. En el otro extremo se encontraría Smalltalk, sin tipado y con clasificación dinámica.

© Universidad Internacional de La Rioja (UNIR)

Jerarquía

La jerarquía es el principio de OO que **permite ordenar las abstracciones** (*Booch et al.,* 2007, p. 58). La **herencia** sería un ejemplo de jerarquía que se da entre las clases y denota la relación «es-un/a», como, por ejemplo, un empleado «es-una» persona. El mecanismo de la herencia es el que va a permitir crear clases nuevas a partir de clases ya existentes.

Que una clase herede de otra clase quiere decir que se **reutiliza la definición** de la clase de la que se hereda en lo que se refiere a atributos, operaciones y relaciones que pueda tener con otras clases. La subclase sería la clase que hereda de otra clase, siendo ésta la superclase para dicha clase. Según el ejemplo anterior, **empleado** sería la subclase (es decir, hereda) de la clase **persona,** la cual representaría a la superclase.

De esta manera, la subclase hereda la estructura y el comportamiento de la superclase. Una clase puede heredar de una o varias clases, las limitaciones en el diseño del sistema *software* se podrán tener a la hora de elegir el lenguaje de programación, ya que Java, por ejemplo, no soporta la herencia múltiple. Estas características asociadas a la herencia son las que favorecerán la **reutilización de código** y que no haya que volver a codificar lo mismo por cada nueva aplicación que se tenga que implementar. La Figura 3 ilustra el principio de jerarquía para el enfoque OO.

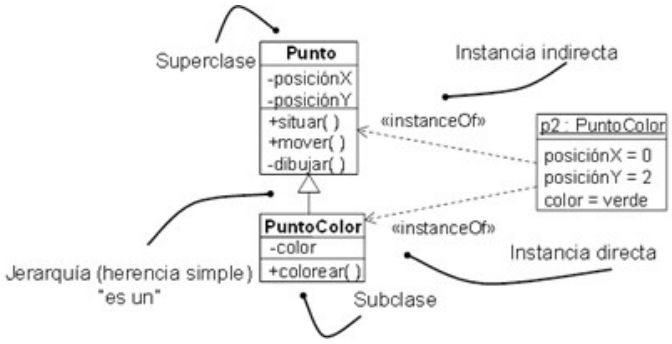


Figura 3. Principio de jerarquía. Fuente: elaboración propia.

© Universidad Internacional de La Rioja (UNIR)

Encapsulamiento

El encapsulamiento es el principio de OO que **permite separar** la interfaz de la implementación en una clase. El objetivo del encapsulamiento es conseguir reducir el acoplamiento entre las clases. La idea es que la comunicación entre clases solo se realice a través del intercambio de mensajes y, para ello, los atributos, así como aquellas operaciones que se consideren solo de uso interno, se definirán como **privados,** mientras que aquellas operaciones que se consideren parte de la interfaz nativa de la clase (sus responsabilidades) son las que se definirán como **públicas.**

La interfaz de la clase **encapsula el conocimiento** del que dicha clase es responsable, de tal forma que, si se producen cambios en la clase sin que afecte a la interfaz, el resto del sistema no se verá afectado por dichos cambios (Stevens y Pooley, 2000, p. 8). La Figura 4 ilustra el principio de encapsulamiento para el enfoque OO.

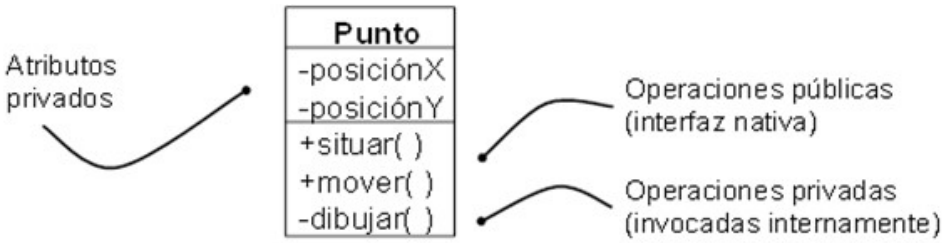


Figura 4. Principio de encapsulamiento. Fuente: elaboración propia.

Polimorfismo

El polimorfismo es el principio de OO que permite a los objetos actuar de distinta manera ante el mismo mensaje. En este caso, las subclases pueden **redefinir el comportamiento** de una operación heredada, lo que hará que dependiendo del tipo de objeto la respuesta a un mismo mensaje podrá tener un comportamiento u otro. Con lo cual, a través del polimorfismo (operaciones polimórficas), para un mismo nombre de operación se pueden designar **implementaciones distintas.**

© Universidad Internacional de La Rioja (UNIR)

Las operaciones que no sean polimórficas serán consideradas **operaciones concretas** (Rumbaugh, Jacobson y Booch, 2007). Desde un punto de vista más técnico, el polimorfismo permite que toda referencia a un objeto que pertenece a una determinada clase (superclase), tome la forma de una referencia a un objeto de una clase que hereda de la anterior (subclase). La Figura 5 ilustra el principio de polimorfismo para el enfoque OO.

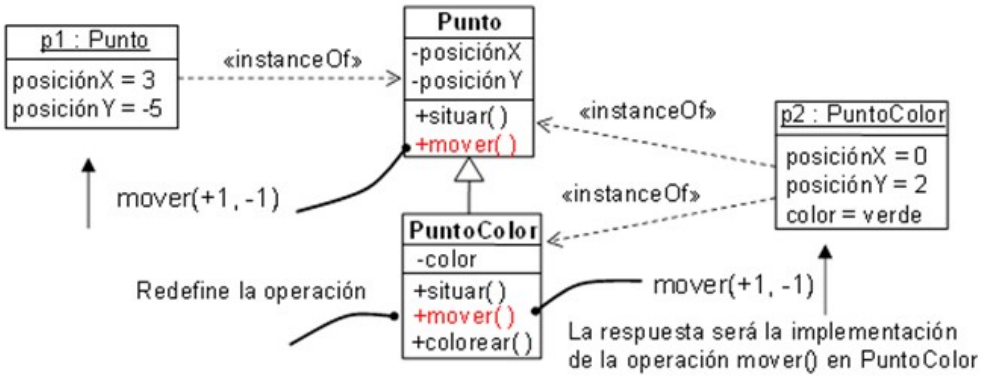


Figura 5. Principio de polimorfismo. Fuente: elaboración propia.

© Universidad Internacional de La Rioja (UNIR)

3.4. Objeto y clase

De acuerdo con la definición de **Rumbaugh** *et al.* (2007), un **objeto** es «una entidad discreta con identidad, estado y comportamiento que se puede invocar. Los objetos son las piezas individuales de un sistema en tiempo de ejecución» (p. 44).

Un poco más elaborada es la definición de **Sommerville** (2005), que define un objeto como:

> «[…] Una entidad que tiene un estado y un conjunto de operaciones definidas que operan sobre ese estado. El estado se representa como un conjunto de atributos del objeto. Las operaciones asociadas al objeto proveen servicios a otros objetos (clientes) que solicitan estos servicios cuando se requiere llevar a cabo algún cálculo» (p. 165).

Bajo el principio de abstracción, un objeto se podría definir como la abstracción de una **entidad real o conceptual,** en la que se seleccionan los aspectos relevantes para el sistema en estudio. Desde un punto de vista de programación, el objeto representaría el **código** compuesto de propiedades y métodos que se podrán manipular de manera independiente.

Por tanto, de todas las definiciones anteriores se puede deducir que un objeto será una entidad del mundo real que tendrá asociado un estado (atributos), un comportamiento (operaciones) y una identidad que le diferenciará del resto de objetos, independientemente de los valores de los atributos que pueda contener cada objeto (es decir, aunque los valores de los atributos sean iguales en objetos distintos, la identidad que los define es lo que los hará independientes).

© Universidad Internacional de La Rioja (UNIR)

Así, por ejemplo, Juan, Ana y María son instancias (es decir, objetos) de la clase persona:

- **Identidad.** La identidad representa la existencia única del objeto en el tiempo y en el espacio. Se deberá determinar qué información del objeto es la que le define y le proporciona una identidad única (Arlow y Neustadt, 2005, p. 127).

- **Estado.** El estado de un objeto se determina a través de los valores de los atributos del objeto y las relaciones que mantiene con otros objetos en un momento determinado (Arlow y Neustadt, 2005, p. 127).

- **Comportamiento.** El comportamiento define las capacidades que el objeto puede ofrecer. Por ejemplo, para el caso de la impresora anterior, se podrían definir, entre otras, las siguientes capacidades: encender, apagar, imprimir documento y cargar página. Las capacidades se definen en una clase en forma de operaciones. Cuando se invoca una operación de un objeto normalmente cambiarán los valores de sus atributos, así como las asociaciones que pueda mantener con otros objetos, lo que, en algunos casos, podrá provocar un cambio de estado del objeto (Arlow y Neustadt, 2005, p. 127).

Por otro lado, de acuerdo a la definición de Rumbaugh, Jacobson y Booch (2007), una clase es el descriptor para un conjunto de objetos con similar estructura, comportamiento y relaciones.

Con lo cual, cuando se describe una clase, se está describiendo a **todos los objetos** que pertenecen a ella. Así, todos los objetos se clasificarán y agruparán en clases, de tal forma que un objeto siempre será una instancia de una clase (es decir, la clase es su tipo).

Bajo el principio de abstracción, una clase se podría definir a dos niveles:

- **Primer nivel.** En un primer nivel, la clase se podría definir como representación de entidades concretas (objetos). Por ejemplo, la clase «perro», donde los objetos que

© Universidad Internacional de La Rioja (UNIR)

serían las entidades concretas, tangibles, podrían ser los perros: Fido, Rex, Luna y Anubis.

- **Segundo nivel.** En un segundo nivel, la clase se podría definir como representación de conjuntos de entidades (clases). Por ejemplo, la clase «raza perruna», donde los objetos representan a un conjunto de entidades (no es algo tangible en este caso), como podrían ser las razas de perro: pastor alemán, bóxer, boston terrier y dálmata.

Desde un punto de vista de programación, la clase representaría el **conjunto de especificaciones** que definen cómo se va a crear un objeto de dicha clase. La clase sería como la **plantilla de creación** de objetos que ha de incluir las declaraciones de todos los atributos y operaciones asociados con un objeto de dicha clase (Sommerville, 2005, p. 288).

Además, los objetos de una clase se podrán comunicar con objetos que pertenezcan a otras clases (es decir, podrán enviar mensajes a otros objetos) mediante el establecimiento de **asociaciones** entre las clases. Una asociación es la especificación de un conjunto de conexiones entre las instancias de las clases que están asociadas. Las asociaciones entre clases representan la estructura y posibilidades de comunicación del sistema.

En definitiva, todas las entidades que tengan características comunes se abstraen en una única entidad (clase), que contendrá toda la información relevante para el sistema en estudio. Así, todos los objetos que sean instancias de una clase compartirán los mismos atributos, operaciones, relaciones y semántica.

© Universidad Internacional de La Rioja (UNIR)

3.5. Análisis y diseño orientado a objetos

Una gran ventaja de la OO es el hecho de que se puede **aplicar la misma notación** para representar los objetos, así como sus relaciones, tanto en la fase de análisis como en la fase de diseño de un proyecto *software.* El análisis va a permitir desarrollar el sistema correcto, mientras que el diseño permitirá desarrollar el sistema de la manera correcta (Larman, 2005, p. 7).

Durante el AOO se hace especial énfasis en **localizar y describir** los objetos o conceptos del dominio del problema. Por ejemplo, para el caso del *software* que gestiona los vuelos en un aeropuerto, algunos de los conceptos serían: avión, vuelo y piloto.

De acuerdo a Arlow y Neustadt (2005, p. 158), las clases de análisis:

- Representan una abstracción a muy alto nivel del dominio del problema.
- Deberían corresponderse con los conceptos del negocio del mundo real y ser nombrados, por tanto, en concordancia, sin ningún tipo de ambigüedad ni inconsistencia.

En la **fase de análisis,** las clases deberían definir los atributos a muy alto nivel, sin entrar en gran detalle, incluyendo la información más relevante que se identifica a primera vista. Arlow & Neustadt (2005, p. 159-160) definen lo mínimo que debería definir una clase de análisis:

- **Nombre.** En una clase de análisis el nombre siempre ha de aparecer.
- **Atributos.** La clasificación de los atributos es opcional a este nivel.
- **Operaciones.** Las operaciones han de recoger a muy alto nivel las responsabilidades que se consideran para una clase. Definir los parámetros de las operaciones y el tipo de valor que puede devolver una operación sería opcional a

© Universidad Internacional de La Rioja (UNIR)

este nivel.

- **Visibilidad.** A este nivel, la visibilidad de atributos y operaciones no se suele mostrar, aunque para mantener el principio de encapsulamiento, todos los atributos deberían ser definidos como privados.

- **Estereotipos.** Los estereotipos se podrán mostrar a este nivel si realmente mejoran la comprensión del dominio del problema.

- **Valores etiquetados.** Al igual que los estereotipos, solo se deberán mostrar si realmente mejoran la comprensión del dominio del problema.

Según Pressman (2010, p. 143), una técnica para identificar las clases de análisis consiste en subrayar, en el documento o modelo de requisitos, cada **sustantivo o frase** que incluya lo que se considera que debería ser una clase e introducirlo en una tabla (también se deberán anotar los sinónimos). Solo se marcará aquello que se considere que forma parte del dominio del problema y no del dominio de la solución (ya que estas clases formarían parte de la fase de diseño). Para Pressman (2010, p. 143), las clases de análisis podrán adoptar, entre otras, las siguientes formas:

- **Entidades externas** que van a producir o harán uso de la información que se obtiene del *software* a desarrollar, como, por ejemplo, sistemas externos, dispositivos y personas.

- **Cosas** que forman parte de dominio de información que proporciona o de la que se alimenta el sistema, como, por ejemplo, informes, pantallas y señales.

- **Ocurrencias o eventos** que se van a producir dentro del contexto del comportamiento del sistema, como, por ejemplo, una trasferencia bancaria o el movimiento de un robot.

- **Roles** que desempeñan los actores que interactúan con el *software* a desarrollar, como, por ejemplo, un cliente y un vendedor.

© Universidad Internacional de La Rioja (UNIR)

- **Unidades organizacionales** que son de interés para el *software* a desarrollar, como, por ejemplo, una división, un departamento y un equipo.
- **Lugares** que determinan el contexto y el objetivo principal del *software* a desarrollar, como, por ejemplo, una plataforma de carga y un aeropuerto.
- **Estructuras** que definen un conjunto de objetos o las clases relacionadas a dichos objetos, como, por ejemplo, sensores, ordenadores y vehículos.

Respecto a cómo comprobar la calidad de las clases de análisis identificadas, Arlow y Neustadt (2005, p. 160) listan un conjunto de condiciones que se han de satisfacer para que una clase de análisis sea considerada **válida** en la definición del dominio del problema en un proyecto *software:*

- Su nombre ha de reflejar su intención.
- Se corresponde con una abstracción a muy alto nivel que modela un elemento específico del dominio del problema.
- Se corresponde con una característica totalmente identificable dentro del dominio del problema.
- Contiene un conjunto pequeño y bien definido de responsabilidades.
- Presenta alta cohesión.
- Presenta bajo acoplamiento.

En lo que respecta al DOO, en este caso, el énfasis se da en la **definición** de los objetos *software* que se traducirán a código en un lenguaje de programación concreto y en la forma en la que estos objetos **colaboran** entre sí para satisfacer los requisitos del sistema (Larman, 2005, p. 7).

Para las clases de diseño, existen dos formas de extraerlas, propuestas por Arlow y Neustadt (2005, p. 344):

© Universidad Internacional de La Rioja (UNIR)

- Como un refinamiento de las clases de análisis, en la que se añaden los detalles de implementación (de cara a posibles cambios, habrá que mantener la trazabilidad de cada clase de análisis a la clase o clases de diseño que describe su implementación).

- Del dominio de la solución, que proporciona las herramientas técnicas que van a permitir implementar el *software.*

La Figura 6 ilustra la **transformación** de una clase de análisis en una clase de diseño teniendo en cuenta los dos factores descritos anteriormente, donde se refina la clase de análisis y se tiene en cuenta la plataforma de desarrollo a la hora de definir la clase de diseño (para este ejemplo, el lenguaje de programación a utilizar sería Java).

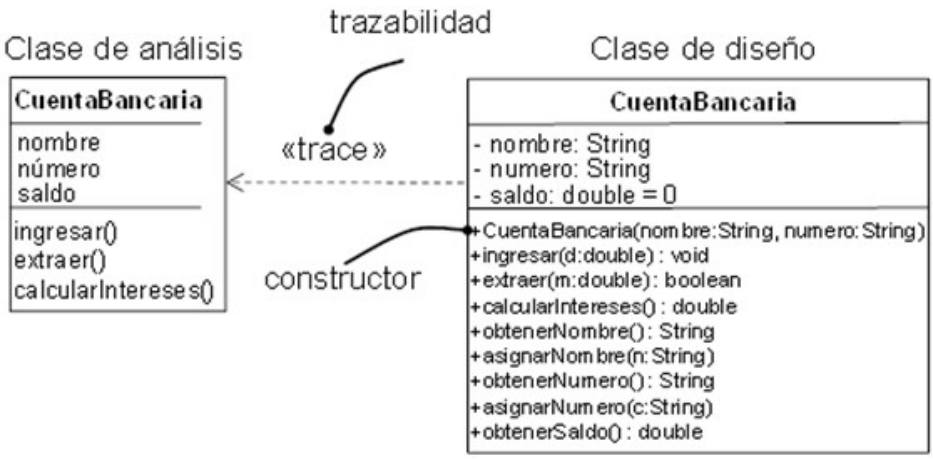


Figura 6. Refinamiento de una clase de análisis en una clase de diseño. Fuente: Arlow y Neustadt, 2005.

El **proceso de extracción** de las clases de diseño, propuesta por Arlow y Neustadt (2005, p. 345), es el que se ilustra en la Figura 7.

© Universidad Internacional de La Rioja (UNIR)

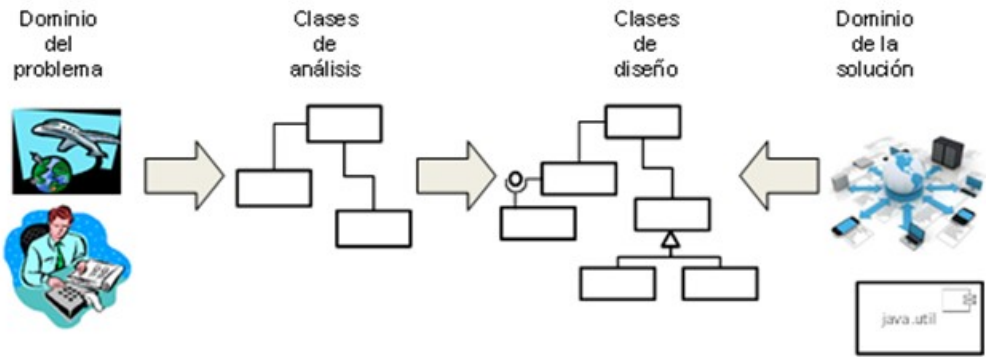


Figura 7. Extracción de las clases de diseño para un sistema software. Fuente: Arlow y Neustadt, 2005.

Por su parte, Pressman (2005, p. 196), define **cinco tipos** distintos de clases de diseño, donde cada tipo representa una capa distinta de la arquitectura elegida para el diseño del *software* a desarrollar:

- **Clases de la interfaz de usuario** que definen todas las abstracciones necesarias para la interacción hombre-máquina, o «interfaz de usuario» como también se le denomina (*Human-Machine Interface,* HMI).
- **Clases del dominio del negocio** que se suelen corresponder con el refinamiento de las clases de análisis para su implementación.
- **Clases de proceso** que implementan las abstracciones del negocio a más bajo nivel de tal forma que se posibilite su gestión completa.
- **Clases persistentes** que representan los repositorios de datos, como, por ejemplo, las bases de datos, que seguirán existiendo, aunque el sistema *software* no esté en ejecución.
- **Clases del sistema** que implementan las funciones de administración y control del *software,* que permitirán que la aplicación opere y se comunique dentro y fuera de la frontera del sistema.

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

Tal y como se destaca en Pressman (2005, p. 196), a medida que se va creando la arquitectura el nivel de abstracción se va reduciendo y las clases de análisis del dominio del problema se transforman en clases de diseño que permitirán su implementación.

Mientras que la terminología empleada en las clases de análisis es **propia del modelo de negocio** que describe el problema a resolver, la terminología empleada para las clases de diseño está ligada a **detalles más técnicos,** vinculados a la plataforma de desarrollo, que describen cómo se ha de implementar el sistema *software.*

El hecho de plantearse el mundo real, o el dominio del negocio, como un conjunto de objetos que colaboran entre sí para elaborar las clases de análisis ha dado lugar a que haya autores que pongan en tela de juicio si esa propuesta que se está haciendo del sistema, considerada como parte integrante de la fase de análisis, no debería ser ya considerada como **parte del diseño** (la orientación a objetos vista así como una tecnología de desarrollo más).

Hay (1999), es uno de los autores que pone en duda que el dominio del negocio solo puede ser visto y tratado exclusivamente desde un **punto de vista de análisis.** Argumenta la dificultad de algunos eruditos de OO para poder explicar conceptos utilizados en análisis, como el concepto de herencia, sin utilizar código en algún lenguaje de programación, lo cual imposibilita su entendimiento para gente que solo sea experta en el dominio del negocio y no sepa programar.

Esto es consecuencia de las diferentes vistas que se tiene, para un mismo sistema, por parte diferentes personas, lo que complica la comunicación y que puede impedir que el análisis orientado a objetos se vea como un análisis del sistema en estado puro.

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

Zachman (1987), dentro de su marco de trabajo, clasifica las **diferentes vistas** o perspectivas que se van a dar en cualquier proyecto de desarrollo de *software* para entender esta problemática:

- **Alcance.** Esta vista se refiere a entender los objetivos de la organización, cómo es con respecto a otras compañías del mismo negocio y qué es lo que la hace diferente.
- **Vista del propietario del negocio.** Esta vista hace referencia, no a los *stakeholders,* sino a las personas que operan el negocio. Estas personas utilizan un vocabulario determinado y son quienes realmente saben cómo funciona el negocio.
- **Vista del arquitecto.** Esta vista se refiere al reconocimiento de que se dan estructuras fundamentales y tecnológicamente independientes, y a la representación del negocio a través de estas estructuras.
- **Vista del diseñador.** Esta vista se refiere a la aplicación de la tecnología para poder gestionar los requisitos del sistema que se han descubierto en la vista del arquitecto.
- **Vista del desarrollador.** Esta vista se refiere a la parte interna de los programas y la tecnología. El desarrollador conoce a la perfección el lenguaje de programación que se va a utilizar, las tecnologías de las comunicaciones, etc.
- **Vista de producción.** La vista del sistema completo, ya finalizado.

© Universidad Internacional de La Rioja (UNIR)

3.6. Lenguaje unificado de modelado

E l **lenguaje unificado de modelado** (*Unified Modeling Language*, UML) (OMG, 2011, 2017) 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 & 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** (UP), que utiliza UML como la notación de su modelado visual. UML se puede utilizar con todas las **metodologías o ciclos de vida** existentes, 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 setenta y finales de los ochenta. 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 noventa, 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.

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

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** *(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

© Universidad Internacional de La Rioja (UNIR)

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 8 resume la historia de los lenguajes de modelado hasta la versión actual más estable de UML (2.5.1, que es una versión con mínimos cambios respecto a la 2.4.1) (OMG, 2017).

© Universidad Internacional de La Rioja (UNIR)

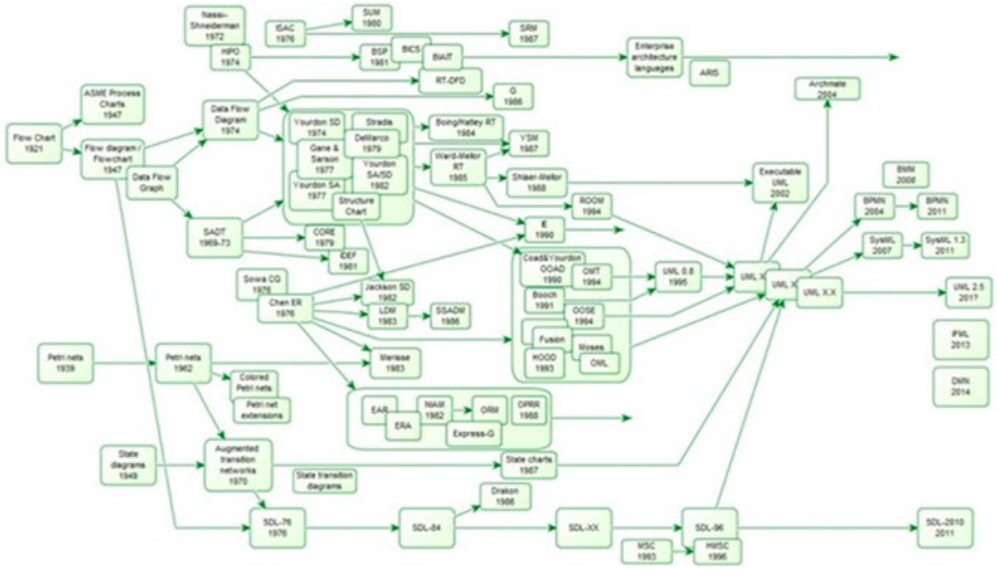


Figura 8. Evolución de los lenguajes de modelado. Fuente: Cabot, 2014.

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** (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.

© Universidad Internacional de La Rioja (UNIR)

- ***Blueprint.*** Aquí se clasificarían los diagramas de diseño más o menos detallados utilizados para:
  - Ingeniería inversa, que permitirá visualizar y comprender mejor el código ya existente.
  - 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 automática y normalmente, no será visto ni modificado por los desarrolladores, que solamente trabajarán con UML como si de un lenguaje de programación se tratase.

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 indican 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 (ver Figura 9).

© Universidad Internacional de La Rioja (UNIR)

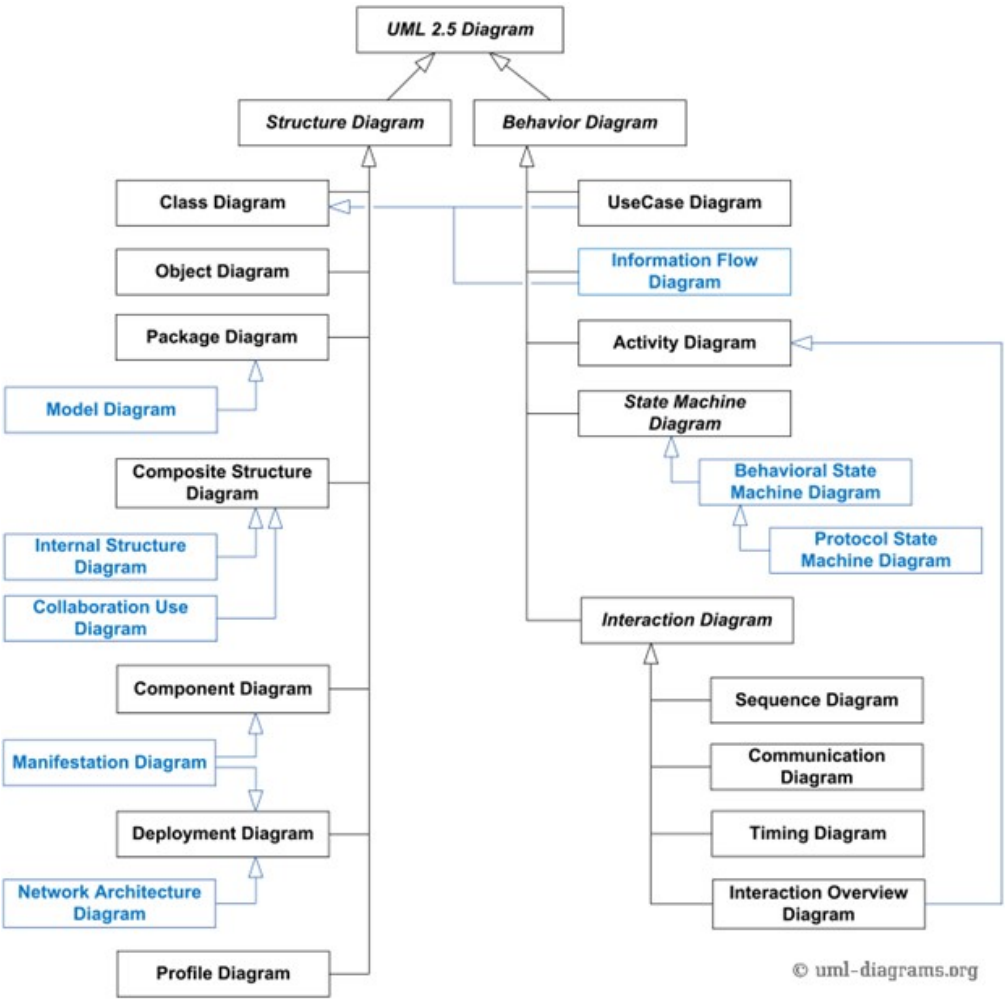


Figura 9. Clasificación de diagramas en UML 2.5.1. Fuente: uml-diagrams.org, s. f.

Erickson y Siau (2007), realizaron un estudio en el que concluyeron que la mayoría de los usuarios de UML consideran como fundamentales los siguientes **cinco tipos** de diagrama:

- **Diagramas de actividad,** que muestran actividades realizadas en un proceso.
- **Diagramas de casos de uso,** que muestran las interacciones entre el sistema y actores de su entorno.
- **Diagramas de secuencia,** que muestran interacciones entre los actores y distintos componentes del sistema.

© Universidad Internacional de La Rioja (UNIR)

- **Diagramas de clases,** que muestran clases de objetos del sistema y sus relaciones.
- **Diagramas de estado,** que muestran el comportamiento y reacciones del sistema ante eventos internos y externos.

Génova, Valiente y Nubiola (2005) definen un modelo UML 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» (p. 4).

Con lo cual, se puede decir que UML va a estar definido por **tres elementos:**

- Sintaxis abstracta (metamodelo).
- Sintaxis concreta (notación).
- Semántica.

El **metamodelo** de UML (OMG, 2011) es un modelo UML que define un lenguaje de modelado para explicar la generación de modelos UML válidos. La Figura 10 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.

© Universidad Internacional de La Rioja (UNIR)

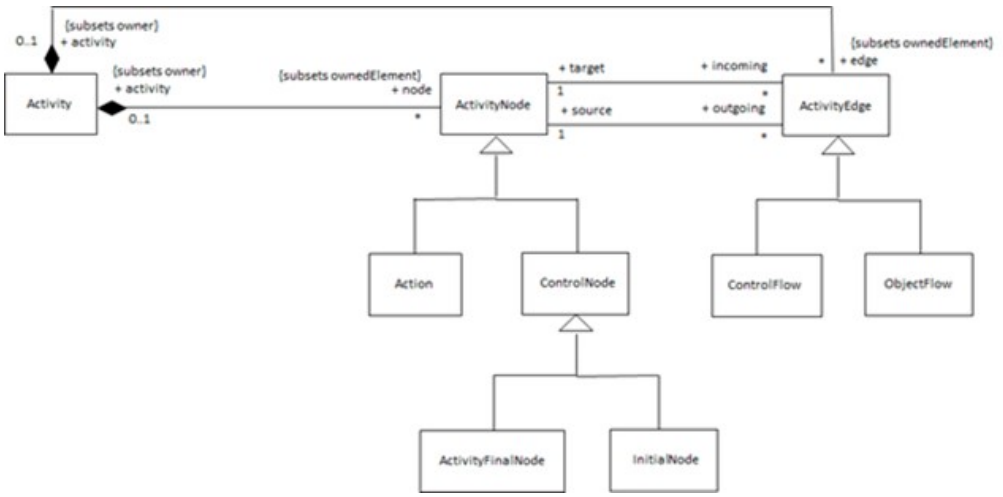


Figura 10. Ejemplo de sintaxis abstracta para el modelo de actividad. Fuente: OMG (2011).

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 11 muestra un ejemplo de sintaxis concreta para algunos elementos que pueden aparecer en los diagramas de actividad.

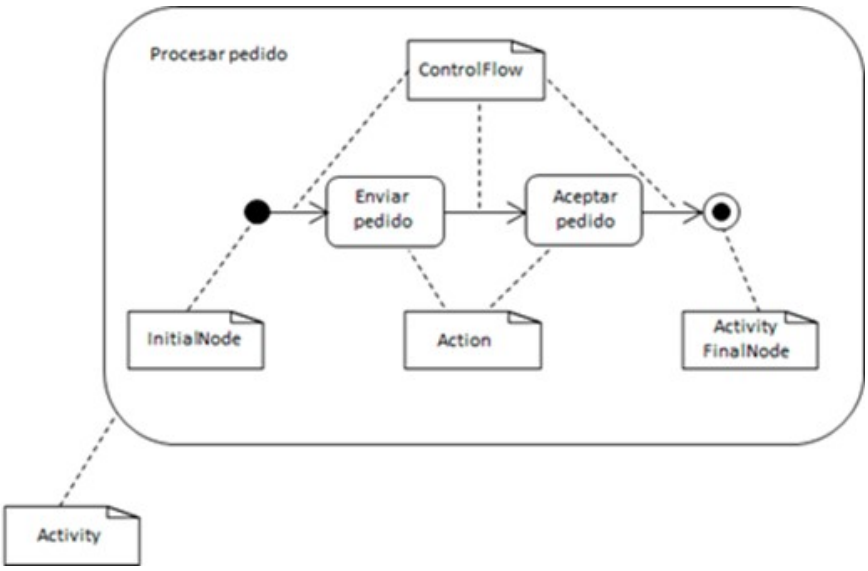


Figura 11. Ejemplo de sintaxis concreta para un diagrama de actividad. Fuente: OMG, 2011.

© Universidad Internacional de La Rioja (UNIR)

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, 2014). Una restricción se presenta como una cadena de caracteres entre llaves junto al elemento asociado. La Figura 12 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.

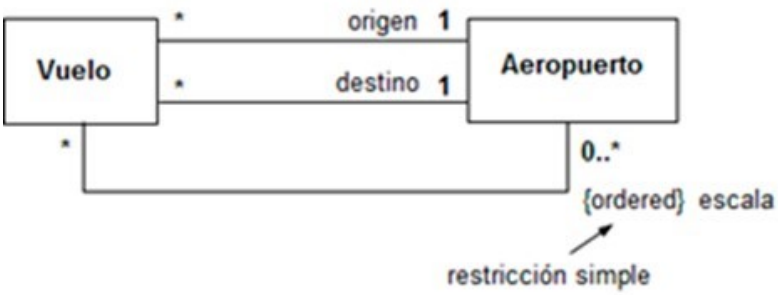


Figura 12. Restricciones en UML. Fuente: OMG, 2014.

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

© Universidad Internacional de La Rioja (UNIR)

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.

© Universidad Internacional de La Rioja (UNIR)

3.7. Diagrama de clases

Las clases, son la **base de la POO.** Son la forma de representar las características comunes de elementos que van a formar parte de un *software.*

Para modelar las clases con sus atributos y operaciones, y las relaciones con otras clases, se utilizan los **diagramas de clases de UML** (Rumbaugh, Jacobson y Booch, 2007).

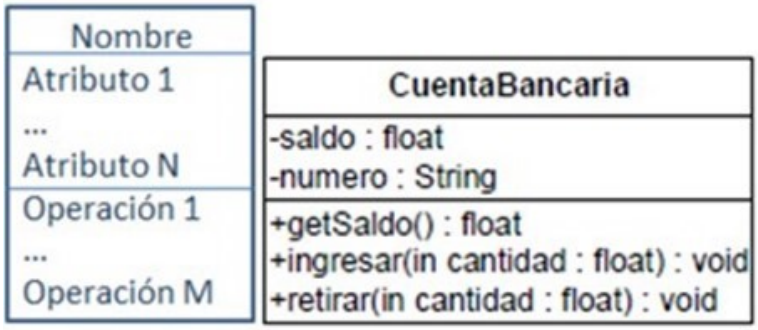


Figura 13. Representación de una clase. Fuente: Minguillón, s. f.

Un diagrama de clases recoge las clases de objetos y sus asociaciones, representando la **estructura** (estática) y el **comportamiento** de cada uno de los objetos del *software* y sus relaciones con los demás objetos, sin mostrar información temporal (Consejo Superior de Informática, 2001).

No existe un método o proceso que nos permita detectar las clases necesarias de un diagrama de clases, pero sí se pueden tener en cuenta una serie de cuestiones (Minguillón, s.f.):

- Respecto al nombre de la clase, hay que tener en cuenta que debe ser un sustantivo en singular y debe de ser significativo, simple, tener intención y tener relación con su semántica.

- Los atributos son las propiedades de las clases y para cada uno se puede indicar su

© Universidad Internacional de La Rioja (UNIR)

nombre, tipo y **visibilidad.** El nombre de los atributos debe ser significativo; el tipo puede ser un tipo de datos común o cualquier clase, el nombre y el tipo vienen separados por el símbolo «:»; y la visibilidad nos indica como pueden relacionarse las operaciones de otras clases con el atributo

- **Público («+»):** puede relacionarse con otras clases. En OO los atributos por defecto no suelen ser públicos por el principio de encapsulamiento. Para que sean accesibles por otras clases, se suelen utilizar las operaciones `get` y `set`.
- **Protegido («#»):** puede relacionarse con las clases descendientes
- **Privado («-»):** no puede relacionarse con otras clases.

- Las operaciones o métodos representan el comportamiento de la clase. La estructura sintáctica de las operaciones es la siguiente:

```
visibilidad nombreOperacion (Parámetro, Parámetro, ...) [: tipo de retorno]
```

-
  - Al igual que los atributos tienen la misma tipología y semántica de visibilidad.
  - Una operación puede tener cero o más parámetros y tienen la siguiente estructura sintáctica:

```
{[dirección] nombreParametro: tipo [=valor por defecto]}
```

-
  - La dirección puede ser `in` (identifica qué es de entrada y no se puede modificar), `out` (identifica qué es de salida y se puede modificar) o `inout` (especifica qué es de entrada y salida).

Relaciones entre clases

Las **principales relaciones** que puede haber entre dos o más clases en OO son: generalización (herencia), asociación, agregación, composición y dependencia.

© Universidad Internacional de La Rioja (UNIR)

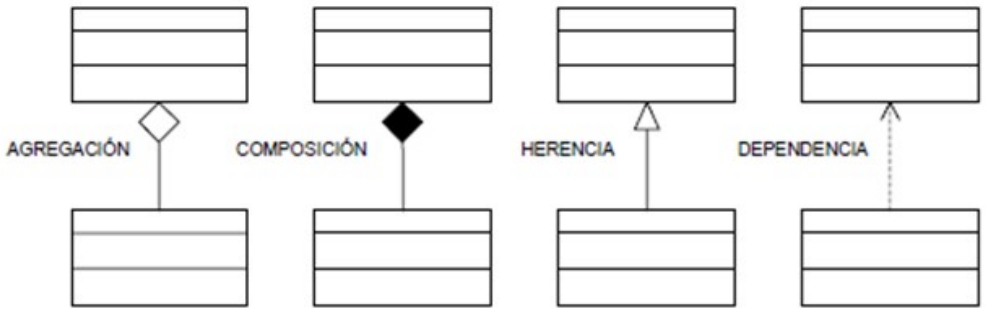


Figura 14. Algunas de las relaciones más comunes entre clases. Fuente: Consejo Superior de Informática, 2001.

Asociación

Relación más común en un diagrama de clases. Representa un **conjunto de enlaces entre objetos,** es decir, las instancias de una clase pueden relacionarse con las instancias de otras clases con las que están asociadas.

Este tipo de relación se representa mediante una **línea continua** entre las clases asociadas. Puede ser **unidireccional** o **bidireccional**. Si es unidireccional se indica con una flecha en uno de los extremos e indica que la relación es solamente en uno de los dos sentidos.

Figura 15. Relación de asociación unidireccional. Fuente: Minguillón, s. f.

Cada relación de asociación puede presentar **elementos adicionales** como la direccionalidad, los nombres de la asociación o la multiplicidad que doten de mayor detalle al tipo de relación:

- **Nombre o rol:** describe la semántica de la asociación. Debe colocarse junto al extremo de la línea que está unida a una clase, para expresar cómo esa clase hace uso de la otra clase con la que mantiene la asociación. El nombre suele

© Universidad Internacional de La Rioja (UNIR)

corresponderse con expresiones verbales presentes en las especificaciones.

- **Multiplicidad o cardinalidad:** describe la cardinalidad de la relación, es decir, especifica cuántas instancias de una clase están asociadas a una instancia de la otra clase. La multiplicidad se puede indicar con un número concreto, un rango o una colección de números. La multiplicidad puede ser uno a uno, uno a muchos y muchos a muchos.

    - **Uno a uno:** una instancia de la clase A se relaciona con una única instancia de la clase B y cada instancia de la clase B se relaciona con una única instancia de la clase A.

    - **Uno a muchos:** una instancia de la clase A se relaciona con varias instancias de la clase B, y cada instancia de la clase B se relaciona con una única instancia de la clase A.

    - **Muchos a muchos:** una instancia de la clase A se relaciona con varias instancias de la clase B, y cada instancia de la clase B se relaciona con varias instancias de la clase A.

En la Tabla 1 se muestran las posibles multiplicidades y sus notaciones. También pueden existir combinaciones de las notaciones de la tabla.

© Universidad Internacional de La Rioja (UNIR)

Notación	Descripción	Ejemplos
1	Exactamente uno	Una persona tiene un único país de origen.
0..1	Cero o uno	Un empleado tiene un responsable directo en una empresa excepto si es un director, en cuyo caso tiene cero.
0..*	Cero o más	Una persona puede no tener casa (vivir alquilado), tener una o tener más de una.
1..*	Uno o más	Un alumno de un centro está matriculado de una o más asignaturas (si no tuviera ninguna no sería alumno del centro).
Número exacto	El número exacto indicado	Un segmento tiene dos puntos asociados: el origen y el destino del segmento.

Tabla 1. Descripción de las notaciones en la relación de asociación. Fuente: elaboración propia a partir de Minguillón (s. f.).

Generalización

L a **generalización o herencia** es una relación entre una clase más general (superclase) y una clase más específica (subclase). Esto refleja relaciones «`es un tipo de`», de manera que las instancias de la subclase son un tipo específico de instancias de la superclase.

La subclase hereda las propiedades, el comportamiento y las relaciones de la superclase, a la vez que puede añadir sus propias propiedades, relaciones y comportamiento.

Esta relación se representa mediante una línea continua con una **punta triangular** hueca que apunta a la superclase.

© Universidad Internacional de La Rioja (UNIR)

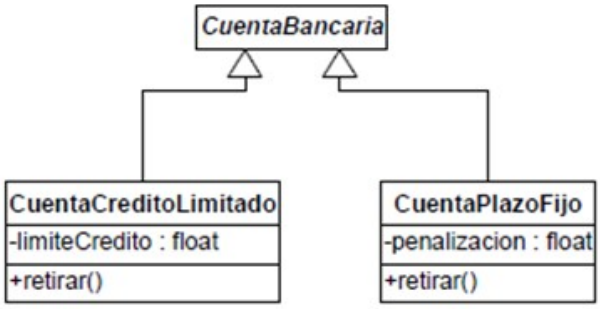


Figura 16. Ejemplo relación de herencia. Fuente: Minguillón, s. f.

Agregación

La relación de agregación es un tipo **especial de asociación.** En este tipo de relaciones se asume una subordinación conceptual del tipo «todo/parte» reflejando relaciones «son parte de», de manera que es un tipo de **relación jerárquica** entre un objeto que representa la totalidad de ese objeto y las partes que lo componen.

En este tipo de relación se puede eliminar la instancia que representa al «todo» y seguir existiendo las instancias que eran sus «partes».

Esta relación se representa mediante un **rombo hueco** en la clase cuya instancia es una agregación de las instancias de la otra, es decir, indicando la relación «todo/parteodo/parte».

Composición

La relación de composición es un **tipo de agregación** más específico en el cual se requiere que una instancia de la clase que representa a las partes esté asociada como mucho con una de las instancias que representan el «todo» (nótese que no puede haber multiplicidad «muchos» en el extremo del agregado) (Minguillón, s.f.).

Esta relación se representa mediante un **rombo relleno** en el lado de la clase que representa al «todo».

En este tipo de relación, si se elimina la instancia que representa al «todo», se

© Universidad Internacional de La Rioja (UNIR)

eliminan indirectamente el resto de las instancias que eran sus «partes».

Dependencia

La relación de dependencia entre clases denota una **relación de uso entre las mismas.** Esto quiere decir que, si la clase de la que se depende cambia, es posible que se tengan que introducir cambios en la clase dependiente.

Esta relación se representa mediante una **línea discontinua** con una flecha apuntando a la clase cliente. La relación puede tener un estereotipo que se coloca junto a la línea y entre el símbolo: «...».

Es el tipo de relación entre clases más **débil** y, como recomendación general, solo debería mostrarse en los diagramas cuando aporten alguna información importante.

© Universidad Internacional de La Rioja (UNIR)

3.8. Herramientas CASE

La tecnología **ingeniería de *software* asistida por ordenador** (*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 (*debuggers*), herramientas que facilitan la construcción del sistema *software,* etc.

De acuerdo a Sommerville (2005, p. 79-82), las herramientas CASE se pueden clasificar en distintas categorías (ver Figura 17):

- **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 (*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.

© Universidad Internacional de La Rioja (UNIR)

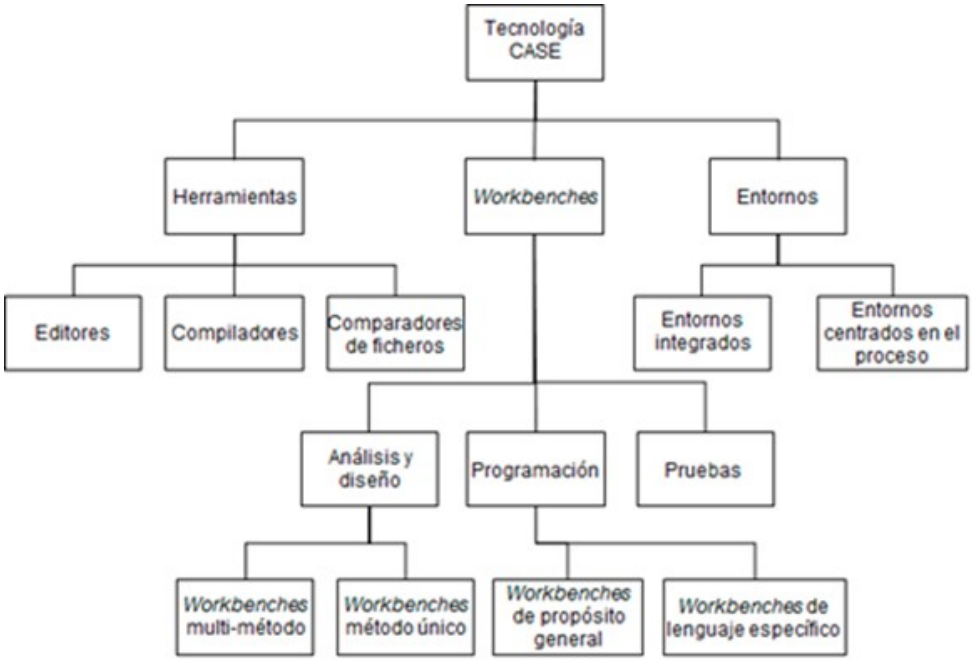


Figura 17. Tecnología CASE: herramientas, workbenches y entornos. Fuente: Sommerville, 2005.

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), donde los modelos son los que guían la construcción del *software* (no es lo mismo que el

© Universidad Internacional de La Rioja (UNIR)

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, 2005):

- Especificar un sistema *software* independientemente de la plataforma sobre la 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.

Herramientas CASE para modelado con UML

Desde la creación del lenguaje de modelado unificado a mediados de los noventa, han aparecido multitud de herramientas para modelar con UML, ya que, a pesar de que ha pasado bastante tiempo desde su primera versión, sigue siendo el **principal lenguaje** para el modelado de *software.*

Según Cabot (2019), antes de utilizar una herramienta de modelado UML, hay que ver cuál realmente te va a ser útil en tu desarrollo, ya que no hay una herramienta que sirva para todo.

Ionos España (2021), piensa de la misma manera. Seleccionar la herramienta de modelado adecuada para tu desarrollo no es una tarea fácil, a pesar de la multitud de herramientas existentes, ya que no todas disponen de las mismas **características y funcionalidades.** Unas son buenas para la generación de código, otras trabajan con cualquier tipo de diagrama, otras permiten trabajar con ingeniería inversa, otras trabajan con múltiples lenguajes de programación, etc.

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

Cómo especifica OMG, primero hay que determinar qué diagramas se quieren desarrollar y qué se quiere representar en esos diagramas, y después será el momento de seleccionar la **herramienta más adecuada** para tu proyecto de desarrollo de *software.*

A continuación, se listan algunas de las herramientas más conocidas o utilizadas, algunas de las cuales son herramientas online que no necesitan instalación (Cabot, 2017, 2019, 2021; Ionos España, 2021; Modeling Languages, 2020):

- StarUML.
- ArgoUML.
- Gliffy.
- Papyrus UML.
- Visual Paradigm Online.
- Modelio.
- MagicDraw.
- Lucidchart.
- UML Designer.
- GenMyModel.
- Draw.io.
- Creately.
- Cacoo.
- UMLet.

© Universidad Internacional de La Rioja (UNIR)

Ideas clave

- PlantUML.
- yUML.

© Universidad Internacional de La Rioja (UNIR)

3.9. Referencias bibliográficas

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

Booch, G. (1981). Describing software design in Ada. *Journal of SIGPLAN Notices, 16*(9), 42-47.

Booch, G. (1982). Object oriented design. *Journal of Ada Letters, 1*(3), pp. 64-76.

Booch, G. (1986). Object oriented development. *Journal of IEEE Transactions on Software Engineering, 12*(2), pp. 211-221.

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J. y Houston, K. A. (2007). *Object-Oriented Analysis and Design with Applications*. (3ª ed.). Addison-Wesley.

Cabot, J. (2014). *History of modeling languages in one picture (by J-P. Tolvanen)*. https://modeling-languages.com/history-modeling-languages-one-picture-j-p-tolvanen/.

Cabot, J. (2017). *Las mejores herramientas para modelar en tu navegador diagramas UML, ER y BPMN.* https://ingenieriadesoftware.es/herramientas-modelar-diagramas-uml-online-navegador/.

Cabot, J. (2019). *Las mejores herramientas UML – Edición 2019.* https://ingenieriadesoftware.es/herramientas-uml/.

Cabot, J. (2021). *25+ Herramientas para modelar programando.* https://ingenieriadesoftware.es/la-manera-mas-rapida-de-crear-diagramas-uml-10-herramientas-online-para-el-modelado-textual/.

Consejo Superior de Informática (2001). *Técnicas y prácticas*.

© Universidad Internacional de La Rioja (UNIR)

https://administracionelectronica.gob.es/pae_Home/pae_Documentacion/pae_Metodolog/pae_Metrica_v3.html#.W1IuUNgzbUI.

Dahl, O. J. y Nygaard, K. (1967). *Simula begin. Technical report*, Norsk Regnesentral.

Erickson, J. y Siau, K. (2007). Theoretical and practical complexity of modeling methods. *Communications of the ACM, 50*(8), 46-51.

Fowler, M. (2003). *UML Distilled. A Brief Guide to the Standard Object Modeling Language.* 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.), pp. 547-557.

Goldberg, A. y Kay, A. (1976). *Smalltalk 72 Instruction Manual*. Xerox PARC.

Hay, D. C. (1999). There is No Object-Oriented Analysis. *Data to Knowledge Newsletter, 27*(1).

Ionos España. (2021). *Las mejores herramientas UML.* https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/las-mejores-herramientas-uml/.

Larman, C. (2005). *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and Iterative Development*, (3ª ed.). Pearson Prentice-Hall.

Minguillón, J. (s. f.). *Introducción al lenguaje de modelado unificado*. Fundación FUOC. http://openaccess.uoc.edu/webapps/o2/bitstream/10609/9121/1/Intro_UML.pdf.

Modeling Languages. (2020). *Curated list of UML tools*. https://modeling-languages.com/uml-tools/.

© Universidad Internacional de La Rioja (UNIR)

OMG (2005). *MDA Guide Revision Draft*, Version 00.03. Document ormsc/05-11-03. https://www.omg.org/mda/specs.htm.

OMG (2011). *OMG Unified Modeling Language. About the unified modeling language specification version 2.4.1.* https://www.omg.org/spec/UML/2.4.1.

OMG (2014). *Object Constraint Language*. Version 2.4. Document formal/2014-02-03. https://www.omg.org/cgi-bin/doc?formal/2014-02-03.

OMG (2017). *OMG Unified Modeling Language. About the unified modeling language specification version 2.5.1.* https://www.omg.org/spec/UML/.

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

Pressman, R. S. (2010). *Ingeniería del software. Un enfoque práctico*, (7ª ed.). McGraw-Hill.

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

Shlaer, S y Mellor S. J. (1992). *Object LifeCycles. Modeling the World in States*. Prentice-Hall.

Sommerville, I. (2005). *Ingeniería del Software*. (7ª ed.). Pearson Addison-Wesley.

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

uml-diagrams.org. (s. f.). *UML 2.5 Diagrams Overview*. https://www.uml-diagrams.org/uml-25-diagrams.html.

Zachman, J. A. (1987). A Framework for Information Systems Architecture. I*BM Systems Journal, 26*(3).

© Universidad Internacional de La Rioja (UNIR)

Ingeniería de software un enfoque práctico

Pressman, R. S. (2010). *Ingeniería del software. Un enfoque práctico*. (7ª ed.). McGraw-Hill.

Libro de Roger S. Pressman, una persona reconocida internacionalmente en el mundo de la ingeniería de *software.* En este libro se da una visión general de los aspectos más relevantes vinculados a la ingeniería de *software.* Es un referente en el desarrollo de *software* orientado a objetos y en el modelado de *software.*

© Universidad Internacional de La Rioja (UNIR)

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 ed.). 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.

© Universidad Internacional de La Rioja (UNIR)

Diagramas UML

Página de UML Diagramas.

En esta página web se detallan las diferentes versiones del lenguaje de modelado unificado y se describen las características principales de cada una de ellas. Junto a esto, puede verse una descripción detallada de cada uno de los diagramas que podemos encontrar en UML y varios ejemplos ilustrativos de cada tipo de diagrama.

© Universidad Internacional de La Rioja (UNIR)

Ejemplos de diagramas de clases

YouTube. (s. f.). *Diagrama de clases*. https://www.youtube.com/results?search_query=diagrama+de+clases.

En la página web de YouTube hay disponibles una amplia variedad de vídeos en lo que se explica todo lo referente a los diagramas de clases con ejemplos muy ilustrativos.

© Universidad Internacional de La Rioja (UNIR)

1. En la programación orientada a objetos:

   A. Las clases heredan atributos y operaciones de los objetos.

   B. El *software* se organiza en forma de objetos, pero no siempre serán instancia de una clase.

   C. Los elementos principales son los algoritmos que definen los objetos.

   D. Las clases se organizan en jerarquías a través de la herencia.

2. ¿Qué afirmación satisface todo lenguaje orientado a objetos?

   A. Las clases son de un tipo concreto.

   B. Los objetos son de un tipo concreto.

   C. Las clases no pueden heredar atributos de sus superclases.

   D. Las clases siempre tendrán atributos y operaciones.

3. El análisis orientado a objetos (AOO):

   A. Define las clases que servirán siempre de base para el diseño orientado a objetos (DOO).

   B. Define las clases que se traducirán siempre automáticamente en una clase de diseño.

   C. En realidad no es un análisis del sistema.

   D. Ninguna de las anteriores definiciones es correcta.

4. Un sistema debería enfocarse con una orientación a objetos siempre que:

   A. Se quiera reutilizar.

   B. El mundo real en el que está basado esté formado por objetos.

   C. Haya una colaboración de objetos.

   D. Siempre hay que utilizar la orientación a objetos para desarrollar sistemas, es lo más eficiente.

© Universidad Internacional de La Rioja (UNIR)

5. ¿Cuál de las siguientes afirmaciones no es correcta?

A. UML es un lenguaje visual.

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

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

D. UML es un lenguaje de propósito general.

6. ¿Cuál de las siguientes afirmaciones es correcta?

A. Un objeto puede que no tenga tipo.

B. Un objeto es una instancia de una clase.

C. Cuando dos objetos tienen los mismos valores en los atributos es porque son iguales.

D. Los objetos se comunican con otros objetos a través del envío de mensajes solo si son instancias de la misma clase.

7. ¿Cuál de las siguientes afirmaciones es correcta?

A. Una clase representa un conjunto de objetos tangibles.

B. Una clase recoge las características y el comportamiento comunes de un conjunto de objetos.

C. Una clase no puede heredar de varias clases, solo de una.

D. Una clase no define la comunicación con otras clases, eso se define a nivel de objeto.

© Universidad Internacional de La Rioja (UNIR)

8. La identidad de un objeto viene definida por:

A. La información del objeto en el mundo real que le identifica de manera unívoca.

B. El atributo identificador de la clase.

C. La clase a la que pertenece.

D. Cuando la clase a la que pertenece hereda de otra clase, entonces tiene varias identidades.

9. El comportamiento de una clase viene definido por:

A. Las operaciones que se hayan definido en la clase.

B. Los estados por los que puede pasar la clase.

C. Las responsabilidades que tenga definidas.

D. Todas las respuestas anteriores son correctas.

10. Las herramientas CASE:

A. Solo son válidas para introducir código.

B. Facilitan la extracción de requisitos.

C. No facilitan la ingeniería de requisitos.

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

© Universidad Internacional de La Rioja (UNIR)