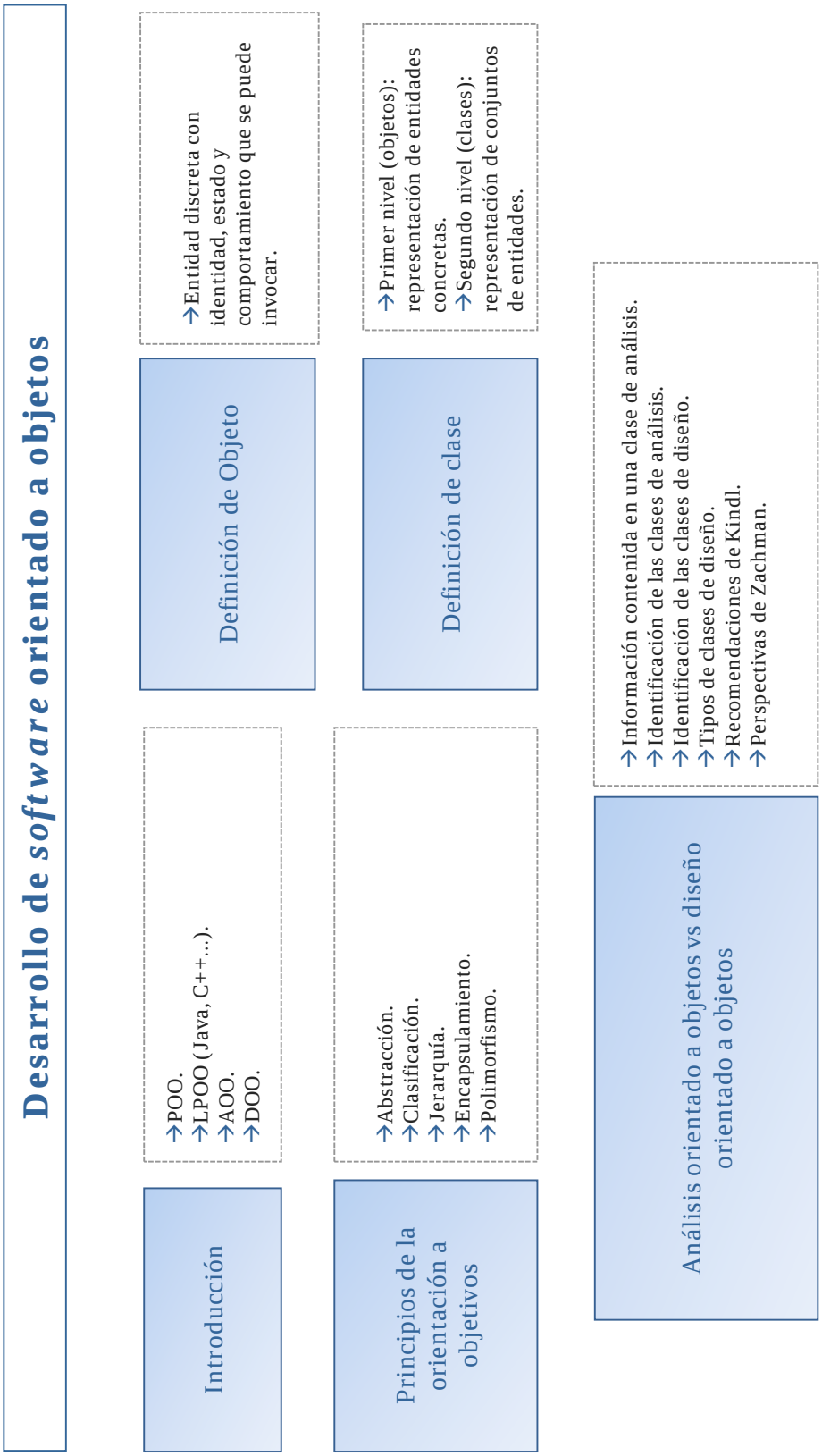
Desarrollo de *software* orientado a objetos

TEMA 3

Esquema



Ideas clave

3.1. ¿Cómo estudiar este tema?

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

En este tema se estudian y consolidan conceptos asociados a la orientación a objetos (OO, del inglés *Object Orientation*, OO), con sus principios fundamentales: abstracción, jerarquía, clasificación, encapsulamiento y polimorfismo. Además, se analiza la necesidad de diferenciar entre análisis orientado a objetos (AOO, del inglés *Object-Oriented Analysis*, OOA) y diseño orientado a objetos (DOO, del inglés *Object-Oriented Design*, OOD) y la dificultad que ello supone.

3.2. Introducción

Aunque el concepto de programación orientada a objetos (del inglés *Object-Oriented Programming*, OOP) 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. OOP 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 (POO, del inglés *Object-Oriented Programming*, OOP) 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».

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.

Los lenguajes de programación orientados a objetos (LPOO, del inglés *Object-Oriented Programming Languages*, OOPLs) tuvieron su punto álgido durante los años 80 con la aparición de lenguajes como C++, Objective C, Self, y Eiffel. Actualmente, Java constituye 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 80 cuando los conceptos introducidos por los OOPL 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 en (Booch, 1981) y (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, del inglés *Object-Oriented Analysis*, OOA) 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, del inglés *Object-Oriented Design*, OOD) propone una solución al problema a través de la creación y combinación de objetos de diseño.

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

3.3. Principios de la orientación a objetos

Dentro de la **orientación a objetos** (OO) 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 contento de la OO, Booch et al. (2007) definen la abstracción como, «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» (ver Figura 1).

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 sistema a implementar. De acuerdo a Booch et al. (2007, 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 un 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.

Figura 1. El nivel de abstracción varía dependiendo de cada punto de vista. Fuente: Booch et al. (2007).

Dentro del contexto de abstracción, se denomina **cliente** a cualquier objeto que utilice los recursos de otro objeto (conocido en este rol como **servidor**). De tal manera que,

para que un cliente pueda acceder a los recursos del servidor se deberá definir un protocolo que constituya la vista externa (estática y dinámica) de la abstracción.

El protocolo, por tanto, se define en Booch et al. (2007) como, «el conjunto de operaciones que un cliente puede utilizar para acceder a un objeto, así como las restricciones que pueda llevar asociadas para su invocación». Por otro lado, todos los requisitos del sistema que sean delegados a un objeto servidor se denominan **responsabilidades** que se han de cumplir y ofrecer a los objetos cliente (Jacobson, Ericsson y Jacobson, 2005, 51)

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».

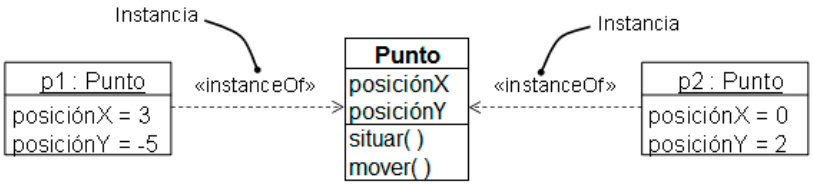


Figura 2. Principio de clasificación

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.

Jerarquía

La jerarquía es el principio de OO que permite ordenar las abstracciones (Booch et al., 2007, 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, y 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.

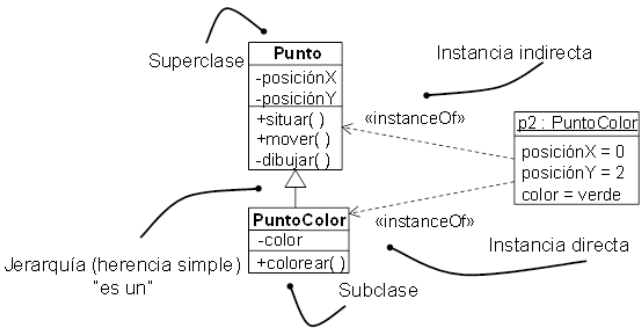


Figura 3. Principio de jerarquía

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 de la 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, 8). La Figura 4 ilustra el principio de encapsulamiento para el enfoque OO.

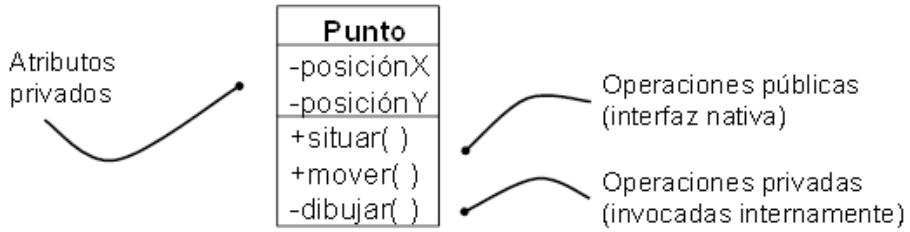


Figura 4. Principio de encapsulamiento

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.

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.

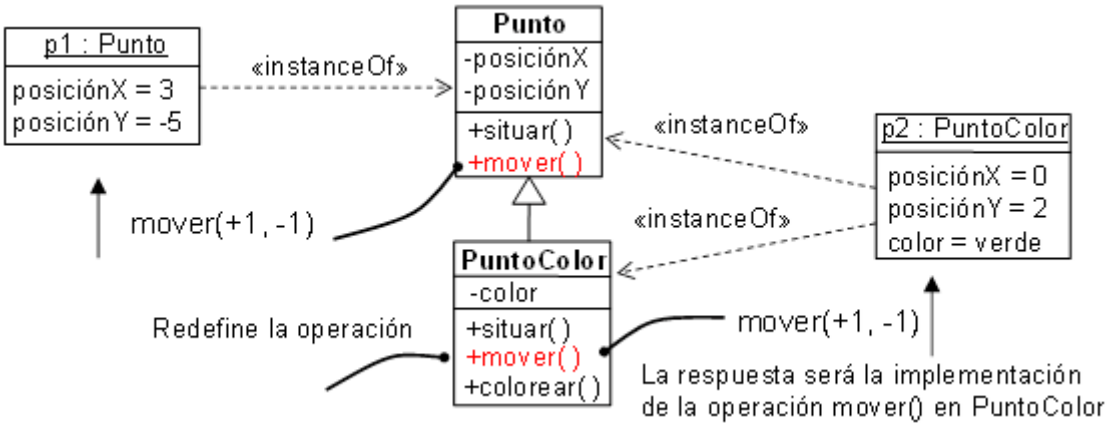


Figura 5. Principio de polimorfismo

3.4. Definición de objeto

De acuerdo a la definición de Rumbaugh, Jacobson y Booch (2007, 44), 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». 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».

Bajo el principio de abstracción, un objeto se podría definir como una abstracción de una entidad real o conceptual en la que se seleccionan los aspectos relevantes para el sistema en estudio (del inglés *System under Study, SuS*). 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 les hará independientes).

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, 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, 127). La Tabla 1 muestra un ejemplo sencillo de los estados que podría tener pasar un objeto de tipo Impresora.
- **Comportamiento.** El comportamiento define las capacidades que el objeto puede ofrecer. Por ejemplo, para el caso de una impresora, 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, 127).

Tabla 1: Estados que puede tener un objeto de tipo Impresora Fuente: Arlow y Neustadt, 2005.

Estado	Atributo	Valor del atributo	Relaciones
On	power	encendido	N/A
Off	power	apagado	N/A
SinTintaNegra	blackInkCartridge	vacío	N/A
SinTintaColor	colorInkCartridge	vacío	N/A
Conectado	N/A	N/A	Conectado a un objeto de tipo *Ordenador*
NoConectado	N/A	N/A	No conectado a un objeto de tipo *Ordenador*.

3.5. Definición de clase

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 serían las entidades concretas, tangibles, como 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, 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.

3.6. Análisis orientado a objetos vs 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, 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, 158) las clases de análisis: (i) representan una abstracción a muy alto nivel del dominio del problema; y (ii) deberían corresponderse con los conceptos del negocio del mundo real, y 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: 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 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, 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, 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.
- **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 la 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.

Esta clasificación no es exclusiva y se podrían considerar otras categorías de clases que no han sido contempladas en la lista anterior, como clases **productoras y consumidoras** propuestas por Budd (1996), o la técnica de análisis **clase-responsabilidad-colaborador** (del inglés *class-responsibility-collaborator*, CRC) tal y como se ilustra en la Figura 6.

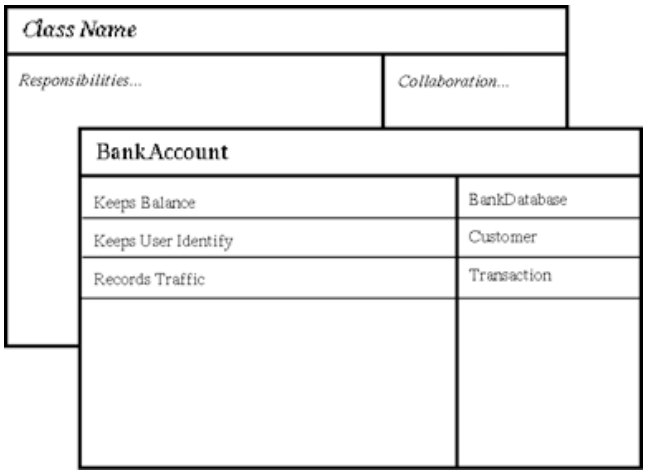


Figura 6. Ejemplo de análisis CRC. Fuente: http://vinci.org/rlv/d/uml/crc.html

Respecto a cómo comprobar la calidad de las clases de análisis identificadas, Arlow y Neustadt (2005, 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, 7).

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

- 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 7 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).

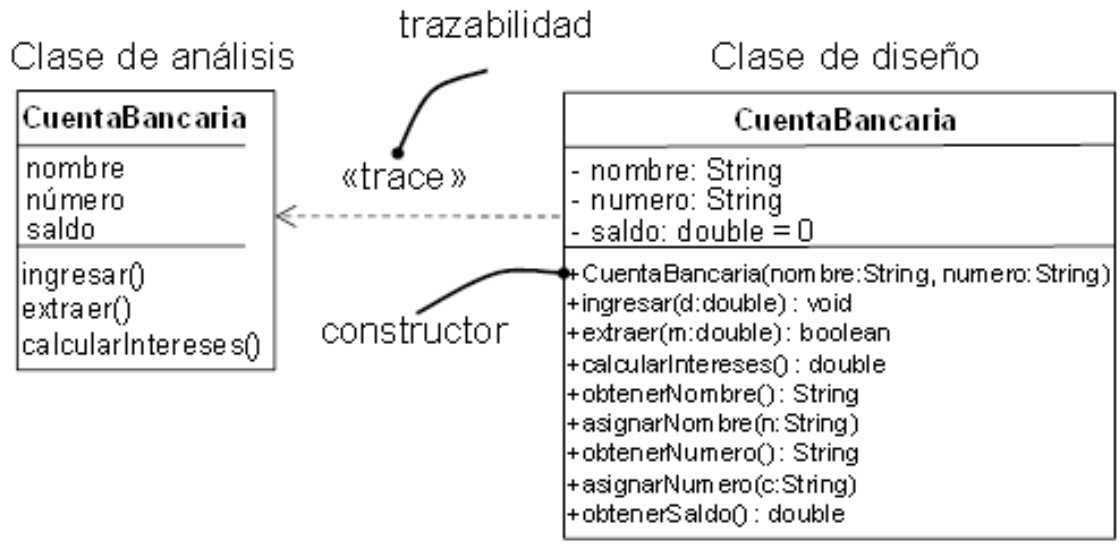


Figura 7. Refinamiento de una clase de análisis en una clase de diseño. Fuente: Arlow y Neustadt, 2005.

El proceso de extracción de las clases de diseño propuesta por Arlow y Neustadt (2005, 345) es el que se ilustra en la Figura 8.

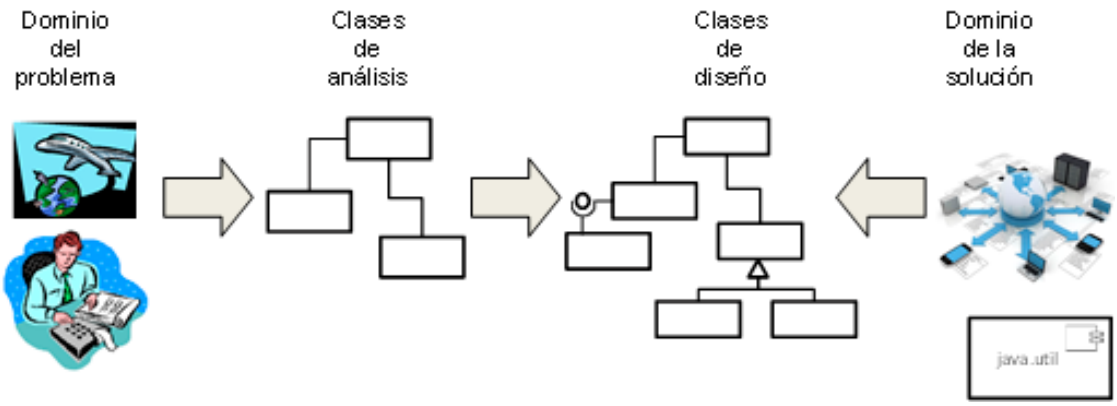


Figura 8. Extracción de las clases de diseño para un sistema software .Fuente: Arlow y Neustadt, 2005.

Por su parte, Pressman (2005, 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 (del inglés *Human-Machine Interface*, HMI).
- **Clases del dominio del negocio** que 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.

Tal y como se destaca en (Pressman, 2005, 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*.

Aunque ya se ha mencionado anteriormente, conviene recordar que para la extracción de clases en la fase de análisis, solo hay que considerar aquéllas que formen parte del dominio del problema, pero no de la solución. Lo que en un principio parece una tarea más o menos sencilla, en su origen, algunos eruditos de la OO proponían una correspondencia directa entre las clases del dominio del problema y las clases del dominio de la solución, o en cualquier caso, que la información proporcionada por las clases de análisis constituyeran una parte integral para la definición de las clases de diseño.

Cuando se desarrollan proyectos reales surgen varios problemas cuando se ha de decidir qué información del mundo real se ha de almacenar en la solución a desarrollar, y la arquitectura tecnológica más adecuada para dicha solución, lo cual no es nada trivial ya que resulta difícil de deducir partiendo solamente de las clases de análisis. Kaindl (1999) analiza este problema ilustrado en la Figura 9 con el ejemplo de un cajero automático, y propone unas recomendaciones para realizar el DOO a partir del AOO. La Tabla 2 muestra las recomendaciones realizadas por Kindl (1999) para el AOO, y la tabla 3 lista las recomendaciones que propone para el DOO.

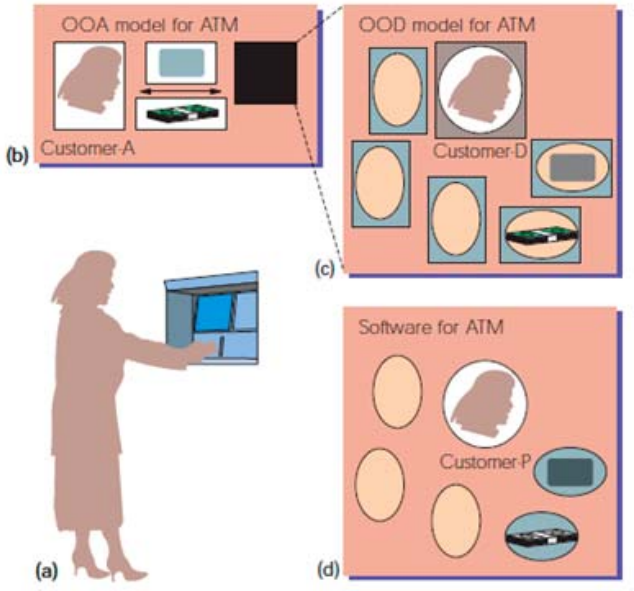


Figura 9. Relaciones entre entidades de distintos modelos del sistema y con el mundo real. Fuente: Kindl, 1999.

Tabla 2: Recomendaciones para el AOO

Recomendación	Descripción
Adquisición de los requisitos preliminares	Realizar un informe breve del problema, pero no confundirlo con los requisitos reales que se utilizan en el DOO y en el desarrollo de la POO.
Familiarizarse con el dominio tal cual es	Analizar tareas ya existentes, objetivos y procesos de negocio, pero sin llegar a crear un modelo completo del dominio.
Imaginarse el dominio una vez que el software haya sido desplegado	Centrarse en el uso que se le va a dar al sistema con ayuda de los usuarios potenciales, lo cual permitirá definir clases nuevas y eliminar, en algunos casos, clases ya existentes en el dominio.
Dependiendo del dominio, enfocarse primero en los aspectos estáticos y dinámicos	En vez de empezar con un modelo de objetos o con escenarios modelados con casos de uso, intentar definir, a muy alto nivel, la vista estática y dinámica del *software* a implementar.

Tabla 3: Recomendaciones para el DOO partiendo del modelo AOO

Recomendación	Descripción
Profundizar en la caja negra que representa el sistema propuesto en el modelo AOO	El DOO debe crear un modelo de la parte del modelo AOO que se debe desarrollar a través del *software*.
Crear la arquitectura inicial del sistema propuesto por el modelo AOO	Para ello hay que enfocarse en la parte dinámica del modelo AOO que especifica el comportamiento externo.
Por cada clase incluida en el modelo AOO, hay que responder la siguiente pregunta: ¿El sistema propuesto necesita información de los objetos del mundo real?	En caso afirmativo, se deberá crear la clase de diseño correspondiente. Lo que es importante es no confundir el actor (en términos de caso de uso) que aparece representado en el modelo AOO, con la clase de diseño que le «representa» en el modelo DOO. Por eso, a veces es conveniente no ponerles el mismo nombre, pues puede generar confusión.
Por cada asociación incluida en el modelo AOO, hay que responder la siguiente pregunta: ¿Las clases asociadas en el modelo AOO tienen las correspondientes clases en el modelo DOO?	En caso afirmativo, se creará también la correspondiente asociación entre clases en el modelo DOO.
Investigar si se necesitan asociaciones adicionales	Puedes ser necesario a veces, por temas de optimización.
Definir la arquitectura haciendo uso de la parte dinámica del modelo AOO, la arquitectura inicial creada anteriormente, y el modelo DOO que se va generando	Se debe ir refinando la arquitectura a medida que se va avanzando en el modelo DOO.
Si la arquitectura así lo requiere, definir nuevas clases en el modelo DOO.	En algunos casos puede ser necesario añadir más clases *controladoras*.
Asignar responsabilidades a las clases DOO para satisfacer la funcionalidad requerida	Puede ser necesario añadir nuevas colaboraciones para cumplir con la funcionalidad requerida y que no estaba completamente definida en el modelo AOO.

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 vista y tratada desde exclusivamente desde un punto de vista de análisis. Las notaciones que se suelen utilizar para el modelo de análisis ya tienen limitaciones a la hora poder incluir ciertos caracteres, poner un espacio en blanco entre cada palabra... lo que no tiene mucho sentido si se está realizando exclusivamente un análisis como se argumenta, y lo que se pretende generar es un modelo basado en el mundo real. Tiene más un aspecto de diseño que ya tiene en cuenta detalles de implementación.

Hay (1999) argumenta la dificultad de algunos eruditos de OO para poder explicar conceptos utilizados en análisis, como por ejemplo, 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 tienen para un mismo sistema por 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. 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 que es lo que le hace diferente.
- **Vista del propietario del negocio**. Esta vista hace referencia, no a los *stakeholders*, si no 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.

3.7. Referencias bibliográficas

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

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), 64-76.

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

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

Budd, T. (1996). *An Introduction to Object-Oriented Programming, 2da. Edition*. Addison-Wesley.

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

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).

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

Kindl, H. (1999). Difficulties in the Transition from OO Analysis to Design. *IEEE Software*, 16(5), 94-102.

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

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

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

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

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

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

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

Lo + recomendado

No dejes de leer...

Ingeniería del *software*. Séptima edición

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

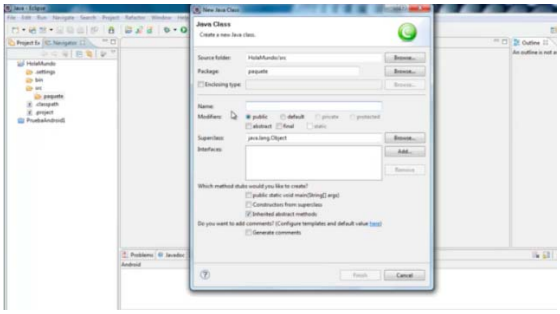
Libro orientado a estudiantes de grado y másteres de los estudios universitarios de informática, así como a los profesionales del *software*, donde se da una visión general de los aspectos más relevantes vinculados a la ingeniería del *software*. Para profundizar en lo estudiado en el tema se recomienda la lectura del apartado 8.4: *Modelos de objetos*, del capítulo 8: *Modelos del sistema* (páginas 164-169) y el capítulo 14: *Diseño orientado a objetos* (páginas 285-307) de este libro.

Accede al artículo, disponible bajo licencia CEDRO, a través del aula virtual.

No dejes de ver...

Tutorial Java 1.Introducción y primer programa «Hola Mundo»

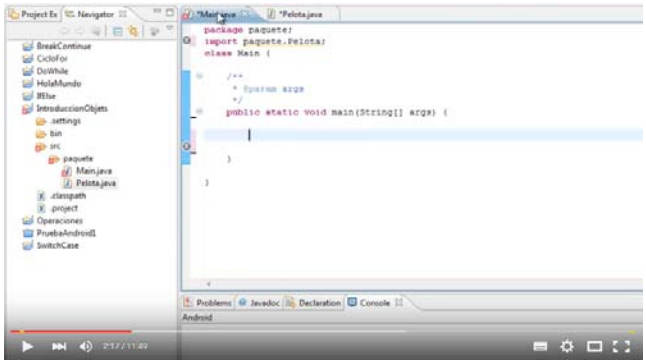
Este vídeo es un recordatorio de la programación en Java haciendo uso de entorno de desarrollo integrado (del inglés *integrated development environment*, IDE) Eclipse, lenguaje y plataforma de desarrollo que vamos a tomar como referencia para la asignatura.



Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://youtu.be/Z0F7sJaOQtw

Programación orientada a objetos en Java

Este vídeo es continuación anterior para ver las características de la orientación a objetos a través del lenguaje de programación Java.



Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://youtu.be/mPBm19gf2Lc

+ Información

A fondo

A research typology for object-oriented analysis and design

Monarchi, D.E. y Puhr, G.I. (1992). *A research typology for object-oriented analysis and design*. Communications of the ACM. Special issue on analysis and modeling in software development, 35(9), 35-47.

En este artículo se muestran investigaciones realizadas en los campos de AOO y DOO, e identifica los aspectos a mejorar y las partes débiles del análisis y diseño orientado a objetos (ADOO, del inglés *Object-Oriented Analysis and Design*, OOA/D).

Disponible en el aula virtual bajo licencia CEDRO:

Grady Booch on Design Patterns, OOP, and Coffee

O´Brien, L. y Booch, G. (2009). *Grady Booch on Design Patterns, OOP, and Coffee*. Recuperado de http://www.informit.com/

Artículo de InformIT que presenta una interesante entrevista realizada en 2009 por Larry O'Brien a Grady Booch, creador de la metodología Booch (el enfoque orientado a objetos para las fases de análisis y diseño del proceso de desarrollo de *software*), con motivo del 15 aniversario del famoso libro *Design Patterns*.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://www.informit.com/articles/article.aspx?p=1405569

On the purpose of object-oriented analysis

Høydalsvik, G.M. y Sindre, G. (1993). On the Purpose of Object-Oriented Analysis. VIII Conf. on Object-Oriented Prog., Syst., Lang., and Appl. (OOPSLA-93). *ACM SIGPLAN Notices*, 28(10), 240-255.

En este artículo se profundiza en el análisis orientado a objetos y trata de comprobar si realmente genera lo que se espera de un proceso de análisis.

Enlaces relacionados

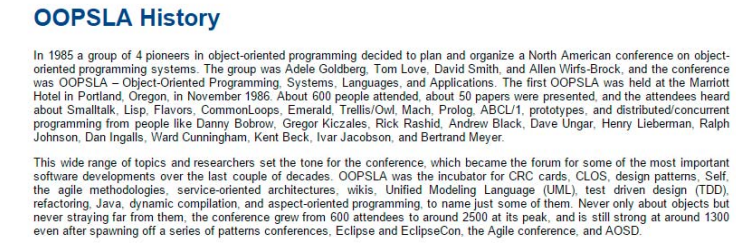
Java

Web de Oracle para Java, el lenguaje de programación orientado a objetos más extendido en la actualidad.

Accede a la página desde el aula virtual o a través de la siguiente dirección web: https://www.oracle.com/java/index.html

OOPSLA

Congreso *Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA). Este congreso constituye una referencia para todo lo que tenga que ver con el desarrollo de software orientado a objetos.



OOPSLA History

In 1985 a group of 4 pioneers in object-oriented programming decided to plan and organize a North American conference on object-oriented programming systems. The group was Adele Goldberg, Tom Love, David Smith, and Allen Wirfs-Brock, and the conference was OOPSLA – Object-Oriented Programming, Systems, Languages, and Applications. The first OOPSLA was held at the Marriott Hotel in Portland, Oregon, in November 1986. About 600 people attended, about 50 papers were presented, and the attendees heard about Smalltalk, Lisp, Flavors, CommonLoops, Emerald, Trellis/Owl, Mach, Prolog, ABCL/1, prototypes, and distributed/concurrent programming from people like Danny Bobrow, Gregor Kiczales, Rick Rashid, Andrew Black, Dave Ungar, Henry Lieberman, Ralph Johnson, Dan Ingalls, Ward Cunningham, Kent Beck, Ivar Jacobson, and Bertrand Meyer.

This wide range of topics and researchers set the tone for the conference, which became the forum for some of the most important software developments over the last couple of decades. OOPSLA was the incubator for CRC cards, CLOS, design patterns, Self, the agile methodologies, service-oriented architectures, wikis, Unified Modeling Language (UML), test driven design (TDD), refactoring, Java, dynamic compilation, and aspect-oriented programming, to name just some of them. Never only about objects but never straying far from them, the conference grew from 600 attendees to around 2500 at its peak, and is still strong at around 1300 even after spawning off a series of patterns conferences, Eclipse and EclipseCon, the Agile conference, and AOSD.

Accede a la página desde el aula virtual o a través de la siguiente dirección web:
http://www.oopsla.org/oopsla-history/

Bibliografía

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

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

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

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

Actividades

Caso práctico: Extracción de las clases de análisis a partir de un documento de información con un enfoque orientado a objetos

Para esta actividad suponemos que nos han solicitado poder gestionar de manera automatizada la tramitación y realización de los exámenes de Karate para cinturón negro, en cualquier de sus grados (Cinturón Negro, 1º DAN, 2º DAN, etc.) en la Federación Madrileña de Karate (F.M.K.).

A partir del documento que define la normativa de grados F.M.K., proporcionado a través de aula virtual, se deberá extraer la siguiente información a un primer nivel de análisis, es decir, con el mayor nivel de abstracción posible y utilizando exclusivamente el vocabulario del dominio en estudio:

- Nombre de las clases de análisis que formarán parte del sistema.
- Atributos básicos asociados a cada clase.
- Operaciones o métodos asociados a cada clase.
- Relaciones de cada clase identificada.

Accede al documento desde el aula virtual o a través de la siguiente dirección web: http://www.fmkarate.com/2001/obj/2011/Normativa%20examen%20de%20cinturon%20negro%20FMK.pdf

Extensión máxima

20 páginas en un documento de Word, tipo de letra Georgia, tamaño 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.
CB7. Que los estudiantes sepan aplicar los conocimientos adquiridos y su capacidad de resolución de problemas en entornos nuevos o poco conocidos dentro de contextos más amplios (o multidisciplinares) relacionados con su área de estudio.
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.
CG1. Capacidad para proyectar, calcular y diseñar productos, procesos e instalaciones en el ámbito de la ingeniería de *software*.
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*.
CE3. Capacidad para analizar las necesidades de información que se plantean en un entorno y llevar a cabo en todas sus etapas el proceso de construcción de un sistema de información.
CE4. Capacidad para crear y diseñar sistemas software que resuelvan problemas del mundo real.
CE5. Capacidad para evaluar y utilizar entornos de ingeniería de *software* avanzados, métodos de diseño, plataformas de desarrollo y lenguajes de programación.
CT3. Aplicar los conocimientos y capacidades aportados por los estudios a casos reales y en un entorno de grupos de trabajo en empresas u organizaciones.
CT4. Adquirir la capacidad de trabajo independiente, impulsando la organización y favoreciendo el aprendizaje autónomo.

Test

1. En la programación orientada a objetos:

A. Los objetos son instancias de alguna clase.

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. Todo lenguaje orientado a objetos satisface la siguiente afirmación afirmaciones:

A. Las clases son de un tipo concreto.

B. Los objetos son de un tipo concreto.

C. Las clases 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.

5. ¿Cuáles son los principios fundamentales de la OO?

A. Abstracción, generalización, encapsulamiento.

B. Abstracción, generalización, encapsulamiento y polimorfismo.

C. Abstracción, herencia, clasificación, encapsulamiento y polimorfismo.

D. Abstracción, jerarquía, encapsulamiento y polimorfismo.

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

A. Un objeto puede que no tenga tipo.

B. Un objeto es una instancia de una clase, es decir, la clase es su tipo.

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.

7. Marcar cuál(es) 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.

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 relaciones que tiene con otras clases.

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

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

D. Las responsabilidades que tenga definidas.

10. El *framework* de Zachman proporciona, entre otras cosas:

A. Recomendaciones para hacer el análisis de un sistema.

B. Perspectivas que se dan en todo proyecto de desarrollo de *software* en lo que se refiere a la comunicación.

C. Categorías de clases de diseño, donde cada tipo representa una capa distinta de la arquitectura elegida para el diseño del *software* a desarrollar.

D. Una nueva forma de programar con el enfoque orientado a objetos.

