Orientación a objetos

Jordi Pradel Miquel
Jose Raya Martos

PID_00171153

Universitat Oberta de Catalunya

Índice

Introducción

Tal y como hemos visto en el módulo "Introducción a la ingeniería del software", los programadores necesitan un lenguaje de programación que les permita escribir programas en términos de más alto nivel de abstracción de lo que la máquina puede ejecutar por sí misma. Uno de los tipos de lenguajes de programación más importantes son los orientados a objetos.

La orientación a objetos propone realizar una abstracción del problema que se está intentando resolver en lugar de hacerla de la máquina que finalmente ejecutará el programa. Gracias a este enfoque, resulta un paradigma muy útil no sólo para la programación, sino también para el análisis.

A lo largo de este módulo estudiaremos cómo podemos crear una descripción del mundo real siguiendo el paradigma de la orientación a objetos y cómo esta descripción nos puede ayudar a elaborar análisis de requisitos pero también en otras actividades del software.

Empezaremos introduciendo la orientación a objetos y el contexto histórico en el que nació.

A continuación iremos viendo, de manera progresiva, cómo podemos utilizar el paradigma de la orientación a objetos para describir el mundo real. Empezaremos hablando de la clasificación y la abstracción, mediante las cuales estructuramos la información en lo que denominamos *clases*, *asociaciones*, *atributos* y *operaciones*, y continuaremos estudiando el encapsulamiento, que nos permite ocultar información dentro de los objetos, y la herencia y el polimorfismo, que nos permiten llevar a cabo abstracciones más potentes.

Finalmente, veremos cómo podemos aplicar la orientación a objetos para hacer una descripción del mundo real por medio de un caso práctico.

Objetivos

El principal objetivo de este módulo es presentar el paradigma de la orientación a objetos y cómo se puede aplicar a diferentes actividades de la ingeniería del software de la programación. En concreto, el estudiante debe alcanzar los siguientes objetivos:

1. Entender los conceptos fundamentales de la orientación a objetos: clasificación, abstracción, ocultación de información, encapsulamiento, herencia y polimorfismo.

2. Saber cómo se pueden describir los objetos de un sistema mediante el uso de clases, atributos, operaciones y asociaciones.

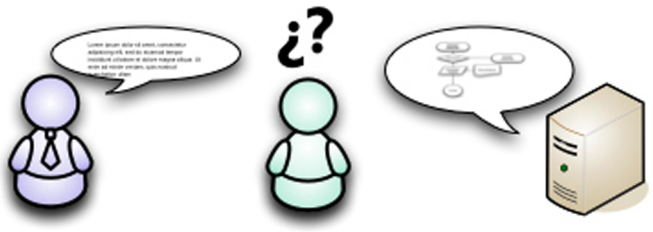
1. ¿Qué es la orientación a objetos?

Uno de los problemas comunes a todos los proyectos de desarrollo de software es describir la tarea que se debe llevar a cabo en un lenguaje que puedan entender tanto el programador como el ordenador. Los lenguajes de programación se crearon justamente por este motivo.

Ved también

En el módulo "Introducción a la ingeniería del software" se tratan los diferentes problemas comunes a todos los proyectos de desarrollo de software.

El problema de la comunicación



En el mundo de la programación informática se ha intentado solucionar el problema creando nuevos lenguajes de programación más próximos al lenguaje natural (el que utilizamos las personas para comunicarnos) y al modo que tenemos de entender la realidad que nos rodea.

Ejemplo de lenguaje ensamblador

A continuación, tenéis un ejemplo de programa en lenguaje ensamblador. No os preocupéis, no es necesario entenderlo para comprender el resto del módulo.

```
 .MODEL Small
 .STACK 100h
 .DATA
msg db 'Hello, world!$'
 .CODE
start:
mov ah, 09h
lea dx, msg
int 21h
mov ax,4C00h
int 21h
end start
```

A continuación, tenéis un ejemplo del mismo programa, esta vez escrito en C.

```
#include<stdio.h>
int main(int arg_count,char ** arg_values) {
 printf("Hello World\n");
 return 0;
}
```

Y a continuación en BASIC.

```
 10 PRINT "Hello World"
```

1.1. Evolución histórica de los lenguajes de programación

Ya sabemos que los procesadores pueden entender un conjunto limitado de instrucciones (el denominado *código máquina*). Los primeros lenguajes se limitaban a instrucciones muy sencillas que prácticamente se correspondían una a una con las operaciones que podía entender el procesador. Los lenguajes que implementan este paradigma se conocen como *lenguaje ensamblador*.

Los lenguajes desarrollados en ensamblador son muy fáciles de interpretar para la máquina, pero son muy complicados de escribir y de entender para las personas. Por este motivo, programar utilizando estos lenguajes resulta muy pesado y este enfoque sólo es viable para tareas muy concretas.

Enseguida se fueron elaborando lenguajes más ricos que permitían a los programadores utilizar un lenguaje más próximo al lenguaje natural para describir las instrucciones que debe seguir el ordenador. Es el caso de lenguajes como BASIC, COBOL o C.

Estos lenguajes se conocen como *lenguajes de alto nivel* porque permiten expresarse a un nivel de abstracción más alto que los lenguajes ensambladores. También se conocen como *lenguajes procedimentales* porque se basan en la definición de procedimientos para descomponer el problema que se quiere tratar en unidades más pequeñas.

Los lenguajes procedimentales elevan el nivel de abstracción con respecto a los lenguajes ensambladores creando una abstracción de la máquina sobre la que se ejecutan, que entiende instrucciones más abstractas que las que realmente puede ejecutar.

1.2. Los lenguajes de programación orientados a objetos

A finales de los años sesenta los investigadores noruegos Ole-Johan Dahl y Kristen Nygaard, mientras trabajaban en un software de simulación de barcos, decidieron probar un enfoque diferente. En lugar de hacer una abstracción de la máquina sobre la que se ejecutaría el programa, harían una abstracción del problema sobre el que estaban trabajando. Sus programas, en vez de tener procedimientos y estructuras de datos, tendrían objetos y cada uno representaría un objeto del mundo real; así nacía el lenguaje Simula 67 y la programación orientada a objetos.

Sin embargo, hasta finales de los años 70 y comienzos de los 80 el paradigma de la orientación a objetos no se popularizó masivamente, primero con el lenguaje Smalltalk (creado por un grupo de investigadores dirigido por Alan Kay en el mítico laboratorio de Xerox en Palo Alto), después con el C++ (creado por

Bjarne Stroustrup en los no menos míticos laboratorios Bell de AT&T en 1983) y, por último, con el Java, uno de los lenguajes más populares actualmente (creado por James Gosling en Sun Microsystems en 1995).

La diferencia fundamental de los **lenguajes orientados a objetos** con respecto a los lenguajes procedimentales radica en el hecho de que, en lugar de basarse en lo que la máquina puede entender, se basan en lo que las personas queremos comunicar.

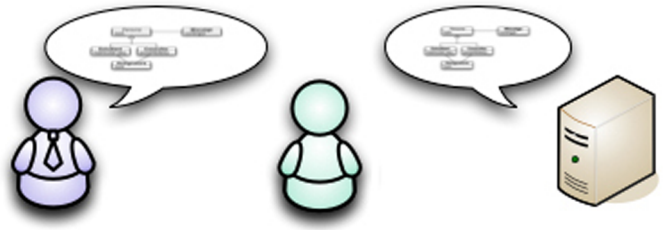
No obstante, pronto se vio que el hecho de que este paradigma se centrara en el ámbito del problema para resolver y no en el de la solución lo convertía en una buena herramienta, tanto para las tareas de implementación como durante el análisis, ya que la orientación a objetos nos ofrece una manera de hacer la descripción de cualquier sistema fácilmente trasladable a software mediante los lenguajes de programación orientados a objetos.

1.3. Análisis orientado a objetos

El análisis orientado a objetos consiste en aplicar los conceptos y las ideas de los lenguajes orientados a objetos al análisis de sistemas. Esto nos permitirá utilizar un lenguaje similar durante las diferentes etapas del desarrollo.

Este uso de la orientación a objetos durante las tareas de análisis se popularizó, especialmente, a partir de la aparición en 1997 del lenguaje unificado de modelización (UML), creado por James Rumbaugh, Grady Booch e Ivar Jacobson.

Comunicación con UML



2. Clasificación y abstracción

Un **sistema orientado a objetos** está formado por una serie de elementos autónomos (los **objetos**) que se comunican entre sí con el fin de llevar a cabo sus tareas.

Por lo tanto, lo primero que debemos saber en un sistema como éste es qué puede hacer cada uno de estos objetos: qué cosas sabe, con qué otros objetos se puede comunicar o qué tareas se le pueden pedir.

Denominaremos **responsabilidades** del objeto a estas características.

2.1. Clasificación

Cuando describimos un determinado sistema utilizando orientación a objetos describiremos los objetos que forman este sistema. Sin embargo, si intentamos describir uno por uno estos objetos, pronto nos damos cuenta de que hay objetos que son de un mismo tipo y tienen una misma estructura y responsabilidades.

Ejemplo de clasificación

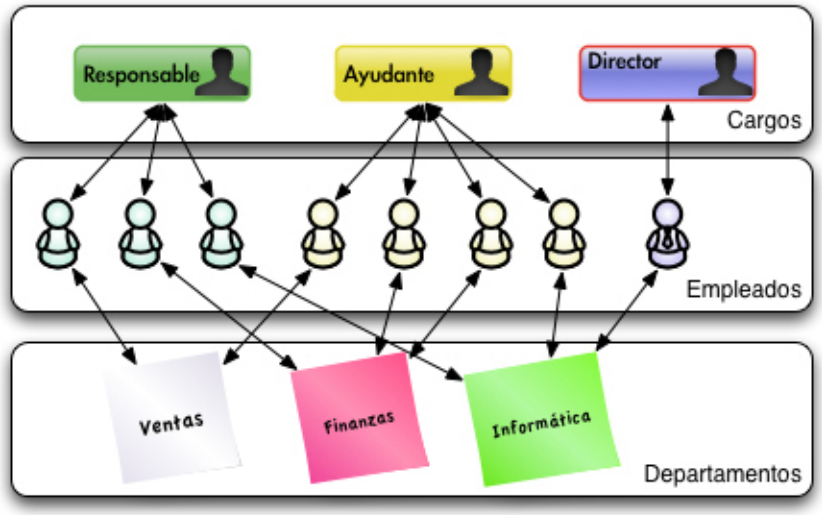
Si estamos describiendo una empresa, seguramente tendremos algún objeto que representa a un trabajador concreto. Este trabajador puede tener responsabilidades como "saber cuál es su nombre", "saber cuál es su departamento" o "calcular cuántos compañeros tiene (es decir, cuántos trabajadores pertenecen al mismo departamento que él)". Pero enseguida observamos que existen muchos otros objetos en la misma empresa que también representan trabajadores y que, por lo tanto, también tienen las responsabilidades de "saber cuál es su nombre", "saber cuál es su departamento" y "calcular cuántos compañeros tiene".

Por lo tanto, a la hora de describir nuestro sistema no nos interesará tanto el objeto en sí (en este caso, una persona, el individuo), como la clase de objetos que tienen las mismas responsabilidades (en este caso, los trabajadores).

La **clasificación** es el mecanismo por el que simplificamos la descripción de los objetos del sistema e identificamos aquellas clases de objetos que comparten las mismas responsabilidades.

En el caso anterior, todos los trabajadores conocen su nombre (aunque cada trabajador tenga un nombre diferente, cada uno sabe cuál es el suyo), saben cuál es su departamento y pueden calcular cuántos compañeros tienen.

Clasificación de los objetos



Denominamos **clase** al grupo e **instancia**, al objeto.

Clases de objetos

En el ejemplo anterior tenemos tres clases de objetos:

- Cargos
- Empleados
- Departamentos

De la clase *Cargo* tenemos tres instancias (*Director*, *Ayudante* y *Responsable*), de *Empleado*, ocho, y de *Departamento*, tres (*Ventas*, *Finanzas* e *Informática*).

A la hora de definir las responsabilidades de los objetos de una clase, las podemos dividir en tres tipos:

1) la información que conocen (los datos)

2) los objetos que conocen (las asociaciones)

3) las tareas que pueden llevar a cabo (las operaciones)

En el ejemplo anterior, cada instancia de *Empleado* sabe cuál es la información del empleado (el nombre, el teléfono, etc.), está asociada con otras instancias (su cargo y su departamento) y existe una serie de tareas que puede llevar a cabo (redactar una solicitud, aprobarla, etc.).

Instancias

2.2. Abstracción

En orientación a objetos, la **abstracción** es el mecanismo por el que decidimos qué responsabilidades asociadas a una clase formarán parte de su definición.

A la hora de describir un sistema mediante la orientación a objetos, deberemos aplicar el mecanismo de abstracción a los tres tipos de responsabilidad que hemos visto anteriormente para decidir qué datos, relaciones y comportamiento son relevantes para nuestro problema.

En el caso del ejemplo anterior, es muy probable que rasgos como el color de los ojos, la altura o quién es amigo de quién entre los trabajadores, así como el hecho de que a algunos trabajadores podamos pedirles que hagan la declaración de renta, no sean relevantes y, por lo tanto, no se tengan en cuenta.

2.3. Descripción de las clases de objetos

Si aplicamos la clasificación y la abstracción, llegaremos a una primera descripción de los objetos que forman parte de nuestro sistema, en el que tendremos un conjunto de clases y, para cada clase, un conjunto de responsabilidades. Estas responsabilidades tomarán la forma de atributos, operaciones y asociaciones.

2.3.1. Atributos

Cuando todos los objetos de una clase tienen un mismo elemento de información (como la fecha de contratación en la empresa), aunque cada objeto tenga un valor diferente, decimos que la clase tiene un atributo.

Los **atributos** nos dicen qué información (qué datos) conocemos sobre los objetos de una clase determinada.

El atributo tendrá un **nombre** que nos permite distinguirlo del resto de los atributos (por ejemplo, podemos denominar *Alta* al atributo que guarda la fecha en la que un empleado fue dado de alta en la empresa) y suele tener un **tipo** que nos ayuda a entender mejor el tipo de información que se almacena y a saber qué valores son válidos para el atributo (por ejemplo, si decimos que *Alta* es de tipo *Fecha*, sabemos que los valores que puede tomar deben ser fechas y no números enteros).

String

Denominamos *String* al tipo de atributos en el que los valores son secuencias de caracteres; es decir, textos sin ningún formato.

Formalmente, un String es una secuencia de símbolos que se eligen de un conjunto o de un alfabeto.

Ejemplo de clase con atributos y de dos de sus instancias

Ámbito del atributo

Los atributos pueden almacenar un valor diferente para cada instancia o un mismo valor para todas las instancias de la clase. Por ejemplo, el atributo *Alta* que almacena la fecha en la que se incorporó el trabajador a la empresa debe tener un valor diferente para cada trabajador. En este caso, decimos que el atributo tiene **ámbito de instancia**.

En cambio, un atributo denominado *mediaEdades*, que almacenará la media de las edades de todos los trabajadores, no tendría sentido que diera valores diferentes para diferentes instancias, ya que, sea cual sea el trabajador a quien preguntemos, la media de las edades debe ser la misma. En este caso, decimos que el atributo tiene **ámbito de clase**. Sin embargo, estos atributos (con ámbito de clase) son poco habituales.

Número de valores de un atributo

Hasta ahora hemos visto atributos tales que cada instancia sólo puede tener un valor y lo debe tener obligatoriamente. Decimos que son atributos univaluados y obligatorios. Pero algunos atributos pueden no tener valor en algunas instancias (atributos **opcionales**) o pueden tener más de uno (atributos **multivaluados**).

Ejemplos de atributos opcionales y multivaluados

El número de fax de un trabajador podría ser un atributo opcional (ya que un trabajador puede no tener número de fax); por otra parte, el número de teléfono podría ser multivaluado, ya que puede ser que un trabajador tuviera más de un número de teléfono en el que se pudiera localizar.

2.3.2. Asociaciones

Las asociaciones nos dicen qué otros objetos conocen los objetos de una clase (cuáles son sus relaciones con otros objetos). Por lo tanto, forman parte del conjunto de información (los datos) que gestiona un objeto y, como tales, los podríamos considerar atributos. Sin embargo, a diferencia de los atributos que hemos considerado hasta ahora, una asociación no hace referencia a un único objeto, sino que siempre hará referencia, como mínimo, a dos objetos. Por esta razón, distinguimos entre atributos y asociaciones.

Por ejemplo, hemos dicho que nos interesa saber, para una instancia concreta de departamento, qué empleados pertenecen a éste. En este caso diríamos que existe una asociación entre las clases *Empleado* y *Departamento*. La asociación puede tener un nombre (por ejemplo, podemos denominar *trabajaEn* a la relación entre un empleado y su departamento). A diferencia de un atributo, la asociación *trabajaEn* siempre hace referencia a dos objetos: un empleado y un departamento.

Una **asociación** entre dos clases indica que las instancias de cada clase saben, como parte de sus responsabilidades, con qué instancia o instancias de la otra clase están relacionadas.

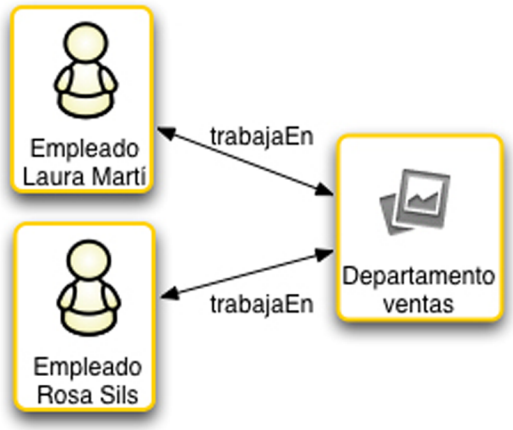
La relación de asociación entre dos instancias es recíproca, de tal manera que si una instancia *a* de una clase *A* está asociada con una instancia *b* de la clase B, la instancia *b* estará también asociada a la instancia *a*.

Cuando una clase está asociada con otra, podemos consultar el vínculo entre las instancias relacionadas. Así, si decimos que la clase *Empleado* está asociada a la clase *Departamento*, estamos diciendo que, dada una instancia concreta de la clase *Empleado*, le podremos preguntar con qué instancias de la clase *Departamento* está asociada o, si lo queremos hacer al revés, dada una instancia concreta de *Departamento*, le podremos preguntar con qué instancias de *Empleado* está asociada.

Ejemplo de asociaciones

En el ejemplo de la figura siguiente podemos saber que la empleada Laura Martí tiene asociado el Departamento de Ventas, la empleada Rosa Sils también y que el Departamento de Ventas tiene asociados a los empleados Laura Martí y Rosa Sils.

Asociación Empleado-Departamento(trabajaEn)



Aunque la asociación es una característica de la clase, los valores que toma serán específicos de las instancias (al igual que sucede con los atributos). Pero, a diferencia de los atributos, las asociaciones nunca tendrán ámbito de clase.

Roles de las instancias

Cuando una instancia participa en una asociación con otras instancias, siempre lo debe hacer teniendo un rol concreto. En ocasiones, este rol tendrá un nombre explícito e incluso podrá tener una visibilidad (público, privado, etc.).

El rol es especialmente importante en el caso de las **asociaciones recursivas**, que son aquellas asociaciones que conectan una clase con ella misma.

Ejemplo de asociación recursiva

Si tenemos la clase *Persona* y queremos definir la relación entre padres e hijos, tendríamos una asociación recursiva en la clase *Persona*, ya que estamos conectando la clase *Persona* con ella misma. En este caso, una instancia de persona puede tener el rol de padre o de hijo en relación con otra instancia. Sin embargo, es importante ver que una misma instancia de *Persona* puede tener el rol de padre en relación con una instancia y, al mismo tiempo, tener el rol de hijo con relación a otra.

Asociación recursiva

El papel también es importante cuando tenemos más de una asociación entre dos clases.

Por ejemplo, la clase *Persona* y la clase *Ciudad* podrían tener dos asociaciones: la que nos dice dónde nació una persona (ciudad natal) y la que nos dice dónde vive la persona (ciudad de residencia). En este caso no basta con decir que una ciudad está asociada a una persona, hay que distinguir cuál de los dos roles tiene la ciudad en su asociación con la persona (es decir, si es su ciudad natal o si es su ciudad de residencia). A priori, nada impide que una misma ciudad tenga los dos roles, pero aun así consideraríamos que la ciudad está asociada dos veces a la persona, una como ciudad natal y otra como ciudad de residencia.

Clases asociativas

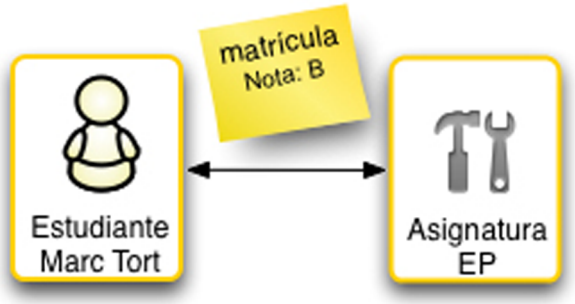
Nota de un estudiante

Supongamos el siguiente caso. Un estudiante está matriculado de una asignatura y, al finalizar el curso, obtiene una nota B. ¿De qué clase es atributo la nota que el estudiante saca de una asignatura? Si hacemos que la nota sea un atributo de *Estudiante*, entonces no dependería de la asignatura; diríamos, por ejemplo, que Juan (un estudiante) tiene una nota B, pero ¿de qué asignatura? Si, en cambio, definimos la nota como un atributo de *Asignatura*, entonces no dependería del estudiante. En realidad, la nota pertenece a la asociación entre el estudiante y la asignatura.

Una **clase asociativa** es aquella clase cuyas instancias son instancias de una asociación. En el ejemplo anterior, tendríamos una clase asociativa denominada *Matrícula* cuyas instancias serían las parejas (*Estudiante*, *Asignatura*) que están relacionadas.

Una **instancia de una asociación** es una pareja de instancias de las clases que la asociación relaciona. Podemos hablar de instancias de asociación tanto si la asociación es una clase asociativa como si no lo es. Sin embargo, en el primer caso estas instancias son, al mismo tiempo, objetos de una clase y, por lo tanto, pueden tener, a su vez, atributos, operaciones, métodos y asociaciones.

Clase asociativa



Asociaciones *n*-arias

Hasta ahora, todas las asociaciones que hemos visto eran entre dos clases, pero a veces se puede dar el caso de tener asociaciones entre tres o más clases.

Ejemplo de asociación *n*-aria

La matrícula de un estudiante la podemos ver como una asociación binaria (*Estudiante*, *Asignatura*) o como una asociación ternaria (*Estudiante*, *Semestre*, *Asignatura*).

En el segundo caso, no existe una única asociación entre el estudiante y la asignatura, sino varias, cada una correspondiente a un semestre diferente.

En el caso de tener una clase asociativa, las instancias de la clase siempre estarán relacionadas con una instancia de cada una de las clases de la asociación. Así, no tiene sentido hablar de la nota del estudiante en esta asignatura, sino hacerlo de la nota del estudiante en aquella asignatura y en aquel semestre.

2.3.3. Operaciones

Las **operaciones** nos dicen qué tareas pueden llevar a cabo las instancias de una clase (cuál es su comportamiento observable).

A la hora de definir el comportamiento de un objeto, distinguimos entre la operación (la definición de qué puede hacer) y el método (el conjunto de instrucciones que indica cómo lo tiene que hacer). Las operaciones forman parte de la **interfaz**, mientras que los métodos forman parte de la **implementación**.

Ved también

Podéis encontrar más información sobre la diferencia entre interfaz e implementación en el módulo "Introducción a la ingeniería del software".

Cuando un objeto quiere que otro ejecute el comportamiento definido en una de sus operaciones, decimos que **invoca** (o **llama**) una operación o que **envía** tal **mensaje**.

Operaciones, ámbito de clase o de instancia

Las operaciones también pueden tener ámbito de clase (el mensaje lo recibe la clase) o de instancia (el mensaje lo recibe una instancia en concreto), aunque, de momento, no es necesario que lo tengamos en cuenta.

Ejemplo

Por ejemplo, si la clase *Trabajador* tiene una operación denominado *cuántosCompañeros*, decimos que se invoca o se llama a la operación *cuántosCompañeros* o que se le envía el mensaje *cuantosCompañeros*.

Ejemplo de clase con atributos y operaciones y de dos de sus instancias

3. Ocultación de información y encapsulamiento

Hemos visto que, gracias a la clasificación y la abstracción, podemos simplificar la descripción de los objetos agrupándolos en clases y centrándonos sólo en aquellas responsabilidades que sean relevantes para nuestro caso.

También hemos visto que nos interesa "ocultar información" con el fin de separar la interfaz de un objeto de su implementación y poder ofrecer al resto de los objetos del sistema una versión todavía más simplificada de éste.

Ved también

Hemos visto en el módulo "Introducción a la ingeniería del software" que nos interesa separar la interfaz de un objeto de su implementación.

El **encapsulamiento** es el mecanismo de la orientación a objetos que nos permite **ocultar información**.

3.1. Ocultación de información

El concepto de ocultación de información es muy anterior a los lenguajes orientados a objetos. De hecho, fue mencionado por primera vez por David Parnas (1972).

Ocultación de información según David Parnas

Cada elemento (módulo o subsistema) del sistema contiene una cierta información que "oculta" al resto de los módulos y les presenta una visión simplificada.

Una descripción en la que no explicamos el funcionamiento interno del objeto descrito se conoce como descripción de "caja negra".

Ejemplo de descripción de "caja negra"

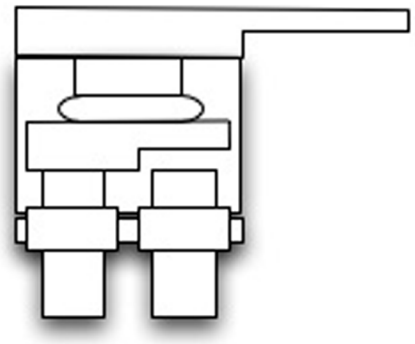
Supongamos que queremos describir a otra persona cómo funciona un grifo de palanca. Seguramente nuestra descripción sería algo así como: "El grifo tiene una palanca que, cuando la empujas, hace que salga agua (cuanto más la subes, más agua sale) y cuando la bajas, cierra el grifo. Para regular la temperatura debes mover la palanca a la izquierda (si quieres el agua más caliente) o a la derecha (si la quieres más fría)".

Grifo visto como caja negra

Sin embargo, con esta descripción sólo hemos descrito cómo se utiliza el grifo, no cómo funciona por dentro. Si fuera un fontanero quien le pregunta a otro cómo funciona un grifo de palanca, seguramente nos contestaría algo como lo siguiente:

"El grifo tiene dos entradas de agua, una fría y otra caliente. La palanca está conectada a una rótula que regula el paso de agua de las dos entradas: en la posición central deja pasar agua de los dos circuitos, mientras que si movemos la palanca iremos ampliando el paso de un circuito al mismo tiempo que cerramos el paso del otro".

Grifo visto como caja blanca



Las descripciones que incluyen los detalles de funcionamiento las conocemos como descripciones de caja blanca.

Las dos son descripciones válidas del grifo de palanca, pero mientras que la primera es muy fácil de entender, la segunda requiere un poco más esfuerzo y conocimientos sobre fontanería. Aplicando el principio de ocultación de información, la primera descripción omite el detalle sobre cómo se regulan los dos circuitos de agua y se limita a explicarnos que, moviendo la palanca, alteramos la temperatura del agua.

De la misma manera, al describir las responsabilidades de una clase, lo podemos hacer o bien desde un punto de vista externo (como una caja negra), indicando sólo cómo se pueden utilizar los objetos de aquella clase, o bien desde un punto de vista interno (o de caja blanca), indicando cómo está estructurado el objeto y cómo funciona por dentro.

3.2. Encapsulamiento

El **encapsulamiento** nos permite ocultar información en otros objetos definiendo qué atributos, operaciones y asociaciones de un objeto les serán visibles y cuáles estarán ocultos.

De esta manera, los atributos y las operaciones que no hemos hecho visibles quedan ocultos para el resto de los objetos, que, por lo tanto, no necesitan (de hecho no pueden) saber de su existencia. El hecho de que los datos y las operaciones formen parte de la misma entidad (el objeto) facilita todavía más la ocultación de información.

Se denomina **visibilidad** de un atributo, asociación u operación a la propiedad que define qué objetos pueden verlo.

La visibilidad, pues, determina qué objetos pueden consultar el valor del atributo o de la asociación o invocar la operación. En general, decimos que la visibilidad determina qué objetos tienen acceso a una responsabilidad determinada. Los diferentes lenguajes de programación definen diferentes tipos de visibilidad, pero los casos más típicos son:

Ved también

Veremos el concepto de subclase en el apartado 4 de este módulo.

- Visibilidad pública: cualquier objeto tiene acceso.
- Visibilidad privada: sólo los objetos de la misma clase tienen acceso.
- Visibilidad protegida: sólo los objetos de la misma clase o de alguna subclase tienen acceso.
- Visibilidad de paquete: sólo los objetos del mismo grupo o "paquete" tienen acceso.

Paquete

Un paquete es una agrupación de elementos en un sistema orientado a objetos para facilitar su organización. Un paquete puede contener clases y otros paquetes y puede resultar útil para definir la visibilidad de paquete de los atributos, las asociaciones y las operaciones de las clases que contiene.

Como efecto colateral, la ocultación de información tiene la ventaja de que alguien podría crear una implementación alternativa (por ejemplo, fabricar un grifo de palanca que sustituyera la rótula por algún otro mecanismo), sin que ello afectara a los usuarios de la clase, ya que el comportamiento observable (la interfaz) es el mismo.

Ejemplo de encapsulamiento

En la figura siguiente tenemos dos posibles implementaciones para una misma interfaz. En los dos casos, los atributos públicos son los mismos (Nombre, Teléfono, Antigüedad),

pero se calcularían de manera diferente a partir de conjuntos de atributos privados diferentes.

Encapsulamiento

4. Herencia y polimorfismo

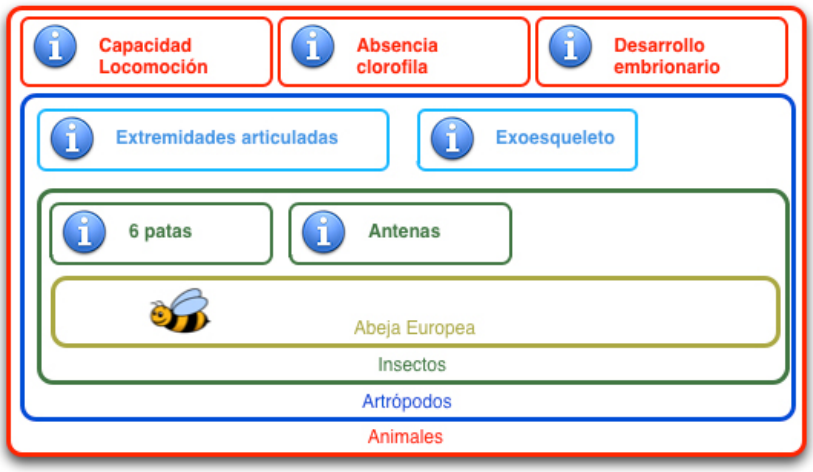
4.1. Herencia

Hasta ahora, sólo hemos tenido en cuenta un nivel de clasificación (un objeto pertenece a una clase), pero de la misma manera que la clasificación nos permite simplificar la descripción que hacemos de un objeto agrupando los objetos en clases, a veces podemos simplificar la descripción de una clase identificando responsabilidades comunes entre las instancias de diferentes clases.

Clasificación de los seres vivos

Podemos considerar el caso de la clasificación de los seres vivos. Una abeja cualquiera podría ser un ejemplar de abeja europea (*Apis mellifera*). Pero a la hora de describir cómo es la abeja, sabemos que ésta tiene características comunes con otros seres vivos que no son abejas (los otros insectos) y que, a su vez, comparten características con otros seres vivos que no son necesariamente insectos (como los artrópodos), que a su tiempo comparten características con el resto de los animales.

Clasificación de la abeja europea



Este tipo de clasificación jerárquica lo podemos reproducir en un sistema orientado a objetos mediante el mecanismo de **herencia** entre clases.

Decimos que una clase (**subclase**) hereda la definición de otra clase (**superclase**) cuando queremos reutilizar la definición de la superclase, de manera que ésta se aplique también a las instancias de la subclase. Este mecanismo nos permite, a la hora de definir una subclase, indicar sólo aquellas características que son específicas, mientras que las que son comunes con otras subclases de la misma superclase, decimos que las hereda.

Volviendo al ejemplo de las abejas, cuando estamos definiendo la clase *abeja*, no es necesario que indiquemos que tiene tres pares de patas, sino que es suficiente con decir que es una subclase de *insecto* (y, por lo tanto, la clase *abeja* hereda esta característica). Si modificamos la definición de la clase *insecto* y añadimos que tiene dos antenas, automáticamente todas las subclases de *insecto* (entre ellas, *abeja*) pasan a heredar esta nueva característica.

Relaciones *is-a* e *is-like-a*

La relación de herencia se conoce también como relación *is-a* (es-un), ya que las instancias de la subclase son también de la superclase (una abeja *is-a* un insecto). Sin embargo, a veces nos puede interesar que la subclase añada características que no son aplicables a la superclase (por ejemplo, una bomba de calor es como un aparato de aire acondicionado, pero que además de enfriar también puede dar calor). En estos casos, la relación sería *is-like-a* (es-como-un) y existe cierta controversia sobre si éste es o no un uso correcto de la herencia.

La relación de herencia no se debe limitar necesariamente a un solo nivel ni simplemente a dos subclases, sino que podemos llegar a tener clasificaciones arbitrariamente complejas.

4.1.1. Generalización y especialización

Existen dos procesos que nos permiten identificar fácilmente las relaciones de herencia en nuestro dominio: la generalización y la especialización. El proceso de **generalización** consiste en encontrar clases más generales a partir de las que ya tenemos, mientras que la **especialización** consiste en encontrar clases más específicas a partir de las que ya tenemos.

La generalización es un proceso que se lleva a cabo en dos etapas: el primer paso es identificar responsabilidades comunes a dos o más clases.

Ejemplo de primera etapa de la generalización

En la universidad tenemos instancias de *Estudiante*, de las que sabemos que tienen un nombre, unos apellidos, una edad y que están matriculados de un número concreto de créditos. También tenemos instancias de *Profesor*, de las que sabemos que tienen un nombre, unos apellidos, una edad y una antigüedad en la universidad. Por lo tanto, podemos comprobar que los estudiantes y los profesores, si bien no son la misma clase (por ejemplo, ningún profesor está matriculado de créditos ni sabemos la antigüedad de ningún estudiante), sí que comparten algunas características (nombre, apellidos y edad).

El segundo paso de la generalización consiste en crear una nueva clase y crear una relación de herencia entre las clases que teníamos y la clase que hemos añadido.

A nuestro ejemplo añadiríamos una clase *Persona* y haríamos que tanto *Estudiante* como *Profesor* fueran subclases de *Persona*.

La **especialización** es el proceso simétrico al anterior: primero identificamos un subconjunto de las instancias de una clase que comparten alguna característica común (por ejemplo, hay un subconjunto de personas que se pueden matricular de asignaturas, mientras que el resto no lo pueden hacer) y después creamos una subclase nueva que incluye este conjunto de instancias y añade a su definición la característica que le es específica.

Si estudiáramos el ADN de las abejas europeas, veríamos que no son todas iguales, sino que existen varios grupos, como el grupo africano, el mediterráneo o el de Oriente Medio. En este caso, a partir de la clase *abeja europea*, iríamos creando subclases por especialización a medida que fuéramos descubriendo los diferentes grupos.

4.2. Polimorfismo

En el caso de las abejas, un mismo espécimen lo podemos ver como una abeja (por ejemplo, si no entendemos mucho) o como una abeja europea del grupo africano (si, por ejemplo, estuviéramos interesados en estudiar la distribución geográfica de los diferentes grupos). Esto es muy útil porque nos permite trabajar con diferentes niveles de abstracción según sea nuestro interés en los detalles.

El **polimorfismo** es la capacidad de un objeto de presentarse con formas diferentes (interfaces diferentes) según el contexto.

En el ejemplo anterior, el espécimen se presenta con forma (o interfaz) de abeja para el observador no experto, mientras que para el observador experto se presenta como abeja europea del grupo africano.

Polimorfismo

Sin embargo, es importante ver que el espécimen (el individuo, la instancia) es el mismo en los dos casos y que, por lo tanto, el polimorfismo está relacionado con la visión que los otros objetos tienen del objeto más que con el objeto en sí, que siempre pertenece a la clase más concreta de la jerarquía, aunque algunos observadores no lo vean.

En el ejemplo anterior, aunque el observador no experto sólo vea una abeja, la abeja que está viendo siempre es una abeja europea del grupo africano.

Ejemplo de polimorfismo

Si queremos conocer el nombre de una persona que participa en el campus virtual de la UOC, no nos importa saber si es un consultor o un estudiante y, por lo tanto, utilizaremos la interfaz de persona independientemente del tipo concreto de persona que sea. En cambio, si queremos saber si la persona está matriculada de una asignatura, debemos tener en cuenta el tipo de persona que es, ya que los consultores no se matriculan y, por lo tanto, la pregunta sólo tiene sentido para los estudiantes (y debemos utilizar la interfaz de estudiante).

El polimorfismo afecta tanto a los datos como al comportamiento y permite que las subclases redefinan algunas operaciones con el fin de comportarse de manera diferente del resto de las instancias (es lo que denominamos *operaciones polimórficas*).

Una **operación polimórfica** es una operación que tiene un comportamiento diferente en función de la clase concreta a la que pertenezca un objeto.

Por ejemplo, existen grupos de abejas que son más agresivos que otros y que, por lo tanto, responderán de manera diferente al mismo estímulo. En este caso, podemos decir que la clase *abeja* define la operación *molestar* con el comportamiento *huir*, pero que hay una subclase que la redefine con el comportamiento *atacar*; decimos que la operación *molestar* es polimórfica porque, si molestamos a una abeja, su comportamiento será diferente según el tipo concreto de abeja que sea (podría huir si es una de las normales o atacarnos si es una de las agresivas).

4.2.1. Clases abstractas y redefinición del comportamiento

En una clasificación jerárquica como las que estamos proponiendo, podría suceder que una superclase no tuviera ninguna instancia que no perteneciera a una subclase (por ejemplo, la clase *abeja* no tiene ninguna instancia que no sea de alguna subclase como la abeja europea).

Denominamos **clases abstractas** a las clases que no se pueden instanciar directamente; estas clases se deben instanciar por medio de sus subclases.

En el ejemplo anterior, si queremos una instancia de abeja, debemos decidir qué subtipo de abeja debe ser.

Las clases abstractas pueden definir tanto datos como el comportamiento que heredarán sus subclases. Sin embargo, una característica específica de las clases abstractas es que pueden definir **operaciones abstractas**.

Una **operación abstracta** es una operación que no tiene asociado un método (es decir, no se ha definido un comportamiento) en la clase en la que se define.

Esto obliga a las subclases a definir un método (comportamiento) para tal operación o a ser igualmente abstractas y dejar que sean sus subclases las que definan el comportamiento.

Ejemplo de operación abstracta

Si volvemos al ejemplo de los grifos, podemos considerar que tenemos una clase abstracta *Grifo* y dos subclases, *Grifo de palanca* y *Grifo termostático*. En este caso, la clase *Grifo* puede ofrecer una operación denominada *cerrar*, que cerrará el grifo y hará que deje de salir agua.

Sin embargo, el comportamiento de esta operación será diferente en función del tipo concreto de grifo. En este ejemplo, la operación *cerrar* de la clase *Grifo* es abstracta porque no define ninguna implementación definida y las subclases *Grifo de palanca* y *Grifo termostático* la implementan de manera diferente (cada uno definiendo su comportamiento propio).

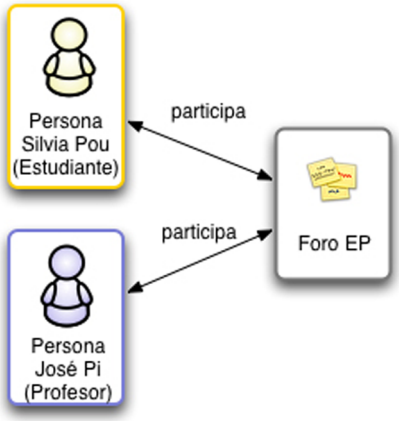
4.2.2. Asociaciones polimórficas

Las asociaciones nos permiten dar una nueva utilidad al polimorfismo: podemos tener asociaciones con una superclase, de manera que no nos debemos preocupar de la subclase concreta de los objetos que tenemos asociados.

Ejemplo de asociación polimórfica

Un aula del campus virtual puede tener una lista de personas participantes, para las que no necesita saber si son estudiantes o profesores, sino sólo saber si son personas, si participan en el aula y cuál es su nombre.

Asociación polimórfica



Este uso del polimorfismo, además, nos ayudará a evolucionar en el sistema si, en un futuro, descubrimos otros tipos de personas (siempre que éstas también tengan nombre y puedan recibir mensajes).

5. Caso práctico: un foro virtual

A continuación, aplicaremos los principios y las técnicas vistas anteriormente a un dominio muy pequeño pero suficientemente representativo: un sistema de foros virtuales en el que los usuarios envían mensajes a un foro y pueden leer los mensajes que han enviado a los otros usuarios.

Antes de aplicar lo que hemos visto, es muy importante que consideremos cuál es la finalidad de la tarea que estamos llevando a cabo. Como hemos dicho con anterioridad, el paradigma de la orientación a objetos se puede aplicar en ámbitos diferentes, como la descripción del dominio durante el análisis o la implementación del software del sistema. En este ejemplo, nos centraremos en el uso de la orientación a objetos como técnica para la descripción del dominio del problema y, por lo tanto, nuestro objetivo es identificar las clases del dominio y no las clases de software que formarán parte de la solución. Así, veremos clases como *Mensaje* o *Foro* pero no *BaseDeDatosDeMensajes*.

Observación

El modelo del dominio que hacemos durante el análisis utiliza los objetos para representar conceptos del mundo real y, por lo tanto, no nos interesa identificar operaciones ni asignarles métodos, ya que estas tareas se harán durante el diseño y la implementación del sistema.

5.1. Clasificación de los objetos

Empezaremos identificando las clases y los atributos de los objetos. En este primer paso, crearemos las abstracciones básicas de nuestro sistema y descartaremos todos los detalles que no consideremos relevantes.

5.1.1. Clases

En este caso, haciendo un ejercicio de abstracción, podemos identificar fácilmente tres clases bien diferenciadas: *Usuario*, *Mensaje* y *Foro*. Usuarios son aquellas personas que utilizan el sistema, mensajes son lo que se envían los usuarios y los foros sirven para agrupar los mensajes.

Todos los objetos que tengamos en el dominio pertenecerán a alguna de estas tres clases y, por lo tanto, no consideraremos ninguna clase más si no es que incorporamos más información que la que tenemos actualmente.

Clases

5.1.2. Atributos

Para determinar los atributos de las diferentes clases, debemos determinar qué detalles son relevantes para nuestro sistema de las instancias de cada clase.

De los usuarios podríamos conocer mucha información (el nombre, la edad, sus aficiones, el color de los ojos, dónde trabajan, si les gusta la ópera, etc.), pero mucha de esta información no es relevante para el sistema que estamos desarrollando y, por lo tanto, no la debemos considerar como atributos de la clase.

En cambio, seguro que necesitaremos una manera de identificar a los usuarios (por ejemplo, utilizando un nombre que podría ser su nombre real o un alias), con el fin de poder saber quién ha creado un mensaje; del mismo modo, necesitaremos poder determinar que un usuario es quien dice ser (por ejemplo, mediante una palabra de paso). En este caso diríamos que la clase *Usuario* tiene dos atributos: *nombreUsuario* (el que utiliza para autenticarse en el sistema) y *palabraDePaso* (la contraseña).

Otros detalles del usuario, como el nombre, los apellidos y la edad, podrían ser atributos o no según los consideremos relevantes para el sistema que estemos describiendo. Para simplificar el ejemplo, los consideraremos no relevantes y, por lo tanto, no los convertiremos en atributos.

De los mensajes también podríamos saber muchas cosas, como la longitud del contenido, si es interesante o es aburrido, etc., pero nos quedaremos con dos atributos que consideramos relevantes: título y contenido.

Finalmente, de los foros nos interesará saber el nombre (para poder distinguirlos de los otros).

Por lo tanto, tenemos las clases y los atributos siguientes:

Clases y atributos

5.2. Herencia

En este caso, no es posible generalizar ninguna de las clases identificadas, ya que representan conceptos muy diferentes entre sí. En cambio, podría suceder que un análisis más completo de la funcionalidad nos llevara a pensar que algunos de los usuarios tienen permisos especiales y pueden borrar los mensajes

de otros usuarios. Como no todos los usuarios tienen este permiso especial, podríamos identificar un subconjunto de usuarios que denominaríamos *Moderadores*, que son aquellos que pueden borrar mensajes de otros usuarios.

En este caso, pues, habríamos añadido una clase nueva por especialización:

Especialización de Moderador

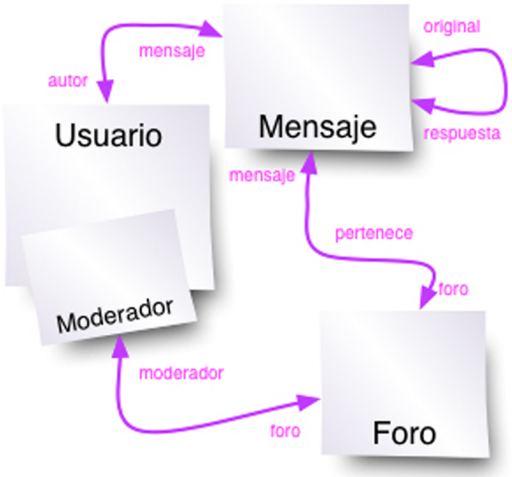
5.3. Asociaciones

Respecto a las asociaciones, podríamos identificar una asociación entre Usuario y Mensaje, en la que Usuario tiene el rol de autor y Mensaje el de mensaje y donde guardaremos qué usuario es el autor de un mensaje. También necesitaremos saber a qué foro pertenece un mensaje (denominaremos *pertenece* a esta asociación y *mensaje* y *foro*, a los roles del mensaje y del foro, respectivamente).

Algunos mensajes pueden ser respuestas a mensajes anteriores. Esta asociación es una asociación recursiva, ya que asocia dos instancias de la misma clase (Mensaje): una tiene el rol de mensaje original y la otra, el rol de respuesta.

Finalmente, también nos interesará saber quién es el moderador de un foro en concreto. En este caso, en vez de asociar Foro y Usuario asociaremos Foro y Moderador, ya que sabemos que sólo el subconjunto de usuarios que son moderadores están capacitados para moderar un foro.

Asociaciones



Resumen

Hemos visto cómo una herramienta que se creó como mecanismo de abstracción para los lenguajes de programación nos permite simplificar la descripción de cualquier sistema del mundo real. Por lo tanto, hemos de ver la orientación a objetos no sólo como un paradigma de programación, sino también como un paradigma de análisis que podemos aplicar en el mundo real.

La orientación a objetos nos proporciona una manera de abstraer las características importantes de las entidades que forman nuestro dominio, clasificarlas de manera que podamos maximizar la reutilización y variar el nivel de abstracción en el que trabajamos y, al mismo tiempo, establecer relaciones entre éstas mediante asociaciones.

Hemos visto cómo podemos clasificar los objetos en clases y hacer abstracción de los aspectos relevantes a la hora de describir las clases mediante las diferentes responsabilidades que se traducirán en atributos, asociaciones y operaciones.

También hemos visto cómo el encapsulamiento nos permite establecer la visibilidad de cada una de las responsabilidades y ocultar así los detalles de implementación en el resto de las clases.

Finalmente, la herencia nos permite establecer jerarquías de clase de tal manera que podemos describir los rasgos comunes a diferentes clases en una clase más general, mientras que el polimorfismo nos permite ver un mismo objeto como parte de una u otra clase de la jerarquía, en función del contexto.

Un modelo orientado a objetos de nuestro dominio nos facilitará, más adelante, la creación de clases de software que reflejen este dominio, al mismo tiempo que nos puede ayudar a la creación de otros modelos (como la estructura de la base de datos) y mantener, al mismo tiempo, la posibilidad de utilizarlo para comunicarnos con las personas expertas en el dominio, sin necesidad de que sean expertas en programación orientada a objetos.

Actividades

1. En el ejemplo del subapartado 2.3.1 hemos dicho que el teléfono es un atributo de tipo String. ¿Por qué no lo hemos indicado como de tipo numérico?

2. Estamos haciendo el análisis orientado a objetos para un sistema de comercio electrónico. La empresa venderá una serie de productos de los que tendremos un nombre, su marca comercial, su descripción, una fotografía y el precio. Algunos productos son electrónicos, en el sentido de que son documentos binarios que se pueden descargar y no objetos físicos que hay que enviar; para éstos también queremos tener el fichero y el formato, que puede ser música, libro o película. Cada producto podrá pertenecer a una o más categorías y cada uno tendrá un nombre y una descripción.

Los usuarios tendrán un nombre de usuario, una contraseña y una dirección y harán compras de las que guardaremos la fecha y el conjunto de productos comprados. De un mismo producto, en una compra, se podrán comprar una o más unidades. No tendremos en cuenta el pago.

- Elaborad una lista de las clases que podemos encontrar en este dominio.
- Para cada clase, indicad los atributos que tiene, el tipo de cada atributo y cuál es su número posible de valores.
- Identificad las asociaciones entre clases. Para cada una indicad los nombres de los roles y, si existen, los atributos que tenga la clase asociativa.
- Identificad las relaciones de herencia entre clases y qué clases creéis que son abstractas.

3. Tenemos un sistema orientado a objetos con las clases *Vehículo*, *Coche*, *Moto* y *Conductor*. La clase *Vehículo*, que es abstracta, tiene el atributo público *matrícula*, el atributo protegido *marca* y el atributo privado *modelo*. *Coche* y *Moto* son subclases de la clase *Vehículo*.

a) ¿Qué atributos de *Vehículo* son visibles desde la clase *Vehículo*?
b) ¿Y desde la clase *Coche*?
c) ¿Y desde *Persona*?

4. Tenemos un sistema orientado a objetos con las clases *Bien*, *BienInmueble* (que es subclase de *Bien*) y *Contribuyente*. Entre *Contribuyente* y *Bien* hay una asociación, de tal manera que un contribuyente tiene cualquier número de bienes y cada uno pertenece a uno o más contribuyentes. *Bien* tiene el atributo *valor*, mientras que *BienInmueble* tiene un atributo *referenciaCatastral*. *Bien* tiene una operación *calcularImpuestos* que calcula el impuesto para aplicar a los contribuyentes que tienen el bien asociado. *BienInmueble* redefine la operación *calcularImpuestos* y, además, una operación propia denominada *hipotecar*, con varios parámetros.

a) ¿Puede ocurrir que un contribuyente tenga asociado un bien inmueble?
b) ¿Un contribuyente podría pedir la operación *hipotecar* sobre todos sus bienes asociados?
c) Según este esquema, ¿los bienes inmuebles tienen valor?
d) ¿Los bienes tienen referencia catastral?

5. Suponed que tenemos el mismo sistema orientado a objetos del ejercicio anterior. Pero ahora sabemos que el atributo *valor* de la clase *Bien* es privado. ¿Los bienes inmuebles tienen valor ahora? ¿Desde la clase *BienInmueble* se puede acceder al atributo *valor*?

Ejercicios de autoevaluación

1. ¿Cuál es la diferencia fundamental entre los lenguajes de programación procedimentales y los orientados a objetos con respecto a la abstracción?

2. ¿Qué relación existe entre el análisis orientado a objetos y la programación orientada a objetos?

3. En orientación a objetos, ¿con qué criterio agrupa objetos el mecanismo de clasificación?

4. ¿Qué tipos de responsabilidades pueden tener los objetos?

5. ¿Qué mecanismo de la orientación a objetos nos permite descartar rasgos característicos de los objetos que no sean relevantes para el problema que queremos resolver?

6. ¿Qué elemento de la clase nos indica la información que conocemos de sus instancias?

7. ¿Qué información conocemos de los atributos de una clase, además del nombre?

8. ¿Qué significa que la relación de asociación es recíproca?

9. ¿Puede existir una asociación de una clase consigo misma? En tal caso, ¿cómo distinguimos una instancia de otra entre dos que están relacionadas?

10. En orientación a objetos, ¿cómo modelizamos las tareas que pueden llevar a cabo los objetos?

11. ¿Qué relación existe entre operaciones, métodos, interfaz e implementación?

12. ¿Qué otra metáfora se utiliza en orientación a objetos para hacer referencia a la invocación o llamamiento de una operación?

13. ¿Qué relación existe entre el encapsulamiento y la ocultación de la información?

14. ¿Qué visibilidades podemos definir para atributos y operaciones en un sistema orientado a objetos?

15. ¿Qué mecanismo de clasificación nos permite simplificar la descripción de las clases?

16. ¿Cómo podemos identificar relaciones de herencia en dominio?

17. ¿Qué mecanismo nos permite trabajar con un objeto en varios niveles de abstracción según nuestras necesidades?

18. ¿Qué es una clase abstracta?

19. ¿Una clase que no es abstracta puede tener operaciones abstractas? ¿Por qué?

20. ¿Una clase abstracta puede tener operaciones que no lo sean? ¿Por qué?

21. ¿Qué relación puede existir entre los conceptos de asociación y polimorfismo?

22. ¿Cómo podemos solucionar, en orientación a objetos, el caso en el que una asociación tiene información adicional que no depende de una de las instancias relacionadas, sino de todas?

Solucionario

Actividades

1. Los teléfonos son cadenas de caracteres que, aparentemente, sólo pueden contener números, pero no son números en el sentido de que no los percibimos como tales y no los utilizamos como tales en ningún momento. Así, por ejemplo, algunos rasgos que los distinguen de los números son:

- No es lo mismo el teléfono 003 que el 3, a pesar de que sí son el mismo número.
- Los teléfonos no admiten ninguna de las operaciones habituales de los números. Nunca sumamos, restamos, multiplicamos, dividimos, etc. teléfonos, aunque sí lo hacemos con los números.
- Algunos caracteres no numéricos pueden formar parte de un número de teléfono. Como mínimo, el carácter +, que podemos utilizar para señalizar el prefijo internacional como un número de teléfono y que realmente se indica al marcar el número.

2. Clases que podemos encontrar y atributos:

- Producto: nombre:String, marcaComercial:String, descripcion:String, fotografia:Imagen, precio:Importe
- Producto electrónico: fichero:Fichero, formato:Formato
- Categoría: nombre:String, descripcion:String
- Usuario: nombreUsuario:String, contraseña:String, direccion:Direccion
- Compra: fecha:Fecha

Asociaciones:

- PerteneceA: cada producto (con nombre de rol *productos*) pertenece a una o más categorías (con nombre de rol categorías).
- Hace: cada usuario (nombre de rol *usuario*) realiza una serie de compras (nombre de rol *compras*).
- Incluye: una compra (nombre de rol *compra*) incluye una serie de productos (nombre de rol *productos*). Ésta es una clase asociativa con el atributo *numUnidades:nat*.

Existe una relación de herencia entre *Producto* y *Producto electrónico*, de tal manera que *Producto electrónico* es una subclase de *Producto*. Sin embargo, *Producto* no es una clase abstracta, ya que hay productos que no son electrónicos.

3.

a) Todos.
b) Matrícula y marca, ya que *marca* es protegido y visible desde las subclases, pero *modelo* es privado y no es visible desde ninguna otra clase que no sea *Vehículo*.
c) *Matrícula*, ya que *marca* y *modelo* no son públicos.

4.

a) Sí. Un contribuyente puede tener asociado cualquier número de bienes y los bienes inmuebles también son bienes.
b) No, ya que los bienes que no son inmuebles no se pueden hipotecar; es decir, la operación sólo está definida para la clase *BienInmueble* y, por lo tanto, sólo se puede invocar sobre instancias de esta clase.
c) Sí. Todo bien inmueble es un bien y, por lo tanto, como todos los bienes, tiene un valor. Visto desde el punto de vista de la herencia, la clase *BienInmueble* hereda el atributo valor de la clase *Bien*.
d) En general, no. Sólo aquellos bienes que sean inmuebles tendrán referencia catastral. Pero todos los otros no tendrán.

5. La visibilidad de un atributo no cambia el comportamiento con respecto a la herencia. Por lo tanto, si *valor* es un atributo privado, todos los bienes continuarán teniendo valor, incluyendo los bienes inmuebles. Ahora bien, como el atributo *valor* es ahora privado, aunque los bienes inmuebles también tengan, no podemos acceder a ellos desde la clase *BienInmueble*.

Ejercicios de autoevaluación

1. Los lenguajes de programación procedimentales hacen una abstracción de lo que la máquina puede entender, mientras que los lenguajes de programación orientados a objetos realizan una abstracción de lo que las personas quieren comunicar. (Podéis ver el subapartado 1.2).

2. El análisis orientado a objetos consiste en aplicar los conceptos y las ideas de los lenguajes de programación orientados a objetos al análisis de sistemas. Por lo tanto, el análisis orien-

tado a objetos utiliza el mismo paradigma, la orientación a objetos, que la programación orientada a objetos. (Podéis ver el subapartado 1.3).

3. El criterio por el que la clasificación agrupa objetos en clases es el conjunto de responsabilidades en común que tienen estos objetos. Así, todos los objetos de una misma clase tienen unas mismas responsabilidades. (Podéis ver el subapartado 2.1).

4. Un objeto puede tener responsabilidades de datos (cosas que debe saber), asociaciones (otros objetos que conoce) y comportamiento (tareas que se le pueden pedir). (Podéis ver el subapartado 2.1).

5. La abstracción. Es el mecanismo por el que decidimos cuáles de las posibles responsabilidades asociadas a una clase formarán parte de su definición. (Podéis ver el subapartado 2.2).

6. Los atributos. (Podéis ver el subapartado 2.3.1).

7. Los atributos suelen tener un tipo, que nos ayuda a entender el tipo de información que se almacena, un ámbito (que puede ser de instancia o de clase) y un número de valores (puede ser o no opcional y puede ser o no multivaluado). (Podéis ver el subapartado 2.3.1).

8. Significa que si una instancia a_1 de una clase A está asociada con una instancia b_1 de la clase B, la instancia b_1 estará también asociada a la instancia a_1. (Podéis ver el subapartado 2.3.2).

9. Sí, y se denomina *asociación recursiva*. Para distinguir el rol que tiene cada instancia de la asociación utilizamos los roles de asociación. (Podéis ver el apartado 2.3.2).

10. Mediante operaciones. (Podéis ver el subapartado 2.3.3).

11. Las operaciones indican qué tareas puede llevar a cabo un objeto y forman la interfaz del objeto. Cada método es el conjunto de instrucciones que indica cómo un objeto lleva a cabo una tarea (una operación) y forma la implementación del objeto. (Podéis ver el subapartado 2.3.3).

12. También se utiliza la metáfora del envío de un mensaje. Cuando un objeto invoca o llama a una operación de otro objeto también podemos decir que le envía un mensaje. (Podéis ver el subapartado 2.3.3).

13. El encapsulamiento es el mecanismo de la orientación a objetos que nos permite ocultar información.

14. Visibilidad pública (cualquier objeto tiene acceso), privada (sólo los objetos de la misma clase), protegida (sólo los objetos de la misma clase o de alguna subclase) y de paquete (sólo los objetos del mismo grupo o paquete). (Podéis ver el subapartado 3.2).

15. La herencia. Nos permite simplificar la descripción de una clase identificando responsabilidades comunes entre las instancias de diferentes clases y evitando la obligación de describirlas en más de una clase. (Podéis ver el apartado 4.1).

16. Mediante la generalización (encontramos clases más generales a partir de las que ya tenemos) o la especialización (encontramos clases más específicas a partir de las que ya tenemos). (Podéis ver el subapartado 4.1.1).

17. El polimorfismo. Es la capacidad de trabajar con un objeto con formas diferentes (interfaces diferentes) según nuestras necesidades. (Podéis ver el subapartado 4.2).

18. Una clase abstracta es una clase que no podemos instanciar directamente; es decir, sólo podemos crear instancias de alguna de sus subclases. (Podéis ver el subapartado 4.2.1).

19. Una clase no abstracta no puede tener operaciones abstractas porque, para éstas, el comportamiento (el método) se debe definir en las subclases. Por lo tanto, los objetos de la clase no tendrán ningún comportamiento asociado, a no ser que pertenezcan a una subclase. Por ello, a la fuerza, deberían pertenecer a una subclase; es decir, la clase debería ser abstracta. (Podéis ver el subapartado 4.2.1).

20. Una clase abstracta puede tener operaciones que no lo sean. Sencillamente, sus subclases heredarán las operaciones con su comportamiento y lo podrán redefinir o dejarlo tal cual. (Podéis ver el subapartado 4.2.1).

21. Podemos tener asociaciones polimórficas: una asociación polimórfica es una asociación con una clase que tiene subclases. En este caso, un objeto puede tener asociados, por una

misma asociación, objetos de la clase asociada y de sus subclases. Por lo tanto, tendrá asociados objetos de varias clases (con una superclase en común). (Podéis ver el subapartado 4.2.2).

22. Utilizamos una clase asociativa, es decir, una clase cuyas instancias son instancias de la asociación. (Podéis ver el apartado 4.3.2).

Glosario

abstracción *f* El acto de reunir las cualidades esenciales o generales de cosas similares. También las características esenciales provenientes de una cosa.

abstracta (clase) *f* Clase que no se puede instanciar directamente; sólo se pueden instanciar las subclases no abstractas que tenga.

abstracta (operación) *f* Operación que no tiene asociado un método; es decir, para la que no se ha definido un comportamiento.

ámbito (de un atributo) *m* Característica que distingue si un atributo almacena un valor diferente para cada instancia (ámbito de instancia) o un mismo valor para todas las instancias de una clase (ámbito de clase).

análisis orientado a objetos *f* Aplicación de los conceptos e ideas de los lenguajes orientados a objetos al análisis de sistemas.

asociación *f* Relación estructural entre dos (o más) clases que nos indica qué objetos de la segunda clase conocen los objetos de la primera. Una asociación, por lo tanto, constituye una responsabilidad de cada una de las clases implicadas.

asociación polimórfica *f* Ved **polimórfica (asociación)**

asociación recursiva *f* Asociación que conecta una clase con ella misma.

atributo *m* Elemento estructural de una clase que indica qué información tienen las instancias de aquella clase; por lo tanto, es una de las responsabilidades de la clase.

caja blanca *f* Tipo de descripción de un objeto en el que se describe, además de cualquier aspecto externo del objeto, su estructura y funcionamiento internos.

caja negra *f* Tipo de descripción de un objeto en el que se describe su funcionamiento desde un punto de vista externo, sin explicar la estructura ni el funcionamiento internos.

clase abstracta *f* Ved **abstracta (clase)**

clase asociativa *f* Clase cuyas instancias pertenecen a una asociación; es decir, cada instancia de una clase asociativa representa una instancia de una asociación, en el sentido de que está formada por el par (en el caso de asociaciones binarias) de instancias asociadas, además de tener los atributos, las asociaciones y las operaciones propias.

clase (de objetos) *f* Descriptor de un conjunto de objetos que comparten los mismos atributos, asociaciones, operaciones y métodos.

clasificación *f* Mecanismo de identificar clases de objetos como descriptoras de conjuntos de objetos con las mismas responsabilidades.

código máquina *m* Código formado por el conjunto de instrucciones que los procesadores (en general, el hardware) pueden entender y ejecutar.

de paquete (visibilidad) Visibilidad de un atributo, asociación u operación tal que sólo es visible por los objetos del mismo grupo o paquete.

encapsulamiento *m* Mecanismo de la orientación a objetos que nos permite ocultar información definiendo qué atributos, operaciones y asociaciones de un objeto son visibles y cuáles quedan ocultos.

especialización *f* Proceso de identificación de una subclase de una clase existente que sirve para describir las responsabilidades comunes a un subconjunto de las instancias de la clase original, pero no a todas.

generalización *f* Proceso de identificación de una superclase de dos o más clases ya existentes que sirve para describir las responsabilidades comunes a las clases ya identificadas inicialmente.

herencia *f* Característica de la orientación a objetos por la que podemos identificar una relación entre una superclase y una subclase, de manera que todas las instancias de la subclase lo son también de la superclase. Decimos que la subclase hereda la definición de la superclase, por lo que para definir la subclase sólo debemos indicar aquellas características que le son específicas y que no están presentes en la superclase.

implementación (de una clase) *f* Comportamiento interno de las instancias de una clase que queda definido por el conjunto de métodos de la clase. La implementación queda oculta detrás de la interfaz.

instancia *f* Ved **objeto**

interfaz *f* Conjunto de operaciones ofrecidas por las instancias de una clase y que, por lo tanto, pueden ser invocadas por otros objetos. Sin embargo, la interfaz no incluye la implementación, que está formada por los métodos que implementan estas operaciones.

invocar *v* Pedir a un objeto la ejecución de su comportamiento para una determinada operación.

lenguaje ensamblador *m* Lenguaje de programación en el que las instrucciones se corresponden prácticamente una a una con las operaciones que puede entender el procesador. Se traduce a código máquina para ser ejecutado.

lenguaje de programación *m* Lenguaje artificial que está diseñado para expresar los cómputos que queremos que lleve a cabo un ordenador.

lenguaje de programación de alto nivel *m* Lenguaje de programación más rico que el lenguaje ensamblador, ya que permite expresar las instrucciones con un nivel de abstracción más alto.

lenguaje de programación orientado a objetos *m* Lenguaje de programación de alto nivel que eleva el nivel de abstracción para crear una abstracción del problema que se quiere resolver utilizando el paradigma de la orientación a objetos.
Ved **orientación a objetos**

lenguaje natural *m* Lenguaje que utilizamos las personas para comunicarnos entre nosotros (como el español o el inglés), por oposición a los lenguajes que utilizamos para comunicarnos con los ordenadores (como los lenguajes de programación).

lenguaje procedimental *m* Lenguaje de programación de alto nivel que eleva el nivel de abstracción y crea una abstracción de la máquina sobre la que se ejecutan. Permite utilizar un juego de instrucciones más abstractas de las que el procesador es directamente capaz de ejecutar.

lenguaje unificado de modelización *m* Ved ***universal modeling language***

llamar *v* Ved **invocar**

mensaje (orientación a objetos) *m* Metáfora que representa un llamamiento o invocación de una operación de un objeto. Bajo esta metáfora decimos que el objeto que hace el llamamiento (el que pide la ejecución de la operación) envía un mensaje al objeto que recibe el llamamiento (el que debe ejecutar su comportamiento para aquella operación).

método *m* Conjunto de instrucciones que para una operación, en una clase, indica cómo se comportarán los objetos de esa clase. El conjunto de los métodos de la clase forma la implementación.

multivaluado (atributo) *m* Atributo que, para una determinada instancia, puede tener más de un valor en lugar de estar obligado a tener sólo uno. Antónimo de *univaluado*.

objeto *m* Cada uno de los elementos autónomos que forma un sistema orientado a objetos.

obligatorio (atributo) *m* Atributo en el que todas las instancias de la clase deben tener valor. Antónimo de *opcional*.

ocultación de información *f* Capacidad de hacer que cada elemento de un sistema contenga cierta información que se oculta al resto de los elementos y les muestra una visión simplificada. En el caso de las clases de un sistema orientado a objetos, el encapsulamiento permite indicar qué elementos son visibles y ocultar el resto.

opcional (atributo) *m* Atributo que, para una determinada instancia, puede no tener valor. Si un atributo no es opcional, entonces debe tener valor para todas las instancias de la clase. Antónimo de *obligatorio*.

operación *f* Especificación de una tarea que pueden llevar a cabo todas las instancias de una clase. La operación se puede pedir mediante el llamamiento o la invocación y el com-

portamiento de la instancia que reciba el llamamiento se define en un método. Por lo tanto, es una de las responsabilidades de la clase.

operación abstracta *f* Ved **abstracta (operación)**

operación polimórfica *f* Ved **polimórfica (operación)**

orientación a objetos *f* Paradigma consistente en descomponer un sistema complejo en un conjunto de elementos autónomos (los objetos) que se comunican unos con otros para llevar a cabo sus tareas.

paquete *m* Agrupación de elementos en un sistema orientado a objetos para facilitar su organización. Un paquete puede contener clases y otros paquetes y puede resultar útil para definir la visibilidad de paquete de los atributos, las asociaciones y las operaciones de las clases que contiene.

polimórfica (asociación) *f* Asociación en la que una (o más) de las clases que participan tiene subclases, de tal manera que un objeto puede tener asociados, por la misma asociación, objetos pertenecientes a diferentes clases (siempre que pertenezcan a la jerarquía de herencia de la clase asociada).

polimórfica (operación) *f* Operación que tiene un método asociado a una clase pero cuyo método es diferente para alguna de las subclases de la clase. De esta manera, las diferentes instancias de la clase, al ejecutar la operación, se comportarán de manera diferente en función de si son instancias sólo de la clase o de qué subclase sean instancias.

polimorfismo *m* Capacidad de un objeto de presentarse con formas diferentes (interfaces diferentes) según el contexto. Así, un objeto que pertenezca a una clase cualquiera se puede ver como perteneciente a aquella clase, a la superclase de ésta, a la superclase de la superclase de aquélla, etc.

privada (visibilidad) *f* Visibilidad de un atributo, asociación u operación que sólo es visible para la misma clase en la que está definida.

protegida (visibilidad) *f* Visibilidad de un atributo, asociación u operación que sólo es visible para la clase en la que está definida y para sus subclases.

pública (visibilidad) *f* Visibilidad de un atributo, asociación u operación que es visible para todos los objetos del sistema.

redefinir (una operación) *v* Definir, para una operación de una subclase, un método (un comportamiento) diferente del método que tenía asociada aquella operación a la superclase. Al redefinir una operación en una subclase la convertimos en una operación polimórfica.

responsabilidad (de un objeto) *f* Características u obligaciones de una clase, que pueden ser información que las instancias deben conocer (atributos), otras instancias que han de conocer (asociaciones) o tareas que deben poder llevar a cabo (operaciones).

rol *m* Extremo de una asociación al que se asigna un nombre para indicar su propósito; es decir, para indicar cuál es el papel que tienen las instancias cuando se asocian.

subclase *f* En una relación de herencia entre dos clases, la clase que hereda las responsabilidades de la otra clase; por lo tanto, es la clase más específica.

superclase *f* En una relación de herencia entre dos clases, la clase de la que se heredan las responsabilidades; por lo tanto, es la clase más genérica.

tipo (de un atributo) *m* Descripción del conjunto de valores que puede tomar un atributo. Algunos ejemplos de tipo de atributos pueden ser el tipo fecha (que define un conjunto de valores que incluye 1/3/2011 o 31/3/1414), entero (como 14 o 356) o carácter (como "a" o "@").

unified modeling language *m* Lenguaje de modelización de propósito general estandarizado por el OMG y utilizado en el campo de la ingeniería del software.
Sigla **UML**

univaluado (atributo) *m* Atributo tal que ninguna instancia de la clase puede tener más de un valor en ese atributo. Antónimo de *multivaluado*.

visibilidad *f* Propiedad de un atributo, asociación u operación que define qué objetos pueden verlo o verla. La visibilidad puede ser pública, privada, protegida o de paquete.

Bibliografía

La bibliografía sobre orientación a objetos está muy relacionada con el mundo de la programación.

Meyer, Bertrand (2002). *Construcción de software orientado a objetos*. Madrid: Prentice Hall.

Este libro es la referencia académica en orientación a objetos y, por lo tanto, es muy adecuado para el estudio detallado de los diferentes conceptos.

Bibliografía complementaria

Muchas guías y libros sobre programación en un lenguaje orientado a objetos incluyen una sección introductoria sobre la programación a objetos. A continuación os recomendamos dos:

Eckel, Bruce (2006). *Thining in Java*. Prentice Hall.

Object Oriented Programming with Objective C (Apple Corp.) http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/OOP_ObjC/OOP_ObjC.pdf

Referencias bibliográficas

Parnas, David L. (1972). "On the Criteria to Be Used in Decomposing Systems Into Modules". *Communications of the ACM*.