Introducción al paradigma de la programación orientada a objetos

PID_00269659

David García Solórzano

Tiempo mínimo de dedicación recomendado: 2 horas

Universitat Oberta de Catalunya

David García Solórzano

Graduado Superior en Ingeniería en Multimedia e Ingeniero en Informática por la Universitat Ramon Llull desde 2007 y 2008, respectivamente. Es también Doctor por la Universitat Oberta de Catalunya desde 2013, donde realizó una tesis doctoral relacionada con el ámbito del e-learning. Desde 2008 es profesor de la Universitat Oberta de Catalunya en los Estudios de Informática, Multimedia y Telecomunicación.

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: David García Solórzano (2020)

Primera edición: febrero 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

Este módulo pretende situarnos en el punto de partida desde el que empezarán estos materiales docentes, que no es otro que el paradigma de la programación estructurada que ya deberíamos conocer (y dominar). Una vez que estemos situados, describiremos las ventajas del paradigma de la programación orientada a objetos (POO) respecto al paradigma de la programación estructurada. Además, explicaremos los elementos básicos de la POO y veremos cómo se debe pensar a la hora de diseñar una solución (es decir, un programa) siguiendo el paradigma de la POO.

Este módulo es introductorio y, por consiguiente, muchos de los conceptos que se comentan en él serán vistos con más detalle en futuros módulos.

Estos materiales didácticos no son:

- de algorítmica (damos por hecho que tienes experiencia previa en programación estructurada y conoces perfectamente qué es un condicional, un `int`, un `boolean`, un bucle, un *array*, una función, un paso por valor y por referencia, un puntero, una pila, una cola, una lista, etc.),

- principalmente de programación, como podrían ser los materiales de una asignatura inicial,

- únicamente de diseño de programas, como podrían ser los materiales de una asignatura de ingeniería de software,

- de Java, ni C#, ni C++, ni Python, ni PHP, ni Scala... No son de ningún lenguaje de programación orientado a objetos y lo son de todos.

Estos materiales son un paso más allá, ya que se centran en la enseñanza de un paradigma, concretamente del paradigma de la programación orientada a objetos (POO). Por lo tanto, estos materiales:

- combinan conocimientos de diseño y programación para dar una solución software a problemas reales,

- explican conceptos generales de la POO que sirven para «cualquier» lenguaje de programación que soporte dicho paradigma (veremos que hay características que a veces no son soportadas por todos los lenguajes o si las soportan, lo hacen con matices),

- intentan, tanto como sea posible, alejarse de la codificación para ahondar en los conceptos, en las ideas y, en definitiva, en la filosofía que hay detrás de cada aspecto de la POO,

- si muestran código, utilizan un pseudocódigo que simplifica la verbosidad de los lenguajes de programación para resaltar los conceptos más que la sintaxis a emplear. No obstante, hay aspectos cuya codificación sí es mostrada en diferentes lenguajes reales, predominando los ejemplos en Java, C# y C++.

Aclarado qué son estos materiales y qué no, te formulamos una pregunta: ¿Estás dispuesto a trabajar estos materiales casi a diario y durante meses? Esperamos que tu respuesta haya sido «sí», sin condiciones ni peros. Nosotros nos comprometemos a explicarte las cosas de la manera más sencilla y amena que seamos capaces (¡eso sí, tú también tendrás que poner de tu parte!). Para lograrlo, sobre todo al principio, seguiremos a la familia POO –ya te la presentaremos– en su vida cotidiana para poder hacer analogías del mundo real con algunos conceptos de la POO.

Por otro lado decirte que, como ya hemos dicho, estos materiales son principalmente teóricos. ¿Y la práctica? La única manera de trasladar los conocimientos teóricos a software real es utilizando un lenguaje programación concreto. Por este motivo, junto a estos materiales didácticos, te facilitamos una guía que explica el lenguaje de programación escogido durante el semestre y que detalla cómo los conceptos vistos en estos materiales didácticos son puestos en práctica.

Finalmente, esperamos que disfrutes del aprendizaje de un paradigma tan potente como es la programación orientada a objetos. Sabemos que es un reto que exige mucho esfuerzo por tu parte, pero te aseguramos que la recompensa final lo merece.

Objetivos

El objetivo principal de este módulo es contextualizar el material docente, concretamente:

1. Ser consciente del punto de partida de estos materiales: el paradigma de la programación estructurada.

2. Conocer el concepto de *paradigma de programación* haciendo énfasis en la programación estructurada –ya conocida por el estudiante– y su evolución hacia la programación orientada a objetos (POO) –paradigma que se estudiará en estos materiales.

3. Comprender de modo general los beneficios que nos aporta el paradigma de la programación orientada a objetos.

4. Entender los elementos básicos de la POO, así como el razonamiento, en líneas generales, que se sigue al diseñar un programa mediante el paradigma POO.

5. Ser consciente de las dificultades a las que un programador novel se afronta al intentar aprender el paradigma de la programación orientada a objetos y la actitud (y esfuerzo) que este exige.

1. Paradigma de la programación estructurada: ¿de dónde venimos?

Si estás leyendo estas líneas es porque tienes nociones básicas de programación. Si es así, entonces seguro que eres consciente de que un programa debe ser diseñado cuidadosamente (es decir, pensado con detenimiento) antes de codificarlo. Esto es más cierto cuanto más compleja es la tarea que debe realizar el programa.

Para garantizar que los programas complejos funcionan –es decir, hacen lo que se espera de ellos–, son fiables y están bien codificados apareció, a finales de los años cincuenta, una nueva disciplina llamada **ingeniería del software**. Esta disciplina propone métodos y teorías que sirven tanto para analizar problemas complejos como para diseñar programas que resuelvan dichos problemas. A cada conjunto de métodos y teorías que determinan una manera específica de analizar y resolver un problema se le llama *enfoque de programación* o, más formalmente, **paradigma de programación**.

Definición de paradigma de programación

Un paradigma de programación es una colección de conceptos, métodos y teorías que guían el proceso de construcción de un programa determinando su estructura.

Así pues, un paradigma de programación determina la manera en que pensamos y formulamos los programas.

En los años setenta, el paradigma estrella era el llamado **programación estructurada**. Corrado Böhm y Giuseppe Jacopini formalizaron este paradigma en 1966. La programación estructurada se basa en seguir un enfoque descendiente (*top-down*) para diseñar un programa que resuelva un problema complejo. Para ello, se descompone el problema o programa inicial en varios problemas o programas más pequeños y sencillos llamados *subproblemas*. Cada subproblema se trabaja individualmente y se considera un nuevo problema que puede ser, a su vez, descompuesto en problemas más pequeños y así sucesivamente. Finalmente, se llega a problemas tan sencillos que pueden ser resueltos directamente sin necesidad de descomponer más.

Este enfoque *top-down* implica tener diferentes niveles de descomposición que van de lo general –el problema complejo inicial– a lo particular –el subproblema de más bajo nivel de descomposición.

En este punto, si decimos que el problema complejo inicial es el programa, entonces podemos también decir que un subproblema es un subprograma. A los subprogramas del primer nivel de descomposición se les suele llamar **módulos**. Los niveles de descomposición posteriores a los módulos están formados por **funciones** y **acciones** –es decir, funciones que no devuelven nada.

Definición de módulo

Un módulo es una colección que agrupa constantes, variables, funciones y acciones relacionadas entre sí.

El número de módulos, así como de funciones y acciones, que tiene un programa depende de cada programador. Cada módulo suele encargarse de una funcionalidad del programa y estos pueden interactuar entre sí. Así pues, si nuestro programa es un videojuego de baloncesto, el módulo de física le dirá al módulo de pintado dónde dibujar la pelota.

El paradigma de la programación estructurada tiene las siguientes ventajas respecto a no seguir ningún paradigma:

- **Mejora la legibilidad del código:** tener el código organizado en módulos que siguen una cierta lógica y en funciones de pocas líneas que hacen una tarea concreta es mejor que tener un único módulo con miles de líneas de código. Asimismo, los programas quedan mejor documentados internamente.

- **Facilita el trabajo en grupo:** varios programadores pueden trabajar en el mismo programa simultáneamente encargándose cada uno de ellos de un módulo concreto.

- **Aumento de la productividad:** como el código está organizado en módulos, se puede aplazar el desarrollo de algunas tareas, destinar más programadores a un módulo concreto según las necesidades del momento y aumentar así la productividad del equipo.

- **Mantenimiento más sencillo:** debido a que la lógica del programa es más visible gracias a su propia estructuración, los errores se pueden localizar y corregir más fácilmente. Así pues, si el videojuego de baloncesto mencionado anteriormente falla a la hora de pintar la pelota, sabemos que el error está en alguna función o acción del módulo de pintado. Además, si se requiere modificar algún segmento de código (p. ej. un módulo), esto se puede hacer a menudo sin tocar el resto del programa.

- **Codificación más simple:** la propia organización o estructura del programa implica que la codificación sea más fácil para el programador, puesto

que en cada momento se debe centrar en resolver programas o problemas sencillos y pequeños.

Es importante tener presente que el grado de éxito que podemos tener al seguir el paradigma de la programación estructurada es directamente proporcional a la capacidad de **abstracción** que seamos capaces de aplicar al problema complejo al que nos enfrentamos. Nuestra capacidad de abstracción nos permite reconocer los diferentes subproblemas del problema complejo inicial.

Todo lo explicado hasta ahora te tendría que ser familiar, puesto que a día de hoy ya deberías haber implementado programas siguiendo el paradigma de la programación estructurada utilizando un lenguaje afín a dicho paradigma, como es C. Si no te acuerdas de si has usado o no el paradigma de la programación estructurada, contesta a estas preguntas:

- **¿He usado módulos?** Sí. Cuando escribías un programa (p. ej., en C, PHP, JavaScript, etc.) organizabas el código en ficheros (con extensión `.c`, `.php`, `.js`, etc.) que contenían variables y funciones.

- **¿He seguido un enfoque descendiente (*top-down*)?** Sí. Además de módulos, cuando escribías una función `F` y esta hacía muchas tareas, la subdividías en funciones más sencillas que llamabas desde la función `F`.

- **¿He aplicado abstracción?** Sí. En cada nivel de descomposición eras capaz de considerar solo aquellos aspectos necesarios para analizar el problema y tomar decisiones. Por ejemplo, decidías sobre cuestiones como: en cuántos módulos dividir el programa, qué hacía cada módulo, qué funciones implementabas en cada módulo, qué cabecera o firma tenía cada función, qué variables necesitabas y de qué tipo eran, etc.

Si has contestado «sí» a las tres preguntas anteriores, entonces estás preparado para seguir leyendo. En caso contrario, sería recomendable que repasaras todo lo que has hecho de programación hasta la fecha.

2. Paradigma de la programación orientada a objetos: ¿adónde vamos?

Si bien la programación estructurada es útil, la complejidad que han ido adquiriendo los programas a lo largo de los años ha obligado a pensar nuevos paradigmas. Así es como en los años ochenta irrumpió con fuerza un nuevo paradigma llamado **programación orientada a objetos (POO)** –*object-oriented programming* (OOP).

Para entender el porqué del éxito de la POO hay que pensar en los principales problemas que tiene la programación estructurada y cómo la POO los solventa.

2.1. Reutilización de código

Vamos a ver dos ejemplos cotidianos con la familia POO relacionados con la reutilización:

Ejemplo 1 (Reutilización) – La CPU estropeada

A David, padre de la familia POO, se le ha estropeado la CPU de su ordenador. ¿Qué hace? Va a una tienda que vende componentes hardware, compra una CPU nueva compatible con el resto de componentes de su ordenador, se va a casa y cambia la CPU estropeada por la nueva. Enciende el ordenador y ¡funciona!

Ejemplo 2 (Reutilización) – La silla del coche

Elena, madre de la familia POO, se ha llevado el coche sin acordarse de que David tiene que llevar a su hija Marina a clases de natación a 10 km de casa y no hay un transporte público que les deje cerca. Por suerte, David tiene una silla del grupo 1-2 en el trastero. ¿Qué hace? Llama al abuelo Manel que va con su coche a buscarlos, instalan la silla para niños en los asientos traseros del vehículo y ¡solucionado!, Marina no se perderá su clase de natación.

Ahora piensa, ¿podemos reutilizar el código o software que hemos escrito en el pasado en otro programa tan fácilmente como se reutiliza el hardware? La respuesta es «no». La programación estructurada se centra en los procedimientos y es la función la unidad básica. Las funciones son poco reusables, ya que es muy difícil copiar una función de un programa a otro y que funcione a la primera. Por ejemplo, muchas funciones llaman a otras funciones del módulo en el que están –incluso de otros módulos– y que también se tendrían que llamar (y copiar) en el nuevo programa. Algunas funciones incluso hacen uso de variables globales –sí, ya sabemos que este tipo de variables no son adecuadas, pero son muy frecuentes– y fallarían en su ejecución en el nuevo programa. Así pues, las funciones no son adecuadas en cuanto a reutilización se refiere.

En cambio, **la POO favorece la reutilización de código** en otros programas mediante el uso de clases. Como veremos, una clase combina o encapsula en un mismo elemento estructuras de datos (i.e. variables) y algoritmos (i.e. funciones). Así pues, gracias a la POO, no necesitamos reescribir las mismas

funciones una y otra vez en diferentes programas, sino que podemos reusar un código –es decir, una clase– ya existente –completamente probado– en otros programas, de manera fácil, rápida y segura.

Una de las maneras de facilitar la reutilización, como veremos en el último módulo de estos materiales, es mediante el mecanismo de *herencia*. La herencia permite básicamente dos cosas:

- Crear un módulo –mejor dicho, una clase– a partir de una ya existente y extender así las funcionalidades de la clase reutilizada sin afectarla.

- Definir y utilizar de forma clara clases funcionalmente incompletas, pero que ayudan a definir la solución al problema.

2.2. Abstracción a alto nivel

Los lenguajes de programación vinculados al paradigma estructurado, como C o Pascal, obligan al programador a pensar en términos propios del ordenador (p. ej., uso de la memoria, bucles, condicionales, etc.) más que en términos del problema que intenta resolver. Es decir, el nivel de abstracción al que llegan es muy bajo y se queda a nivel de sentencias de programación. Cuántas veces hemos pensado: ¿qué uso, un `if` o `switch`? ¿un `for` o un `while`? ¿un `int` o un `float`? Como vemos, nuestros programas hasta la fecha han sido tan sencillos que la mayoría de nuestras decisiones se han centrado más en el código en sí que en la funcionalidad del programa.

Sin embargo, los lenguajes de programación orientados a objetos fuerzan a que el programador se centre en el problema que hay que resolver, en vez de en el código. Así pues, el diseño del programa o software gana mucho protagonismo, lo que obliga al programador a pensar en un nivel de abstracción más alto. El hecho de que el programador dedique gran parte de su tiempo en el diseño del software lleva a un desarrollo más productivo y de mayor calidad.

Con la experiencia verás que es mucho más natural y acorde con la realidad pensar un problema en términos de clases u objetos (como hace la POO) que pensar en funciones (como hace la programación estructurada). Así pues, en la POO la manera de diseñar un software cambia de *top-down* a *bottom-up*; de hecho, se parte de entidades reales existentes en el problema que hay que resolver y son «traducidas» a entidades software que interactúan entre sí. No es sencillo hacer este cambio en nuestra manera de pensar y diseñar el software, pero poco a poco, con la práctica, se va acostumbrando nuestra mente.

2.3. Seguridad

Muchos módulos de programas basados en el paradigma de la programación estructurada incluyen variables globales que pueden ser usadas por otros módulos, o bien accediendo a ellas directamente, o bien mediante funciones proporcionadas por el módulo que las declara. Al comienzo de los ochenta, se popularizó la segunda forma de acceder, es decir, los módulos de un programa no accedían directamente a las variables de otro módulo, sino que lo hacían mediante funciones proporcionadas por el módulo que contenía las variables que se iban a usar. Esto tiene una serie de ventajas respecto a acceder directamente, como por ejemplo:

- Se asegura la **consistencia** de la variable. Por ejemplo, si una variable es un *array*, al usar una función para acceder a él, será dicha función quien compruebe si el índice al que se desea acceder (pasado por parámetros) es correcto o no. Además, se garantiza que todos los accesos al *array* hechos mediante la función se harán de igual manera y se realizarán las mismas comprobaciones.

- Se **exime de responsabilidades** al programador que utiliza el módulo, ya que, al usar las funciones proporcionadas, la responsabilidad recae en quien o quienes han programado dichas funciones.

- **Facilidad de mantenimiento**. Si se cambia el tipo de variable o existe algún error en una de las funciones proporcionadas por el módulo, solo hay que cambiar el código del módulo que la contiene. De este modo, los cambios se ven reflejados automáticamente en los módulos que la utilicen.

Como podemos apreciar, poco a poco surgió la tendencia de ir ocultando información acerca de la implementación de los módulos, o, dicho de otro modo, de hacer de un módulo una caja negra de la que se supiera lo mínimo para que un tercero la pudiera usar. Este hábito de ocultar información y controlar el acceso a ella –especialmente datos, es decir, variables– está íntimamente relacionado con los conceptos de **encapsulación** (una de las principales características de la POO) y **ocultación de información (o niveles o modificadores de acceso)**. Así pues, el programador conoce una interfaz bien definida que indica qué funciones contiene un módulo (en POO será una clase) y que, por lo tanto, puede utilizar o llamar. Veamos un ejemplo con el paradigma de la programación estructurada:

Ejemplo 3 (Encapsulación) – ¿Qué preferís...?

Imaginemos que necesitamos usar un módulo hecho por una tercera persona en nuestro programa. Pensemos, en este contexto, qué es lo más importante para nosotros: ¿saber cómo está implementado dicho módulo por dentro o saber qué es lo que podemos hacer con él y cómo utilizarlo? Efectivamente, lo segundo. A cualquier programador que usa código de un tercero lo que le interesa saber es qué funciones puede usar, qué hacen, cómo se usan (es decir, cómo se llaman) y qué devuelven. Por el contrario, no le interesa lo más mínimo si el programador que hizo el módulo al escribir una función utilizó una serie de sentencias `if` o prefirió un `switch`, o si ha usado un `while` en vez de un `for`, etc.

> Así pues, lo ideal para nosotros es incluir el nuevo módulo a nuestro programa, que nos digan que existe una función llamada `parOImpar` que recibe un número e imprime por pantalla si el número es par o impar. Nos da igual cómo dicha función logra saber si el número pasado como argumento es par o impar, lo que nos importa es que en algún lugar de nuestro código haremos `parOImpar(2)` y por pantalla aparecerá "`2 es par`".

Dadas las ventajas, los programadores tenían cada vez más interés en poder escribir módulos que facilitaran la encapsulación y que, por lo tanto, ayudaran a ocultar información interna; así es como surgieron nuevos lenguajes (por ejemplo, C++) que daban soporte a esta técnica y a nuevos paradigmas de programación más adecuados. De esta forma apareció el paradigma de la programación orientada a objetos (POO).

2.4. Escalabilidad

La herencia junto con otras características del paradigma de la programación orientada a objetos permiten que el esfuerzo no aumente exponencialmente ni con el tamaño, ni con la complejidad del proyecto, ni con un cambio de especificaciones.

2.5. Mejor comprensión del código

Aunque ya dijimos que el paradigma de la programación estructurada mejoraba la legibilidad del código respecto a no usar ningún paradigma, no es suficiente. Todavía sigue habiendo demasiadas líneas de código y demasiados elementos interrelacionados dispersos en diferentes ficheros.

La POO, gracias a la encapsulación, mejora la legibilidad del código. Debido a que los datos –las variables– y los procedimientos –funciones o métodos– que conforman un objeto están juntos en un mismo elemento (llamado *clase*), la comprensión del funcionamiento y la lógica del código es más sencilla.

3. Programación orientada a objetos (POO)

3.1. Breve introducción a los elementos básicos de la POO

A nivel muy introductorio –ya veremos todo con más detalle–, cabe destacar que el elemento principal de la programación orientada a objetos (POO) es el **objeto**. La POO modela la funcionalidad de un programa –es decir, del problema que hay que resolver– intentando asemejarse lo máximo posible a la realidad. El modo en que intenta imitar la realidad es definiendo un conjunto de objetos que interactúan entre sí enviándose y respondiendo a **mensajes**. Gracias a la colaboración entre objetos, el programa es capaz de realizar la funcionalidad esperada –o de resolver el problema planteado.

De igual modo que en el lenguaje C existe el tipo *integer* (`int`, entero), *character* (`char`), etc. para definir una variable, en los lenguajes que soportan la POO, el tipo de los objetos es una **clase**. Una clase se asemejaría a lo que en programación estructurada llamamos *módulo*, puesto que una clase contiene datos (en forma de variables) y funciones. A los datos de una clase se les llama **atributos**, mientras que a sus funciones se les llama **métodos**. De igual modo, tanto a los atributos como a los métodos se les suele llamar **miembros de una clase**.

Así pues, cuando definimos un objeto, tenemos que indicar su tipo. Este no será ni un `int`, ni un `char`, sino una clase. De hecho, muchas veces a los objetos se les llama instancias de una clase o, simplemente, **instancias** (del inglés, *instance*), aunque su correcta traducción al español debería ser 'ejemplo o caso'. Por lo tanto, un objeto sería un ejemplo o caso particular de una clase. Es importante entender que podemos tener tantos objetos o instancias como queramos de una misma clase.

Como hemos dicho, las clases –que definen el tipo de los objetos– tendrán que modelar la realidad del problema que estamos tratando, lo cual nos obliga a definir las clases siguiendo una estrategia totalmente diferente a la que seguimos cuando definimos módulos en la programación estructurada. Si en la programación estructurada los módulos los definíamos siguiendo un criterio, más o menos lógico, como agrupar funciones que traten una misma tarea (p. ej., pintado, control de reglas, control de físicas, etc.), en la POO esta estrategia no nos servirá (del todo). Como ya hemos dicho anteriormente, la POO no utiliza un enfoque *top-down*, sino *bottom-up*, es decir, se parte de la identificación de los objetos dentro del problema para abstraer las clases a las que pertenecen.

Veamos a continuación un ejemplo sencillo que nos permita entender los conceptos que han ido apareciendo:

Ejemplo 4 (Conceptos básicos de la POO) – La familia POO

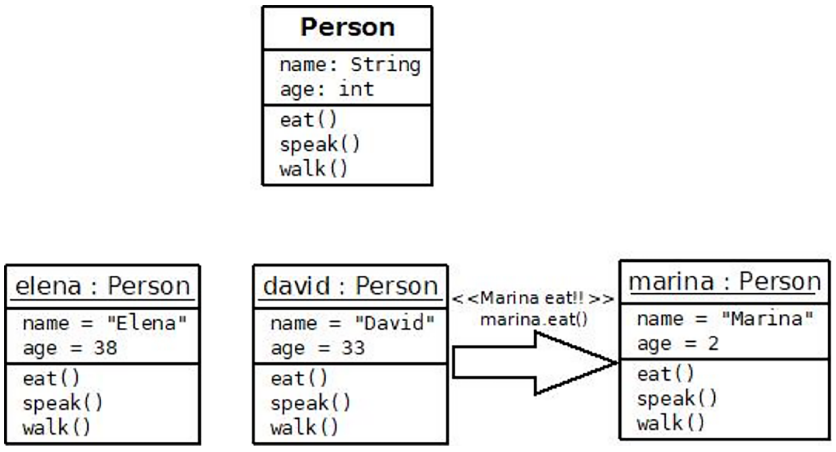
`Elena`, `Marina` y `David` son **objetos o instancias** (ejemplos o casos) de la clase `Person`. Una persona (`Person`) tiene unos datos y unos comportamientos, es decir, unos **atributos** y unos **métodos**. Algunos ejemplos de atributos que todas las personas compartimos son: nombre, apellido, fecha de nacimiento, lugar de nacimiento, género, raza, peso, altura, entre otros. Mientras que ejemplos de métodos que compartimos todas las personas pueden ser: hablar, escuchar, andar, comer, aumentar la altura, aumentar el peso, reducir el peso, entre otros.

Una vez que sabemos que `Elena`, `Marina` y `David` son objetos de `Person`, es fácil intuir que cada uno de ellos tendrá unos valores diferentes para los atributos de la clase `Person`. Por ejemplo, el valor del atributo `nombre` de la clase `Person` ya es diferente para cada uno de los tres objetos, puesto que un objeto tiene el valor `Elena`, otro `Marina` y otro `David`. Asimismo, mientras que el valor del atributo género para `David` es `hombre`, para `Elena` y `Marina` el valor es `mujer`.

De igual modo, si `Marina` es un bebé de veinticinco meses, es fácil imaginar que los valores para los atributos `altura` y `peso` no pueden ser iguales a los de los adultos `Elena` y `David`; y que, a su vez, los valores de estos atributos serán distintos entre `Elena` y `David`.

Por otro lado, como `David` es el padre de `Marina`, a la hora de las comidas este podría pedirle a `Marina` que comiera enviándole el **mensaje** `"Marina, come"` (es decir, llamando al método `comer` de `Marina`). A su vez, `Marina` podría responderle, o bien comiendo (por ejemplo, devolviendo su método `comer` un `true` o un `1`), o bien negándose a ello (por ejemplo, devolviendo un `false` o `0`). De esta manera, vemos cómo interactúan (se comunican o colaboran) los diferentes objetos.

Podemos ver este ejemplo de manera simplificada y visual en la figura siguiente:



El diagrama anterior está dibujado siguiendo la notación UML –profundizaremos en ella más adelante. Una clase está representada como una caja con tres partes que contienen: el nombre, los atributos y los métodos, respectivamente. El nombre de la clase está escrito en negrita y alineada en el centro de la caja. Asimismo, hemos indicado el tipo de los atributos, mientras que los métodos los hemos distinguido abriendo y cerrando paréntesis al lado de su nombre.

Los objetos `Elena`, `David` y `Marina` también se han representado con una caja dividida en tres partes, pero con el nombre de la instancia con el formato `nombreInstancia:nombreClase`. Para los atributos hemos indicado los valores que reciben para cada uno de los objetos. Asimismo, hemos añadido una flecha para mostrar un mensaje del objeto `David` al objeto `Marina`. Este mensaje llama al método `eat` del objeto `Marina`.

Reutilización

Una vez codificada la clase `Person`, esta puede utilizarse en cualquier programa que necesite modelar una persona. Así pues, no hay que volver a reescribir código, solo reutilizarlo.

Vídeo de interés

Para terminar de entender los conceptos básicos de la POO, te recomendamos que veas el vídeo «Clase, objeto, atributo y método» que encontrarás en el aula de la asignatura.

3.2. Dado un problema, ¿cuál es el proceso que hay que seguir?

La programación orientada a objetos (POO) intenta modelar el mundo real –o el problema real que queremos resolver– sobre la base de dos elementos: los objetos y las clases. Para ello, el diseñador del programa sigue un proceso de

abstracción *bottom-up* en el que, a partir de elementos concretos (los objetos), intenta generalizar hasta llegar a diferentes plantillas o abstracciones (las clases). Concretamente, el proceso que se suele seguir es:

1) Identificar los objetos que interactúan en el problema. Dada una descripción del problema, los objetos suelen ser sustantivos o sintagmas nominales.

2) Analizar los objetos identificados con el objetivo de **reconocer sus características y acciones o funcionalidades**. Dada la descripción de un problema, las características se corresponden con sustantivos o sintagmas nominales, mientras que las acciones se suelen corresponder con verbos o sintagmas verbales. En el paradigma de la programación orientada a objetos, a las características las llamaremos **atributos** y a las acciones, **métodos**.

3) Identificar las clases que agrupan a los diferentes objetos. Cada clase tendrá las características (atributos) y acciones o funcionalidades (métodos) comunes de los objetos que representa.

4) Asignar cada objeto a las clases. Una vez que tenemos los objetos y las clases identificadas, debemos relacionar los objetos o instancias con sus correspondientes clases. Un objeto pertenece a una única clase.

Finalmente, cabe tener presente que no todos los sustantivos y verbos de la descripción del problema son objetos, atributos, métodos o clases. Pueden darse los siguientes casos:

- **Redundancia (o sinónimos):** un objeto, clase, atributo o método es denominado de diferentes maneras dentro del texto, es decir, aparecen dos o más sustantivos o verbos para indicar el mismo elemento. En estos casos tenemos que quedarnos con una de las maneras de referirnos a ese elemento.

- **Irrelevancia:** puede darse el caso de que algún elemento que hemos identificado, por lo que sea, tiene poco o nada que ver con el problema que queremos solucionar, es decir, no nos es útil para el problema o contexto en el que estamos o, dicho de otro modo, nos es irrelevante.

- **Roles:** existen sustantivos y verbos que indican cuál es el papel de una clase u objeto dentro del problema. A veces esta información es superflua y, por lo tanto, la podemos descartar; en otras ocasiones, los roles se corresponden con atributos, operaciones o relaciones que son importantes tener en cuenta.

Si nos fijamos en el Ejemplo 4 del apartado anterior, podemos ver que hemos seguido un proceso *bottom-up*. Démonos cuenta de que hemos ido de abajo (objetos: `Elena`, `David` y `Marina`) hacia arriba (clase: `Person`). Concretamente, primero hemos identificado tres objetos –`Elena`, `David` y `Marina`– y a par-

tir de ellos hemos detectado que estos tres objetos compartían una serie de atributos y métodos, lo cual nos estaba indicando que `Elena`, `David` y `Marina` podrían tratarse del mismo tipo de objeto, es decir, que podrían pertenecer a la misma clase. Esta clase la hemos llamado `Person`.

Alguno podría pensar que se podría haber definido una clase para `Elena`, otra para `David` y otra para `Marina`, pero, en vez de esto, hemos hecho un ejercicio de abstracción y hemos buscado elementos comunes cuyo resultado ha sido una clase común llamada `Person`.

La identificación de los «elementos compartidos o comunes» requiere, por nuestra parte, ejercer un nivel de abstracción mayor del que solemos hacer en el paradigma de la programación estructurada.

Vídeo de interés

Para terminar de entender el concepto de *abstracción* y el enfoque *bottom-up*, te recomendamos que veas el vídeo «Abstracción y paradigma bottom-up» que encontrarás en el aula de la asignatura.

Esta forma de pensar un programa nos ayuda a hacer un diseño mejor organizado, más reducido en cuanto a líneas de código, más reutilizable y, por consiguiente, más fácil de mantener y escalar.

A diferencia de la programación estructurada, que se centra en las funciones, la POO se centra en la estructura de los datos –es decir, cómo estos se organizan. Este cambio, aunque puede parecer sutil, es vital que se comprenda y asimile para tener éxito con la POO.

4. ¿Cómo nos enfrentamos a la programación orientada a objetos?

Es posible que estés pensando que esto de la POO parece complicado. Sentimos decirte que tienes razón. Aprender el paradigma POO y un lenguaje orientado a objetos concreto es más complejo que aprender un lenguaje imperativo ligado al paradigma de la programación estructurada. Así que, aunque no sea sencillo de buenas a primeras, tampoco es imposible. Pero tranquilo, somos conscientes de que eres un programador novel y, por ello, iremos paso a paso. Como ya hemos dicho, la POO utiliza un enfoque *bottom-up*, diferente del enfoque *top-down* del paradigma de la programación estructurada. El enfoque *bottom-up* se centra en abstraer los elementos relevantes del dominio de la aplicación y, por desgracia, la capacidad de abstracción de un programador novel suele ser una de sus debilidades.

De igual manera que iremos paso a paso, también te pedimos que seas consciente de que dominar el paradigma de la POO te va a exigir un plus de compromiso. En concreto, te va a exigir y te vamos a exigir:

- **Mucha dedicación:** pasarás decenas y decenas de horas delante del ordenador programando, porque, como se suele decir: «a programar, se aprende programando». Esto es como aprender a tocar la guitarra, si no le dedicas tiempo a la práctica, ya te puedes saber los acordes de memoria, que no tocarás ninguna canción con soltura.

- **Tener autonomía (también conocido como «buscarse la vida»):** esta es una competencia vital en pleno siglo XXI. Hoy más que nunca la tecnología evoluciona muy rápidamente y es muy difícil estar a la última. El equipo docente no lo podemos saber todo y, a menudo, haremos lo mismo que puedes hacer tú: consultar internet, un libro, hacer pruebas, etc. Existen miles de configuraciones y, por si esto resultara poco, los programas suelen contener *bugs*. Así que, antes de preguntar algo, busca por tu cuenta la solución. La satisfacción de resolver un problema por uno mismo no tiene precio.

- **Compartir tu conocimiento:** si resuelves un problema que te ha mantenido en vilo dos días, comparte la solución con el resto de compañeros. Te lo agradecerán. Recuerda: «hoy por ti, mañana por mí». Lo mismo si sabes la respuesta a una duda que se formula en un aula (siempre y cuando no resuelva directamente lo que pregunta una actividad que hay que entregar). Explicar algo a alguien es una oportunidad de oro para profundizar en el conocimiento de un tema, puesto que muchas veces uno se da

cuenta de si entiende perfectamente un concepto cuando se lo explica a otra persona.

Resumen

Punto de partida

- Damos por hecho que conoces y dominas el paradigma de la programación estructurada antes de empezar con el paradigma de la programación orientada a objetos (POO).

Beneficios del paradigma de la programación orientada a objetos

- **Facilita la reutilización de código** gracias a la encapsulación que fuerza de por sí una clase y la herencia.

- Un diseño del software **más abstracto** que aporta una solución más cercana y natural a la realidad.

- **Protección de los datos** de un objeto mediante modificadores de acceso que garantizan la consistencia y facilitan el mantenimiento de las clases.

- Gracias a la propia naturaleza del paradigma de la POO y a los mecanismos que incluyen (p. ej. herencia y encapsulación), es **más fácil escalar un programa** sin que el esfuerzo de llevarlo a cabo aumente exponencialmente.

- **Mejora de la legibilidad del código** debido a que todos los datos y funciones o métodos interrelacionados están juntos o encapsulados en una misma entidad software llamada *clase*.

Elementos básicos del paradigma de la programación orientada a objetos

a) Clase:

b) Objeto o instancia:

- un ejemplar o caso concreto de una clase,

- una vez creado, cada objeto ocupará un espacio de memoria y tendrá sus propios atributos y métodos que la clase a la que pertenece haya definido.

c) Proceso de abstracción *bottom-up* para diseñar programas basados en la POO:

- **identificar** los diferentes **objetos** del problema o contexto,
- **reconocer** las **características** (datos; llamados **atributos**) y **acciones** (funcionalidades; llamadas **métodos**) de los objetos identificados,
- agrupar los objetos por similitud para **identificar** y **definir** las **clases** que los representan,
- **asignar** cada **objeto** a una de las **clases** definidas.

Actitud frente al paradigma de la programación orientada a una de objetos

- **Mucha dedicación:** durante horas tendrás que realizar el proceso iterativo de pensar, programar, probar, buscar información y entender.
- **Autonomía:** buscar y probar diferentes soluciones a los problemas que te encuentres.
- **Compartir conocimiento:** aprendamos entre todos, compartamos lo que sabemos o descubrimos.

Actividades

Ejercicio 1
La liga de balón prisionero (LBP) está formada por diez equipos, como por ejemplo, los Uocquis o los PEC Patrol. Cada equipo tiene un nombre, una fecha de fundación, un presupuesto y una dirección donde juegan los partidos. Así pues, los Uocquis, creados el 6 de octubre de 1995, tienen un presupuesto de 5.000.000 € y juegan en la avenida Tibidabo de la ciudad de Barcelona. En cambio, los PEC Patrol tienen un presupuesto de 16.000.000 € y juegan en la calle Tajo.

Evidentemente, cada equipo está formado por jugadores. La normativa indica que cada jugador tiene que tener un dorsal y un nombre que le identifique. Además, todos los jugadores tienen un salario. Así, por ejemplo, el mejor jugador de la liga, Winnie Poo, juega con el número 13 en los Rangers y cobra 100.000 € al año.

Según la normativa, los equipos solo pueden hacer ofertas a los jugadores, ficharlos cuando estos aceptan las ofertas y vender jugadores. Por su parte, los jugadores solo pueden entrenar, jugar y aceptar o rechazar las ofertas.

Según las últimas noticias, los PEC Patrol han hecho una oferta a Winnie Poo para ficharlo durante cinco años y pagarle 200.000 €/año. Los PEC Patrol están a la espera de recibir una respuesta por parte de Winnie Poo.

Dada la descripción del problema anterior:

a) Indicad los objetos o instancias que aparecen (solo el nombre).

b) Indicad las clases que veis (solo el nombre).

c) Por cada una de las clases que habéis mencionado en el apartado «b», indicad sus atributos y métodos según lo que dice el texto.

d) Para cada objeto identificado en el apartado «a», indicad a qué clase de las mencionadas en el apartado «b» pertenece.

e) Para cada objeto indicad el valor que tienen los atributos de la clase a la que pertenece. Si para un atributo no se indica el valor en el texto, poned «Unknown».

f) ¿Con qué concepto de la POO se relaciona la frase en negrita?

Ejercicio 2
UocDonald's es una cadena de restaurantes de comida rápida que tiene quince sucursales repartidas por todo el mundo. Cada sucursal tiene un identificador, la fecha en que abrió sus puertas por primera vez y la dirección en la que está ubicada. Así pues, el restaurante con identificador 01, que inició su andadura el 8 de noviembre de 2005, está ubicado en la calle Villa Olímpica, número 2, de la ciudad de Barcelona. En cambio, el restaurante 05 sirve comidas en la calle de Fuencarral, número 103, de Madrid.

La especialidad de la empresa son las hamburguesas. Para cada una de ellas se sabe su nombre, su precio en euros, las kilocalorías que aporta y si se puede pedir con pan sin gluten o no. Por ejemplo, la Big Pac cuesta 3,75 €, tiene 522 kcal y no es apta para celíacos, sin embargo, la Básica cuesta 1 €, tiene 254 kcal y sí la pueden disfrutar los celíacos.

La forma de trabajar de UocDonald's se basa en un modelo centralizador, es decir, la empresa solo tiene un almacén que provee a todas las sucursales. Así pues, el almacén central recibe pedidos de cada una de las sucursales y los envía a quien corresponda cuando los tiene listos. Por ejemplo, ayer mismo, el almacén recibió un pedido de la sucursal 01. Es posible que en un futuro se creen más almacenes, pero de momento solo hay uno con el nombre «Almacén Central» y está ubicado en la población de Ateca (Zaragoza).

Dada la descripción del problema anterior:

a) Indicad los objetos o instancias que aparecen (solo el nombre).

b) Indicad las clases que veis (solo el nombre).

c) Por cada una de las clases que habéis mencionado en el apartado «b», indicad sus atributos y métodos según lo que dice el texto.

d) Para cada objeto identificado en el apartado «a», indicad a qué clase de las mencionadas en el apartado «b» pertenece.

e) Para cada objeto indicad el valor que tienen los atributos de la clase a la que pertenece. Si para un atributo no se indica el valor en el texto, poned «Unknown».

Ejercicio 3
El museo MuseUOC quiere guardar sus obras de arte. Cada obra de arte tiene un nombre, un autor, una fecha de creación, una fecha de entrada en el museo y una tasación (i. e. en cuanto está valorada la obra).

Una de las obras más importantes que tiene el MuseUOC es *El GuerniPAC*, un cuadro realizado por PACasso en el año 1994 y que empezó a formar parte del museo a partir del 25 de septiembre de 2013. Se calcula que su valor actual es de 200.000.000 €.

Otra obra importante es el retablo *El Jardín de las PS*, cuyo valor es de 75.000.000 € y está en el museo desde el 10 de octubre de 2009.

El museo está compuesto por varias salas. Cada sala tiene un nombre, unos metros cuadrados, una planta del edificio y un valor que indica si está abierta o cerrada en ese momento. Una de ellas es la «Sala Principal», de 300 m^2, ubicada en la planta 0; también está la «Sala de los Sentidos», de 100 m^2; y la «Sala de Restauración», de 500 m^2. Cada sala puede abrirse o cerrarse, según se desee. Evidentemente, si está abierta no puede estar cerrada y viceversa. Actualmente todas las salas están abiertas.

Dada la descripción del problema anterior:

a) Indicad los objetos o instancias que aparecen (solo el nombre).

b) Indicad las clases que veis (solo el nombre).

c) Por cada una de las clases que habéis mencionado en el apartado «b», indicad sus atributos y métodos según lo que dice el texto.

d) Para cada objeto identificado en el apartado «a», indicad a qué clase de las mencionadas en el apartado «b» pertenece.

e) Para cada objeto indicad el valor que tienen los atributos de la clase a la que pertenece. Si para un atributo no se indica el valor en el texto, poned «Unknown».

Ejercicio 4
La bibliUOCteca (biblioteca de la UOC) quiere guardar todo el inventario de los libros del que dispone. Cada libro tiene un título, un autor, una fecha de creación, una fecha de adquisición y un precio (i. e. lo que costó adquirirlo).

Uno de los libros más importantes que tiene la bibliUOCteca es *La sombra de la PS*, escrito por Carlos Ruiz PACfón en el año 2001 y adquirido, el 8 de noviembre de 2013, por 21,99 €.

Otro libro importante es *PSity*, que se compró por 18 € y está en la biblioteca desde el 3 de octubre de 2008.

La biblioteca está compuesta por varias salas. Cada sala tiene un nombre, unos metros cuadros, una planta del edificio y un valor que indica si está abierta o cerrada en ese momento. Una de ellas es la «Sala de Actos», de 300 m^2, ubicada en la planta 0. También está la «Sala de Estudio», de 150 m^2. Cada sala puede abrirse o cerrarse según se desee. Evidentemente, si está abierta no puede estar cerrada y viceversa. Actualmente todas las salas están abiertas.

Dada la descripción del problema anterior:

a) Indicad los objetos o instancias que aparecen (solo el nombre).

b) Indicad las clases que veis (solo el nombre).

c) Por cada una de las clases que habéis mencionado en el apartado «b», indicad sus atributos y métodos según lo que dice el texto.

d) Para cada objeto identificado en el apartado «a», indicad a qué clase de las mencionadas en el apartado «b» pertenece.

e) Para cada objeto indicad el valor que tienen los atributos de la clase a la que pertenece. Si para un atributo no se indica el valor en el texto, poned «Unknown».

Solucionario

Solución ejercicio 1

a) Indicad los objetos o instancias que aparecen (solo el nombre).

- Uocquis
- PEC Patrol
- Winnie Poo
- Rangers

La LBP no la consideramos un objeto, puesto que el problema que estamos modelando o solucionando es la propia LBP, por lo tanto, no tiene sentido.

b) Indicad las clases que ves (solo el nombre).

- Equipo
- Jugador

c) Por cada una de las clases que habéis mencionado en el apartado «b», indicad sus atributos y métodos según lo que dice el texto.

Clase	Equipo	Jugador
Atributos	Nombre Fecha de fundación Presupuesto Dirección	Dorsal Nombre Salario
Métodos	Hacer ofertas jugadores (hacerOferta) Fichar jugadores (ficharJugador) Vender jugadores (venderJugador)	Entrenar (entrenar) Jugar (jugar) Aceptar o rechazar oferta (aceptarRechazarOferta)

En `Equipo` se podría considerar el método `jugar`. Tanto si se ha puesto como si no, es correcto. Si el método `aceptarRechazarOferta` se ha dividido en dos (p. ej., `aceptarOferta` y `rechazarOferta`), se puede considerar correcto.

Para ser del todo correctos y modelar la frase o relación mencionada en el texto «cada equipo está formado por jugadores», deberíamos haber puesto un atributo tipo `Jugador` en la clase `Equipo`. Este atributo de la clase `Equipo` sería un *array* (o similar). Si lo has puesto, perfecto, pues quiere decir que estás yendo un paso más allá de lo que hemos explicado hasta ahora. Si no lo has puesto, tranquilo/a, pues la relación entre objetos (denominada formalmente *asociación*) se explicará en el módulo «Asociaciones (relaciones entre objetos)» de estos materiales docentes. Por ahora, solo trabajamos las clases de manera aislada, es decir, sin relacionarlas entre sí.

d) Para cada objeto identificado en el apartado «a», indicad a qué clase de las mencionadas en el apartado «b» pertenece.

`Uocquis` es un objeto que pertenece a la clase `Equipo`.

`PEC Patrol` es un objeto que pertenece a la clase `Equipo`.

`Winnie Poo` es un objeto que pertenece a la clase `Jugador`.

`Rangers` es un objeto que pertenece a la clase `Equipo`.

e) Para cada objeto indicad el valor que tienen los atributos de la clase a la que pertenece. Si para un atributo no se indica el valor en el texto, poned «Unknown».

Atributo/Objeto	Uocquis	PEC Patrol	Rangers
Nombre	Uocquis	PAC Patrol	Rangers

Atributo/Objeto	Uocquis	PEC Patrol	Rangers
Fecha de fundación	6 de octubre de 1995	«Unknown»	«Unknown»
Presupuesto	5.000.000 €	16.000.000 €	«Unknown»
Dirección	Avenida Tibidabo (Barcelona)	Calle Tajo	«Unknown»

Atributo/Objeto	Winnie Poo
Dorsal	13
Nombre	Winnie Poo
Salario	100.000 €

f) ¿Con qué concepto de la POO se relaciona la frase en negrita?

Hace referencia al concepto de mensaje. La versión codificada del texto podría ser:

```
pecpatrol.hacerOferta(WinniePoo, 5, 200.000);
```

Solución ejercicio 2

a) Indicad los objetos o instancias que aparecen (solo el nombre).

- Sucursal 01
- Sucursal 05
- Big Pac
- Básica
- Almacén Central

b) Indicad las clases que veis (solo el nombre).

- Sucursal (o también le podías haber llamado «Restaurante»)
- Hamburguesa
- Almacén

c) Por cada una de las clases que habéis mencionado en el apartado «b», indicad sus atributos y métodos según lo que dice el texto.

Clase	Sucursal	Hamburguesa	Almacén
Atributos	Identificador Fecha de apertura Dirección	Nombre Precio Kcal sinGluten	Nombre Dirección (o población)
Métodos			Recibir pedido de sucursal (recibirPedido) Enviar pedido a sucursal (enviarPedido)

El método `recibirPedido` podría haberse considerado como `hacerPedido` y haberlo puesto en la clase `Sucursal`. Igualmente, se podría haber creado el método `recibirPedido` (de almacén) en la clase `Sucursal`. En cualquier caso, la relación entre las sucursales y el almacén debería quedar reflejada en la solución de una manera u otra.

Las direcciones de las sucursales se podrían haber separado en tres atributos: calle, número y población.

d) Para cada objeto identificado en el apartado «a», indicad a qué clase de las mencionadas en el apartado «b» pertenece.

`Sucursal01` es un objeto que pertenece a la clase `Sucursal`.

`Sucursal05` es un objeto que pertenece a la clase `Sucursal`.

`Big Pac` es un objeto que pertenece a la clase `Hamburguesa`.

`Básica` es un objeto que pertenece a la clase `Hamburguesa`.

`Almacén Central` es un objeto que pertenece a la clase `Almacén`.

e) Para cada objeto indicad el valor que tienen los atributos de la clase a la que pertenece. Si para un atributo no se indica el valor en el texto, poned «Unknown».

Atributo/Objeto	Sucursal01	Sucursal05
Identificador	01	05
Fecha de apertura	8 de noviembre de 2005	«Unknown»
Dirección	Calle Villa Olímpica, número 2 (Barcelona)	Calle de Fuencarral, número 103 (Madrid)

Atributo/Objeto	Big Pac	Básica
Nombre	Big Pac	Básica
Precio	3,75 €	1 €
Kcal	522	254
sinGluten	No (false)	Sí (true)

Atributo/Objeto	Almacén Central
Nombre	Almacén Central
Dirección	Ateca (Zaragoza)

Solución ejercicio 3

a) Indicad los objetos o instancias que aparecen (solo el nombre).

- El GuerniPAC
- El Jardín de las PS
- Sala Principal
- Sala de los Sentidos
- Sala de Restauración

b) Indicad las clases que veis (solo el nombre).

- Obra
- Sala

c) Por cada una de las clases que habéis mencionado en el apartado «b», indicad sus atributos y métodos según lo que dice el texto.

Clase	Obra	Sala
Atributos	Nombre Autor Fecha de creación Fecha de entrada Tasación	Nombre m^2 Planta estaAbierta
Métodos		abrirCerrar()

El método `abrirCerrar()` podría haberse hecho con dos métodos: `abrir()` y `cerrar()`. Sin embargo, al tratarse de dos métodos que asignan el valor contrario al atributo `estaAbierta`, se puede hacer todo con un único método que asigna al atributo `estaAbierta` el valor contrario al que tenga en ese momento.

d) Para cada objeto identificado en el apartado «a», indicad a qué clase de las mencionadas en el apartado «b» pertenece.

`El GuerniPAC` es un objeto que pertenece a la clase `Obra`.

`El Jardín de las PS` es un objeto que pertenece a la clase `Obra`.

`Sala Principal` es un objeto que pertenece a la clase `Sala`.

`Sala de los Sentidos` es un objeto que pertenece a la clase `Sala`.

`Sala de Restauración` es un objeto que pertenece a la clase `Sala`.

e) Para cada objeto indicad el valor que tienen los atributos de la clase a la que pertenece. Si para un atributo no se indica el valor en el texto, poned «Unknown».

Atributo/Objeto	El GuerniPAC	El Jardín de las PS
Nombre	El GuerniPAC	El Jardín de las PS
Autor	PACasso	«Unknown»
Fecha de creación	1994	«Unknown»
Fecha de entrada	25 de septiembre de 2013	10 de octubre de 2009
Tasación	200.000.000 €	75.000.000 €

Atributo/Objeto	Sala Principal	Sala de los Sentidos	Sala de Restauración
Nombre	Sala Principal	Sala de los Sentidos	Sala de Restauración
m^2	300	100	500
Planta	0	«Unknown»	«Unknown»
estaAbierta	Sí (true)	Sí (true)	Sí (true)

Solución ejercicio 4

a) Indicad los objetos o instancias que aparecen (solo el nombre).

- La sombra de la PS
- PSity
- Sala de Actos
- Sala de Estudio

b) Indicad las clases que veis (solo el nombre).

- Libro
- Sala

c) Por cada una de las clases que habéis mencionado en el apartado «b», indicad sus atributos y métodos según lo que dice el texto.

Clase	Libro	Sala
Atributos	Título Autor Fecha de creación Fecha de adquisición Precio	Nombre m^2 Planta estaAbierta
Métodos		abrirCerrar()

El método `abrirCerrar()` podría haberse hecho con dos métodos: `abrir()` y `cerrar()`. No obstante, al tratarse de dos métodos que asignan el valor contrario al atributo `estaAbierta`, se puede hacer todo con un único método que asigna al atributo `estaAbierta` el valor contrario al que tenga en ese momento.

d) Para cada objeto identificado en el apartado «a», indicad a qué clase de las mencionadas en el apartado «b» pertenece.

`La sombra de la PS` es un objeto que pertenece a la clase `Libro`.

`PSity` es un objeto que pertenece a la clase `Libro`.

`Sala de Actos` es un objeto que pertenece a la clase `Sala`.

`Sala de Estudio` es un objeto que pertenece a la clase `Sala`.

e) Para cada objeto indicad el valor que tienen los atributos de la clase a la que pertenece. Si para un atributo no se indica el valor en el texto, poned «Unknown».

Atributo/Objeto	La sombra de la PS	PSity
Título	La sombra de la PS	PSity
Autor	Carlos Ruiz PACfón	«Unknown»
Fecha de creación	2011	«Unknown»
Fecha de adquisición	8 de noviembre de 2013	3 de octubre de 2008
Precio	21,99 €	18 €

Atributo/Objeto	Sala de Actos	Sala de Estudio
Nombre	Sala de Actos	Sala de Estudio
m2	300	150
Planta	0	«Unknown»
estaAbierta	Sí (true)	Sí (true)

Bibliografía

Booch, G.; Maksimchuk, R.; Engle, M.; Young, B. J.; Conallen, J.; Houston, K. (2007). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional. ISBN: 978-0201895513.

Griffiths, D.; Griffiths, D. (2019). *Head First Kotlin*. O'Reilly Media, Inc. ISBN: 978-1491996690.

Hunt, A.; Thomas, D. (2019). *The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition* (2.ª ed.). Addison-Wesley Professional. ISBN: 978-0135956977.

Phillips, D. (2018). *Python 3 Object-Oriented Programming* (3.ª ed.). Packt Publishing. ISBN: 978-1789615852.

Pollice, G.; West, D.; McLaughlin, B. (2006). *Head First Object-Oriented Analysis and Design*. O'Reilly Media, Inc. ISBN: 978-0596008673.

Robinson, S.; Nagel, C.; Watson, K.; Glynn, J.; Skinner, M.; Evjen, B. (2004). *Professional C#* (3.ª ed.). Wrox. ISBN: 978-0764557590.

Sharp, J. (2007). *Microsoft Visual C# 2008 Step by Step*. Microsoft Press. ISBN: 978-0735624306.

Weisfeld, M. (2019). *The Object-Oriented Thought Process* (5.ª ed.). Addison-Wesley Professional. ISBN: 978-0135182130.