

# Programación orientada a objetos

En este primer capítulo, comenzaremos viendo qué es la programación orientada a objetos y en qué puede ayudarnos. También veremos sus conceptos principales, la evolución del diseño y una introducción a la construcción de sistemas complejos.

<b>Introducción</b>	<b>16</b>
Programación no estructurada	16
Programación procedural	17
Programación modular	17
Programación orientada a objetos	18
<b>El análisis y el diseño en la programación orientada a objetos</b>	<b>23</b>
El lenguaje de modelado unificado	24
<b>Resumen</b>	<b>27</b>
<b>Actividades</b>	<b>28</b>



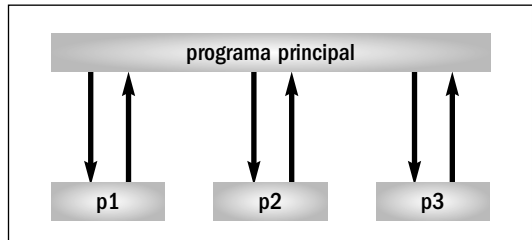
La programación no estructurada consiste en un programa extenso desarrollado dentro de una función principal que utiliza sólo variables del tipo global. Este modo de programación puede ser aceptable para la resolución de problemas triviales, sin embargo, a medida que la complejidad se incrementa, comienza a ser muy engorroso su mantenimiento y el agregado de nuevas características.

## Programación procedural

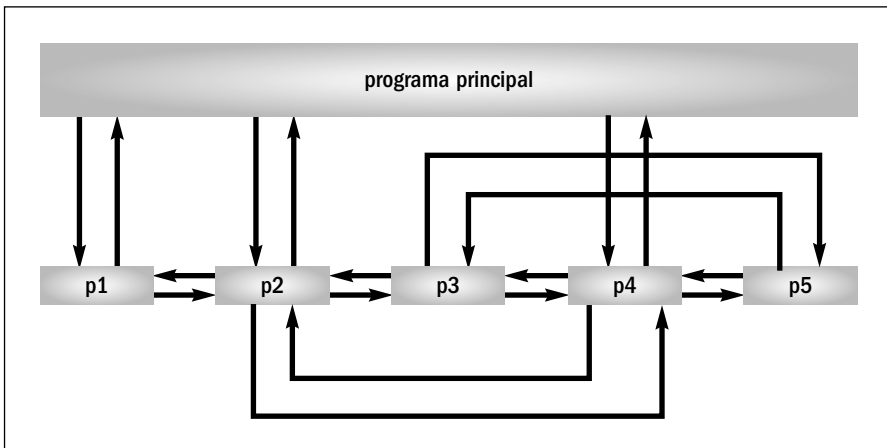
La mayor parte de los lenguajes de alto nivel soportan la creación de procedimientos, que son trozos de código que realizan una tarea determinada.

Un procedimiento podrá ser invocado muchas veces desde otras partes del programa con el fin de aislar la tarea en cuestión y, una vez que finaliza su ejecución, retorna al punto del programa desde donde se realizó la llamada.

Además, cada procedimiento tendrá su propio conjunto de datos, aunque podrá acceder a datos globales y a los pasados como parámetros desde el programa principal o desde otros procedimientos. De esta forma, podemos dividir nuestro problema en problemas más pequeños, y así, llevar la complejidad a un nivel manejable.



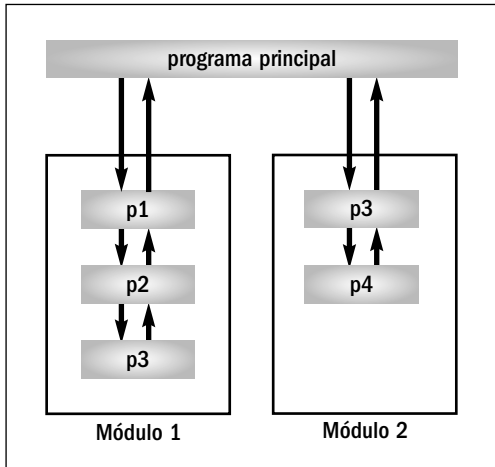
**Figura 3.** La programación procedural, en programas pequeños o medianos.



**Figura 4.** La programación procedural, en un programa de tamaño mayor.

## Programación modular

La programación modular va un paso más allá con la creación de procedimientos: los agrupa en módulos según su función. De esta manera, el modelo es apto para



**Figura 5.** En la programación modular, cada módulo agrupa procedimientos.

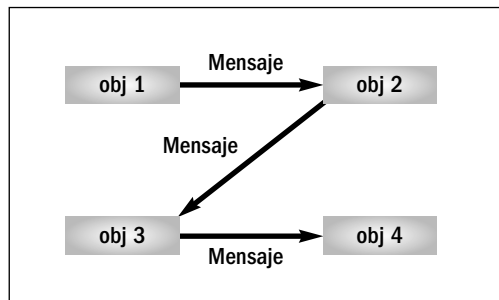
sistemas más complejos que requieren programas más extensos. Cada módulo tendrá su propio conjunto de datos, creando un nivel intermedio de datos globales, ya que el conjunto de datos del módulo será global para los procedimientos del módulo, pero inaccesible a los procedimientos de otros módulos. De este modo, la representación de un determinado concepto del problema podrá ser llevado a un módulo, por ejemplo, la manipulación de archivos donde dentro del módulo en cuestión existirán distintas funciones para su creación, destrucción, lectura, escritura, etc.

## Programación orientada a objetos

En la programación orientada a objetos, el concepto de módulo es profundizado y se transforma en un objeto. Además, allí, un programa no es otra cosa que una colección de objetos que se comunican entre sí para lograr un objetivo común (Figura 6). Veamos entonces antes de continuar, qué es un objeto y cuáles son sus propiedades.

### ¿Qué es un objeto?

Miremos por un instante a nuestro alrededor. ¿Qué tenemos?: un monitor, una computadora, una máquina de café, una calculadora, un televisor... ¡son todos objetos! Cada uno cumple una función específica, y podemos comunicarnos con



**Figura 6.** Los objetos se comunican entre sí para lograr un objetivo común.

## INCONVENIENTES

Tal vez el mayor inconveniente de este tipo de programación sea que el modelo no restringe quién puede invocar qué procedimiento. Entonces, a medida que el programa crece, puede tornarse confusa la relación de llamadas que se realizan a los procedimientos. Éste es un aspecto muy importante a tener en cuenta al momento de sentarnos a programar.

ellos mediante algún mecanismo que el fabricante haya colocado para tal fin. Un televisor tendrá botones y un control remoto por medio del cual accedemos a sus **funciones** (encender, apagar, cambiar de canal, etc.). Otras funciones avanzadas no nos son accesibles, y tendríamos que abrir el aparato para acceder a ellas (ajuste de frecuencia horizontal, calibraciones específicas de componentes, etc.); estas funciones no forman parte de la interfaz del objeto, es decir, existen pero el acceso es vedado al usuario común.

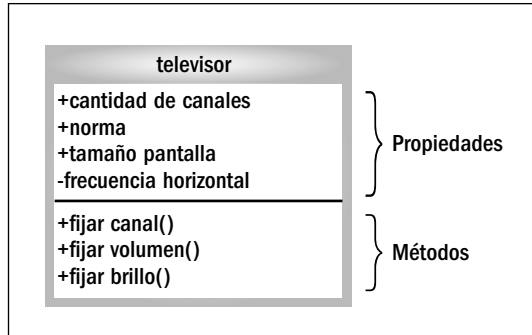
Por otro lado, podríamos pensar que el objeto tiene un estado, es decir, el televisor se encuentra prendido o apagado, y posee ciertas **propiedades** (como la norma en la que trabaja, la cantidad de canales que maneja el sintonizador, el tamaño de la pantalla, el canal actualmente sintonizado, etc.).

En definitiva, visto de manera general, el televisor es un objeto con ciertas funcionalidades y ciertas propiedades. Podríamos establecer, entonces, de manera no rigurosa, que un objeto es un conjunto de propiedades y métodos:

### objeto = propiedades + métodos

Por supuesto, no todas las propiedades serán accesibles desde el exterior del objeto (recordemos el ajuste de la frecuencia horizontal), y aquí llegamos a una de las principales características de este paradigma: el **encapsulamiento**.

El objeto encapsula propiedades y métodos que no formarán parte de su interfaz y que se reservan para su uso interno. De esta manera, ocultamos la implementación y mantenemos una interfaz conformada solamente por un subgrupo de la cantidad total de métodos y propiedades del objeto, evitando errores producidos por accesos indebidos. Internamente, el aparato de televisión podría estar formado por otros



**Figura 7.** Un objeto con sus propiedades y métodos. Sólo los que antepongan el signo + podrán ser accedidos desde otros objetos.

## ESTADO DE UN OBJETO

Un punto muy importante a saber: el estado de un objeto se encuentra determinado por el valor concreto de cada una de sus propiedades.

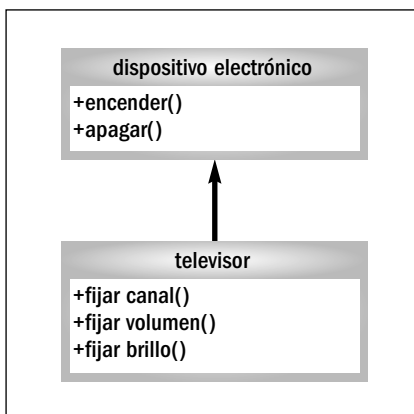
objetos (la pantalla, el sintonizador, amplificadores, etc.), pero eso a nosotros no nos preocupa, no necesitamos tener conocimientos de su contenido o de cómo se acoplan esos subsistemas. De hecho, un televisor convencional de tubo de rayos catódicos se maneja de modo similar que un televisor con pantalla de plasma y, sin embargo, internamente, muchos de sus componentes son distintos. En pocas palabras, ambos aparatos poseen la misma interfaz, pero cada uno encapsula propiedades y funciones diferentes. ¿Esto significa que en un programa podría cambiar un objeto A por un objeto B que tenga la misma interfaz pero distinta implementación, y todo seguiría funcionando correctamente? ¡Exacto! Más adelante, veremos cómo implementar esto en lenguaje C++.

## Más características de los objetos

Ciertamente, los objetos pueden encapsular propiedades y métodos, pero esto no es todo. La programación orientada a objetos en búsqueda de maximizar la capacidad de volver a usarlo de código establece una característica de lo más interesante.

En la tradición aristotélica, si queremos definir un determinado elemento, lo hacemos enunciando su género más próximo conocido junto con la diferencia fundamental que lo distingue de dicho grupo. Así es como la definición de ser humano es “animal racional”, ya que primero ubicamos al ser humano dentro de su género más próximo (animal), y luego, enunciemos la diferencia fundamental que lo distingue de dicha clase (racional).

En la programación orientada a objetos, es posible definir una clase de objetos enunciando que **es como** otra clase de objetos; luego deberemos especificar cuáles son sus diferencias agregando las propiedades y los métodos que correspondan.



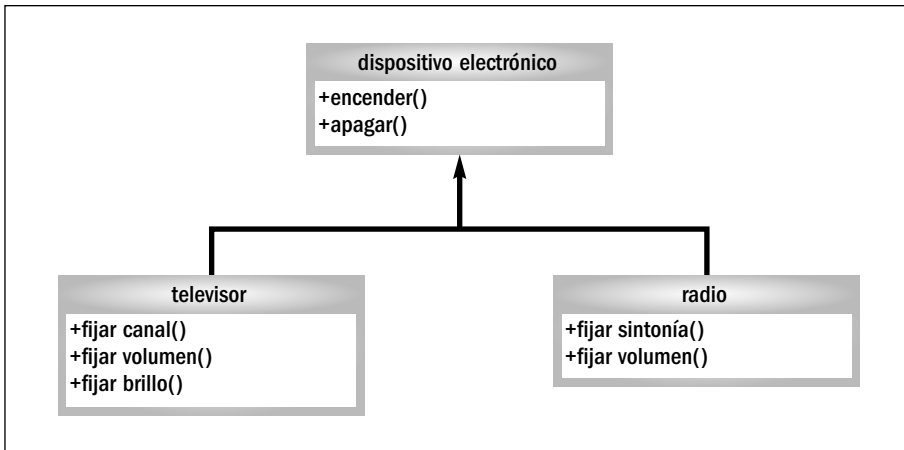
**Figura 8.** El diagrama simboliza que el objeto “televisor” hereda propiedades y métodos de la clase “dispositivo electrónico”.

Volvamos al ejemplo del televisor, ¿cuál es el género más cercano a él? ¡Un dispositivo electrónico! Bueno, podríamos definir una clase de objetos “dispositivo electrónico” que pueda encenderse y apagarse; entonces, el televisor será un dispositivo electrónico que permita sintonizar y visualizar imágenes y sonido. Claro que deberemos agregar muchas funcionalidades, pero el televisor, por ser dispositivo electrónico, **heredará** la posibilidad de ser prendido y apagado, lo que nos evitará codificar una y otra vez métodos idénticos en distintos tipos de objetos.

Usted podrá pensar “¿Qué me he ahorrado? ¿Tuvo sentido hacer la división dispositivo electrónico/televisor?”. Bueno, usualmente, un

sistema es un conjunto de una gran cantidad de objetos. Muchos de ellos poseen propiedades en común que es conveniente especificar de esa manera para ahorrar tiempo de codificación. Además, menos códigos significa menos errores, mayor reutilización y más facilidad de mantenimiento.

Podríamos suponer que dentro de mi pequeño programa debo agregar objetos del tipo “radio”. Una radio es un dispositivo electrónico, ¿no? ¡Muy bien! Ahora simplemente nos resta especificar este tipo de objetos desde una clase ya especificada.

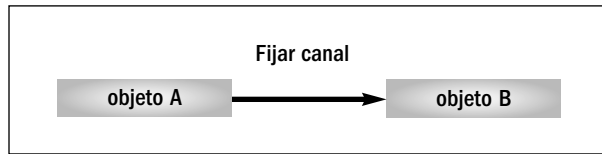


**Figura 9.** Una radio es un dispositivo electrónico.

Como se puede apreciar en la figura, el método “fijar volumen” se encuentra tanto en un televisor como en una radio. ¿Podría este método ser llevado a dispositivo electrónico? Tal vez, el asunto es que no todos los dispositivos electrónicos tienen sonido. ¿Se podría crear, entonces, una clase intermedia entre dispositivo electrónico y televisor/radio? Es otra posibilidad, y esto nos muestra la punta de un iceberg que será tratado en capítulos posteriores y que tiene que ver con la importancia del **análisis** y el **diseño**. Es que cuando queremos resolver un problema determinado por medio de la programación, antes de encender la computadora es necesario realizar un análisis previo; este proceso dejará como salida un documento que nos servirá de mapa en la codificación y que especificará, entre otras cosas, qué objetos posee nuestro sistema y la relación entre ellos.

### Aún más características de los objetos

Hemos comentado que los objetos presentan propiedades y métodos; también, que un programa orientado a objetos es, en realidad, una colección de objetos que interactúan con un fin común. Lo que aún no hemos dicho es **el modo** en que los objetos interactúan: un objeto puede invocar un método de la interfaz de otro objeto por medio del envío de un mensaje, como vemos en la **Figura 10**.



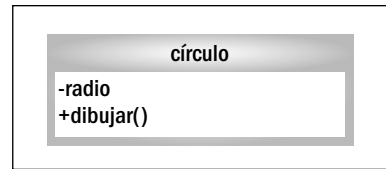
**Figura 10.** El objeto A envía un mensaje al objeto B.

De esta manera, por ejemplo, podríamos crear una clase llamada “círculo”. Ésta tendrá distintas propiedades relacionadas con la figura geométrica en cuestión, como el **radio**, y también tendrá métodos como, por ejemplo, **Dibujar**, que podría dibujar la figura en un determinado medio.

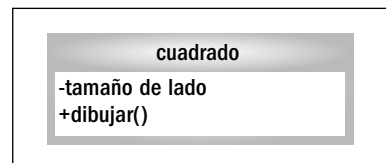
Luego, podremos crear distintos objetos del tipo círculo, asignarle distintos valores a su propiedad radio y ordenarles que se dibujen por medio del envío del mensaje Dibujar.

Supongamos que también queremos, en el programa, representar un cuadrado; crearíamos para esto la clase “cuadrado”, con la propiedad **tamaño de lado** y, nuevamente, el método Dibujar.

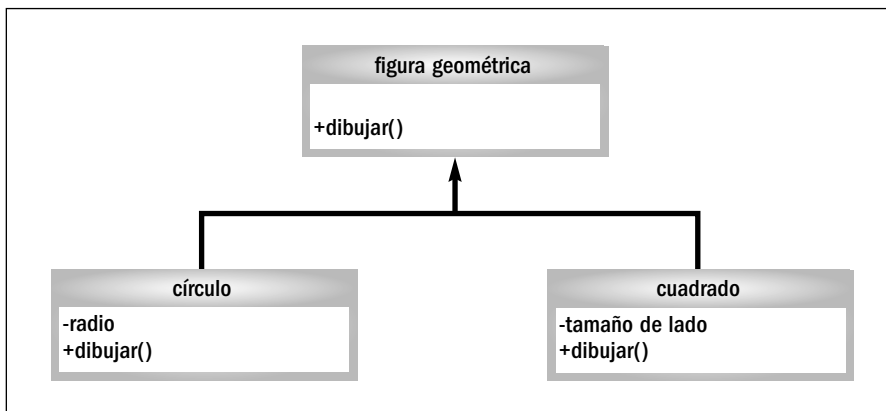
Pero... ¿qué es lo que tienen en común un círculo con un cuadrado? Podemos establecer que ambos son figuras geométricas y que pueden ser dibujadas en un medio determinado. Podríamos crear una clase base común a ambas, llamada **Figura geométrica**, con un método llamado Dibujar.



**Figura 11.** El tipo de objeto “círculo”.



**Figura 12.** El tipo de objeto “cuadrado”.



**Figura 13.** Un círculo y un cuadrado son, al fin de cuentas, figuras geométricas.

¿Y qué se ganaría? ¿Con qué fin se hace esto? Dijimos que los objetos se comunican entre sí por medio de mensajes. Pero para que un objeto pueda enviarle un mensaje a otro, debe conocer su **tipo**, es decir, que no puede invocar el método Dibujar

de un objeto de tipo desconocido en ningún momento. En ciertas circunstancias, esto podría restar flexibilidad, ya que si sabemos que un objeto es del tipo Figura geométrica (ya sea círculo o cuadrado), sabemos que podría invocar su método Dibujar. Es aquí donde hace su aparición el **polimorfismo**.

Gracias al polimorfismo, un objeto podría invocar el método de otro objeto sabiendo que es del tipo Figura geométrica, pero sin saber a ciencia cierta si es un círculo o un cuadrado. El mecanismo citado hará que finalmente se ejecute el código Dibujar que corresponda.

Ciertamente, el polimorfismo es un concepto más amplio y profundo; más adelante, por medio de ejemplos, entenderemos mejor de qué estamos hablando y por qué es tan importante esta característica en la programación orientada a objetos.

## EL ANÁLISIS Y EL DISEÑO EN LA PROGRAMACIÓN ORIENTADA A OBJETOS

Volvamos un poco al principio de todo, cuando nos planteamos crear un programa para satisfacer una necesidad. Modelaremos nuestro problema en función de objetos, y nuestro programa estará basado en su definición y la interrelación existente entre ellos. Pero ¿cómo llegamos a dicho modelo?

El proceso de pasar de un problema real a una definición de objetos y relaciones no es trivial. Un arquitecto no construye una casa indicando al azar dónde deben ir las habitaciones, sino que existe un plano, y se tienen en cuenta distintos factores que podrían ser omitidos si a esta etapa no se le da la importancia que merece. Nosotros también somos arquitectos, pero en lugar de construir edificaciones, construimos software. Sin embargo, por algún motivo, es común que cometamos el error de realizar construcciones sin planos.

Pues bien, **¿qué sucede cuando construimos software sin planos?** Nuestros planos serán, en realidad, diagramas y documentos de diseño. Saltearnos esta etapa podría ocasionar la creación de un modelo incorrecto en el contexto de nuestro programa. Esto, a su vez, podría desembocar en:

- un aumento de complejidad desmesurado en el agregado de nuevas características (decaimiento de la capacidad de volver a usarlo);
- mayor cantidad de errores;
- mal desempeño de rendimiento de nuestro programa;
- uso desmedido de recursos de sistema (esto aumenta, en consecuencia, los requerimientos mínimos solicitados);
- comportamientos no esperados.

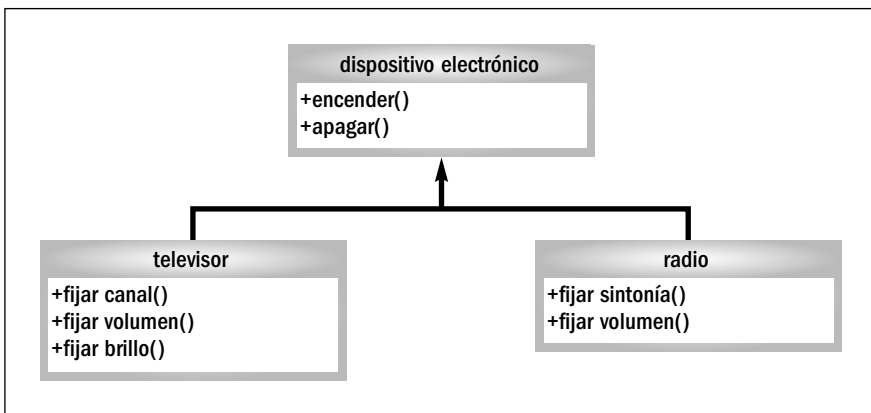
Por eso, antes de abrir nuestro entorno de desarrollo, deberemos analizar cómo modelaremos nuestro problema. Para esto, nos haremos de dos herramientas fundamentales: el **lenguaje unificado de modelado** (UML), y los **patrones de diseño**.

## El lenguaje unificado de modelado

Durante el proceso de diseño, tendremos que bajar nuestro problema a un documento entendible por el programador (papel que también podríamos cumplir nosotros). En ese documento deberemos exponer los objetos que poseerá el sistema, en qué momentos deberán ser creados y destruidos, cómo se comunicarán entre sí, etc.

Para comunicar eso, se nos podrían ocurrir muchos métodos y tipos de diagramas, sin embargo, nuestros diagramas adolecerían de aceptación y no serían entendidos por otras personas, ya que seguirían nuestras normas arbitrarias y tendríamos que explicarlas en detalle. Hace no mucho tiempo, dado que no existía ninguna herramienta estándar en el mercado para documentar diseños orientados a objetos, sucedió que a distintas personas se les ocurrió crear sus propios diagramas.

Por suerte, tres de los autores de los diagramas más populares en aquella época (Grady Booch, Ivar Jacobson y James Rumbaugh) decidieron unir esfuerzos y crear un lenguaje común que fuera el estándar del mercado, y es así como nació el **UML**. El UML está compuesto por distintos diagramas; cada uno de ellos se utiliza para mostrar un aspecto distinto del sistema. Por supuesto que hay diagramas más populares que otros, de hecho, existen algunos que rara vez tendremos la necesidad de utilizar, y otros que serán moneda corriente en la expresión de cualquier idea. El más popular es, quizás, el **diagrama estático de clases**; tan popular es que ya hemos utilizado algunos de sus elementos en los ejemplos vistos. Este diagrama muestra las clases por las cuales estará compuesto nuestro sistema (aunque la vista no tiene por qué ser de todo el sistema) y las relaciones existentes entre ellas.



**Figura 14.** El diagrama estático de clases puede o no mostrar propiedades y métodos. En este ejemplo, sólo mostramos sus métodos.

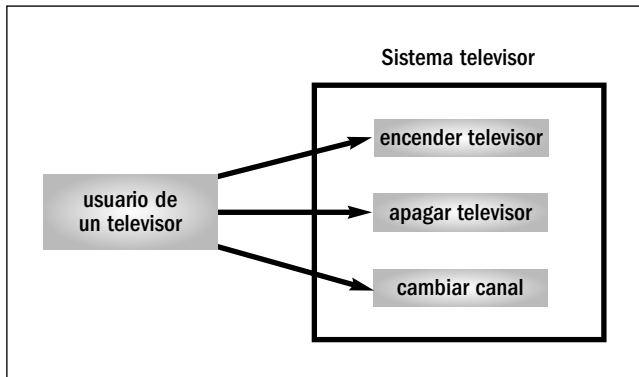
También existe el **diagrama de objetos**, que vemos en la **Figura 15**. Éstos, a diferencia del de clases, no podrán ser estáticos, ya que la cantidad de objetos existentes en nuestro sistema podrá variar con el tiempo, en función del flujo de ejecución del programa. Por lo tanto, cada uno de estos diagramas mostrará como una fotografía de nuestro sistema en un instante dado.



**Figura 15** Propiedades del objeto y sus valores.

El **diagrama de clases de uso** es una visión de la comunicación usuario/sistema. Es especialmente útil en sistemas que poseen un flujo de trabajo, donde distintos usuarios (actores) acceden a éstos desde distintas interfaces.

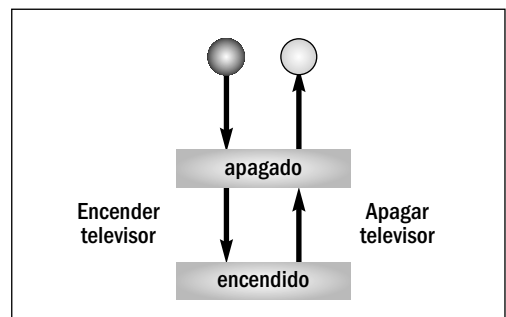
Un objeto podrá actuar de distintas maneras en función de su estado (como habíamos mencionado, éste se encuentra determinado por el valor de sus propiedades).



**Figura 16.** El diagrama de clases de uso muestra el sistema desde la visión del usuario.

El **diagrama de estados** (**Figura 17**) permite exponer claramente cuáles son y qué mensajes se requieren para pasar de uno a otro estado.

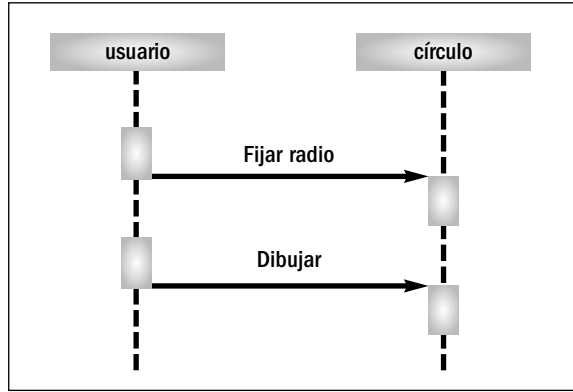
Sabemos que los objetos poseen métodos; sin embargo, podría existir cierta dependencia de un método sobre otro. Podríamos tener nuestro objeto del tipo círculo donde el método Dibujar no debería ser invocado antes de fijarle al objeto un tamaño.



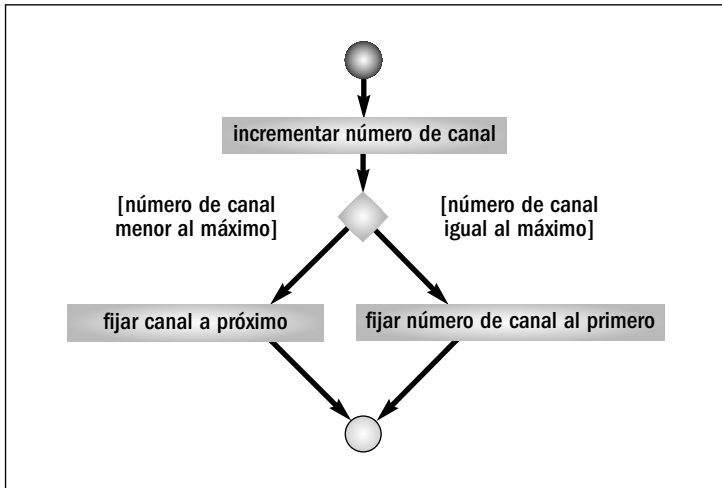
**Figura 17.** El círculo negro señala el estado inicial y el círculo con doble borde, el estado final. En este caso, es el mismo.

El **diagrama de secuencias** permite documentar la interacción entre distintos objetos en una línea de tiempos.

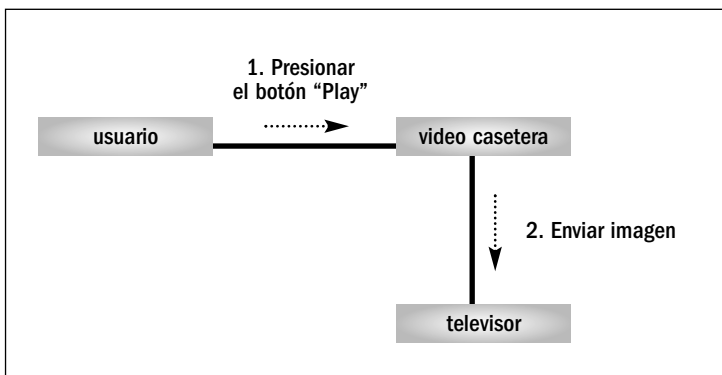
El **diagrama de actividades**, que se utiliza para exponer los pasos que debe efectuar un objeto –por ejemplo, en la ejecución de una tarea–, es una extensión del diagrama de estados, y frecuentemente se muestran divisiones en el flujo de ejecución de un proceso por medio de una toma de decisión.



**Figura 18.** El diagrama de secuencias expone en qué orden deben invocarse ciertos métodos.



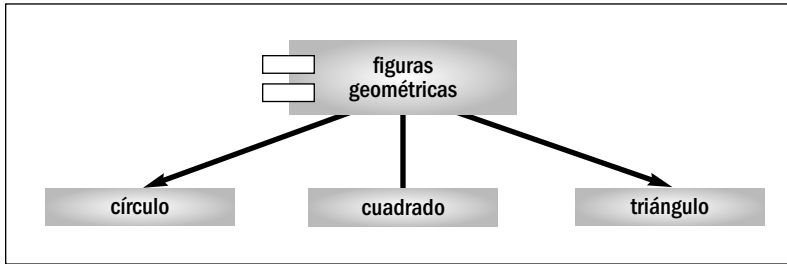
**Figura 19.** Llegado el número máximo de canal, se retorna nuevamente al primero.



**Figura 20.** Vemos cómo los objetos se relacionan entre sí.

El diagrama de colaboraciones permite documentar de qué forma distintos objetos que colaboran entre sí logran un determinado objetivo (**Figura 20**).

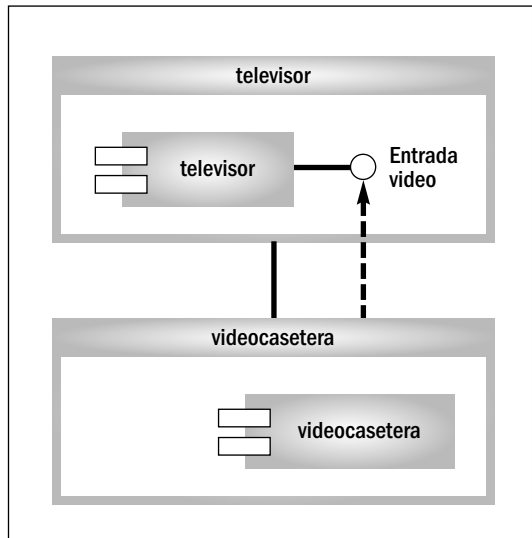
Los sistemas complejos no son monolíticos, es decir, no se encuentran armados a partir de una sola pieza, sino de componentes. El **diagrama de componentes** brinda la posibilidad de informar los componentes de un sistema complejo (**Figura 21**).



**Figura 21.** En este caso, el diagrama muestra el componente “Figuras geométricas”.

Finalmente, los **diagramas de distribución** exponen de qué forma está compuesto un sistema, pero de manera física (**Figura 22**).

Es importante notar que los diagramas UML son herramientas, y, como tales, deben actuar en nuestro favor. Utilizaremos el diagrama que nos permita exponer determinadas aristas de nuestro sistema, es decir, el que nos sea más útil en un momento determinado; por lo tanto, es posible que algunos de los diagramas no nos sean inútiles o no tengan el suficiente valor para incorporarlo a nuestro documento.



**Figura 22.** Este diagrama muestra la ubicación de los grupos de componentes.

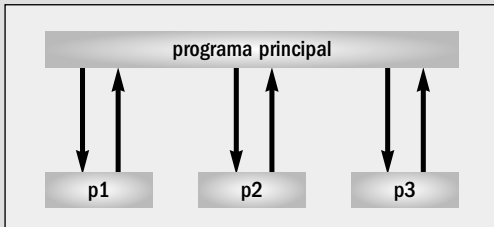
## RESUMEN

Como vemos, la programación orientada a objetos tiene características muy particulares. En este primer capítulo, hemos tratado de poner toda la teoría sobre la mesa, para ya si tener un apoyo que nos permita comenzar con la práctica concreta de la programación en C++.



## TEST DE AUTOEVALUACIÓN

- 1 ¿Cuál es la característica de la programación no estructurada?
- 2 ¿En qué se basa la programación modular?
- 3 ¿En qué se basa la programación procedural? ¿Qué diferencia existe con la programación modular?
- 7 ¿Cuáles son las principales características de los objetos?
- 8 ¿Qué significa “lenguaje de modelo unificado” (UML)?
- 9 Mencione los casos en que es posible emplear los diagramas UML.



- 10 ¿Qué son los patrones de diseño?

- 4 ¿Cuál es la característica de la programación orientada a objetos?
- 5 ¿A qué nos referimos puntualmente cuando hablamos de objetos?
- 6 ¿Podemos decir que un objeto es un conjunto de métodos y propiedades?

