

procesadores, grandes capacidades de los discos duros, potencia de gráficos, facilidades de conexión en redes, etc., incrementan la complejidad de los programas. Además de todo lo anterior, los usuarios esperan interfaces gráficas basados en ventanas, acceso transparente a los datos almacenados en computadoras mini o grandes y, naturalmente, la capacidad de trabajo en red. Para enfrentarse a esta complejidad, los programadores de la década de los noventa utilizan la *programación orientada a objetos (POO)*¹. POO es un nuevo medio de organizar código y datos que auguran un control creciente sobre la complejidad del proceso de desarrollo del software.

Programación orientada a objetos (POO) no es una idea nueva; sus conceptos fundamentales ya existían en lenguajes tales como Simula 67 y Smalltalk. Las propiedades fundamentales de la POO son: *objetos* (abstracción de datos), *herencia* y *polimorfismo*. Sin embargo, sí es nuevo el interés de los programadores, por POO y por los lenguajes OO, especialmente C++.

Si usted es un programador experimentado de C, encontrará fácil aprender la sintaxis de C++. Caso de tener conocimientos básicos de C o Pascal, la *emigración* a C++ le puede resultar un poco más ardua, pero de cualquier forma, el esfuerzo suplementario estamos seguros le producirá grandes beneficios.

El mejor medio para aprender C++, consideramos que es comprender bien los conceptos de POO y ver como C++ los soporta. Este libro explica POO a través de ejemplos, a la vez enseña el lenguaje de programación C++, y este capítulo explica en particular la terminología básica POO y los lenguajes de programación orientados a objetos.

LA EVOLUCION DE LA PROGRAMACION

POO (*programación orientada a objetos*) es un importante conjunto de técnicas que se pueden utilizar para hacer el desarrollo de programas más eficiente mientras se mejora la fiabilidad de los programas resultantes. En esencia, *POO* es un nuevo medio de enfocar el trabajo de programación. Sin embargo, a fin de comprender lo que es POO, es necesario comprender sus raíces. Así pues, comencemos examinando la historia del proceso de programación, analizando cómo evolucionó POO y deduciendo, en consecuencia, por qué es tan importante este concepto.

Programación lineal

La programación de computadoras es una joven disciplina. Las primeras computadoras se desarrollaron hace cuarenta años. En estos cuarenta años la evolución de la programación ha sido enorme y se ha dirigido hacia las necesidades del programador, es decir, se ha humanizado.

¹ En inglés, *Object Oriented Programming (OOP)*.

Los primeros lenguajes de programación se diseñaron para desarrollar programas que realizaban tareas relativamente simples. La mayoría de estos programas eran cortos, menos de 100 líneas de código fuente.

A medida que aumentaba la potencia de las computadoras, se requerían programas cada vez más complejos. Los lenguajes de programación primitivos eran inadecuados para esas tareas de programación. Los lenguajes de programación lineal (primeras versiones de BASIC, COBOL y FORTRAN) no tenían facilidad para reutilizar el código existente de programas. De hecho, se duplicaban segmentos de software cada vez más en muchos programas. Los programas se ejecutaban en secuencias lógicas, haciendo su lógica difícil de comprender. El control del programa era difícil y se producían continuos saltos a lo largo del referido programa. Aún más, los lenguajes lineales no tenían capacidad de controlar la visibilidad de los elementos **dato**. Todos los datos eran *globales*, por consiguiente podían ser modificados por cualquier parte del programa. No existían *datos locales* y en consecuencia cualquier ligera modificación en un dato podía afectar a todo el programa.

La evolución de los lenguajes produjo la aparición de los lenguajes estructurados (Pascal, C, Ada, ...) así como una nueva estructura, el **procedimiento** (*subprograma* o *rutina*) que consistía en secuencias de instrucciones que realizaban una tarea determinada. Estos procedimientos tienen un nombre y suelen ser realizados por un programador. Sin embargo, este nacimiento no supuso la solución a la resolución de problemas complejos mediante programas, ya que se requerían numerosos procedimientos y prolíficos sistemas de comunicación entre ellos.

Programación modular

En principio, la solución a estos problemas era evidente: romper los programas grandes en componentes más pequeños que pueden ser construidos indepen-

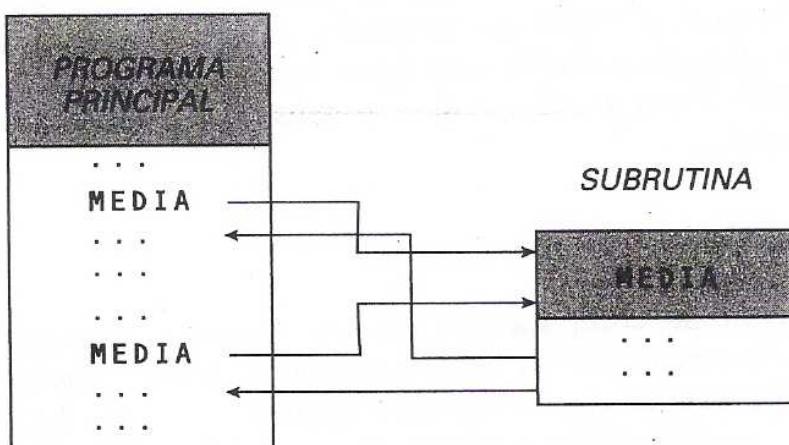


FIGURA 1-1. Una subrutina llamada desde dos posiciones.

dientemente, a continuación combinarlos para formar el sistema completo. Esta estrategia general se conoce como **programación modular**, y ha sido el paradigma que más ha influido en la construcción de software durante muchos años.

El soporte más elemental para la programación modular llegó con la aparición de la *subrutina* al principio de la década de los cincuenta. Una subrutina ha creado una secuencia de instrucciones a las que se les da un nombre independiente; una vez que se ha definido, la subrutina se puede ejecutar simplemente incluyendo su nombre en el programa siempre que se requiera. Las subrutinas proporcionan una división natural de la tarea: diferentes programadores escriben las diferentes subrutinas, y a continuación se ensamblan las subrutinas terminadas en un único programa.

Aunque las subrutinas proporcionan el mecanismo básico de la programación modular, se necesita mucha disciplina para crear software bien estructurado. Sin esta disciplina, es fácil escribir programas complicados y tortuosos difíciles de modificar y comprender y casi imposible de mantener. Esta ha sido la panorámica durante muchos años en la industria del software.

Programación estructurada

En la década de los sesenta y principio de los setenta, se produjo la primera gran revolución: la *introducción de la programación estructurada*. Los programas estructurados se organizan de acuerdo a las operaciones que ellos ejecutan. En esencia, el programa se descompone en procedimientos individuales (también se conocen como funciones) cada uno de los cuales se descompone en subprocedimientos hasta llegar al nivel de procedimiento individual sencillo. Equipo de diversos programadores escriben los diferentes procedimientos que se ensamblan posteriormente en el programa completo. La información se pasa entre procedimientos utilizando parámetros y los procedimientos pueden tener datos locales a los que no se puede acceder desde fuera del ámbito del procedimiento.

El objetivo final era hacer el desarrollo de software más fácil para el programador mientras se mejoraba la fiabilidad y mantenibilidad. Aislando los procesos dentro de funciones, un programa estructurado minimiza el riesgo de que un procedimiento afecte a otro. Esto facilita también la detección de problemas mediante la localización de errores, y hace el programa más claro. Las variables globales desaparecen y se han sustituido por parámetros y variables locales que tienen un ámbito controlable más pequeño.

Un concepto muy importante introdujo la programación estructurada: **abstracción**. La **abstracción** se puede definir como la capacidad de examinar algo sin preocuparse de los detalles internos. En un programa estructurado, es suficiente conocer que un procedimiento dado realiza una tarea específica. El *cómo* se realiza la tarea no es importante; mientras el procedimiento sea fiable, se puede utilizar sin tener que conocer cómo funciona su interior. Esto se conoce como una abstracción funcional y es el núcleo de la programación estructurada. Hoy casi todos los lenguajes de programación tienen construcciones que facilitan la programación estructurada.

Una debilidad de la programación estructurada aparece cuando programadores diferentes trabajan en una aplicación como un equipo. En un programa estructurado, a cada programador se le asigna la construcción de un conjunto específico de funciones y tipos de datos. Dado que programadores diferentes manipulan funciones separadas que pueden referirse a tipos de datos mutuamente compartidos, los cambios de un programador se deben reflejar en el trabajo del resto del equipo. Los problemas de la comunicación entre los miembros del equipo pueden producir graves errores. Otro problema serio de la programación estructurada es que raramente es posible anticipar el diseño de un sistema completo antes de que se implemente realmente.

En esencia, *un defecto de la programación estructurada*, como se acaba de ver, consiste en la separación conceptual de datos y código. Este defecto se agrava a medida que el tamaño del programa se hace más complejo.

Abstracción de datos

La abstracción se define como la «*extracción de las propiedades esenciales de un concepto*». Con la abstracción de datos, las estructuras de datos e ítems se pueden utilizar sin preocuparse sobre los detalles exactos de la implementación. Por ejemplo, los números en coma flotante son abstracciones en todos los lenguajes de programación; no tiene que preocuparse de la representación binaria exacta de un número en coma flotante cuando se le asigna un valor. No se requiere conocer las complejidades de la multiplicación binaria para multiplicar valores en coma flotante; lo más importante es que los números en coma flotante funcionen de modo correcto.

La abstracción de datos permite no preocuparse de los detalles no esenciales. La abstracción de datos existe en casi todos los lenguajes de programación. Las estructuras de datos y los tipos de datos son un ejemplo de abstracción. Los procedimientos y las funciones son otro ejemplo. Sin embargo, sólo recientemente han emergido lenguajes que soportan sus propios tipos abstractos de datos (**TAD**), como Pascal, Ada y Modula-2, y naturalmente C++.

¿QUE ES LA PROGRAMACION ORIENTADA A OBJETOS?

El término *programación orientada a objetos (POO)*, hoy día ampliamente utilizado, es difícil de definir, ya que no es un concepto nuevo sino que ha sido el desarrollo de técnicas de programación desde principios de la década de los setenta, aunque sea en la década de los noventa cuando ha aumentado su difusión, uso y popularidad. No obstante, se puede definir POO como una técnica o estilo de programación que utiliza objetos como bloque esencial de construcción.

Los objetos son en realidad como los tipos abstractos de datos, ampliamente utilizados por los programadores en la década de los setenta y ochenta. Un *tipo abstracto de datos (TAD)*² es un tipo de dato definido por el progra-

² En inglés, ADT (*Abstract Data Type*).

mador junto con un conjunto de operaciones que se pueden realizar sobre ellos. Se denominan *abstractos* para diferenciarlos de los tipos de datos fundamentales o básicos definidos, por ejemplo en C, tales como `int`, `char` y `double`. En C se puede definir un tipo abstracto de dato utilizando `typedef` y `struct` y la implementación de las operaciones con un conjunto de funciones. C++ tiene muchas facilidades para definir y utilizar tipo TAD. Al igual que los tipos de datos definidos por el usuario, un objeto es una colección de datos, junto con las funciones asociadas, utilizadas para operar sobre esos datos. Sin embargo, la potencia real de los objetos reside en las propiedades que soporan: *herencia*, *encapsulación* (o *encapsulamiento*) y *polimorfismo* junto con los conceptos de *objetos*, *clases*, *métodos* y *mensajes*.

Trabajando con objetos

En programación convencional, los programas se dividen en dos componentes: procedimientos y datos. Cada procedimiento actúa como una *caja negra*; es un componente que realiza una tarea específica tal como convertir un conjunto de números o visualizar una ventana. Si las cajas negras se dividen correctamente, se pueden escribir código para cada una sin preocuparse de lo que internamente hacen otras cajas negras. La principal ventaja de utilizar este método es que ayuda a desarrollar programas que son modulares y transportables.

Este método permite empaquetar código de programa en procedimientos. Pero ¿qué sucede con los datos? Las estructuras de datos utilizadas en programas son, frecuentemente, globales o se pasan explícitamente con parámetros. En esencia los datos, se tratan separadamente de los procedimientos.

Cuando se utilizan métodos de **POO**, un programa se divide en componentes que contienen procedimientos y datos. Cada componente se considera un *objeto*.

Un **objeto** es una unidad que contiene datos y las funciones que operan sobre esos datos. A los elementos de un objeto se les conoce como *miembros*; las funciones que operan sobre los objetos se denominan *métodos* (en C++ los métodos se denominan también *funciones miembro*) y los datos se denominan *miembros datos*. En C++ un programa consta de objetos. Los objetos de un programa se comunican entre sí mediante el *paso* o *envío de mensajes* (acciones que debe ejecutar el objeto).

¿Qué son objetos en **POO**? La respuesta es cualquier entidad del mundo real que se pueda imaginar:

- *Objetos físicos*

Automóviles en una simulación de tráfico.

Aviones en un sistema de control de tráfico aéreo.

Componentes electrónicos en un programa de diseño de circuitos.

Animales mamíferos.

- *Elementos de interfaces gráficos de usuarios*

- Ventanas.
- Iconos.
- Menús.
- Objetos gráficos (líneas, rectángulos, círculos).
- Ratones.
- Teclados.

- *Estructuras de datos*

- Arrays.
- Pilas.
- Colas.
- Arboles binarios.

- *Tipos de datos definidos por el usuario*

- Números complejos.
- Hora del día.
- Puntos de un plano.

Examinaremos un ejemplo para ver como **POO** actúa de modo diferente a la programación convencional. Supongamos que se dispone de un objeto tal como una ventana en la pantalla, y se desea definir las cuatro operaciones requeridas para mover el objeto ventana en la pantalla:

1. Mover el objeto ventana a la derecha.
2. Mover el objeto ventana a la izquierda.
3. Mover el objeto ventana hacia arriba.
4. Mover el objeto ventana hacia abajo.

Utilizando métodos de programación convencional, se puede escribir un procedimiento independiente para cada operación (Figura 1-2). Estos procedimientos actúan como cajas negras y suponen un mecanismo idóneo para dividir operaciones. Desgraciadamente, los datos que definen el objeto son independientes de cada uno de los procedimientos.

Examinaremos ahora el método para definir el objeto visual utilizando métodos orientados a objetos. Aplicando este método combinamos datos y procedimientos en un único paquete. Para ello, el primer paso es crear un objeto llamado *ventana*. Esta ventana, contiene los procedimientos que definen cómo se mueve la ventana. Cuando la ventana recibe una orden tal como *mover a izquierda* se ejecuta el procedimiento correspondiente.

Los objetos soportan una serie de características específicas de los mismos:

- Se agrupan en tipos denominados *clases*.
- Contienen datos internos que definen su estado actual.
- Soportan *ocultación de datos*.

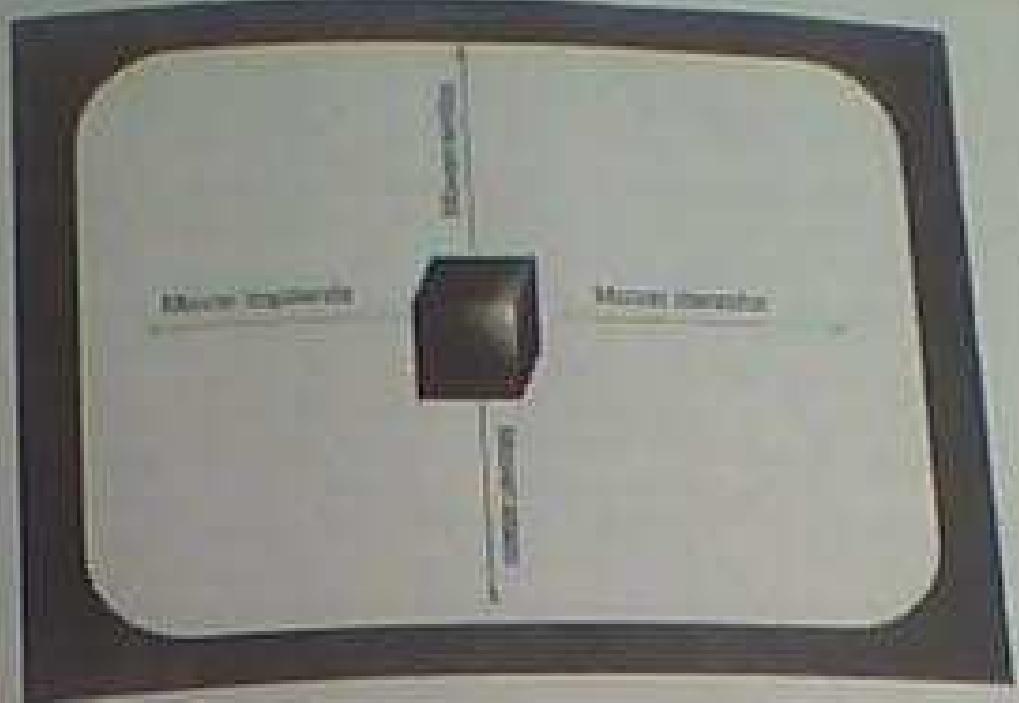


FIGURA 1-2. Captura de pantalla de un televisor mostrando una aplicación.

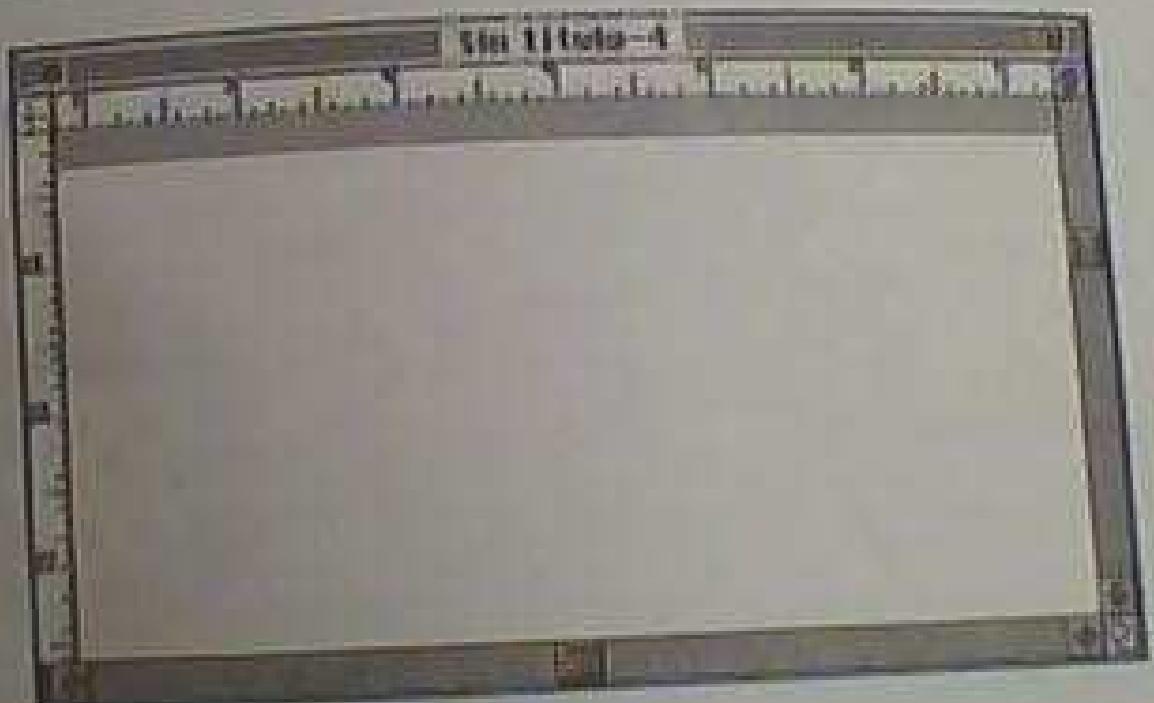


FIGURA 1-3. Captura de pantalla.

- Pueden *heredar* propiedades de otros objetos.
- Pueden comunicarse con otros objetos enviando o pasando mensajes.
- Tienen *métodos* que definen su *comportamiento*.

Definición de objetos

* Un *objeto* es una unidad que contiene *datos* y *las funciones que operan* sobre esos datos. Los datos se denominan *miembros dato* y las funciones, *miembros función* (también *funciones miembro*) o *métodos*.

Los datos y las funciones se encapsulan en una única entidad. Los datos están ocultos y solo mediante las funciones miembro es posible acceder a ellos. Los términos *encapsulación (encapsulamiento) de datos* y *ocultación de datos* son términos claves empleados en los lenguajes de programación.

Un método gráfico para representar los objetos se representa en la Figura 1.4. Un objeto se representa con un cuadro. El cuadro se etiqueta con el nombre del objeto. El cuadro (perímetro del mismo) representa el límite o frontera entre el interior y el exterior de un objeto. En el interior de un objeto están las variables locales —los campos miembro— y las funciones. Un campo se representa por un cuadro rectangular, una función por un hexágono. Los campos y las funciones se rotulan con sus nombres. Todos los campos miembros y las funciones que están completamente en el interior del cuadro objeto son ocultos desde el exterior, lo que significa que estas características están *encapsuladas*. Los campos o funciones que se extienden fuera del cuadro son accesibles, desde el exterior y actúan de interfaz. El acceso a las características miembro (campos y funciones) es posible a través del interfaz del objeto.

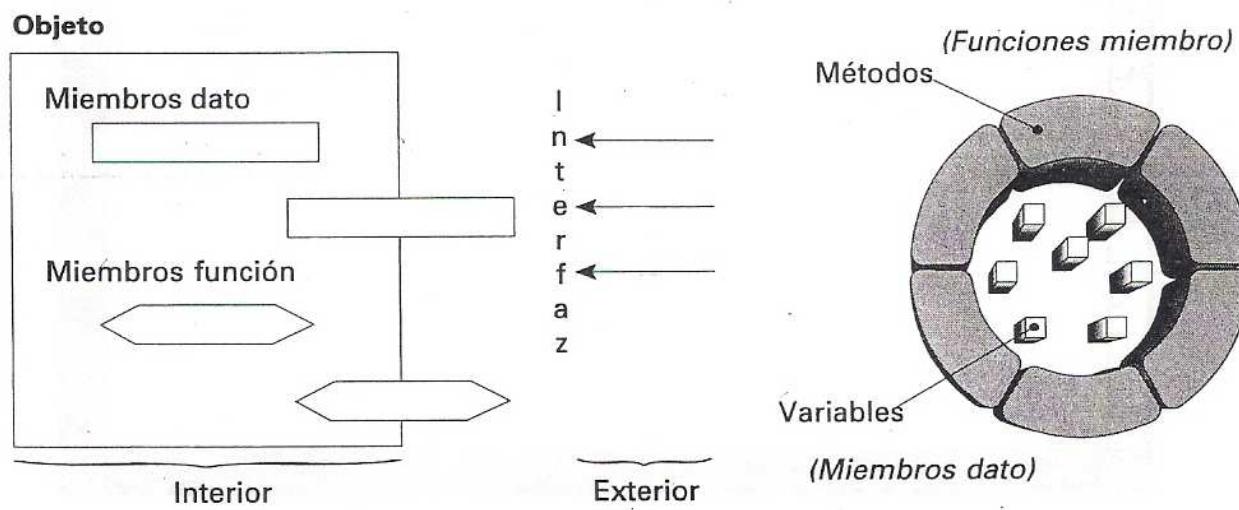


FIGURA 1-4. Diagrama de un objeto.

Un programa, típicamente, consta de un número de objetos que se comunican entre sí mediante métodos que son, a su vez, llamadas funciones miembro del objeto invocado. La organización de un programa en C++ se muestra en la Figura 1-5.

En C++ las funciones se denominan *funciones miembro* y en otros lenguajes orientados a objetos (OO) —tales como Smalltalk— *métodos*.

Clases

En POO se suele decir que los objetos son *miembros de clases*. Una *clase* es un tipo definido por el usuario que determina las estructuras de datos y las

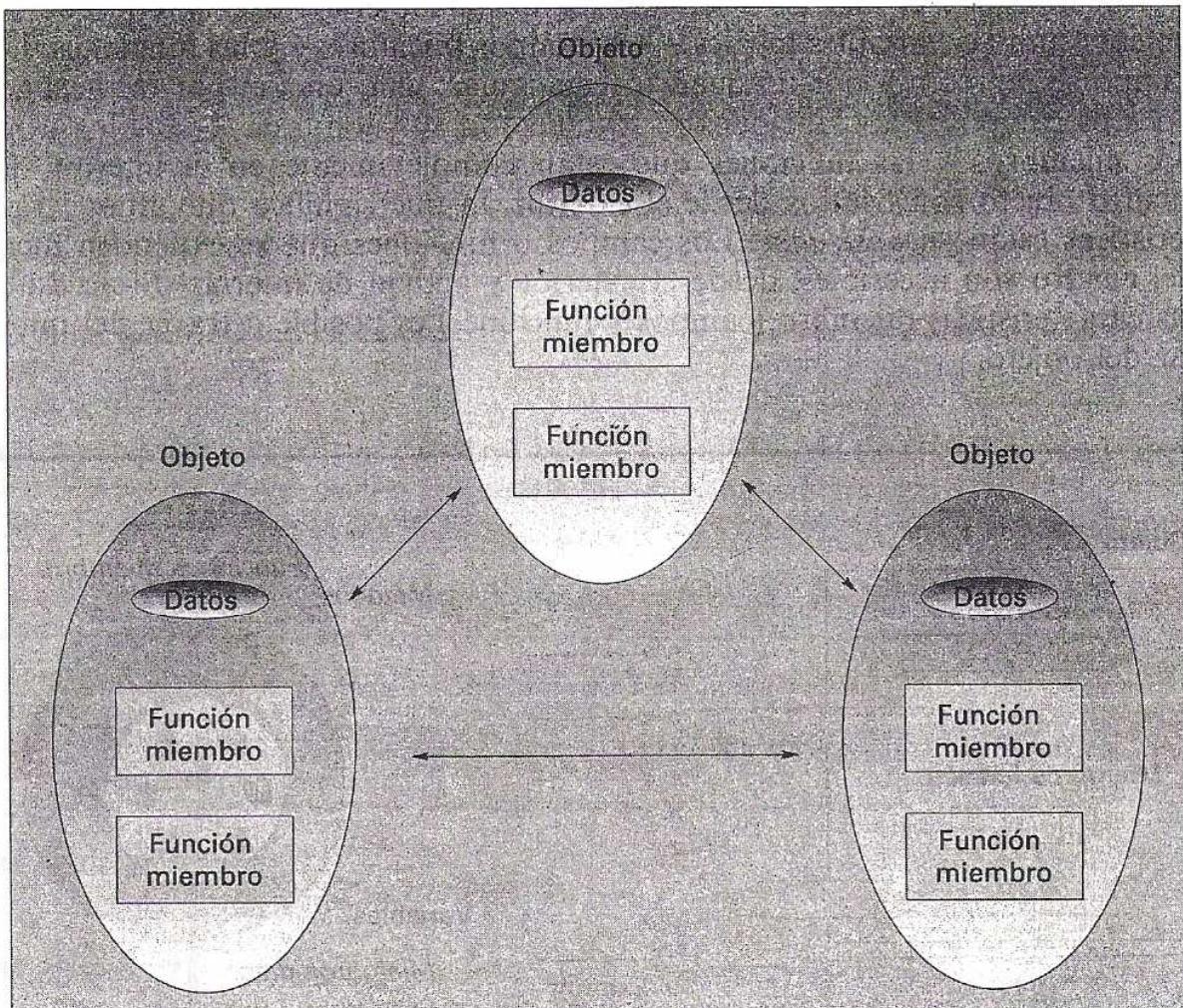


FIGURA 1-5. Programa orientado a objetos.

operaciones asociadas con ese tipo. Las clases son como plantillas o modelos que describen cómo se construyen ciertos tipos de objetos. Cada vez que se construye un objeto de una clase, se crea una *instancia* («instance») de esa clase. Por consiguiente, los objetos son instancias de clases. En general, los términos *objetos* e *instancias de una clase* se pueden utilizar indistintamente.

- Una clase es una colección de objetos similares y un objeto es una instancia de una definición de una clase. Una clase puede tener muchas instancias y cada una es un objeto independiente.

Una clase es simplemente un modelo que se utiliza para describir uno o más objetos del mismo tipo.

Así, por ejemplo, sea una clase ventana, un tipo de dato, que contenga los miembros dato:

```
posx, posy
tipo_ventana
tipo_borde
color_ventana
```

y unas funciones miembro:

```
mover_horizontal
mover_vertical
```

Un objeto de la clase ventana es una ventana concreta —*una instancia de la clase*— cuyos datos tienen por valores x, y; desplegable; línea doble, amarillo.

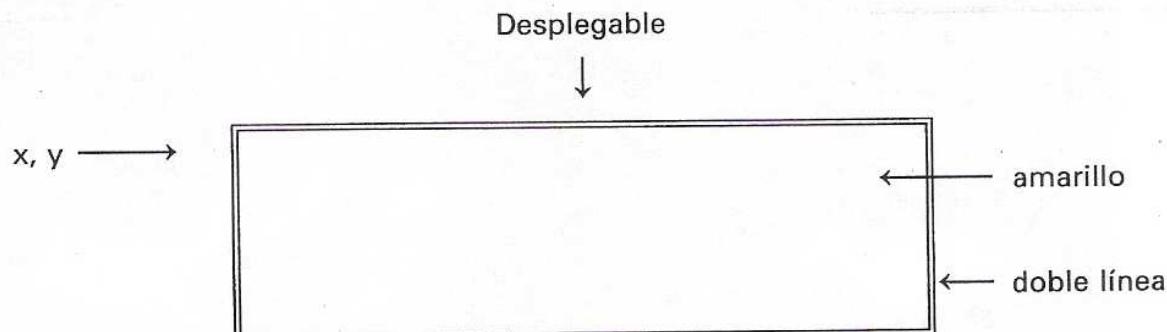


FIGURA 1-6. Relación entre una clase ventana y un objeto ventana.

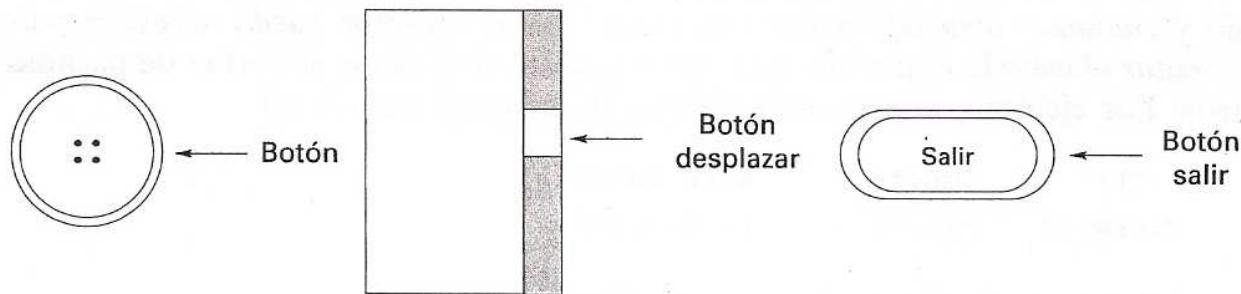


FIGURA 1-7. Construcción de botones en entornos de ventanas (Windows).

La comunicación con el objeto se realiza a través del paso de mensajes. «*El envío de un mensaje a una instancia de una clase produce la ejecución de un método (función miembro en C++)*». El paso de mensajes es el término utilizado para referirnos a la invocación o llamada de una función miembro de un objeto. La noción de paso de mensajes es fundamental en todos los lenguajes de programación orientada a objetos.

Algunos ejemplos típicos de clases en los interfaces gráficos de usuarios, son los populares botones de los populares interfaces Windows, o X-Windows.

Mensajes: activación de objetos

A diferencia de los métodos tradicionales, cuyos elementos datos son pasivos, los objetos pueden ser activados mediante la recepción de mensajes. Un *mensaje* es simplemente una petición de un objeto a otro objeto para que éste se comporte de una determinada manera, ejecutando uno de sus métodos. La técnica de enviar mensajes se conoce como *paso de mensajes*. La Figura 1-8 representa un objeto A (*emisor*) que envía un mensaje **Test** al objeto B (*receptor*).

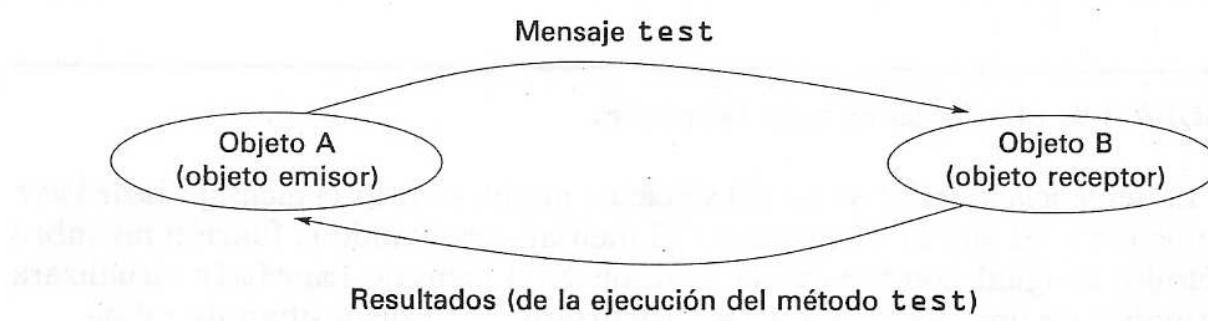


FIGURA 1-8. Paso de mensajes entre objetos.

Estructuralmente, un mensaje consta de tres partes: *la identidad del objeto receptor, el método (función miembro) del receptor cuya ejecución se ha solicitado y cualquier otra información adicional que el receptor pueda necesitar para ejecutar el método requerido*. Esta información suele darse en forma de parámetros. Por ejemplo, una posible sintaxis de mensaje podría ser

"T1 Imprimir Hola Mundo"
receptor método parámetros

En el caso de C++, la notación utilizada es

nombre_del_objeto.función_miembro o método

Así, en el caso de un objeto T1 (miembros *dato*: nombre de un alumno y *curso* en que está matriculado; miembros *función*: LeerNombre, obtiene el nombre del alumno e Imprimir, visualiza el citado nombre), que recibe un mensaje Imprimir, se expresaría así:

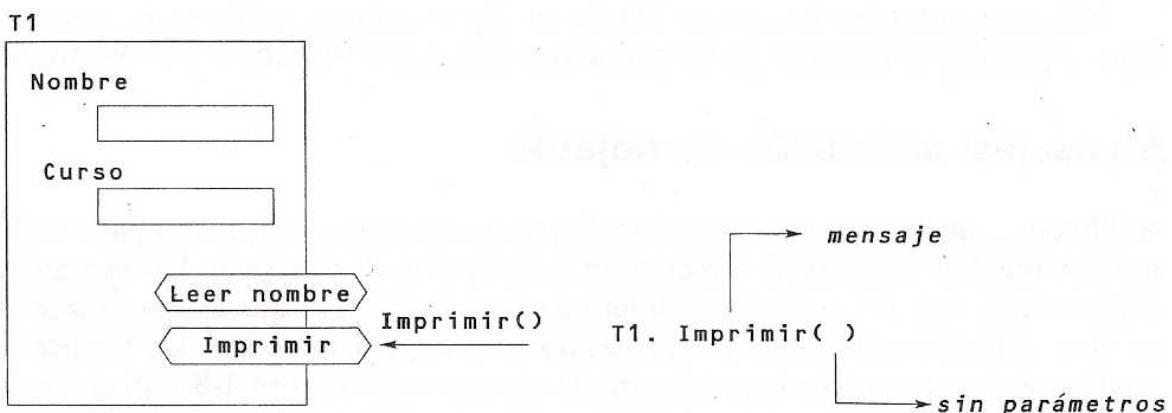


FIGURA 1-9. Envío del mensaje Imprimir.

La sentencia anterior se lee del siguiente modo: «enviar el mensaje **imprimir** al objeto **t1**. El objeto **t1** reacciona al mensaje ejecutando la función miembro (método) de igual nombre que el mensaje. Si el mensaje **Imprimir** visualizara el nombre de una persona, éste se visualizaría en el dispositivo de salida.

Como se ha comentado anteriormente el mensaje podría llevar unos parámetros. Así, en el caso de enviar el mensaje LeerNombre consideremos la invocación de:

```
t1.LeerNombre("Luis Mackoy Flanagan");
```

Esta sentencia representa el envío del mensaje `LeerNombre` al objeto `t1` con el argumento `"Luis Mackoy Flanagan"`. El objeto `t1` es el *receptor* del mensaje y —como respuesta al mismo— ejecuta su función miembro `LeerNombre` con el parámetro pasado.

Los mensajes juegan un papel vital en POO; sin ellos, los objetos que se definen no se podrán comunicar con otros objetos. Desde un punto de vista convencional, el *paso de mensajes* no es más que el sinónimo de *llamada a una función*. De hecho, ésto es lo que sucede cuando un mensaje se envía a un programa C++.

Programa orientado a objetos

Un programa orientado a objetos es una colección de clases. Necesitará una función principal que cree objetos y comience la ejecución mediante la invocación de sus funciones miembro o métodos.

Esta organización conduce a separar partes diferentes de una aplicación en distintos archivos. La idea consiste en poner la descripción de la clase para cada una de ellas en un archivo separado. La función principal también se pone en un archivo independiente. El compilador ensamblará, entonces, el programa completo a partir de los archivos independientes en una única unidad. Naturalmente, las clases que están en archivos independientes en una única unidad. Naturalmente, las clases que están en archivos independientes se pueden utilizar en más de una aplicación.

En realidad, cuando se ejecuta un programa orientado a objetos, ocurren tres acciones o sucesos. En primer lugar, se crean los objetos cuando se necesitan (el mecanismo para creación de objetos se explicará en el capítulo 5). Segundo, los mensajes se envían desde unos objetos y se reciben en otros, a medida que el programa procesa internamente información o responde a la entrada del usuario. Por último, se borran los objetos cuando ya no son necesarios y se recupera la memoria ocupada por ellos.

Herencia

Una característica muy importante de los objetos y las clases es la *herencia*. La herencia es la propiedad que permite a los objetos construirse a partir de otros objetos. El concepto de herencia está presente en nuestras vidas diarias donde las clases se dividen en subclases. Así por ejemplo, las clases de animales se dividen en mamíferos, anfibios, insectos, pájaros, etc. La clase de vehículos se divide en automóviles (coches o carros), autobuses, camiones y motocicletas.

El principio de este tipo de división es que cada subclase comparte características comunes con la clase de la que se deriva. Los automóviles, camiones, autobuses y motocicletas —que pertenecen a la clase vehículo— tienen ruedas y un motor; son las características de vehículos. Además de las características

compartidas con otros miembros de la clase, cada subclase tiene sus propias características particulares: autobuses, por ejemplo, tienen un gran número de asientos, un aparato de televisión para los viajeros, mientras que las motocicletas tienen dos ruedas, un manillar y un asiento doble.

La idea de herencia se muestra en la Figura 1-10. Observen en la figura que las características A y B que pertenecen a la clase base, también son comunes a

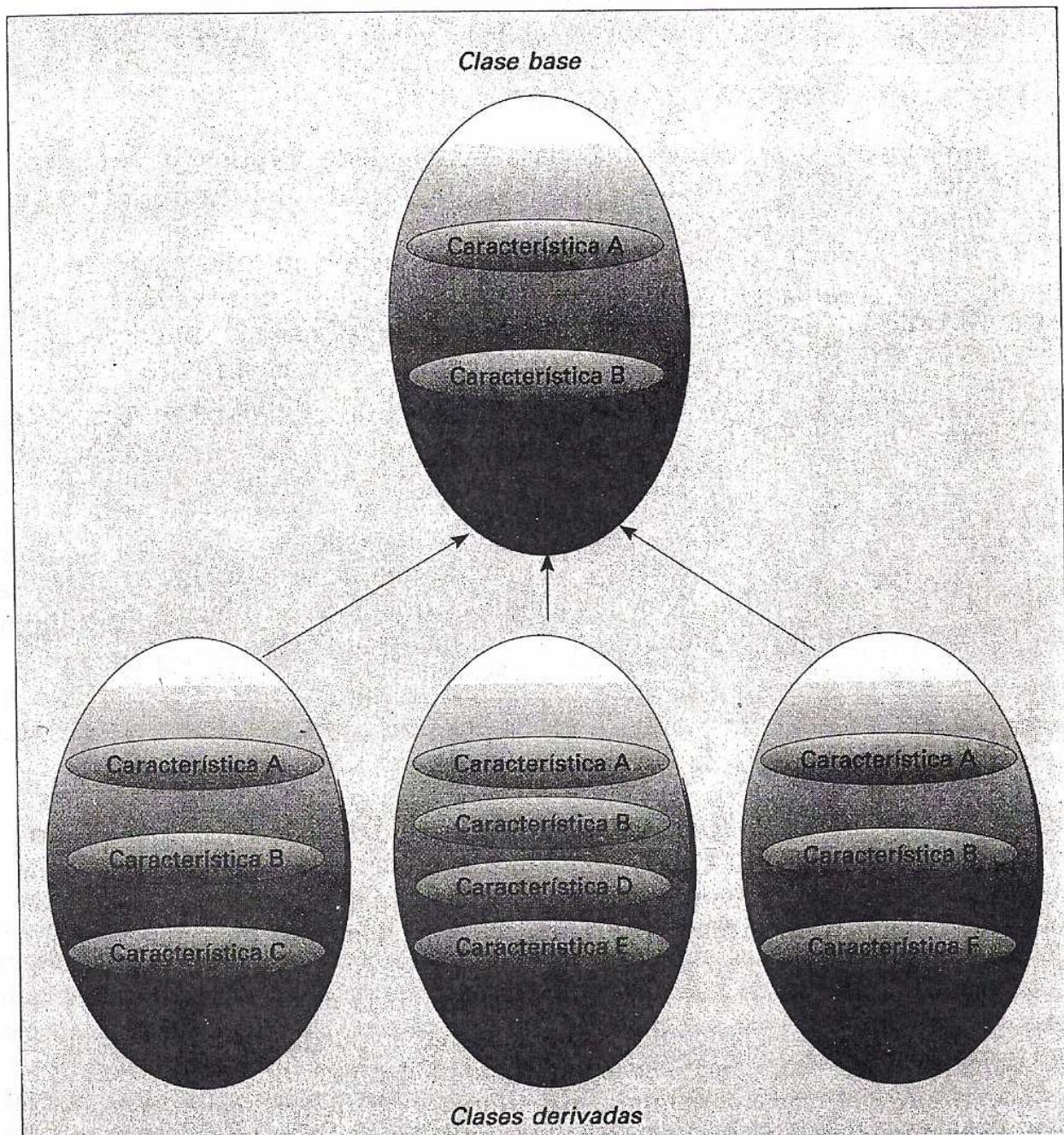


FIGURA 1-10. Herencia.

todas las clases derivadas, y a su vez estas clases derivadas tienen sus propias características.

De modo similar en POO, una clase se puede dividir en *subclases*. En C++ la clase original se denomina *clase base*; las clases que se definen a partir de la clase base, compartiendo sus características y añadiendo otras nuevas, se denominan *clases derivadas*.

Las clases derivadas pueden heredar código y datos de su clase base añadiendo su propio código especial y datos a la misma.

La herencia permite definir nuevas clases a partir de clases ya existentes.

Por ejemplo, si se define una clase denominada `figura_geométrica` se pueden definir las subclases o clases derivadas `cuadrado`, `círculo`, `rectángulo` y `triángulo`.

La clase `figura_geometrica` posee una función miembro `calcular_superficie` que proporciona el valor cero. La clase `cuadrado` posee el atributo `Lado` y redefine la función `calcular_superficie()` que proporciona el valor "`Lado * Lado`". La clase `círculo` posee el atributo `radio` y redefine la función `calcular_superficie()` que proporciona el valor `radio*radio*pi` (Figura 1-11).

La herencia impone una relación jerárquica entre clases en la cual una clase *hija* hereda de su clase *padre*. Si una clase sólo puede recibir características de otra clase base, la herencia se denomina *herencia simple*.

Si una clase recibe propiedades de más de una clase base, la herencia se denomina *herencia múltiple*. Muchos lenguajes orientados a objetos no soportan herencia múltiple, sin embargo C++ sí incorpora esta propiedad.

Polimorfismo

En un sentido literal, *polimorfismo* significa la cualidad de tener más de una forma. En el contexto de **POO**, el polimorfismo se refiere al hecho que una misma operación puede tener diferente comportamiento en diferentes objetos. En otras palabras, diferentes objetos reaccionan al mismo mensaje de modo diferente. Por ejemplo, consideremos la operación sumar. En un lenguaje de programación el operador `+` representa la suma de dos números (`x+y`) de diferentes tipos: enteros, coma flotante, ... Además se puede definir la operación de sumar dos cadenas: *concatenación*, mediante el operador `suma`.

De modo similar, supongamos un número de figuras geométricas que responden todas al mensaje, *dibujar*. Cada objeto reacciona a este mensaje visualizando su figura en una pantalla de visualización. Obviamente, el mecanismo real para visualizar los objetos difiere de una figura a otra, pero todas las figuras realizan esta tarea en respuesta al mismo mensaje.

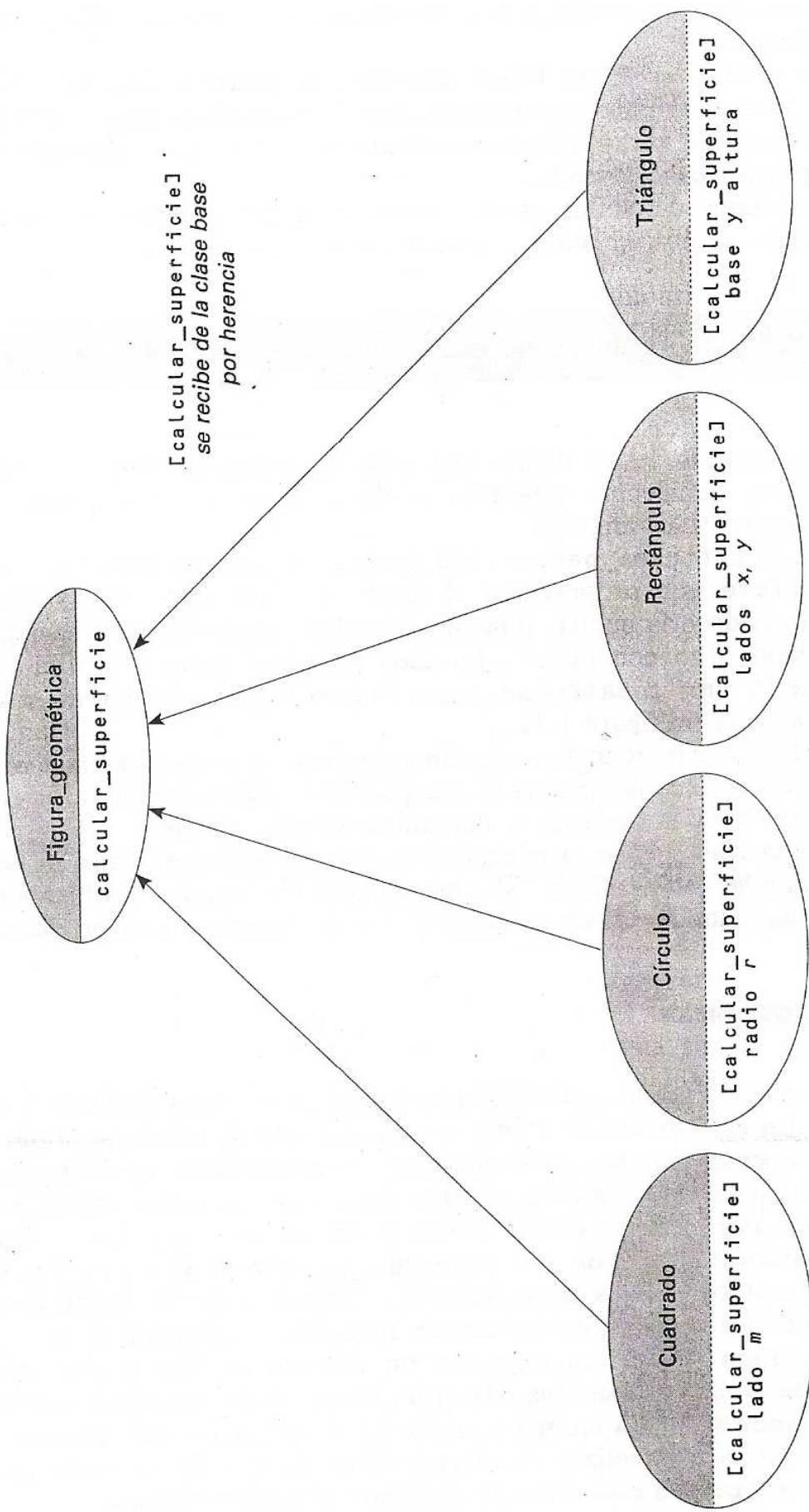


FIGURA 1-11. Clases: base y derivadas.