CAPITULO 3

CONCEPTOS FUNDAMENTALES DE PROGRAMACION ORIENTADA A OBJETOS

CONTENIDO

La programación orientada a objetos es un importante conjunto de técnicas que pueden utilizarse para hacer el desarrollo de programas más eficientes, a la par que mejora la fiabilidad de los programas de computadora. En la programación orientada a objetos, los objetos son los elementos principales de construcción. Sin embargo, la simple comprensión de lo que es un objeto, o bien el uso de objetos en un programa, no significa que esté programado en un modo orientado a objetos. Lo que cuenta es el sistema en el cual los objetos se interconectan y comunican entre sí.

En este texto nos limitaremos al campo de la programación, pero es también posible hablar de sistemas de administración de bases de datos orientadas a objetos, sistemas operativos orientados a objetos, interfaces de usuarios orientados a objetos, etc.

3.1. PROGRAMACION ESTRUCTURADA

La programación estructurada se emplea desde el principio de la década de los setenta y es uno de los métodos más utilizados en el campo de la programación

La *técnica descendente* o *el refinamiento sucesivo* comienza descomponiendo el programa en piezas manejables más pequeñas, conocidas como *funciones* (subrutinas, subprogramas o procedimientos), que realizan tareas menos complejas. Un programa estructurado se construye rompiendo el programa en funciones. Esta división permite escribir código más claro y mantener el control sobre cada función

Un concepto importante se introdujo con la programación estructurada, ya comentado anteriormente: *la abstracción*, que se puede definir como la capacidad para examinar algo sin preocuparse de sus datos internos. En un programa estructurado es suficiente conocer que un procedimiento dado realiza una tarea específica. El cómo se realiza esta tarea no es importante, sino conocer cómo se utiliza correctamente la función y lo que hace.

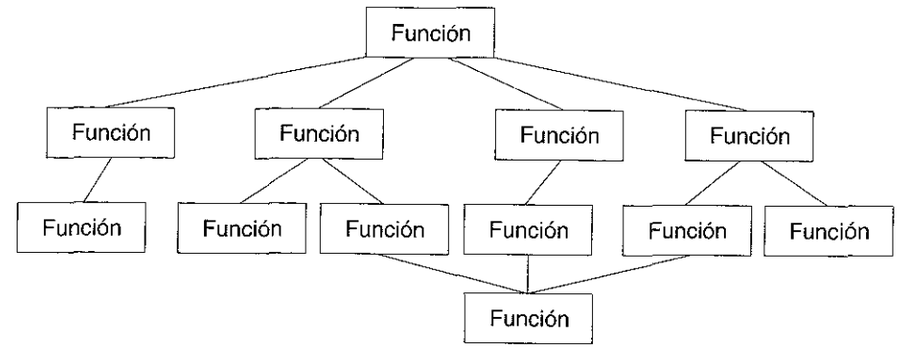


Figura 3.1. Programa estructurado.

A medida que la complejidad de un programa crece, también crece su independencia de los tipos de datos fundamentales que procesa. En un programa estructurado, las estructuras de datos de un programa son tan importantes como las operaciones realizadas sobre ellas. Esto se hace más evidente a medida que crece un programa en tamaño. Los tipos de datos se procesan en muchas funciones dentro de un programa estructurado, y cuando se producen cambios en esos tipos de datos, las modificaciones se deben hacer en cada posición que actúa sobre esos tipos de datos dentro del programa. Esta tarea puede ser frustrante y consumir un tiempo considerable en programas con millones de líneas de código y centenares de funciones.

En un programa estructurado, los datos locales se ocultan dentro de funciones y los datos compartidos se pasan como argumentos (Fig. 3.2)

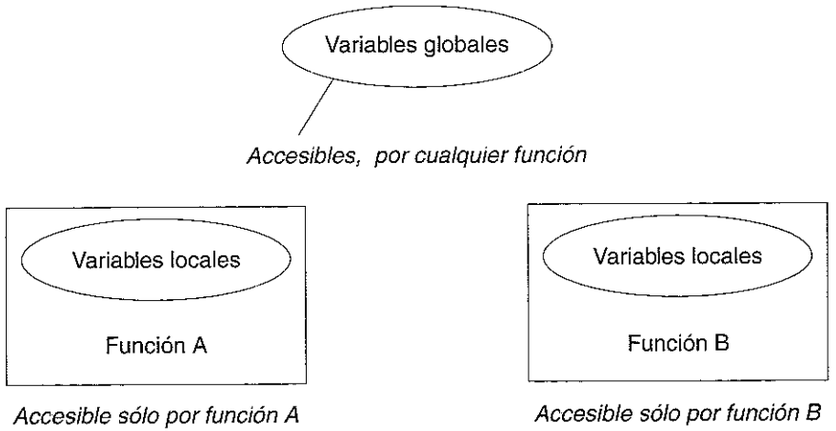


Figura 3.2. Variables globales y locales.

Otro problema es que, dado que muchas funciones acceden a los mismos datos, el medio en que se almacenan los datos se hace más crítico. La disposición de los datos no se pueden cambiar sin modificar todas las funciones que acceden a ellos. Si por ejemplo se añaden nuevos datos, se necesitará modificar todas las funciones que acceden a los datos, de modo que ellos puedan también acceder a esos elementos.

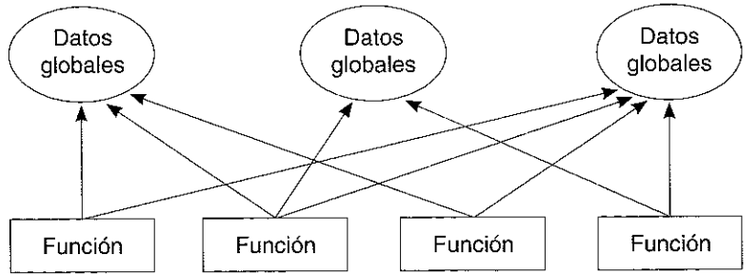


Figura 3.3. Descomposición de un programa en módulos (funciones).

Los programas basados en funciones son difíciles de diseñar. El problema es que sus componentes principales —funciones y estructuras de datos— no modelan bien el mundo real. Por ejemplo, supongamos que se está escribiendo un programa para crear los elementos de un interfaz gráfico de usuario: menús, ventanas, cuadros de diálogo, etc. ¿Qué funciones se necesitarán? ¿Qué estructuras de datos? La solución sería más aproximada si hubiera una correspondencia lo más estrecha posible entre los menús y ventanas y sus correspondientes elementos de programa.

3.1.1. Desventajas de la programación estructurada

Además de los inconvenientes citados anteriormente, comentaremos algunos otros que influyen considerablemente en el diseño.

Cuando diferentes programadores trabajan en equipo para diseñar una aplicación, a cada programador se le asigna la construcción de un conjunto específico de funciones y tipos de datos. Dado que los diferentes programadores manipulan funciones independientes que relacionan a tipos de datos compartidos mutuamente, los cambios que un programador hace a los datos se deben reflejar en el trabajo del resto del equipo. Aunque los programas estructurados son más fáciles de diseñar en grupo, los errores de comunicación entre miembros de equipos pueden conducir a gastar tiempo en rescritura.

Por otra parte, los lenguajes tradicionales presentan una dificultad añadida: *la creación de nuevos tipos de datos*. Los lenguajes de programación típicamente tienen tipos de datos incorporados: enteros, coma flotante, caracteres, etc. ¿Cómo inventar sus propios tipos de datos? Los tipos definidos por el usuario y la facilidad para crear tipos abstractos de datos en determinados lenguajes es la propiedad conocida como *extensibilidad*. Los lenguajes tradicionales no son normalmente extensibles, y eso hace que *los programas tradicionales sean* más complejos de escribir y mantener.

En resumen, con los métodos tradicionales, un programa se divide en dos componentes: *procedimientos y datos*. Cada procedimiento actúa como una caja negra. Esto es, un componente que realiza una tarea específica, tal como convertir un conjunto de números o visualizar una ventana. Este procedimiento permite empaquetar código programa en funciones, pero ¿qué sucede con los datos? Las estructuras de datos utilizadas en programas son con frecuencia globales o se pasan explícitamente con parámetros.

3.2. ¿QUE ES LA PROGRAMACION ORIENTADA A OBJETOS?

Grady Booch, autor del método de diseño orientado a objetos, define la *programación orientada a objetos* (**POO**) como

> «un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representan una instancia de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unidas mediante relaciones de herencia» [1].

Existen tres importantes partes en la definición: la programación orientada a objetos 1) utiliza *objetos*, no algorítmicos, como bloques de construcción lógicos (*jerarquía de objetos*); 2) cada objeto es una instancia de una *clase*, y 3) las clases se relacionan unas con otras por medio de relaciones de herencia.

[1] BOOCH, Grady: *Análisis y diseño orientado a objetos con aplicaciones*. 2ª edición. Addison-Wesley/Díaz de Santos, 1995.

Un programa puede parecer orientado a objetos, pero si cualquiera de estos elementos no existe, no es un programa orientado a objetos Específicamente, la programación sin herencia es distinta de la programación orientada a objetos; se denomina *programación con tipos abstractos de datos o programación basada en objetos*.

El concepto de objeto, al igual que los tipos abstractos de datos o tipos definidos por el usuario, es una colección de elementos de datos, junto con las funciones asociadas utilizadas para operar sobre esos datos Sin embargo, la potencia real de los objetos reside en el modo en que los objetos pueden definir otros objetos. Este proceso, ya comentado en el Capítulo 1: se denomina *herencia* y es el mecanismo que ayuda a construir programas que se modifican fácilmente y se adaptan a aplicaciones diferentes.

Los conceptos fundamentales de programación son: *objetos, clases, herencia, mensajes* y *polimorfismo*.

> Los programas orientados a objetos constan de objetos. Los objetos de un programa se comunican con cada uno de los restantes pasando mensajes.

3.2.1. El objeto

La idea fundamental en los lenguajes orientados a objetos es combinar en una sola unidad *datos y funciones que operan sobre esos datos*. Tal unidad se denomina *objeto*. Por consiguiente, dentro de los objetos residen los datos de los lenguajes de programación tradicionales, tales como números, *arrays*, cadenas y registros, así como funciones o subrutinas que operan sobre ellos.

Las funciones dentro del objeto (*funciones miembro* en C++, *métodos* en Object-Pascal y Smalltalk) son el único medio de acceder a los datos privados de un objeto. Si se desea leer un elemento datos de un objeto se llama a la función miembro del objeto Se lee el elemento y se devuelve el valor No se puede acceder a los datos directamente. Los datos están ocultos, y eso asegura que no se pueden modificar accidentalmente por funciones externas al objeto.

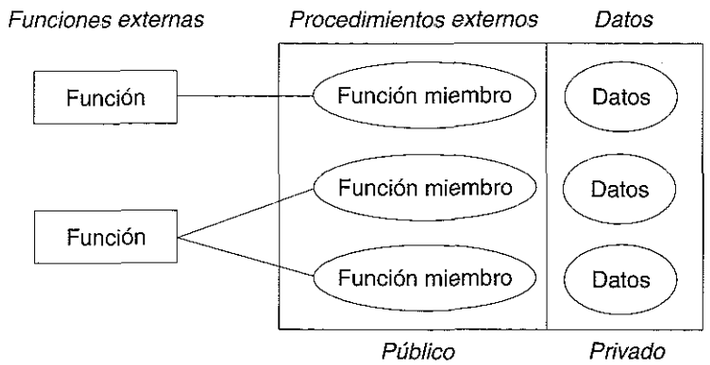


Figura 3.4. El modelo objeto.

Los datos y las funciones (procedimientos en Object-Pascal) asociados se dicen que están *encapsulados* en una única entidad o módulo. La *encapsulación de* datos y ocultación de datos son términos importantes en la descripción de lenguajes orientados a objetos.

Si se desea modificar los datos de un objeto, se conoce exactamente cuáles son las funciones que interactúan con el mismo Ninguna otra función puede acceder a los datos. Esta característica simplifica la escritura, depuración y mantenimiento del programa.

3.2.2. Ejemplos de objetos

¿Qué clase de cosas pueden ser objetos en un programa orientado a objetos? La respuesta está sólo limitada a su imaginación. Algunos ejemplos típicos pueden ser:

- *Objetos físicos*
  Aviones en un sistema de control de tráfico aéreo.
  Automóviles en un sistema de control de tráfico terrestre.
  Casas
- *Elementos de interfaces gráficos de usuario*
  Ventanas
  Menús
  Objetos gráficos (cuadrados, triángulos, etc.).
  Teclado.
  Cuadros de diálogo
  Ratón
- *Animales*
  Animales vertebrados.
  Animales invertebrados.
  Pescados
- *Tipos de datos definidos por el usuario*
  Datos complejos.
  Puntos de un sistema de coordenadas
- *Alimentos*
  Carnes
  Frutas.
  Pescados.
  Verduras.
  Pasteles.

Un objeto es una entidad que contiene los atributos que describen el estado de un objeto del mundo real y las acciones que se asocian con el objeto del mundo real. Se designa por un nombre o identificador del objeto.

Dentro del contexto de un lenguaje orientado a objetos (LOO), un objeto encapsula datos y los procedimientos/funciones (*métodos*) que manejan esos datos. La notación gráfica de un objeto varía de unas metodologías a otras.

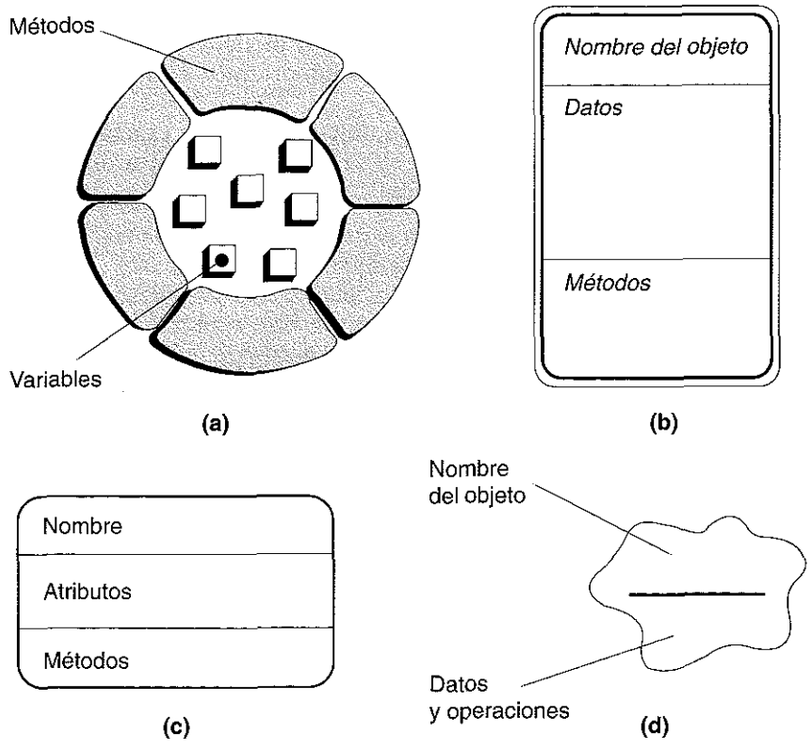


Figura 3.5. Notaciones gráficas de objetos: **(a)** Taylor; **(b)** Yourdon/Coad; **(c)** OMT; **(d)** Booch.

Consideremos una ilustración de un coche vendido por un distribuidor de coches. El identificador del objeto es `Coche1`. Los atributos asociados pueden ser: `número de matrícula`, `fabricante`, `precio_compra`, `precio_actual`, `fecha_compra`. El objeto `Coche1` se muestra en la Figura 3.6.

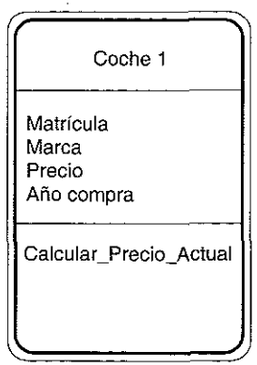


Figura 3.6. El objeto `Coche1`.

Atributos: Datos o variables que caracterizan el estado de un objeto.
Métodos: Procedimientos o acciones que cambian el estado de un objeto.

El objeto retiene cierta información y conoce cómo realizar ciertas operaciones. La encapsulación de operaciones e información es muy importante. Los métodos de un objeto sólo pueden manipular directamente datos asociados con ese objeto. Dicha encapsulación es la propiedad que permite incluir en una sola entidad (el módulo u objeto) la *información* (los datos o atributos) y las *operaciones* (los métodos o funciones) que operan sobre esa información.

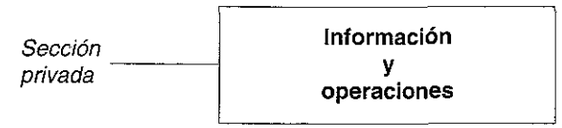


Figura 3.7. Encapsulamiento de datos.

Los objetos tienen un interfaz público y una representación privada que permiten ocultar la información que se desee al exterior.

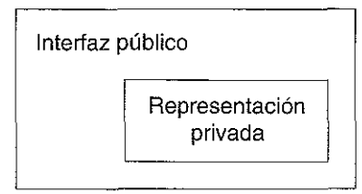


Figura 3.8. Interfaz público de un objeto.

3.2.3. Métodos y mensajes

Un programa orientado a objetos consiste en un número de objetos que se comunican unos con otros llamando a funciones miembro. Las funciones miembro (en C++) se denominan *métodos* en otros lenguajes orientados a objetos (tales como Smalltalk y Turbo Pascal 5.5/6.0/7.0).

Los procedimientos y funciones, denominados *métodos o funciones miembro*, residen en el objeto y determinan cómo actúan los objetos cuando reciben un mensaje. Un *mensaje* es la acción que hace un objeto. Un método es el procedimiento o función que se invoca para actuar sobre un objeto. Un método especifica *cómo* se ejecuta un mensaje.

El conjunto de mensajes a los cuales puede responder un objeto se denomina *protocolo* del objeto. Por ejemplo, el protocolo de un icono puede constar de

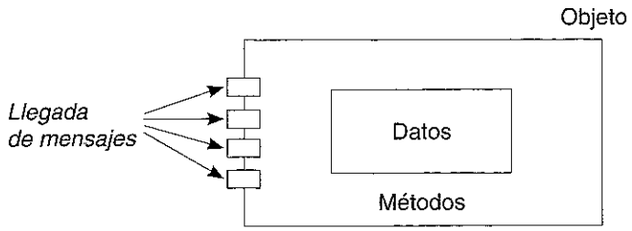


Figura 3.9. Métodos y mensajes de un objeto.

mensajes invocados por el clic de un botón del ratón cuando el usuario localiza un puntero sobre un icono.

Al igual que en las cajas negras, la estructura interna de un objeto está oculta a los usuarios y programadores. Los mensajes que recibe el objeto son los únicos conductos que conectan el objeto con el mundo externo. Los datos de un objeto están disponibles para ser manipulados sólo por los métodos del propio objeto.

Cuando se ejecuta un programa orientado a objetos ocurren tres sucesos. Primero, los objetos se crean a medida que se necesitan. Segundo, los mensajes se mueven de un objeto a otro (o desde el usuario a un objeto) a medida que el programa procesa información internamente o responde a la entrada del usuario. Tercero, cuando los objetos ya no son necesarios, se borran y se libera la memoria.

La Figura 3.10 representa un diagrama orientado a objetos.

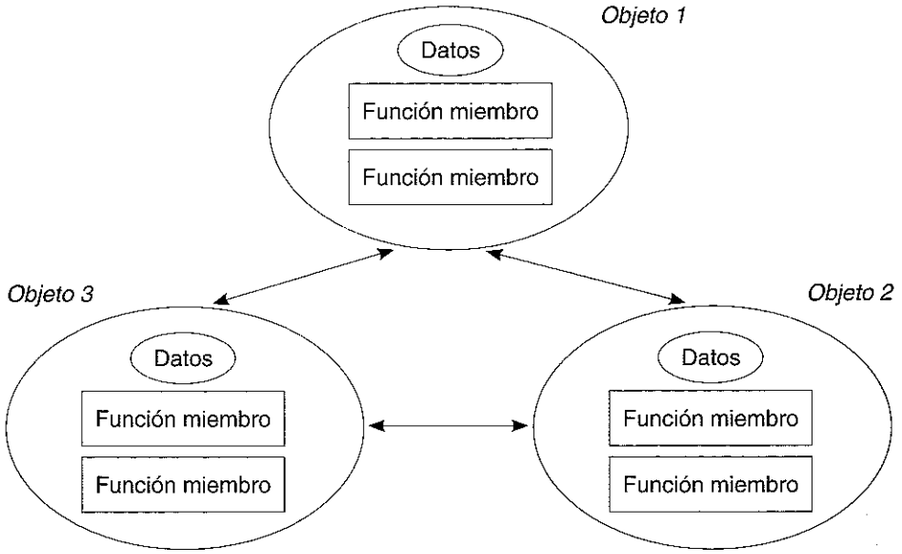


Figura 3.10. Diagrama orientado a objetos.

3.3. CLASES

Una *clase* es la descripción de un conjunto de objetos; consta de métodos y datos que resumen características comunes de un conjunto de objetos. Se pueden definir muchos objetos de la misma clase. Dicho de otro modo, una clase es la declaración de un tipo objeto.

Las clases son similares a los tipos de datos y equivalen a modelos o plantillas que describen cómo se construyen ciertos tipos de objetos. Cada vez que se construye un objeto a partir de una clase, estamos creando lo que se llama una *instancia* de esa clase. Por consiguiente, los objetos no son más que instancias de una clase. *Una instancia es una variable de tipo objeto.* En general, instancia de una clase y objeto son términos intercambiables.

> Un objeto es una instancia de una clase.

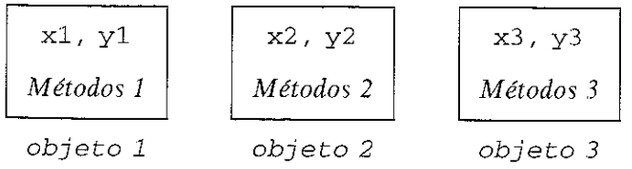
Cada vez que se construye un objeto de una clase, se crea una instancia de esa clase. Los objetos se crean cuando un mensaje de petición de creación se recibe por la clase base.

3.3.1. Implementación de clases en lenguajes

Supongamos una clase `Punto` que consta de los campos dato (coordenadas x e y) y los campos función (métodos leer dichas coordenadas x e y).

clase `Punto`

Objetos de la clase `Punto`



- *Turbo Pascal 5.5/6.0/7.0 llama a la clase objeto* y al objeto una *instancia* de un objeto.
- Un objeto es una variable de una clase dada, y se denomina *instancia* de esa clase.
- Las funciones del objeto se denominan *métodos* (Turbo Pascal) y *funciones miembro* (en C++).

En realidad, una *clase* es un tipo de dato definido por el usuario que determina las estructuras de datos y operaciones asociadas con ese tipo. Dicho de otro modo, *una clase es una colección de objetos similares*. La definición de una clase no crea ningún objeto, de igual modo que la declaración de variables tampoco crea variables.

```
int x;
int pesetas;
```

Tenga cuidado no confundir clases con objetos de esas clases: un automóvil rojo y un automóvil azul no son objetos de clases diferentes, sino objetos de la misma clase con un atributo diferente.

3.3.2. Sintaxis

En C++ se puede declarar una clase de la siguiente forma:

```
class Punto
{
   int x;
   int y;
public:
   void fijarXY (int a, int b)
   {
      x = a;
      y = b;
   }
   int leerX ()     (return x;)
   int leerY ()     (return y;)
};
```

La sintaxis anterior ha definido la clase `Punto`, pero no ha creado ningún objeto. Para crear un objeto de tipo `Punto` tendrá que utilizarse una declaración del tipo correspondiente, al igual que se declara cualquier variable de un tipo incorporado C++

```
Punto P;    //  se define una variable de tipo Punto
```


En Turbo Pascal se define una clase (en terminología de Borland se denomina objeto) con la palabra **object**

```
type
   <nombre-clase> =  object
      <lista de campos de datos>
      <lista de cadenas de funciones y procedimientos>
                     end;
```

y el ejemplo de un objeto `Punto`

```
type Punto =  object
      x,y : Integer;
      procedure operar;
            end;

var p : Punto;
```

- Una clase es una colección de objetos similares.
- Madonna, Michael Jackson, Prince, Mecano y Dire Straits son objetos de una clase, «cantantes de rock»; sin embargo, personas específicas con nombres específicos son miembros de esa clase si poseen ciertas características.

3.4. UN MUNDO DE OBJETOS

Una de las ventajas ineludibles de la orientación a objetos es la posibilidad de reflejar sucesos del mundo real mediante tipos abstractos de datos extensibles a objetos. Así pues, supongamos el fenómeno corriente de la conducción de una bicicleta, un automóvil, una motocicleta o un avión: usted conoce que esos vehículos comparten muchas características, mientras que difieren en otros. Por ejemplo, cualquier vehículo puede ser conducido: aunque los mecanismos de conducción difieren de unos a otros, se puede generalizar el fenómeno de la conducción. En esta consideración, enfrentados con un nuevo tipo de vehículo (por ejemplo una nave espacial), se puede suponer que existe algún medio para conducirla. Se puede decir que *vehículo* es un tipo base y *nave espacial* es un tipo derivado de ella.

En consecuencia, se puede crear un tipo base que representa el comportamiento y características comunes a los tipos derivados de este tipo base.

Un objeto es en realidad una clase especial de variable de un nuevo tipo que algún programador ha creado. Los tipos objeto definidos por el usuario se comportan como tipos incorporados que tienen datos internos y operaciones externas. Por ejemplo, un número en coma flotante tiene un exponente, mantisa y bit de signo y conoce cómo sumarse a sí mismo con otro número de coma flotante.

Los tipos objeto definidos por el usuario contienen datos definidos por el usuario (*características*) y operaciones (*comportamiento*). Las operaciones

definidas por el usuario se denominan *métodos*. Para llamar a uno de estos métodos se hace una petición al objeto: esta acción se conoce como «enviar un mensaje al objeto» Por ejemplo, para detener un objeto automóvil se envía un mensaje de *parada* («stop»). Obsérvese que esta operación se basa en la noción de encapsulación (encapsulamiento): se indica al objeto lo que ha de hacer, pero los detalles de cómo funciona se han encapsulado (ocultado).

3.4.1. Definición de objetos[2]

Un *objeto* (desde el punto de vista formal se debería hablar de **clase**), como ya se ha comentado, es una abstracción de cosas (entidades) del mundo real, tales que:

- Todas las cosas del mundo real dentro de un conjunto —denominadas *instancias*— tienen las mismas características.
- Todas las instancias siguen las mismas reglas

Cada objeto consta de:

- Estado (*atributos*)
- Operaciones o comportamiento (métodos invocados por mensajes).

Desde el punto de vista informático, los objetos son *tipos abstractos de datos* (tipos que encapsulan datos y funciones que operan sobre esos datos).

Algunos ejemplos típicos de objetos:

- *Número racional*
  Estado (valor actual)
  Operaciones (sumar, multiplicar, asignar...)
- *Vehículo*
  Estado (velocidad, posición, precio...).
  Operaciones (acelerar, frenar, parar...).
- *Conjunto*
  Estado (elementos).
  Operaciones (añadir, quitar, visualizar...).
- *Avión*
  Estado (fabricante, modelo, matrícula, número de pasajeros...).
  Operaciones (aterrizar, despegar, navegar...)

3.4.2. Identificación de objetos

El primer problema que se nos plantea al analizar un problema que se desea implementar mediante un programa orientado a objetos es *identificar los objetos*; es decir, ¿qué cosas son objetos?; ¿cómo deducimos los objetos dentro del dominio de la definición del problema?

La identificación de objetos se obtiene examinando la descripción del problema (análisis gramatical somero del enunciado o descripción) y localizando los nombres o cláusulas nominales Normalmente, estos nombres y sus sinónimos se suelen escribir en una tabla de la que luego deduciremos los objetos reales.

Los objetos, según Shlaer, Mellor y Coad/Yourdon, pueden caer dentro de las siguientes categorías:

- *Cosas tangibles* (avión, reactor nuclear, fuente de alimentación, televisor, libro, automóvil).
- *Roles o papeles* jugados o representados por personas (gerente, cliente, empleado, médico, paciente, ingeniero).
- *Organizaciones* (empresa, división, equipo...)
- *Incidentes* (representa un suceso —evento— u ocurrencia, tales como vuelo, accidente, suceso, llamada a un servicio de asistencia técnica...).
- *Interacciones* (implican generalmente una transacción o contrato y relacionan dos o más objetos del modelo: compras —comprador, vendedor, artículo—, matrimonio —esposo, esposa, fecha de boda).
- *Especificaciones* (muestran aplicaciones de inventario o fabricación: refrigerador, nevera...).
- *Lugares* (sala de embarque, muelle de carga...).

Una vez identificados los objetos, será preciso identificar los atributos y las operaciones que actúan sobre ellos.

Los *atributos* describen la abstracción de características individuales que poseen todos los objetos.

AVION
Matrícula
Licencia del piloto
Nombre de avión
Capacidad de carga
Número de pasajeros

EMPLEADO
Nombre
Número de identificación
Salario
Dirección
Nombre del departamento

Las *operaciones* cambian el objeto —su comportamiento— de alguna forma, es decir, cambian valores de uno o más atributos contenidos en el objeto. Aunque existen gran número de operaciones que se pueden realizar sobre un objeto, generalmente se dividen en tres grandes grupos[3]:

- Operaciones que *manipulan* los datos de alguna forma específica (añadir, borrar, cambiar formato...).
- Operaciones que realizan un *cálculo o proceso*

[2] Cuando se habla de modo genérico, en realidad se debería hablar de CLASES, dado que la clase en el tipo de dato y objeto es sólo una instancia, ejemplar o caso de la clase Aquí mantenemos el término objeto por conservar la rigurosidad de la definición «orientado a objetos», aunque en realidad la definición desde el punto de vista técnico sería la clase

[3] PRESSMAN, Roger: *Ingeniería del software Un enfoque práctico* 3.ª edición McGraw-Hill, 1993

- Operaciones que comprueban (*monitorizan*) un objeto frente a la ocurrencia de algún suceso de control.

La identificación de las operaciones se realiza haciendo un nuevo análisis gramatical de la descripción del problema y buscando y aislando los verbos del texto.

3.4.3. Duración de los objetos

Los objetos son entidades que existen en el tiempo; por ello deben ser creados o instanciados (normalmente a través de otros objetos). Esta operación se hace a través de operaciones especiales llamadas *constructores* en C++ o *inicializadores*. Estas operaciones se ejecutarán implícitamente por el compilador o explícitamente por el *programador*, mediante invocación a los citados constructores.

3.4.4. Objetos frente a clases. Representación gráfica (Notación de Ege)

Los objetos y las clases se comparan a *variables* y *tipos* en lenguajes de programación convencional. Una variable es una instancia de un tipo, al igual que un objeto es una instancia de una clase; sin embargo, una clase es más expresiva que un tipo. Expresa la estructura y todos los procedimientos y funciones que se pueden aplicar a una de sus instancias.

En un lenguaje estructurado, un tipo `integer`, por ejemplo, define la estructura de una variable entera, por ejemplo una secuencia de 16 bits y los procedimientos y funciones que se pueden realizar sobre enteros. De acuerdo a nuestra definición de «clase», el tipo `integer` será una clase. Sin embargo, en estos lenguajes de programación no es posible agrupar nuevos tipos y sus correspondientes nuevas funciones y procedimientos en una única unidad. En un lenguaje orientado a objetos una clase proporciona este servicio.

Además de los términos objetos y clases, existen otros términos en orientación a objetos. Las variables o campos que se declaran dentro de una clase se denominan *datos miembro* en C++; otros lenguajes se refieren a ellos como *variables instancia*. Las funciones que se declaran dentro de una clase se denominan *funciones miembro* en C++; otros lenguajes utilizan el término *método*. Las funciones y campos miembro se conocen como *características miembro*, o simplemente *miembros*. A veces se invierten las palabras, y las funciones miembros se conocen como *miembro función* y los campos se denominan *miembro datos*.

Es útil ilustrar objetos y clases con diagramas[4]. La Figura 3.11 muestra el esquema general de un diagrama objeto. Un objeto se dibuja como una caja.

[4] Las notaciones de clases y objetos utilizada en esta sección se deben a Raimund K. Ege, que las dio a conocer en su libro *Programming in an Object-Oriented Environment*. Academic Press (AP), 1992.

La caja se etiqueta con el nombre del objeto y representa el límite o frontera entre el interior y el exterior de un objeto.

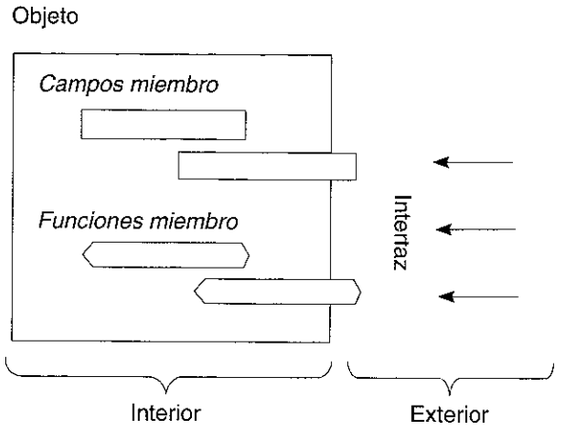


Figura 3.11. Diagrama de un objeto.

Un campo se dibuja por una caja rectangular, una función por un hexágono largo. Los campos y funciones se etiquetan con sus nombres. Si una caja rectangular contiene algo, entonces se representa el valor del campo para el objeto dibujado. Los campos y funciones miembro en el interior de la caja están ocultos al exterior, que significa estar *encapsulados*. El acceso a las características de los miembros (campos y funciones) es posible a través del interfaz del objeto. En una clase en C++, el interfaz se construye a partir de todas las características que se listan después de la palabra reserva **`public`**; puede ser funciones y campos.

La Figura 3.12 muestra el diagrama objeto del objeto `"hola mundo"`. Se llama `Saludo1` y permite acceder a su estado interno a través de las funciones miembro públicas `cambiar` y `anunciar`. El campo miembro privado contiene el valor `Esto es saludo1`.

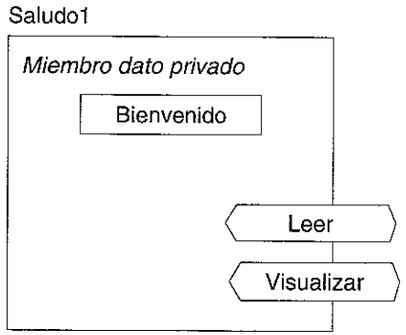


Figura 3.12. El objeto `Saludo1`.

¿Cuál es la diferencia entre clase y objeto?

Un objeto es un simple elemento, no importa lo complejo que pueda ser. Una clase, por el contrario, describe una familia de elementos similares. En la práctica, una clase es como un esquema o plantilla que se utiliza para definir o crear objetos

A partir de una clase se puede definir un número determinado de objetos. Cada uno de estos objetos generalmente tendrá un estado particular propio (una pluma estilográfica puede estar llena, otra puede estar medio llena y otra totalmente vacía) y otras características (como su color), aunque compartan algunas operaciones comunes (como «escribir» o «llenar su depósito de tinta»).

Los objetos tienen las siguientes características:

- Se agrupan en tipos llamados *clases*.
- Tienen *datos internos* que definen su estado actual.
- Soportan *ocultación de datos*
- Pueden *heredar* propiedades de otros objetos.
- Pueden comunicarse con otros objetos pasando *mensajes*
- Tienen *métodos* que definen su *comportamiento*.

La Figura 3.13 muestra el diseño general de diagramas que representan a una clase y a objetos pertenecientes a ella.

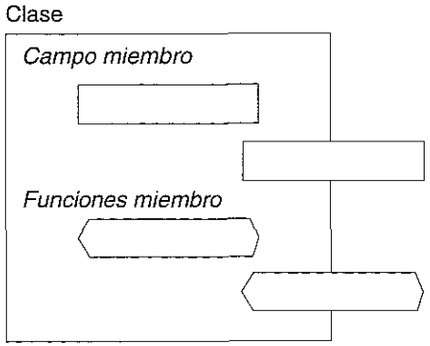


Figura 3.13. Diagrama de una clase.

Una clase es un tipo definido que determina las estructuras de datos y operaciones asociadas con ese tipo. Las clases son como plantillas que describen cómo ciertos tipos de objetos están construidos. Cada vez que se construye un objeto de una clase, estamos creando lo que se llama una *instancia* (modelo o ejemplar) de una clase y la operación correspondiente se llama *instanciación* (creación de instancias). Por consiguiente, los objetos no son más que instancias de clases. En general, los términos *objeto* e *instancia de una clase* se pueden utilizar indistintamente.

- Un objeto es una instancia de una clase.
- Una clase puede tener muchas instancias y cada una es un objeto independiente.
- Una clase es una plantilla que se utiliza para describir uno o más objetos de un mismo tipo.

3.4.5. Datos internos

Una propiedad importante de los objetos es que almacenan información de su *estado* en forma de datos internos. El estado de un objeto es simplemente el conjunto de valores de todas las variables contenidas dentro del objeto en un instante dado. A veces se denominan a las variables que representan a los objetos *variables de estado*. Así por ejemplo, si tuviésemos una clase `ventana` en C++:

```
class ventana {
   int posx, posy;
   int tipo_ventana;
   int tipo_borde;
   int color_ventana;
public:
   move_hor (int dir, int ang);
   move_ver (int dir, int ang);
};
```

Las variables de estado pueden ser las coordenadas actuales de la ventana y sus atributos de color actuales.

En muchos casos, las variables de estado se utilizan sólo indirectamente. Así, en el caso del ejemplo de la ventana, suponga que una orden (mandato) típica a la ventana es:

```
reducir en 5 filas y 6 columnas
```

Esto significa que la ventana se reducirá en tamaño una cantidad dada por 5 filas y 6 columnas:

```
mensaje reducir
```

Afortunadamente, no necesita tener que guardar la posición actual de la ventana, ya que el objeto hace esa operación por usted. La posición actual se almacena en una variable de estado que mantiene internamente la ventana. Naturalmente, se puede acceder a esta variable estado cuando se desee, enviando un mensaje tal como:

```
indicar posicion actual
```

3.4.6. Ocultación de datos

Con el fin de mantener las características de caja negra de POO, se debe considerar cómo se accede a un objeto en el diseño del mismo. Normalmente es una buena práctica restringir el acceso a las variables estado de un objeto y a otra información interna que se utiliza para definir el objeto. Cuando se utiliza un objeto no necesitamos conocer todos los detalles de la implementación. Esta práctica de limitación del acceso a cierta información interna se llama *ocultación de datos*.

En el ejemplo anterior de ventana, el usuario no necesita saber cómo se implementa la ventana; sólo cómo se utiliza. Los detalles internos de la implementación pueden y deben ser ocultados. Considerando este enfoque, somos libres de cambiar el diseño de la ventana (bien para mejorar su eficiencia o bien para obtener su trabajo en un hardware diferente), sin tener que cambiar el código que la utiliza.

> C++ soporta las características de ocultación de datos con las palabras reservadas **`public`**, **`private`** y **`protected`**.

3.5. HERENCIA

La encapsulación es una característica muy potente, y junto con la ocultación de la información, representan el concepto avanzado de objeto, que adquiere su mayor relevancia cuando encapsula e integra datos, más las operaciones que manipulan los datos en dicha entidad. Sin embargo, la orientación a objetos se caracteriza, además de por las propiedades anteriores, por incorporar la característica de *herencia*, propiedad que permite a los objetos ser construidos a partir de otros objetos. Dicho de otro modo, la capacidad de un objeto para utilizar las estructuras de datos y los métodos previstos en antepasados o ascendientes. El objetivo final es la **reutilizabilidad** o **reutilización** (*reusability*)[5], es decir reutilizar código anteriormente ya desarrollado.

La herencia se apoya en el significado de ese concepto en la vida diaria. Así, las clases básicas o fundamentales se dividen en subclases. Los animales se dividen en mamíferos, anfibios, insectos, pájaros, peces, etc. La clase vehículo se divide en subclase automóvil, motocicleta, camión, autobús, etc. El principio en que se basa la división de clases es la jerarquía compartiendo características comunes. Así, todos los vehículos citados tienen un motor y ruedas, que son características comunes; si bien los camiones tienen una caja para transportar mercancías, mientras que las motocicletas tienen un manillar en lugar de un volante.

[5] Este término también se suele traducir por *reusabilidad*, aunque no es un término aceptado por el Diccionario de la Real Academia Española.

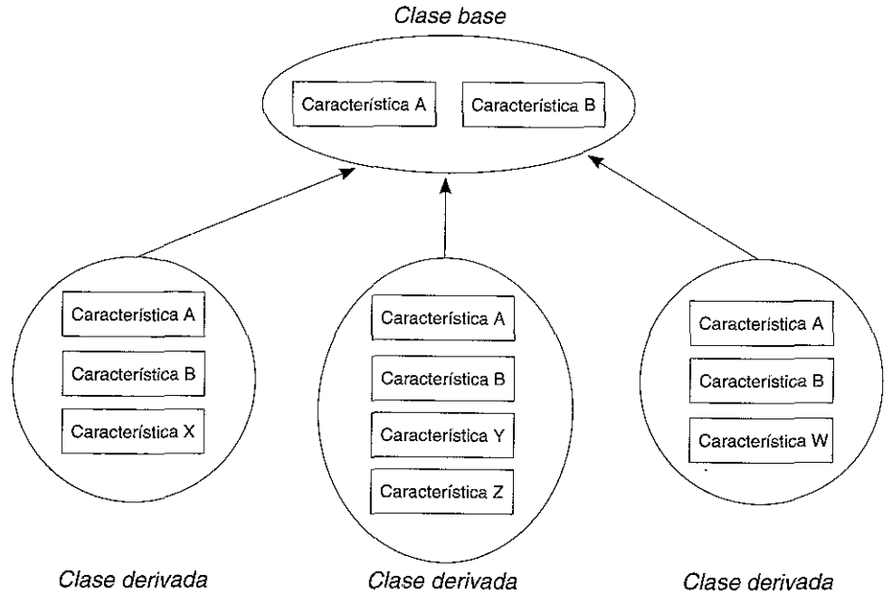


Figura 3.14. Jerarquía de clases.

La herencia supone una *clase base* y una *jerarquía de clases* que contienen las *clases derivadas* de la clase base. Las clases derivadas pueden heredar el código y los datos de su clase base, añadiendo su propio código especial y datos a ellas, incluso cambiar aquellos elementos de la clase base que necesita sean diferentes.

No se debe confundir las relaciones de los objetos con las clases, con las relaciones de una clase base con sus clases derivadas. Los objetos existentes en la memoria de la computadora expresan las características exactas de su clase y sirven como un módulo o plantilla. Las clases derivadas heredan características de su clase base, pero añaden otras características propias nuevas.

> Una clase *hereda* sus características (datos y funciones) de otra clase.

Así, se puede decir que una clase de objetos es un conjunto de objetos que comparten características y comportamientos comunes. Estas características y comportamientos se definen en una clase base. Las clases derivadas se crean en un proceso de definición de nuevos tipos y reutilización del código anteriormente desarrollado en la definición de sus clases base. Este proceso se denomina *programación por herencia*. Las clases que heredan propiedades de una clase base pueden a su vez servir como definiciones base de otras clases. Las jerarquías de clases se organizan en forma de árbol.

3.5.1. Sintaxis

En lenguaje C++ la propiedad de herencia se implementa con la siguiente sintaxis:

```
class <derivadas> : <lista clases base>
{
   <datos propios>
   <funciones miembro propias>
}
```

En lenguaje Pascal orientado a objetos (versiones Turbo 5.5 a 7.0), la sintaxis de una clase (objeto en su terminología) es:

```
type
   <nombre-clase> = object (clase ascendiente)
   <campos propios de nuevo objeto>
   <métodos propios del nuevo objeto>
end;
```

Así, por ejemplo, en C++, si se crea una clase base:

```
class base {
   int x, y;
   public:
   void hacerAlgo ();
}
```

en Turbo Pascal se escribiría la clase equivalente:

```
base = object
   x, y : integer;
   procedure hacerAlgo;
end;
```

Se desea construir un nuevo tipo derivado del tipo base existente, tal que el tipo derivado sea idéntico al tipo base, con una excepción: extender base añadiendo un método llamado `hacerOtraCosa`. Se construye una clase derivada. En C++:

```
class derivada : class base {
public:
   void hacerOtraCosa ();
};
```

En Turbo Pascal,

```
derivada = object (base)
   procedure hacerOtraCosa;
end;
```


Dado que `derivada` hereda todos los datos y métodos de `base`, no se necesita redefinirlos; simplemente *indicar* al compilador que desea derivar un nuevo tipo (*derivado*) de un tipo base (base) y añadir el nuevo método. La herencia permite construir tipos de datos complejos sin repetir mucho código. El nuevo tipo hereda unas características y comportamiento de un ascendiente o antepasado. Puede también reimplementar o sobrescribir cualquier método que elija. Esta reimplementación de métodos del tipo base en los tipos derivados es fundamental el concepto de polimorfismo, que se verá más tarde.

3.5.2. Tipos de herencia

Existen dos mecanismos de herencia utilizados comúnmente en programación orientada a objetos: *herencia simple* y *herencia múltiple*.

En *herencia simple*, un objeto (*clase*) puede tener sólo un ascendiente, o dicho de otro modo, una subbase puede heredar datos y métodos de una única clase, así como añadir o quitar comportamientos de la clase base. Turbo Pascal sólo admite este tipo de herencia. C++ admite herencia simple y múltiple.

La *herencia múltiple* es la propiedad de una clase de poder tener más de un ascendiente inmediato, o lo que es igual, adquirir datos y métodos de más de una clase.

Una representación gráfica de los tipos de herencia con una clase base genérica 01 se muestra en la Figura 3.16.

La Figura 3.15 representa los gráficos de herencia simple y herencia múltiple de la clase `figura` y `persona`, respectivamente.

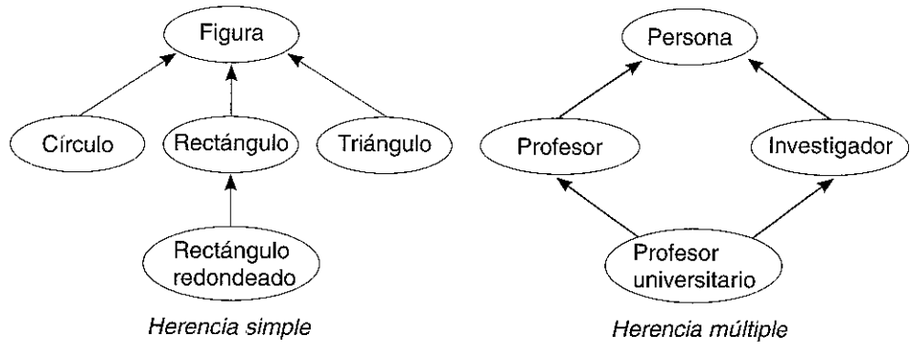


Figura 3.15. Tipos de herencia.

En la Figura 3.16 se muestran gráficamente las relaciones de herencia, apreciándose fácilmente los dos tipos (simple y múltiple).

A primera vista, se puede suponer que la herencia múltiple es mejor que la herencia simple; sin embargo, como ahora comentaremos, no siempre será así.

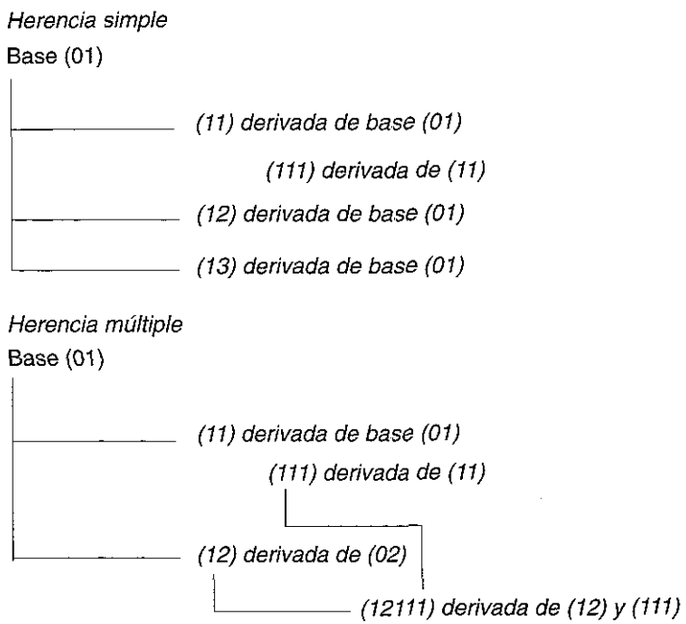


Figura 3.16. Herencia simple y múltiple.

En los lenguajes de POO —incluyendo Object-Pascal, Objective-C, Smalltalk y Acter— implementan sólo herencia simple. Eiffel y C++ (a partir de la versión 2.0), por otra parte, soportan herencia múltiple.

En general, prácticamente todo lo que se puede hacer con herencia múltiple se puede hacer con herencia simple, aunque a veces resulta más difícil. Una dificultad surge con la herencia múltiple cuando se combinan diferentes tipos de objetos, cada uno de los cuales define métodos o campos iguales. Supongamos dos tipos de objetos pertenecientes a las clases `Gráficos` y `Sonidos`, y se crea un nuevo objeto denominado `Multimedia` a partir de ellos. `Gráficos` tiene tres campos datos: `tamaño`, `color` y `mapasdebits`, y los métodos `dibujar`, `cargar`, `almacenar` y `escala`; `sonidos` tiene dos campos datos, `duración`, `voz` y `tono`, y los métodos `reproducir`, `cargar`, `escala` y `almacenar`. Así, para un objeto `Multimedia`, el método escala significa poner el sonido en diferentes tonalidades, o bien aumentar/reducir el tamaño de la escala del gráfico.

Naturalmente, el problema que se produce es la ambigüedad, y se tendrá que resolver con una operación de prioridad que el correspondiente lenguaje deberá soportar y entender en cada caso.

En realidad, ni la herencia simple ni la herencia múltiple son perfectas en todos los casos, y ambas pueden requerir un poco más de código extra que represente bien las diferencias en el modo de trabajo.

3.6. COMUNICACIONES ENTRE OBJETOS: LOS MENSAJES

Ya se ha mencionado en secciones anteriores que los objetos realizan acciones cuando ellos reciben mensajes. El mensaje es esencialmente una orden que se envía a un objeto para indicarle que realice alguna acción. Esta técnica de enviar mensajes a objetos se denomina *pasar mensajes*. Los objetos se comunican entre sí enviando mensajes, al igual que sucede con las personas. Los mensajes tienen una contrapartida denominada métodos. Mensajes y métodos son dos caras de la misma moneda. Los *métodos* son los procedimientos que se invocan cuando un objeto recibe un *mensaje*. En terminología de programación tradicional, un mensaje es una *llamada a una función*. Los mensajes juegan un papel crítico en POO. Sin ellos los objetos que se definen no se podrán comunicar entre sí. Como ejemplo, consideramos enviar un mensaje tal como *subir 5 líneas* el objeto ventana definido anteriormente. El aspecto importante no es cómo se implementa un mensaje, sino cómo se utiliza.

Consideremos de nuevo nuestro objeto `ventana`. Supongamos que deseamos cambiar su tamaño, de modo que le enviamos el mensaje

```
Reducir 3 columnas por la derecha
```

Observe que no le indicamos a la ventana cómo cambiar su tamaño, la ventana maneja la operación por sí misma. De hecho, se puede enviar el mismo mensaje a diferentes clases de ventanas y esperar a que cada una realice la misma acción.

Los mensajes pueden venir de otros objetos o desde fuentes externas, tales como un ratón o un teclado.

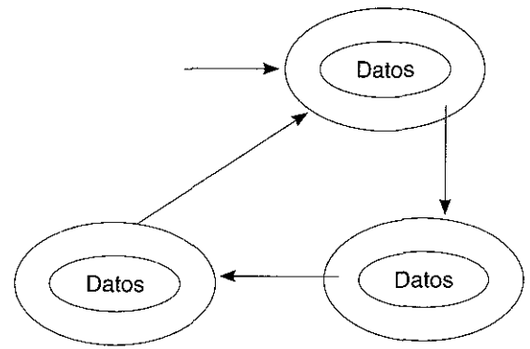


Figura 3.17. Mensajes entre objetos.

Una aplicación Windows es un buen ejemplo de cómo se emplean los mensajes para comunicarse entre objetos. El usuario pulsa un botón para enviar (remitir, despachar) mensajes a otros objetos que realizan una función específica. Si se pulsa el botón `Exit`, se envía un mensaje al objeto responsable de cerrar la aplicación. Si el mensaje es válido, se invoca el método interno. Entonces se cierra la aplicación.

3.6.1. Activación de objetos

A los objetos sólo se puede acceder a través de su interfaz público. ¿Cómo se permite el acceso a un objeto? Un objeto accede a otro objeto enviándole un mensaje.

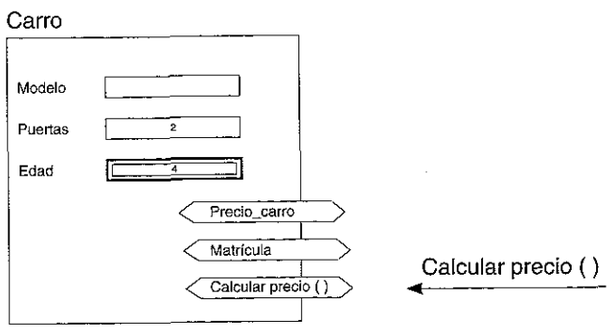


Figura 3.18. Envío de un mensaje.

3.6.2. Mensajes

Un *mensaje* es una petición de un objeto a otro objeto al que le solicita ejecutar uno de sus métodos. Por convenio, el objeto que envía la petición se denomina *emisor* y el objeto que recibe la petición se denomina *receptor*.

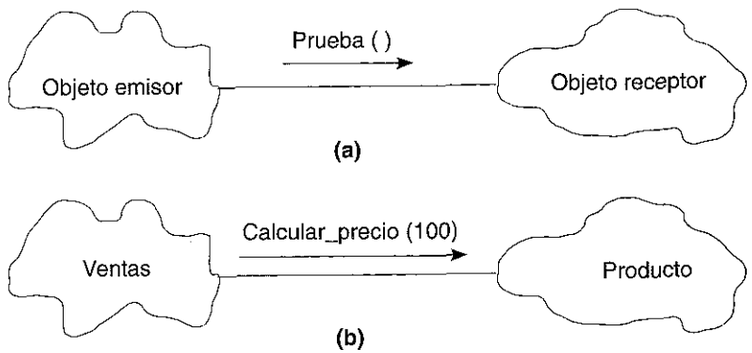


Figura 3.19. Objetos emisor y receptor de un mensaje.

Estructuralmente, un mensaje consta de tres partes:

- *Identidad* del receptor.
- El *método* que se ha de ejecutar.

- *Información especial* necesaria para realizar el método invocado (argumentos o parámetros requeridos).

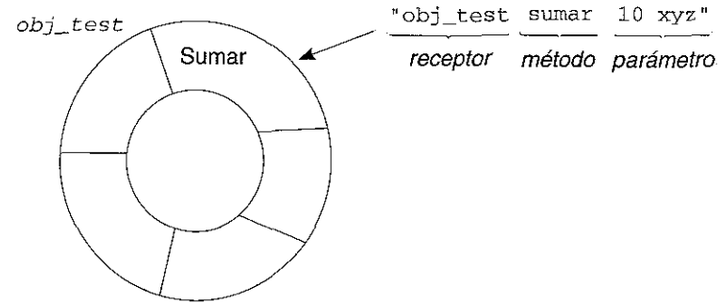


Figura 3.20. Estructura de un mensaje.

Cuando un objeto está inactivo (durmiendo) y recibe un mensaje se hace activo. El mensaje enviado por otros objetos o fuentes tiene asociado un método que se activará cuando el receptor recibe dicho mensaje. La petición no especifica *cómo* se realiza la operación. Tal información se oculta siempre al emisor.

El conjunto de mensajes a los que responde un objeto se denomina *comportamiento* del objeto. No todos los mensajes de un objeto responden; es preciso que pertenezcan al interfaz accesible.

Nombre de un mensaje

Un mensaje incluye el nombre de una operación y cualquier argumento requerido por esa operación. Con frecuencia, es útil referirse a una operación por nombre, sin considerar sus argumentos.

Métodos

Cuando un objeto recibe un mensaje, se realiza la operación solicitada ejecutando un método. Un *método* es el algoritmo ejecutado en respuesta a la recepción de un mensaje cuyo nombre se corresponde con el nombre del método.

La secuencia actual de acontecimientos es que el emisor envía su mensaje; el receptor ejecuta el método apropiado, consumiendo los parámetros; a continuación, el receptor devuelve algún tipo de respuesta al emisor para reconocer el mensaje y devolver cualquier información que se haya solicitado.

> El receptor responde a un mensaje.

3.6.3. Paso de mensajes

Los objetos se comunican entre sí a través del uso de mensajes El interfaz del mensaje se define un interfaz claro entre el objeto y el resto de su entorno.

Esencialmente, el protocolo de un mensaje implica dos partes: el emisor y el receptor. Cuando un objeto emisor envía un mensaje a un objeto receptor, tiene que especificar lo siguiente:

1. Un receptor.
2. Un nombre de mensaje
3. Argumentos o parámetros (si se necesita).

En primer lugar, un objeto receptor que ha de recibir el mensaje que se ha especificado. Los objetos no especificados por el emisor no responderán El receptor trata de concordar el nombre del mensaje con los mensajes que él entiende. Si el mensaje no se entiende, el objeto receptor no se activará. Si el mensaje se entiende por el objeto receptor, el receptor aceptará y responderá al mensaje invocando el método asociado.

Los parámetros o argumentos pueden ser:

1. Datos utilizados por el método invocado.
2. Un mensaje, propiamente dicho

La estructura de un mensaje puede ser:

```
enviar <Objeto A>.<Método1  (parámetro1,     parámetroN)>
```

El ejemplo siguiente muestra algunos mensajes que se pueden enviar al objeto `Coche1`. El primero de éstos invoca al método `Precio_Coche` y no tiene argumentos, mientras que el segundo, `Fijar_precio`, envía los parámetros 8-10-92, y `Poner_en_blanco` no tiene argumentos.

Ejemplo

```
enviar Coche1.Precio_Coche()          envía a Coche1 el mensaje Precio_Coche
enviar Coche1.Fijar_precio(8-10-92)   envía a Coche1 el mensaje Fijar_precio
                                      con el parámetro 8- 10-92
enviar Coche1.Poner_en_blanco()       envía a Coche1 el mensaje Poner_en_blanco
```

3.7. ESTRUCTURA INTERNA DE UN OBJETO

La estructura interna de un objeto consta de dos componentes básicos:

- Atributos.
- Métodos (operaciones o servicios).

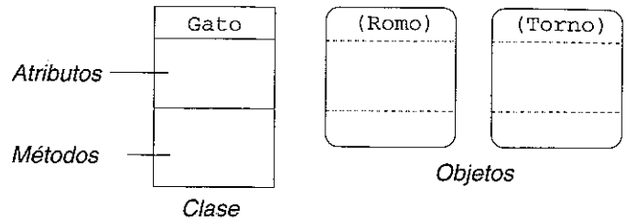


Figura 3.21. Notación gráfica OMT de una clase y de un objeto.

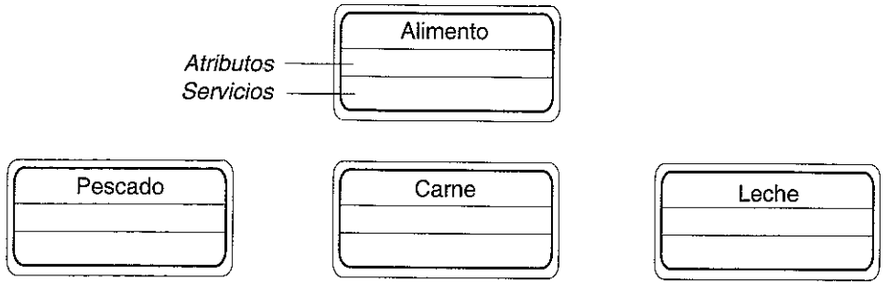


Figura 3.22. Objetos en notación Yourdon/Coad.

3.7.1. Atributos

Los **atributos** describen el estado del objeto. Un atributo consta de dos partes: un nombre de atributo y un valor de atributo

Los objetos simples pueden constar de tipos primitivos, tales como enteros, carácter, boolean, reales, o tipos simples definidos por el usuario Los objetos complejos pueden constar de pilas, conjuntos, listas, array, etc, o incluso estructuras recursivas de alguno o todos los elementos.

Los constructores se utilizan para construir estos objetos complejos a partir de otros objetos complejos.

3.7.2. Métodos

Los **métodos** (operaciones o servicios) describen el *comportamiento* asociado a un objeto. Representan las acciones que pueden realizarse por un objeto o sobre un objeto. La ejecución de un método puede conducir a cambiar el estado del objeto o dato local del objeto.

Cada método tiene un nombre y un cuerpo que realiza la acción o comportamiento asociado con el nombre del método En un LOO, el cuerpo de un método consta de un bloque de código procedimental que ejecuta la acción requerida. Todos los métodos que alteran o acceden a los datos de un objeto se definen dentro del objeto Un objeto puede modificar directamente o acceder a los datos de otros objetos

Un método dentro de un objeto se activa por un mensaje que se envía por otro objeto al objeto que contiene el método. De modo alternativo, se puede llamar por otro método en el mismo objeto por un mensaje local enviado de un método a otro dentro del objeto.

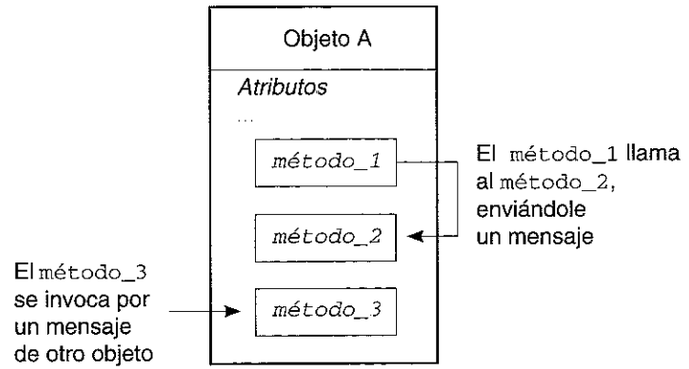


Figura 3.23. Invocación de un método.

```
metodo: Precio_coche
inicio
   Precio_coche := Precio_coste * (Marca+1);
fin.
```

3.8. CLASES

La clase es la construcción del lenguaje utilizada más frecuentemente para definir los tipos abstractos de datos en lenguajes de programación orientados a objetos. Una de las primeras veces que se utilizó el concepto de clase fue en Simula (Dahl y Nygaard, 1966; Dahl, Myhrhang y Nigaard, 1970) como entidad que declara conjuntos de objetos similares. En Simula, las clases se utilizaron principalmente como plantillas para crear objetos de la misma estructura. Los atributos de un objeto pueden ser tipos base, tales como enteros, reales y booleans; o bien pueden ser arrays, procedimientos o instancias de otras clases.

Generalmente, una clase se puede definir como una descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por un *estado* específico y es capaz de realizar una serie de *operaciones*.

Por ejemplo, una pluma estilográfica es un objeto que tiene un estado (llena de tinta o vacía) que puede realizar algunas operaciones (por ejemplo escribir, poner/quitar el capuchón, rellenar si está vacía).

En programación, una clase es una estructura que contiene datos y procedimientos (o funciones) que son capaces de operar sobre esos datos. Una clase pluma estilográfica puede tener, por ejemplo, una variable que indica si está llena o vacía; otra variable puede contener la cantidad de tinta cargada realmente. La clase contendrá algunas funciones que operan o utilizan esas variables.

Dentro de un programa, las clases tienen dos propósitos principales: definir abstracciones y favorecer la modularidad.

¿Cuál es la diferencia entre una clase y un objeto, con independencia de su complejidad? Una clase verdaderamente describe una familia de elementos similares. En realidad, una clase es una plantilla para un tipo particular de objetos. Si se tienen muchos objetos del mismo tipo, sólo se tienen que definir las características generales de ese tipo una vez, en lugar de en cada objeto.

A partir de una clase se puede definir un número de objetos. Cada uno de estos objetos tendrá generalmente un estado peculiar propio (una pluma puede estar rellenada, otra puede estar medio-vacía y otra estar totalmente vacía) y otras características (como su color), aunque compartirán operaciones comunes (como «escribir», «llenar», «poner el capuchón», etc.).

En resumen, un objeto es una instancia de una clase.

3.8.1. Una comparación con tablas de datos

Una *clase* se puede considerar como la extensión de un registro. Aquellas personas familiarizadas con sistemas de bases de datos pueden asociar clase e instancias con tablas y registros, respectivamente. Al igual que una clase, una tabla define los nombres y los tipos de datos de la información que contenga. Del mismo modo que una instancia, un registro de esa tabla proporciona los valores específicos para una entrada particular. La principal diferencia, a nivel conceptual, es que las clases contienen métodos, además de las definiciones de datos.

Una clase es una caja negra o módulo en la que está permitido conocer *lo que hace* la clase, pero no *cómo* lo hace. Una clase será un *módulo* y un *tipo*. Como módulo la clase encapsula los recursos que ofrece a otras clases (sus clientes). Como *tipo* describe un conjunto de *objetos* o *instancias* que existen en tiempo de ejecución.

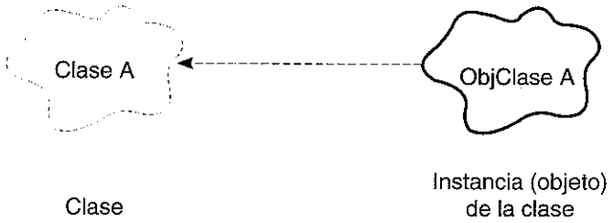


Figura 3.24. Una clase y una instancia (objeto) de la clase (Notación Booch).

Instancias como registros			
Servicio	**Horas**	**Frecuencia**	**Descuento**
S2020	4,5	6	10
S1010	8	2	20
S4040	5	3	15

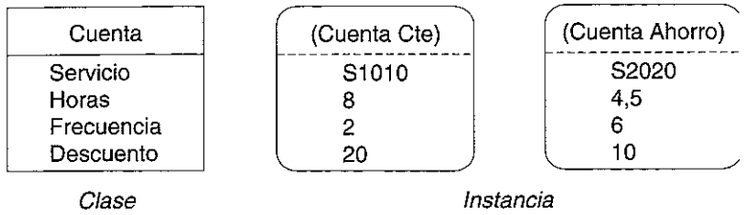


Figura 3.25. Instancias de una clase (notación OMT).

Los objetos ocupan espacio en memoria, y en consecuencia existen en el tiempo, y deberán *crearse o instanciarse* (probablemente a partir de otros objetos). Por la misma razón, se debe liberar el espacio en memoria ocupado por los objetos. Dos operaciones comunes típicas en cualquier clase son:

- *Constructor*: una operación que crea un objeto y/o inicializa su estado.
- *Destructor*: una operación que libera el estado de un objeto y/o destruye el propio objeto.

En C++ los constructores y destructores se declaran como parte de la definición de una clase. La creación se suele hacer a través de operaciones especiales (*constructores* en C++, `Pila`); estas operaciones se aplicarán implícitamente o se deberán llamar explícitamente por otros objetos, como sucede en C++.

Cuando se desea crear una nueva instancia de una clase, se llama a un método de la propia clase para realizar el proceso de construcción. Los métodos constructores se definen como métodos de la clase. De modo similar, los métodos empleados para destruir objetos y liberar la memoria ocupada (*destructores* en C++, `~Pila`) también se definen dentro de la clase.

> Un objeto es una *instancia* (ejemplar, caso u ocurrencia) de una clase.

3.9. HERENCIA Y TIPOS

Los objetos con propiedades comunes (atributos y operaciones) se clasifican en una clase. De igual modo, las clases con propiedades y funciones comunes se agrupan en una **superclase**. Las clases que se derivan de una superclase son las **subclases**.

Las clases se organizan como *jerarquía de clases*. La ventaja de definir clases en una jerarquía es que a través de un mecanismo denominado *herencia*, casos especiales comparten todas las características de sus casos más generales.

La herencia es una característica por la que es posible definir una clase, no de un borrador, sino en términos de otra clase. Una clase *hereda* sus características (datos y funciones) de otra clase. Esta característica proporciona claramente un soporte poderoso para reutilización y extensibilidad, dado que la definición de nuevos objetos se pueden basar en clases existentes.

Como ejemplo, considérese la jerarquía de herencia mostrada en la Figura 3.26.

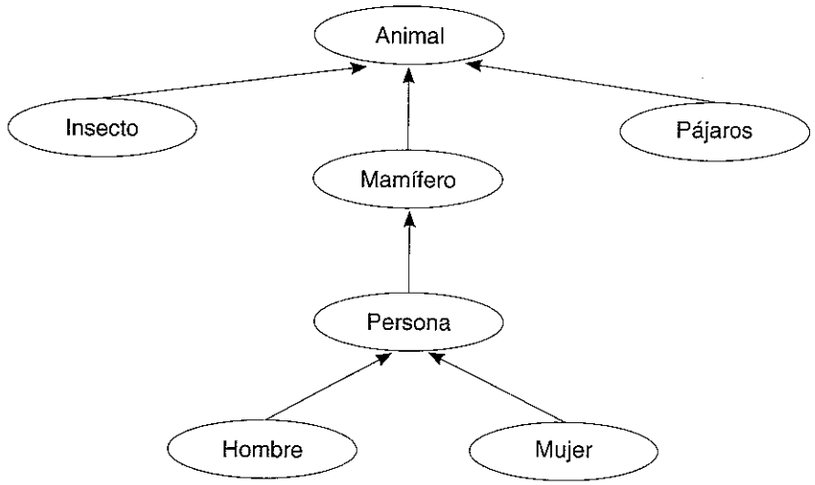


Figura 3.26. Jerarquía de herencia.

Las clases de objeto mamífero, pájaro e insecto se definen como *subclases* de animal; la clase de objeto persona, como una subclase de mamífero, y un hombre y una mujer son subclases de persona.

Las definiciones de clases para esta jerarquía puede tomar la siguiente estructura:

```
clase criatura
   atributos
      tipo : string;
      peso : real;
      habitat : (...algun tipo de habitat... );
   operaciones
      crear() → criatura;
      predadores(criatura) → fijar(criatura);
      esperanza_vida(criatura) → entero;
      ...
end criatura.

clase mamifero inherit criatura;
   atributos (propiedades)
      periodo_gestacion: real;
   operaciones
      ...
end mamifero.

clase persona inherit mamifero;
   propiedades
```

```
        apellidos, nombre: string;
        fecha_nacimiento: date;
        origen: pais;
end persona

clase hombre inherit persona;
   atributos
        esposa: mujer;
        ...
   operaciones
        ...
end hombre

clase mujer inherit persona;
   propiedades
        esposo: man;
        nombre: string;
        ...
end mujer
```

La herencia es un mecanismo potente para tratar con la evolución natural de un sistema y con modificación incremental [Meyer, 1988] Existen dos tipos diferentes de herencia: *simple* y *múltiple*.

3.9.1. Herencia simple (*herencia jerárquica*)

En esta jerarquía cada clase tiene como máximo una sola superclase. La herencia simple permite que una clase herede las propiedades de su superclase en una cadena jerárquica.

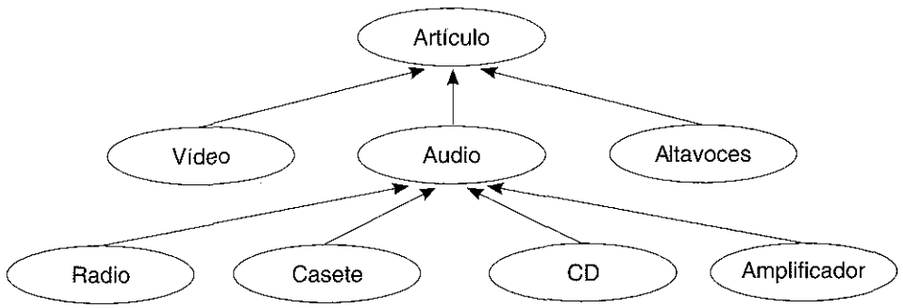


Figura 3.27. Herencia simple.

3.9.2. Herencia múltiple (*herencia en malla*)

Una malla o retícula consta de clases, cada una de las cuales puede tener uno o más superclases inmediatas Una herencia múltiple es aquella en la que cada clase puede heredar métodos y variables de cualquier número de superclase

En la Figura 3 28 la clase C tiene dos superclases, A y D Por consiguiente, la clase C hereda las propiedades de las clases A y D. Evidentemente, esta acción puede producir un conflicto de nombres, donde la clase C hereda las mismas propiedades de A y D.

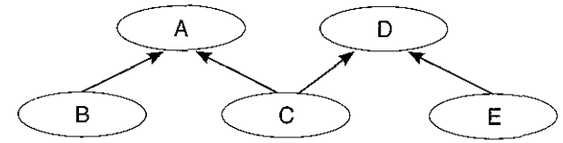


Figura 3.28. Herencia múltiple.

Herencia selectiva

La herencia selectiva es la herencia en que algunas propiedades de las superclases se heredan selectivamente por parte de la clase heredada Por ejemplo, la clase B puede heredar algunas propiedades de la superclase A, mientras que la clase C puede heredar selectivamente algunas propiedades de la superclase A y algunas de la superclase D

Herencia múltiple

Problemas:

1. La propiedad referida solo está en una de las subclases padre.
2. La propiedad concreta existe en más de una superclase.

Caso 1. No hay problemas.

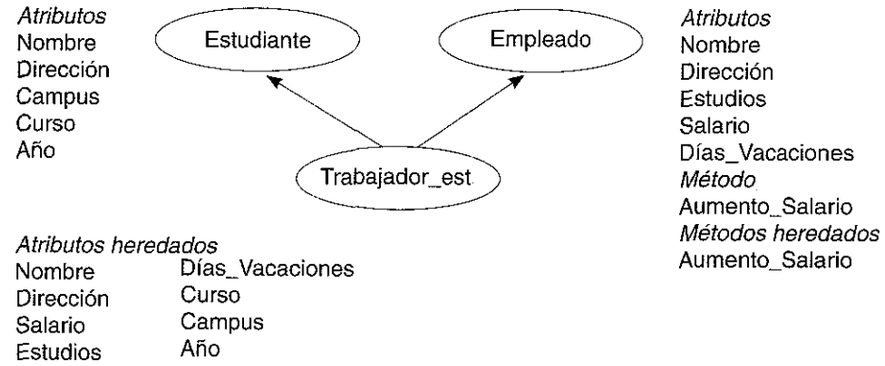


Figura 3.29. Herencia de atributos y métodos.

Caso 2. Existen diferentes tipos de conflictos que pueden ocurrir:

- Conflictos de nombres.

- Conflictos de valores
- Conflictos por defecto
- Conflictos de dominio
- Conflictos de restricciones

Por ejemplo,

Conflicto de nombres	Nombre	Nombre_estudiante
		Nombre_empleado
Valores	Atributos con igual nombre, tienen valores en cada clase	
	Universidad con diversos campus.	

Reglas de resolución de conflictos

1. Una lista de precedencia de clases, como sucede en LOOPS y FLAVORS.
2. Una precedencia especificada por el usuario para herencia, como en Smalltalk.
3. Lista de precedencia del usuario, y si no sucede así, la lista de precedencia de las clases por profundidad.

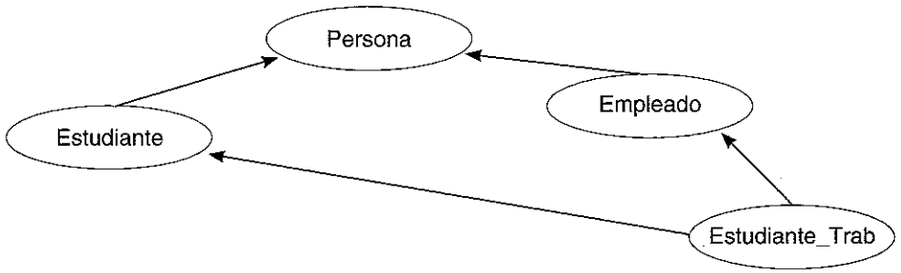


Figura 3.30. Clase derivada por herencia múltiple.

3.9.3. Clases abstractas

Con frecuencia, cuando se diseña un modelo orientado a objetos es útil introducir clases a cierto nivel que pueden no existir en la realidad pero que son construcciones conceptuales útiles. Estas clases se conocen como *clases abstractas*.

Una clase abstracta normalmente ocupa una posición adecuada en la jerarquía de clases que le permite actuar como un depósito de métodos y atributos compartidos para las subclases de nivel inmediatamente inferior.

Las clases abstractas no tienen instancias directamente. Se utilizan para agrupar otras clases y capturar información que es común al grupo. Sin embargo, las subclases de clases abstractas que corresponden a objetos del mundo real pueden tener instancias.

Una clase abstracta es COCHE_TRANSPORTE_PASAJEROS. Una subclase es SEAT, que puede tener instancias directamente, por ejemplo Coche1 y Coche2.

> Una clase abstracta es una clase que sirve como clase base común, pero no tendrá instancias.

Una clase abstracta puede ser una impresora.

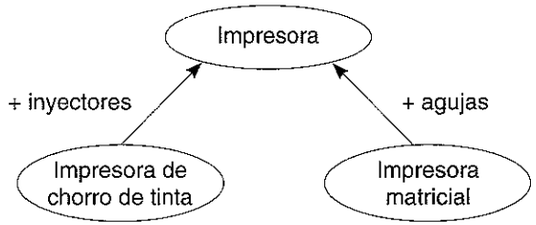


Figura 3.31. La clase abstracta impresora.

Las clases derivadas de una clase base se conocen como *clases concretas*, que ya pueden *instanciarse* (es decir, pueden tener *instancias*).

3.10. ANULACION/SUSTITUCION

Como se ha comentado anteriormente, los atributos y métodos definidos en la superclase se heredan por las subclases. Sin embargo, si la propiedad se define nuevamente en la subclase, aunque se haya definido anteriormente a nivel de superclase; entonces la definición realizada en la subclase es la utilizada en esa subclase. Entonces se dice que anulan las correspondientes propiedades de la superclase. Esta propiedad se denomina **anulación** o **sustitución** (*overriding*).

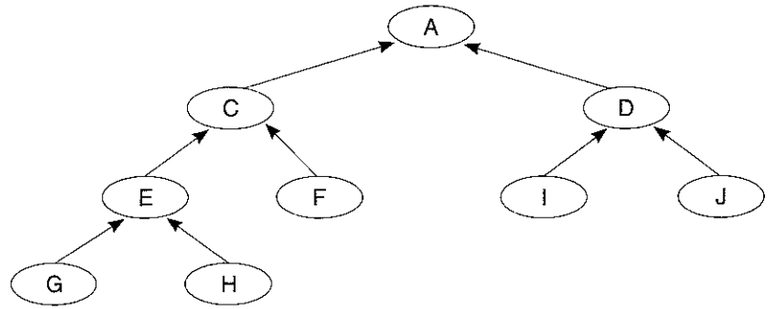


Figura 3.32. Anulación de atributos y métodos en clases derivadas.

Supongamos que ciertos atributos y métodos definidos en la clase A se redefinen en la clase C. Las clases E, F, G y H heredan estos atributos y métodos. La cuestión que se produce es si estas clases heredan las definiciones dadas en la clase A o las dadas en la clase C. El convenio adoptado es que una vez que un atributo o método se redefine en un nivel de clases específico, entonces cualquier hijo de esa clase, o sus hijos en cualquier profundidad, utilizan este método o atributo redefinido. Por consiguiente, las clases E, F, G y H utilizarán la redefinición dada en la clase C, en lugar de la definición dada en la clase A.

3.11. SOBRECARGA

La **sobrecarga** es una propiedad que describe una característica adecuada que utiliza el mismo nombre de operación para representar operaciones similares que se comportan de modo diferente cuando se aplican a clases diferentes. Por consiguiente, los nombres de las operaciones se pueden sobrecargar, esto es, las operaciones se definen en clases diferentes y pueden tener nombres idénticos, aunque su código programa puede diferir.

Si los nombres de una operación se utilizan para nuevas definiciones en clases de una jerarquía, la operación a nivel inferior se dice que anula la operación a un nivel más alto.

Un ejemplo se muestra en la Figura 3.33, en la que la operación `Incrementar` está sobrecargada en la clase `Empleado` y la subclase `Administrativo`. Dado que `Administrativo` es una subclase de `Empleado`, la operación `Incrementar`, definida en el nivel Administrativo, anula la operación correspondiente al nivel Empleado. En `Ingeniero` la operación `Incrementar` se hereda de `Empleado`. Por otra parte, la sobrecarga puede estar situada entre dos clases que no están relacionadas jerárquicamente. Por ejemplo, `Cálculo_Comisión` está sobrecargada en `Administrativo` y en `Ingeniero`. Cuando un mensaje `Calcular_Comisión` se envía al objeto `Ingeniero`, la operación correspondiente asociada con `Ingeniero` se activa.

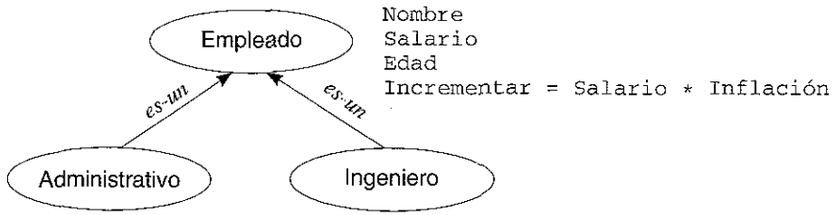


Figura 3.33. Sobrecarga.

Actualmente la sobrecarga se aplica sólo a operaciones. Aunque es posible extender la propiedad a atributos y relaciones específicas del modelo propuesto.

La sobrecarga no es una propiedad específica de los lenguajes orientados a objetos. Lenguajes tales como C y Pascal soportan operaciones sobrecargadas. Algunos ejemplos son los operadores aritméticos, operaciones de E/S y operadores de asignación de valores.

En la mayoría de los lenguajes, los operadores aritméticos «+», «−» y «*» se utilizan para sumar, restar o multiplicar números enteros o reales. Estos operadores funcionan incluso aunque las implementaciones de aritmética entera y real (coma flotante) sean bastante diferentes. El compilador genera código objeto para invocar la implementación apropiada basada en la clase (entero o coma flotante) de los operandos.

Así, por ejemplo, las operaciones de E/S (Entrada/Salida) se utilizan con frecuencia para leer números enteros, caracteres o reales. En Pascal `read(x)` se puede utilizar, siendo `x` un entero, un carácter o un real. Naturalmente, el código máquina real ejecutado para leer una cadena de caracteres es muy diferente del código máquina para leer enteros. `read(x)` es una operación sobrecargada que soporta tipos diferentes. Otros operadores tales como los de asignación («:=» en Pascal o «=» en C) son sobrecargados. Los mismos operadores de asignación se utilizan para variables de diferentes tipos.

Los lenguajes de programación convencionales soportan sobrecarga para algunas de las operaciones sobre algunos tipos de datos, como enteros, reales y caracteres. Los *sistemas orientados a objetos dan un poco más en la sobrecarga y la hacen disponible para operaciones sobre cualquier tipo objeto.*

Por ejemplo, en las operaciones binarias se pueden sobrecargar para números complejos, arrays, conjuntos o listas que se hayan definido como tipos estructurados o clases. Así, el operador binario «+» se puede utilizar para sumar las correspondientes partes reales e imaginarias de los números complejos. Si `A1` y `A2` son dos array de enteros, se pueden definir:

```
A := A1 + A2
```

para sumar:

```
A[i] := A1[i] + A2[i]       //para todo i
```

De modo similar, si `S1` y `S2` son dos conjuntos de objetos, se puede definir:

```
S := S1 + S2
```

como unión de dos conjuntos `S1` y `S2`.

¿Cómo se asocia una operación particular o mensaje a un método? La respuesta es mediante la ligadura dinámica (*dynamic binding*), que se verá posteriormente.

3.12. LIGADURA DINAMICA

Los lenguajes OO tienen la característica de poder ejecutar ligadura tardía (*dinámica*), al contrario que los lenguajes imperativos, que emplean ligadura temprana (*estática*). Por consiguiente, los tipos de variables, expresiones y funciones se conocen en tiempo de compilación para estos lenguajes imperativos. Esto permite el enlazar entre llamadas a procedimientos y los procedimientos utilizados que se establecen cuando se cumple el código. En un sistema OO esto requería el enlace entre mensajes y que los métodos se establezcan en tiempo dinámico.

En el caso de ligadura dinámica o tardía, el tipo se conecta directamente al objeto. Por consiguiente, el enlace entre el mensaje y el método asociado sólo se puede conocer en tiempo de ejecución.

La ligadura estática permite un tiempo de ejecución más rápido que la ligadura dinámica, que necesita resolver estos enlaces en tiempo de ejecución. Sin embargo, en ligadura estática se ha de especificar en tiempo de compilación las operaciones exactas a que responderá una invocación del método o función específica, así como conocer sus tipos.

Por el contrario, en la ligadura dinámica simplemente se especifica un método en un mensaje, y las operaciones reales que realizan este método se determinan en tiempo de ejecución. Esto permite definir funciones o métodos virtuales.

3.12.1. Funciones o métodos virtuales

Las funciones virtuales en C++ permiten especificar un método como virtual en la definición de una clase particular. La implementación real del método se

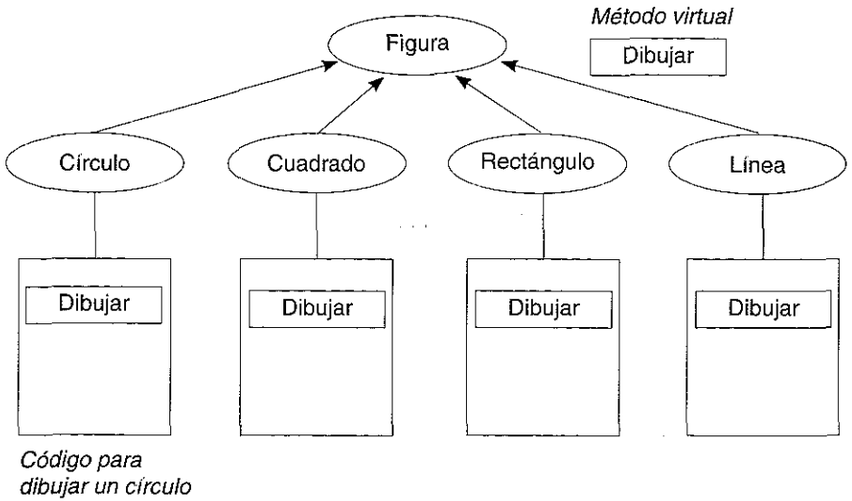


Figura 3.34. Métodos o funciones virtuales.

realiza en las subclases. En este caso, por consiguiente, la selección del método se hace en tiempo de compilación, pero el código real del método utilizado se determina utilizando ligadura dinámica o tardía en tiempo de compilación.

Esto permite definir el método de un número de formas diferentes para cada una de las diferentes clases. Consideremos la jerarquía de clases definida en la Figura 3.34.

Aquí el método virtual se define en la clase `FIGURA` y el código procedimental real utilizado se define en cada una de las subclases `CIRCULO`, `CUADRADO`, `RECTANGULO` y `LINEA`. Ahora, si un mensaje se envía a una clase específica, se ejecuta el código asociado con ella. Esto contrasta con un enfoque más convencional que requiere definir los procedimientos por defecto, con nombres diferentes, tales como `Dibujar_círculo`, `Dibujar_cuadrado`, etc. También se requerirá utilizar una llamada al nombre de la función específica cuando sea necesario.

3.12.2. Polimorfismo

La capacidad de utilizar funciones virtuales y ejecutar sobrecarga conduce a una característica importante de los sistemas OO, conocida como *polimorfismo*, que esencialmente permite desarrollar sistemas en los que objetos diferentes puedan responder de modo diferente al mismo mensaje.

En el caso de un método virtual se puede tener especialización incremental de, o adición incremental a, un método definido anteriormente en la jerarquía.

Más adelante volveremos a tratar este concepto.

3.13. OBJETOS COMPUESTOS

Una de las características que hacen a los objetos ser muy potentes es que pueden contener otros objetos. Los objetos que contienen otros objetos se conocen como *objetos compuestos*.

En la mayoría de los sistemas, los objetos compuestos no «contienen» en el sentido estricto otros objetos, sino que contienen variables que se refieren a otros objetos. La referencia almacenada en la variable se llama identificador del objeto (ID del objeto).

Esta característica ofrece dos ventajas importantes:

1. Los objetos «contenidos» pueden cambiar en tamaño y composición, sin afectar al objeto compuesto que los contiene. Esto hace que el mantenimiento de sistemas complejos de objetos anidados sea más sencillo, que sería el caso contrario.
2. Los objetos contenidos están libres para participar en cualquier número de objetos compuestos, en lugar de estar bloqueado en un único objeto compuesto.

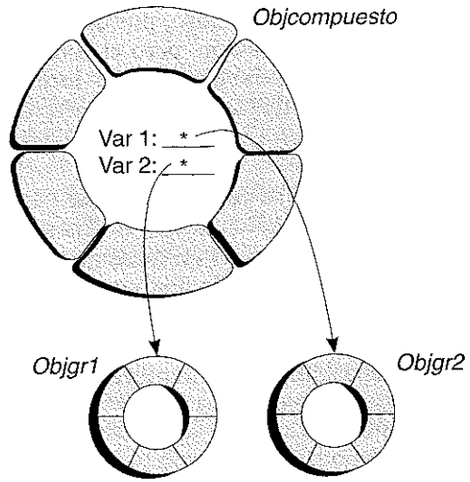


Figura 3.35. Un objeto compuesto.

Un objeto compuesto consta de una colección de dos o más objetos componentes. Los objetos componentes tienen una relación **part-of** (*parte-de*) o **component-of** (*componente-de*) con objeto compuesto. Cuando un objeto compuesto se instancia para producir una instancia del objeto, todos sus objetos componentes se deben instanciar al mismo tiempo. Cada objeto componente puede ser a su vez un objeto compuesto [6].

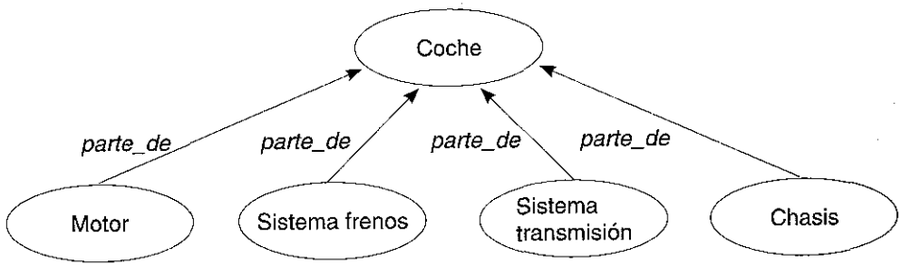


Figura 3.36. Relación de agregación (*parte-de*).

La relación *parte-de* puede representarse también por **has-a** (*tiene-un*), que indica la relación que une al objeto *agregado* o *continente*. En el caso del objeto compuesto `COCHE` se leerá: `COCHE` *tiene-un* `MOTOR`, *tiene-un* `SISTEMA_DE_FRENOS`, etc.

[6] TAYLOR, David: *Object-Oriented Information System*. Wiley, 1992.

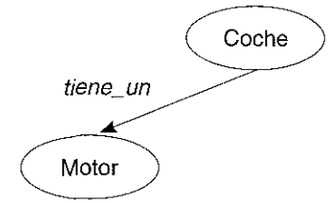


Figura 3.37. Relación de agregación (*tiene-un*).

3.13.1. Un ejemplo de objetos compuestos

La ilustración siguiente muestra dos objetos que representa órdenes de compra. Sus variables contienen información sobre clientes, artículos comprados y otros datos. En lugar de introducir toda la información directamente en los objetos `orden_compra`, se almacenan referencia a estos objetos componentes en el formato del identificador de objeto (IDO).

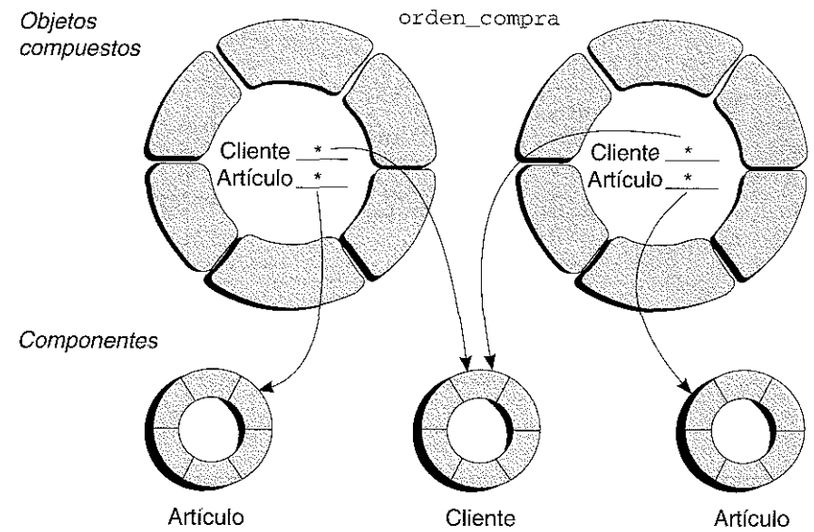


Figura 3.38. Objetos compuestos (dos objetos `orden_compra`[7]).

3.13.2. Niveles de profundidad

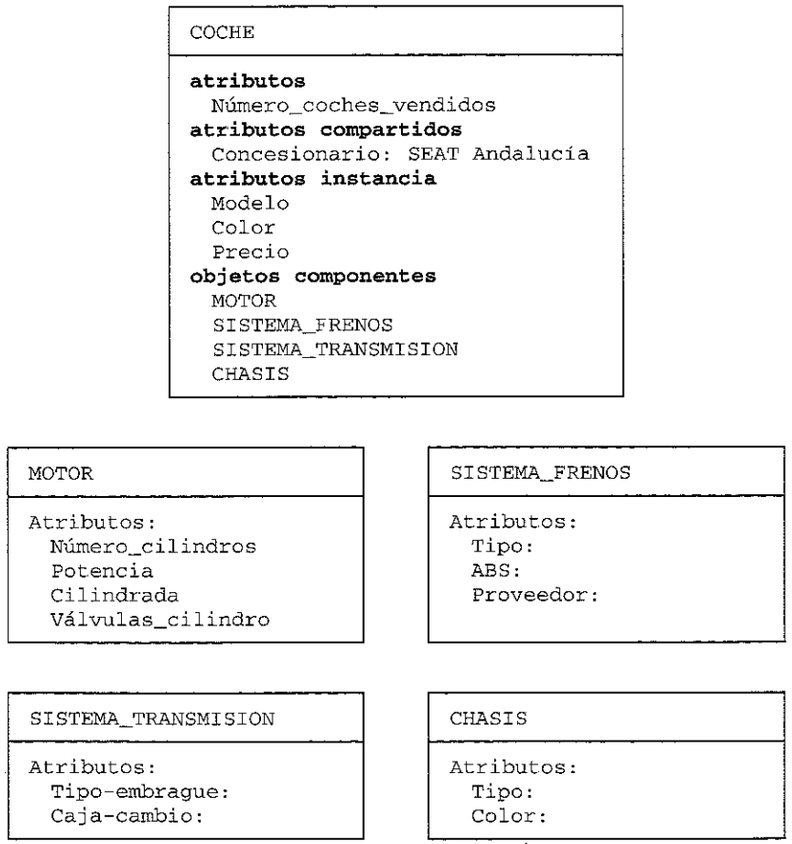
Los objetos contenidos en objetos compuestos pueden, por sí mismos, ser objetos compuestos, y este anidamiento puede ir hasta cualquier profundidad. Esto

[7] Este ejemplo está citado en: TAYLOR, David: *op. cit.* pág. 45. Asimismo, la notación de objetos empleada por Taylor se ha mantenido en varios ejemplos de nuestra obra, ya que la consideramos una de las más idóneas para reflejar el concepto de objeto; esta obra y otras suyas sobre el tema son consideradas como aportaciones muy notables al mundo científico de los objetos.

significa que puede construir estructuras de cualquier complejidad conectando objetos juntos. Esto es importante debido a que normalmente se necesita más de un nivel de modularización para evitar el caos en sistemas a gran escala.

Un objeto compuesto, en general, consta de una colección de dos o más objetos relacionados conocidos como objetos componentes. Los objetos componentes tienen una relación una *parte-de* o un *componente-de* con objeto compuesto. Cuando un objeto compuesto se instancia para producir un objeto instancia, todos sus objetos componentes se deben instanciar al mismo tiempo. Cada objeto componente puede, a su vez, ser un objeto compuesto, resultando, por consiguiente, una jerarquía de *componentes-de*.

Un ejemplo de un objeto compuesto es la clase COCHE. Un coche consta de diversas partes, tales como un motor, un sistema de frenos, un sistema de transmisión y un chasis; se puede considerar como un objeto compuesto que consta de partes diferentes: MOTOR, SISTEMA_FRENOS, SISTEMA_TRANSMISION, CHASIS. Estas partes constituyen los objetos componentes del objeto COCHE, de modo que cada uno de estos objetos componentes pueden tener atributos y métodos que los caracterizan.



La jerarquía *componente-de* (*parte-de*) pueden estar solapadas o anidadas. Una jerarquía de solapamiento consta de objetos que son componentes de más de un objeto padre.

Una jerarquía anidada consta de objetos que son componentes de un objeto padre que, a su vez, puede actuar como componente de otro objeto. El objeto Z es un componente del objeto B, y el objeto B es un componente de un objeto complejo más grande, A.

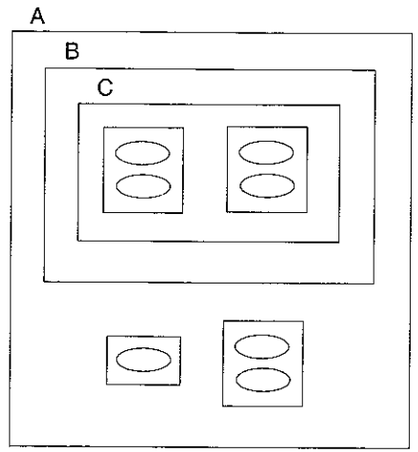


Figura 3.39. Jerarquía de componentes agregados.

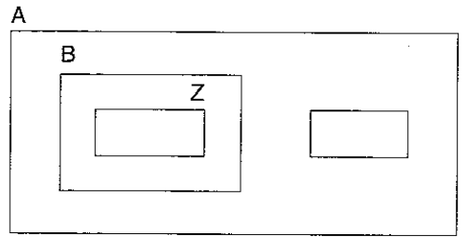


Figura 3.40. Anidamiento de objetos.

Un ejemplo típico de un objeto compuesto anidado es un archivador. Un archivador contiene cajones, un cajón contiene carpetas y una carpeta contiene documentos. El ejemplo COCHE, citado anteriormente, es también un objeto compuesto anidado.

3.14. REUTILIZACION CON ORIENTACION A OBJETOS

Reutilización o reutilizabilidad es la propiedad por la que el software desarrollado puede ser utilizado cuantas veces sea necesario en más de un programa. Así por ejemplo, si se necesita una función que calcule el cuadrado o el cubo de un

número, se puede crear la función que realice la tarea que el programa necesita. Con un esfuerzo suplementario se puede crear una función que pueda elevar cualquier número a cualquier potencia. Esta función se debe guardar para poderla utilizar como herramienta de propósito general en cuantas ocasiones sea necesario.

Las ventajas de la reutilización son evidentes. El ahorro de tiempo es, sin duda, una de las ventajas más considerables, y otra la facilidad para intercambiar software desarrollado por diferentes programadores.

En la programación tradicional, las bibliotecas de funciones (casos de FORTRAN o C) evitan tener que ser escritas cada vez que se necesita su uso.

Ada y Modula-2 incorporan el tipo de dato *paquete* (**package**) y *módulo* (**module**) que consta de definición de tipos y códigos y que son la base de la reutilización de esos lenguajes.

3.14.1. Objetos y reutilización

La programación orientada a objetos proporciona el marco idóneo para la reutilización de las clases. Los conceptos de encapsulamiento y herencia son las bases que facilitan la reutilización. Un programador puede utilizar una clase existente, y sin modificarla, añadirle nuevas características y datos. Esta operación se consigue derivando una clase a partir de la clase base existente. La nueva clase hereda las propiedades de la antigua, pero se pueden añadir nuevas propiedades. Por ejemplo, suponga que se escribe (o compra) una clase menú que crea un sistema de menús (barras de desplazamiento, cuadros de diálogo, botones, etc.); con el tiempo, aunque la clase funciona bien, observa que sería interesante que las leyendas de las opciones de los menús parpadearán o cambiarán de color. Para realizar esta tarea se diseña una clase derivada de menú que añada las nuevas propiedades de parpadeo o cambio de color.

La facilidad para reutilizar clases (y en consecuencia objetos) es una de las propiedades fundamentales que justifican el uso de la programación orientada a objetos. Por esta razón los sistemas y en particular los lenguajes orientados a objetos suelen venir provistos de un conjunto (*biblioteca*) de clases predefinidas, que permite ahorrar tiempo y esfuerzo en el desarrollo de cualquier aplicación. Esta herramienta —*la biblioteca de clases*— es uno de los parámetros fundamentales a tener en cuenta en el momento de evaluar un lenguaje orientado a objetos.

3.15. POLIMORFISMO

Otra propiedad importante de la programación orientada a objetos es el *polimorfismo*. Esta propiedad, en su concepción básica, se encuentra en casi todos los lenguajes de programación. El polimorfismo, en su expresión más simple, es el uso de un nombre o un símbolo —por ejemplo un operador— para representar o significar más de una acción. Así, en C, Pascal y FORTRAN —entre otros lenguajes— los operadores aritméticos representan un ejemplo de esta característica. El símbolo +, cuando se utiliza con enteros, representa un conjunto de instrucciones máquina distinto de cuando los operadores son valores reales de doble precisión. De igual modo, en algunos lenguajes el símbolo + sirve para realizar sumas aritméticas o bien para concatenar (unir) cadenas.

La utilización de operadores o funciones de formas diversas, dependiendo de cómo se estén operando, se denomina *polimorfismo* (múltiples formas). Cuando un operador existente en el lenguaje tal como +, = o * se le asigna la posibilidad de operar sobre un nuevo tipo de dato, se dice que está *sobrecargado*. La *sobrecarga* es una clase de polimorfismo, que también es una característica importante de POO. Un uso típico de los operadores aritméticos es la sobrecarga de los mismos para actuar sobre tipos de datos definidos por el usuario (objetos), además de sobre los tipos de datos predefinidos. Supongamos que se tienen tipos de datos que representan las posiciones de puntos en la pantalla de un computador (coordenadas x e y). En un lenguaje orientado a objetos se puede realizar la operación aritmética

```
posición1 = origen + posición2
```

donde las variables *posición1*, *posición2* y *origen* representan cada una posiciones de puntos, sobrecargando el operador más (+) para realizar suma de posiciones de puntos (x, y). Además de esta operación de suma se podrían realizar otras operaciones, tales como resta, multiplicación, etc., sobrecargando convenientemente los operadores −, *, etc.

En un sentido más general, el polimorfismo supone que un mismo mensaje puede producir acciones (resultados) totalmente diferentes cuando se reciben por objetos diferentes. Con polimorfismo un usuario puede enviar un mensaje genérico y dejar los detalles de la implementación exacta para el objeto que recibe el mensaje. El polimorfismo se fortalece con el mecanismo de herencia.

Supongamos un tipo objeto llamado `vehículo` y tipos de objetos derivados llamados `bicicleta`, `automóvil`, `moto` y `embarcación`. Si se envía un mensaje conducir al objeto `vehículo`, cualquier tipo que herede de `vehículo` puede también aceptar ese mensaje. Al igual que sucede en la vida real, el mensaje `conducir` reaccionará de modo diferente en cada objeto, debido a que cada vehículo requiere una forma distinta de conducir.

RESUMEN

El tipo abstracto de datos se implementa a través de clases. Una clase es un conjunto de objetos que constituyen instancias de la clase, cada una de las cuales tienen la misma estructura y comportamiento. Una clase tiene un nombre, una colección de operaciones para manipular sus instancias y una representación. Las operaciones que manipulan las instancias de una clase se llaman *métodos*. El estado o representación de una instancia se almacena en variables de instancia. Estos métodos se invocan mediante el envío de *mensajes* a instancias. El envío de mensajes a objetos (instancias) es similar a la llamada a procedimientos en lenguajes de programación tradicionales.

El mismo nombre de un método se puede sobrecargar con diferentes implementaciones; el método `Imprimir` se puede aplicar a enteros, arrays y cadenas de caracteres. La sobrecarga de operaciones permite a los programas ser extendidos de un modo elegante. La sobrecarga permite la ligadura de un mensaje a la implementación de código del mensaje y se hace en tiempo de ejecución. Esta característica se llama ligadura dinámica.

El *polimorfismo* permite desarrollar sistemas en los que objetos diferentes pueden responder de modo diferente al mismo mensaje. La ligadura dinámica, sobrecarga y la herencia permite soportar el polimorfismo en lenguajes de programación orientados a objetos.

Los programas orientados a objetos pueden incluir *objetos compuestos*, que son objetos que contienen otros objetos, anidados o integrados en ellos mismos.

Los principales puntos clave tratados son:

- La programación orientada a objetos incorpora estos seis componentes importantes:

  Objetos
  Clases.
  Métodos.
  Mensajes.
  Herencia.
  Polimorfismo

- Un objeto se compone de datos y funciones que operan sobre esos objetos.
- La técnica de situar datos dentro de objetos de modo que no se puede acceder directamente a los datos se llama *ocultación de la información*
- Una clase es una descripción de un conjunto de objetos Una instancia es una variable de tipo objeto y un objeto es una instancia de una clase.
- La herencia es la propiedad que permite a un objeto pasar sus propiedades a otro objeto, o dicho de otro modo, un objeto puede heredar de otro objeto
- Los objetos se comunican entre sí pasando mensajes.
- La clase padre o ascendiente se denomina clase base y las clases descendientes clases derivadas.
- La reutilización de software es una de las propiedades más importantes que presenta la programación orientada a objetos.
- El polimorfismo es la propiedad por la cual un mismo mensaje puede actuar de diferente modo cuando actúa sobre objetos diferentes ligados por la propiedad de la herencia.

CAPITULO **4**

LENGUAJES DE PROGRAMACION ORIENTADOS A OBJETOS

CONTENIDO

Este capítulo describe:

- La evolución de los lenguajes de programación orientados a objetos, desde Simula a Object COBOL.
- Las características de un lenguaje orientado a objetos.
- La clasificación de los lenguajes orientados a objetos basados en objetos, orientados a objetos (*puros* e *híbridos*) basados en objetos, orientados a objetos basados en clases.
- Una descripción breve de los lenguajes más utilizados: Smalltalk, C++, Objective-C, Eiffel, Object Pascal, Visual BASIC y Ada.