

# FUNDAMENTOS DE PROGRAMACIÓN

**Algoritmos, estructura  
de datos y objetos**

**Cuarta edición**

**Luis Joyanes Aguilar**

Catedrático de Lenguajes y Sistemas Informáticos  
*Facultad de Informática, Escuela Universitaria de Informática  
Universidad Pontificia de Salamanca campus de Madrid*



MADRID • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO  
NUEVA YORK • PANAMÁ • SAN JUAN • SANTIAGO • SÃO PAULO  
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS  
SAN FRANCISCO • SIDNEY • SINGAPUR • ST LOUIS • TOKIO • TORONTO

PARTE

# Algoritmos y herramientas de programación

## CONTENIDO

**Capítulo 1.** Introducción a las computadoras y los lenguajes de programación

**Capítulo 2.** Metodología de la programación y desarrollo de software

**Capítulo 3.** Estructura general de un programa

**Capítulo 4.** Flujo de control I: Estructuras selectivas

**Capítulo 5.** Flujo de control II: Estructuras repetitivas

**Capítulo 6.** Subprogramas (subalgoritmos): Funciones

# Metodología de la programación y desarrollo de software

- 2.1. Fases en la resolución de problemas
- 2.2. Programación modular
- 2.3. Programación estructurada
- 2.4. Programación orientada a objetos
- 2.5. Concepto y características de algoritmos

- 2.6. Escritura de algoritmos
  - 2.7. Representación gráfica de los algoritmos
- RESUMEN
- EJERCICIOS

## INTRODUCCIÓN

Este capítulo le introduce a la metodología que hay que seguir para la resolución de problemas con computadoras.

La resolución de un problema con una computadora se hace escribiendo un programa, que exige al menos los siguientes pasos:

- 1. Definición o análisis del problema.
- 2. Diseño del algoritmo.

- 3. Transformación del algoritmo en un programa.
- 4. Ejecución y validación del programa.

Uno de los objetivos fundamentales de este libro es el *aprendizaje y diseño de los algoritmos*. Este capítulo introduce al lector en el concepto de algoritmo y de programa, así como las herramientas que permiten “dialogar” al usuario con la máquina: *los lenguajes de programación*.

## 2.1. FASES EN LA RESOLUCIÓN DE PROBLEMAS

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución en la misma. Aunque el proceso de diseñar programas es, esencialmente, un proceso creativo, se puede considerar una serie de fases o pasos comunes, que generalmente deben seguir todos los programadores.

Las fases de resolución de un problema con computadora son:

- *Análisis del problema.*
- *Diseño del algoritmo.*
- *Codificación.*
- *Compilación y ejecución.*
- *Verificación.*
- *Depuración.*
- *Mantenimiento.*
- *Documentación.*

Las características más sobresalientes de la resolución de problemas son:

- **Análisis.** El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por la persona que encarga el programa.
- **Diseño.** Una vez analizado el problema, se diseña una solución que conducirá a un *algoritmo* que resuelva el problema.
- **Codificación (implementación).** La solución se escribe en la sintaxis del lenguaje de alto nivel (por ejemplo, Pascal) y se obtiene un programa fuente que se compila a continuación.
- **Ejecución, verificación y depuración.** El programa se ejecuta, se comprueba rigurosamente y se eliminan todos los errores (denominados “*bugs*”, en inglés) que puedan aparecer.
- **Mantenimiento.** El programa se actualiza y modifica, cada vez que sea necesario, de modo que se cumplan todas las necesidades de cambio de sus usuarios.
- **Documentación.** Escritura de las diferentes fases del ciclo de vida del software, esencialmente el análisis, diseño y codificación, unidos a manuales de usuario y de referencia, así como normas para el mantenimiento.

Las dos primeras fases conducen a un diseño detallado escrito en forma de algoritmo. Durante la tercera fase (*codificación*) se *implementa*<sup>1</sup> el algoritmo en un código escrito en un lenguaje de programación, reflejando las ideas desarrolladas en las fases de análisis y diseño.

Las fases de *compilación y ejecución* traducen y ejecutan el programa. En las fases de *verificación y depuración* el programador busca errores de las etapas anteriores y los elimina. Comprobará que mientras más tiempo se gaste en la fase de análisis y diseño, menos se gastará en la depuración del programa. Por último, se debe realizar la *documentación del programa*.

Antes de conocer las tareas a realizar en cada fase, se considera el concepto y significado de la palabra **algoritmo**. La palabra *algoritmo* se deriva de la traducción al latín de la palabra *Alkhô-warîzmi*<sup>2</sup>, nombre de un matemático y astrónomo árabe que escribió un tratado sobre manipulación de números y ecuaciones en el siglo IX. Un **algoritmo** es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.

### Características de un algoritmo

- *preciso* (indica el orden de realización en cada paso),
- *definido* (si se sigue dos veces, obtiene el mismo resultado cada vez),
- *finito* (tiene fin; un número determinado de pasos).

<sup>1</sup> En la última edición (21.<sup>a</sup>) del **DRAE** (Diccionario de la Real Academia Española) se ha aceptado el término *implementar*: (Informática) “Poner en funcionamiento, aplicar métodos, medidas, etc. para llevar algo a cabo”.

<sup>2</sup> Escribió un tratado matemático famoso sobre manipulación de números y ecuaciones titulado *Kitab al-jabr w’almugabala*. La palabra álgebra se derivó, por su semejanza sonora, de *al-jabr*.

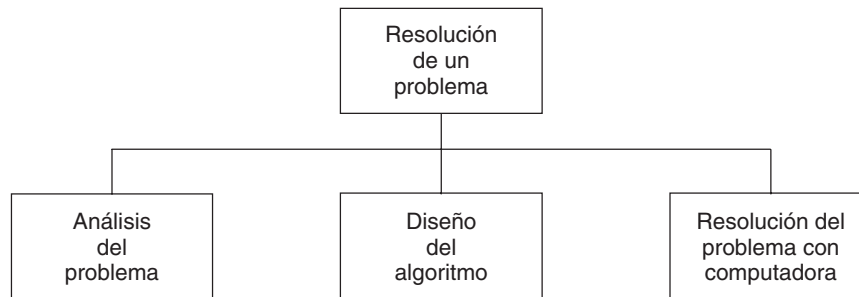
Un algoritmo debe producir un resultado en un tiempo finito. Los métodos que utilizan algoritmos se denominan *métodos algorítmicos*, en oposición a los métodos que implican algún juicio o interpretación que se denominan *métodos heurísticos*. Los métodos algorítmicos se pueden *implementar* en computadoras; sin embargo, los procesos heurísticos no han sido convertidos fácilmente en las computadoras. En los últimos años las técnicas de inteligencia artificial han hecho posible la *implementación* del proceso heurístico en computadoras.

Ejemplos de algoritmos son: instrucciones para montar en una bicicleta, hacer una receta de cocina, obtener el máximo común divisor de dos números, etc. Los algoritmos se pueden expresar por *fórmulas*, *diagramas de flujo* o *N-S* y *pseudocódigos*. Esta última representación es la más utilizada para su uso con lenguajes estructurados como Pascal.

### 2.1.1. Análisis del problema

La primera fase de la resolución de un problema con computadora es el *análisis del problema*. Esta fase requiere una clara definición, donde se contemple exactamente lo que debe hacer el programa y el resultado o solución deseada.

Dado que se busca una solución por computadora, se precisan especificaciones detalladas de entrada y salida. La Figura 2.1 muestra los requisitos que se deben definir en el análisis.



**Figura 2.1.** Análisis del problema.

Para poder identificar y definir bien un problema es conveniente responder a las siguientes preguntas:

- ¿Qué entradas se requieren? (tipo de datos con los cuales se trabaja y cantidad).
- ¿Cuál es la salida deseada? (tipo de datos de los resultados y cantidad).
- ¿Qué método produce la salida deseada?
- Requisitos o requerimientos adicionales y restricciones a la solución.

#### PROBLEMA 2.1

Se desea obtener una tabla con las depreciaciones acumuladas y los valores reales de cada año, de un automóvil comprado por 20.000 euros en el año 2005, durante los seis años siguientes suponiendo un valor de recuperación o rescate de 2.000. Realizar el análisis del problema, conociendo la fórmula de la depreciación anual constante  $D$  para cada año de vida útil.

$$D = \frac{\text{coste} - \text{valor de recuperación}}{\text{vida útil}}$$

$$D = \frac{20.000 - 2.000}{6} = \frac{18.000}{6} = 3.000$$

$$\text{Entrada} \quad \left\{ \begin{array}{l} \text{coste original} \\ \text{vida útil} \\ \text{valor de recuperación} \end{array} \right.$$

Salida	$\left\{ \begin{array}{l} \text{depreciación anual por año} \\ \text{depreciación acumulada en cada año} \\ \text{valor del automóvil en cada año} \end{array} \right.$
Proceso	$\left\{ \begin{array}{l} \text{depreciación acumulada} \\ \text{cálculo de la depreciación acumulada cada año} \\ \text{cálculo del valor del automóvil en cada año} \end{array} \right.$

La tabla siguiente muestra la salida solicitada

Año	Depreciación	Depreciación acumulada	Valor anual
1 (2006)	3.000	3.000	17.000
2 (2007)	3.000	6.000	14.000
3 (2008)	3.000	9.000	11.000
4 (2009)	3.000	12.000	8.000
5 (2010)	3.000	15.000	5.000
6 (2011)	3.000	18.000	2.000

### 2.1.2. Diseño del algoritmo

En la etapa de análisis del proceso de programación se determina *qué* hace el programa. En la etapa de diseño se determina *cómo* hace el programa la tarea solicitada. Los métodos más eficaces para el proceso de diseño se basan en el conocido *divide y vencerás*. Es decir, la resolución de un problema complejo se realiza dividiendo el problema en subproblemas y a continuación dividiendo estos subproblemas en otros de nivel más bajo, hasta que pueda ser *implementada* una solución en la computadora. Este método se conoce técnicamente como **diseño descendente** (*top-down*) o **modular**. El proceso de romper el problema en cada etapa y expresar cada paso en forma más detallada se denomina *refinamiento sucesivo*.

Cada subprograma es resuelto mediante un **módulo** (*subprograma*) que tiene un solo punto de entrada y un solo punto de salida.

Cualquier programa bien diseñado consta de un *programa principal* (el módulo de nivel más alto) que llama a subprogramas (módulos de nivel más bajo) que a su vez pueden llamar a otros subprogramas. Los programas estructurados de esta forma se dice que tienen un *diseño modular* y el método de romper el programa en módulos más pequeños se llama *programación modular*. Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente (incluso por diferentes programadores) y a continuación combinarlos entre sí. El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina:

1. Programar un módulo.
2. Comprobar el módulo.
3. Si es necesario, depurar el módulo.
4. Combinar el módulo con los módulos anteriores.

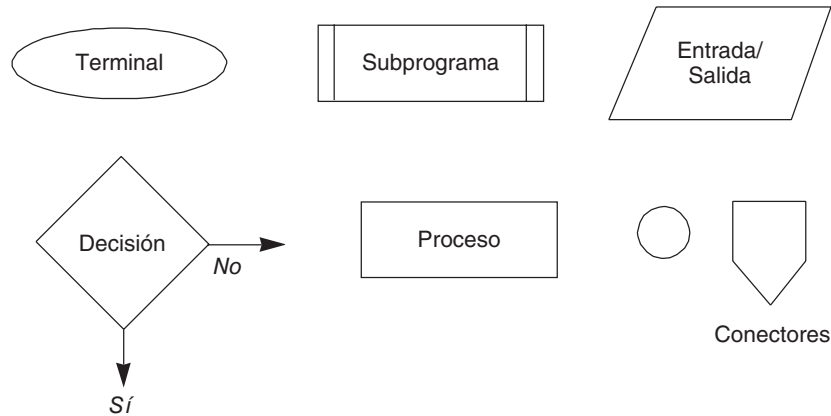
El proceso que convierte los resultados del análisis del problema en un diseño modular con refinamientos sucesivos que permitan una posterior traducción a un lenguaje se denomina **diseño del algoritmo**.

El diseño del algoritmo es independiente del lenguaje de programación en el que se vaya a codificar posteriormente.

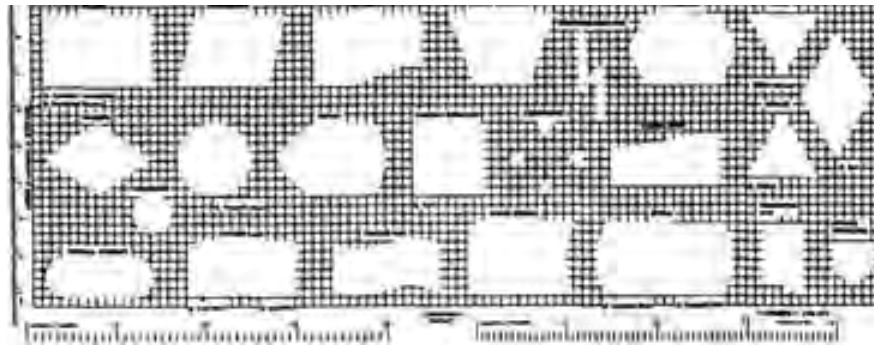
### 2.1.3. Herramientas de programación

Las dos herramientas más utilizadas comúnmente para diseñar algoritmos son: *diagramas de flujo* y *pseudocódigos*.

Un **diagrama de flujo** (*flowchart*) es una representación gráfica de un algoritmo. Los símbolos utilizados han sido normalizados por el Instituto Norteamericano de Normalización (ANSI), y los más frecuentemente empleados se muestran en la Figura 2.2, junto con una plantilla utilizada para el dibujo de los diagramas de flujo (Figura 2.3). En la Figura 2.4 se representa el diagrama de flujo que resuelve el Problema 2.1.



**Figura 2.2.** Símbolos más utilizados en los diagramas de flujo.



**Figura 2.3.** Plantilla para dibujo de diagramas de flujo.

El **pseudocódigo** es una herramienta de programación en la que las instrucciones se escriben en palabras similares al inglés o español, que facilitan tanto la escritura como la lectura de programas. En esencia, el pseudocódigo se puede definir como *un lenguaje de especificaciones de algoritmos*.

Aunque no existen reglas para escritura del pseudocódigo en español, se ha recogido una notación estándar que se utilizará en el libro y que ya es muy empleada en los libros de programación en español<sup>3</sup>. Las palabras reservadas básicas se representarán en letras negritas minúsculas. Estas palabras son traducción libre de palabras reservadas de lenguajes como C, Pascal, etc. Más adelante se indicarán los pseudocódigos fundamentales para utilizar en esta obra.

El pseudocódigo que resuelve el Problema 2.1 es:

```
Previsiones de depreciacion
Introducir coste
    vida util
    valor final de rescate (recuperacion)
imprimir cabeceras
Establecer el valor inicial del año
Calcular depreciacion
```

<sup>3</sup> Para mayor ampliación sobre el *pseudocódigo*, puede consultar, entre otras, algunas de estas obras: *Fundamentos de programación*, Luis Joyanes, 2.ª edición, 1997; *Metodología de la programación*, Luis Joyanes, 1986; *Problemas de Metodología de la programación*, Luis Joyanes, 1991 (todas ellas publicadas en McGraw-Hill, Madrid), así como *Introducción a la programación*, de Clavel y Biondi. Barcelona: Masson, 1987, o bien *Introducción a la programación y a las estructuras de datos*, de Braunstein y Groia. Buenos Aires: Editorial Eudeba, 1986. Para una formación práctica puede consultar: *Fundamentos de programación: Libro de problemas* de Luis Joyanes, Luis Rodríguez y Matilde Fernández, en McGraw-Hill (Madrid, 1998).

```

mientras valor año =< vida util hacer
    calcular depreciacion acumulada
    calcular valor actual
    imprimir una linea en la tabla
    incrementar el valor del año
fin de mientras

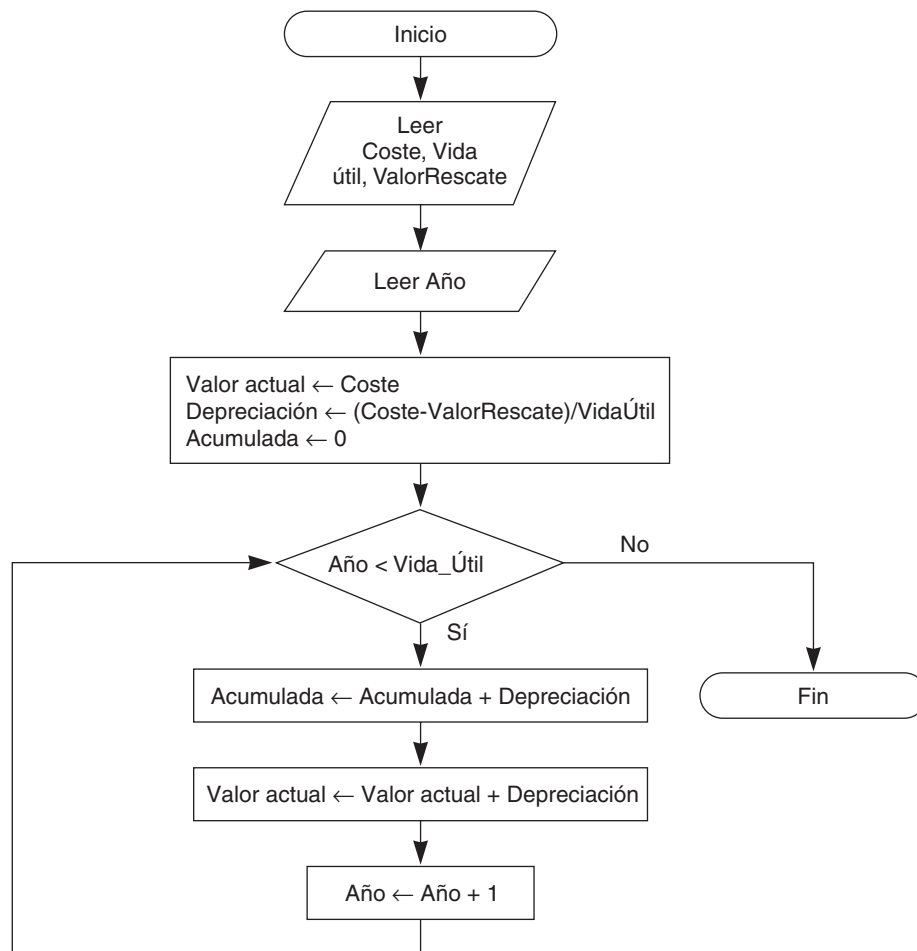
```

## EJEMPLO 2.1

Calcular la paga neta de un trabajador conociendo el número de horas trabajadas, la tarifa horaria y la tasa de impuestos.

### Algoritmo

1. Leer Horas, Tarifa, Tasa
2. Calcular  $\text{PagaBruta} = \text{Horas} * \text{Tarifa}$
3. Calcular  $\text{Impuestos} = \text{PagaBruta} * \text{Tasa}$
4. Calcular  $\text{PagaNeta} = \text{PagaBruta} - \text{Impuestos}$
5. Visualizar PagaBruta, Impuestos, PagaNeta



**Figura 2.4.** Diagrama de flujo (Problema 2.1).



**EJEMPLO 2.2**

Calcular el valor de la suma  $1+2+3+\dots+100$ .

**algoritmo**

Se utiliza una variable `Contador` como un contador que genere los sucesivos números enteros, y `Suma` para almacenar las sumas parciales 1, 1+2, 1+2+3...

1. Establecer `Contador` a 1
2. Establecer `Suma` a 0
3. **mientras** `Contador`  $\leq$  100 **hacer**  
     Sumar `Contador` a `Suma`  
     Incrementar `Contador` en 1  
   **fin\_mientras**
4. Visualizar `Suma`

**2.1.4. Codificación de un programa**

La *codificación* es la escritura en un lenguaje de programación de la representación del algoritmo desarrollada en las etapas precedentes. Dado que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, el código puede ser escrito con igual facilidad en un lenguaje o en otro.

Para realizar la conversión del algoritmo en programa se deben sustituir las palabras reservadas en español por sus homónimos en inglés, y las operaciones/instrucciones indicadas en lenguaje natural por el lenguaje de programación correspondiente.

```
{Este programa obtiene una tabla de depreciaciones
acumuladas y valores reales de cada año de un
determinado producto}
```

**algoritmo** primero

```
Real: Coste, Depreciacion,
      Valor_Recuperacion
      Valor_Actual,
      Acumulado
      Valor_Anual;
entero: Año, Vida_Util;
inicio
  escribir('introduzca coste, valor recuperación y vida útil')
  leer(Coste, Valor_Recuperacion, Vida_Util)
  escribir('Introduzca año actual')
  leer(Año)
  Valor_Actual  $\leftarrow$  Coste;
  Depreciacion  $\leftarrow$  (Coste-Valor_Recuperacion)/Vida_Util
  Acumulado  $\leftarrow$  0
  escribir('Año Depreciación Dep. Acumulada')
  mientras (Año < Vida_Util)
    Acumulado  $\leftarrow$  Acumulado + Depreciacion
    Valor_Actual  $\leftarrow$  Valor_Actual - Depreciacion
    escribir('Año, Depreciacion, Acumulado')
    Año  $\leftarrow$  Año + 1;
  fin mientras
fin
```

### Documentación interna

Como se verá más tarde, la documentación de un programa se clasifica en *interna* y *externa*. La *documentación interna* es la que se incluye dentro del código del programa fuente mediante comentarios que ayudan a la comprensión del código. Todas las líneas de programas que comiencen con un símbolo / \* son *comentarios*. El programa no los necesita y la computadora los ignora. Estas líneas de comentarios sólo sirven para hacer los programas más fáciles de comprender. El objetivo del programador debe ser escribir códigos sencillos y limpios.

Debido a que las máquinas actuales soportan grandes memorias (512 Mb o 1.024 Mb de memoria central mínima en computadoras personales) no es necesario recurrir a técnicas de ahorro de memoria, por lo que es recomendable que se incluya el mayor número de comentarios posibles, pero eso sí, que sean significativos.

### 2.1.5. Compilación y ejecución de un programa

Una vez que el algoritmo se ha convertido en un programa fuente, es preciso introducirlo en memoria mediante el teclado y almacenarlo posteriormente en un disco. Esta operación se realiza con un programa editor. Posteriormente el programa fuente se convierte en un *archivo de programa* que se guarda (graba) en disco.

El **programa fuente** debe ser traducido a lenguaje máquina, este proceso se realiza con el compilador y el sistema operativo que se encarga prácticamente de la compilación.

Si tras la compilación se presentan errores (*errores de compilación*) en el programa fuente, es preciso volver a editar el programa, corregir los errores y compilar de nuevo. Este proceso se repite hasta que no se producen errores, obteniéndose el **programa objeto** que todavía no es ejecutable directamente. Suponiendo que no existen errores en el programa fuente, se debe instruir al sistema operativo para que realice la fase de **montaje** o **enlace** (*link*), carga, del programa objeto con las bibliotecas del programa del compilador. El proceso de montaje produce un **programa ejecutable**. La Figura 2.5 describe el proceso completo de compilación/ejecución de un programa.

Una vez que el programa ejecutable se ha creado, ya se puede ejecutar (correr o rodar) desde el sistema operativo con sólo teclear su nombre (en el caso de DOS). Suponiendo que no existen errores durante la ejecución (llamados **errores en tiempo de ejecución**), se obtendrá la salida de resultados del programa.

Las instrucciones u órdenes para compilar y ejecutar un programa en C, C++,... o cualquier otro lenguaje dependerá de su entorno de programación y del sistema operativo en que se ejecute Windows, Linux, Unix, etc.

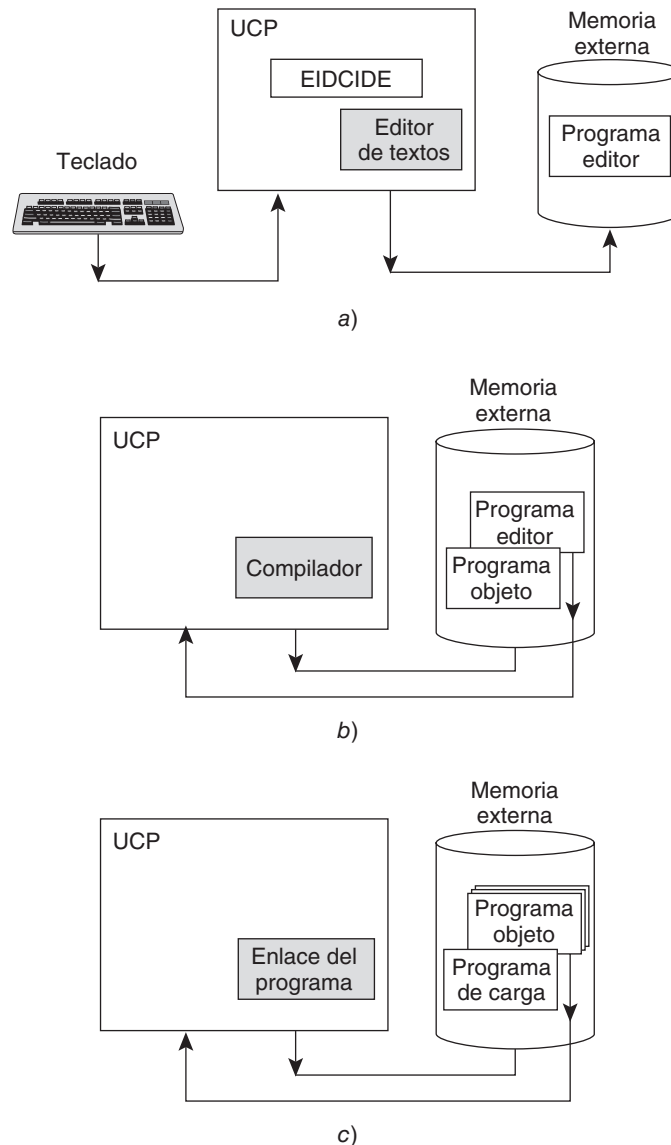
### 2.1.6. Verificación y depuración de un programa

La *verificación* o *compilación* de un programa es el proceso de ejecución del programa con una amplia variedad de datos de entrada, llamados *datos de test* o *prueba*, que determinarán si el programa tiene o no errores (“*bugs*”). Para realizar la verificación se debe desarrollar una amplia gama de datos de test: valores normales de entrada, valores extremos de entrada que comprueben los límites del programa y valores de entrada que comprueben aspectos especiales del programa.

La *depuración* es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores.

Cuando se ejecuta un programa, se pueden producir tres tipos de errores:

1. *Errores de compilación*. Se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación y suelen ser *errores de sintaxis*. Si existe un error de sintaxis, la computadora no puede comprender la instrucción, no se obtendrá el programa objeto y el compilador imprimirá una lista de todos los errores encontrados durante la compilación.
2. *Errores de ejecución*. Estos errores se producen por instrucciones que la computadora puede comprender pero no ejecutar. Ejemplos típicos son: división por cero y raíces cuadradas de números negativos. En estos casos se detiene la ejecución del programa y se imprime un mensaje de error.
3. *Errores lógicos*. Se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y sólo puede advertirse el error por la obtención de resultados incorrectos. En este caso se debe volver a la fase de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente y compilar y ejecutar una vez más.



**Figura 2.5.** Fases de la compilación/ejecución de un programa:  
a) edición; b) compilación; c) montaje o enlace.

### 2.1.7. Documentación y mantenimiento

La documentación de un problema consta de las descripciones de los pasos a dar en el proceso de resolución de dicho problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Programas pobremente documentados son difíciles de leer, más difíciles de depurar y casi imposibles de mantener y modificar.

La documentación de un programa puede ser *interna* y *externa*. La *documentación interna* es la contenida en líneas de comentarios. La *documentación externa* incluye análisis, diagramas de flujo y/o pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros o bien cambiar el programa. Tales cambios se denominan *mantenimiento del programa*. Después de cada cambio la documentación debe ser actualizada para facilitar cambios posteriores. Es práctica frecuente numerar las sucesivas versiones de los programas **1.0, 1.1, 2.0, 2.1**, etc. (Si los cambios introducidos son importantes, se varía el primer dígito [**1.0, 2.0**,...]; en caso de pequeños cambios sólo se varía el segundo dígito [**2.0, 2.1**...].)

## 2.2. PROGRAMACIÓN MODULAR

La *programación modular* es uno de los métodos de diseño más flexible y potente para mejorar la productividad de un programa. En programación modular el programa se divide en *módulos* (partes independientes), cada uno de los cuales ejecuta una única actividad o tarea y se codifican independientemente de otros módulos. Cada uno de estos módulos se analiza, codifica y pone a punto por separado. Cada programa contiene un módulo denominado *programa principal* que controla todo lo que sucede; se transfiere el control a *submódulos* (posteriormente se denominarán *subprogramas*), de modo que ellos puedan ejecutar sus funciones; sin embargo, cada submódulo devuelve el control al módulo principal cuando se haya completado su tarea. Si la tarea asignada a cada submódulo es demasiado compleja, éste deberá romperse en otros módulos más pequeños. El proceso sucesivo de subdivisión de módulos continúa hasta que cada módulo tenga solamente una tarea específica que ejecutar. Esta tarea puede ser *entrada*, *salida*, *manipulación de datos*, *control de otros módulos* o alguna *combinación de éstos*. Un módulo puede transferir temporalmente (*bifurcar*) el control a otro módulo; sin embargo, cada módulo debe eventualmente devolver el control al módulo del cual se recibe originalmente el control.

Los módulos son independientes en el sentido en que ningún módulo puede tener acceso directo a cualquier otro módulo excepto el módulo al que llama y sus propios submódulos. Sin embargo, los resultados producidos por un módulo pueden ser utilizados por cualquier otro módulo cuando se transfiera a ellos el control.

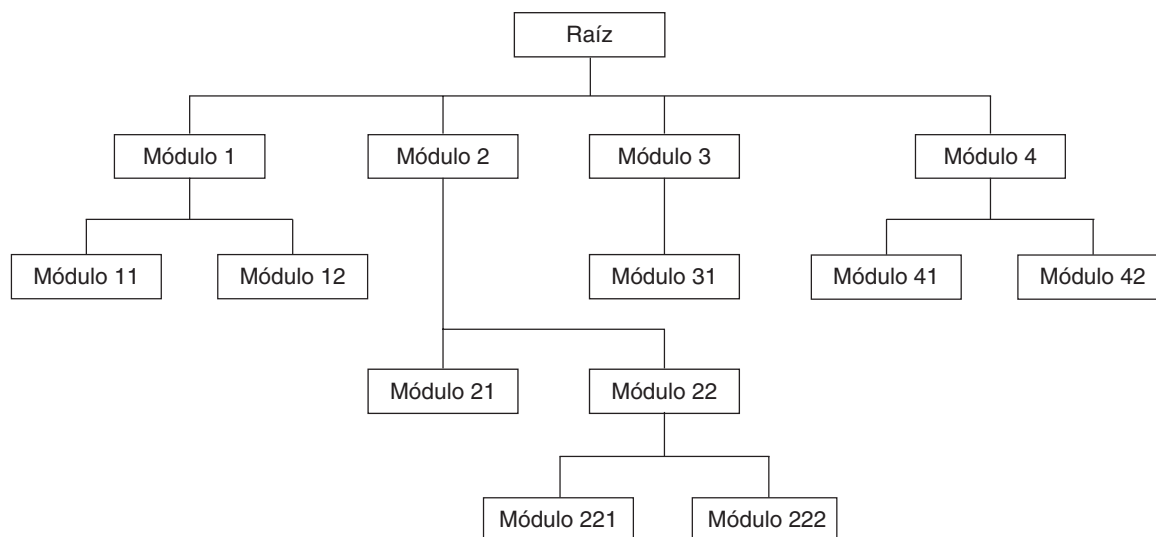


Figura 2.6. Programación modular.

Dado que los módulos son independientes, diferentes programadores pueden trabajar simultáneamente en diferentes partes del mismo programa. Esto reducirá el tiempo del diseño del algoritmo y posterior codificación del programa. Además, un módulo se puede modificar radicalmente sin afectar a otros módulos, incluso sin alterar su función principal.

La descomposición de un programa en módulos independientes más simples se conoce también como el método de *divide y vencerás* (*divide and conquer*). Cada módulo se diseña con independencia de los demás, y siguiendo un método ascendente o descendente se llegará hasta la descomposición final del problema en módulos en forma jerárquica.

## 2.3. PROGRAMACIÓN ESTRUCTURADA

C, Pascal, FORTRAN, y lenguajes similares, se conocen como *lenguajes procedimentales* (por procedimientos). Es decir, cada sentencia o instrucción señala al compilador para que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir por cinco, etc. En resumen, un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias. En el caso de pequeños programas, estos principios de organización (denominados *paradigma*) se demuestran eficientes. El programador sólo tiene que crear esta lista de

instrucciones en un lenguaje de programación, compilar en la computadora y ésta, a su vez, ejecuta estas instrucciones.

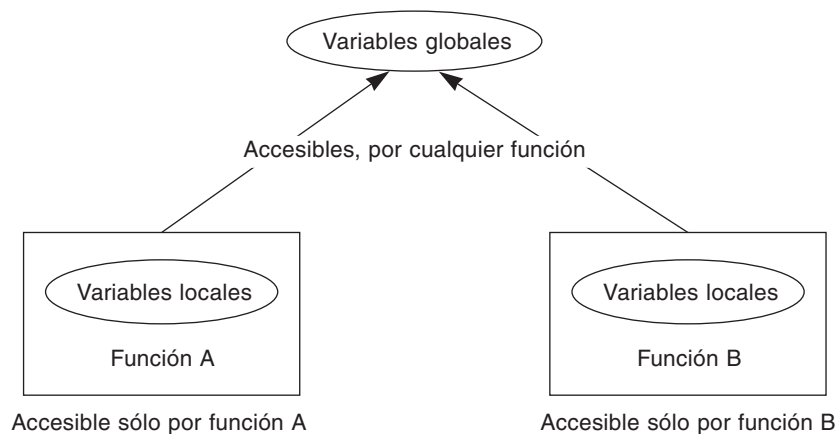
Cuando los programas se vuelven más grandes, cosa que lógicamente sucede cuando aumenta la complejidad del problema a resolver, la lista de instrucciones aumenta considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Los programadores pueden controlar, de modo normal, unos centenares de líneas de instrucciones. Para resolver este problema los programas se descompusieron en unidades más pequeñas que adoptaron el nombre de *funciones* (*procedimientos*, *subprogramas* o *subrutinas* en otros lenguajes de programación). De este modo en un programa orientado a procedimientos se divide en funciones, de modo que cada función tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de romper el programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas *módulos* (normalmente, en el caso de C, denominadas **archivos** o **ficheros**); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias). Esta característica hace que a medida que los programas se hacen más grandes y complejos, el paradigma estructurado comienza a dar señales de debilidad y resultando muy difícil terminar los programas de un modo eficiente. Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos. Tal vez las dos razones más evidentes son éstas. Primero, las funciones tienen acceso ilimitado a los datos globales. Segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental proporcionan un modelo pobre del mundo real.

### 2.3.1. Datos locales y datos globales

En un programa procedimental, por ejemplo escrito en C, existen dos tipos de datos. *Datos locales* que están ocultos en el interior de la función y son utilizados, exclusivamente, por la función. Estos datos locales están estrechamente relacionados con sus funciones y están protegidos de modificaciones por otras funciones.

Otro tipo de datos son los *datos globales* a los cuales se puede acceder desde *cualquier* función del programa. Es decir, dos o más funciones pueden acceder a los mismos datos siempre que estos datos sean globales. En la Figura 2.7 se muestra la disposición de variables locales y globales en un programa procedimental.



**Figura 2.7.** Datos locales y globales.

Un programa grande (Figura 2.8) se compone de numerosas funciones y datos globales y ello conlleva una multitud de conexiones entre funciones y datos que dificulta su comprensión y lectura.

Todas estas conexiones múltiples originan diferentes problemas. En primer lugar, hacen difícil concepcuar la estructura del programa. En segundo lugar, el programa es difícil de modificar ya que cambios en datos globales pueden necesitar la reescritura de todas las funciones que acceden a los mismos. También puede suceder que estas modificaciones de los datos globales pueden no ser aceptadas por todas o algunas de las funciones.

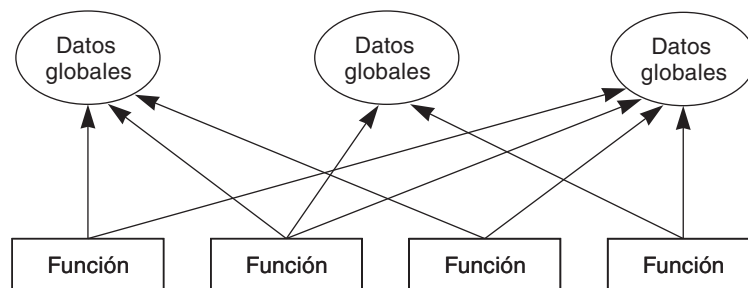


Figura 2.8. Un programa procedimental.

### 2.3.2. Modelado del mundo real

Un segundo problema importante de la programación estructurada reside en el hecho de que la disposición separada de datos y funciones no se corresponden con los modelos de las cosas del mundo real. En el mundo físico se trata con objetos físicos tales como personas, autos o aviones. Estos objetos no son como los datos ni como las funciones. Los objetos complejos o no del mundo real tienen *atributos* y *comportamiento*.

Los **atributos** o características de los objetos son, por ejemplo: en las personas, su edad, su profesión, su domicilio, etc.; en un auto, la potencia, el número de matrícula, el precio, número de puertas, etc; en una casa, la superficie, el precio, el año de construcción, la dirección, etc. En realidad, los atributos del mundo real tienen su equivalente en los datos de un programa; tienen un valor específico, tal como 200 metros cuadrados, 20.000 dólares, cinco puertas, etc.

El **comportamiento** es una acción que ejecutan los objetos del mundo real como respuesta a un determinado estímulo. Si usted pisa los frenos en un auto, el coche (carro) se detiene; si acelera, el auto aumenta su velocidad, etcétera. El comportamiento, en esencia, es como una función: se llama a una función para hacer algo (visualizar la nómina de los empleados de una empresa).

Por estas razones, ni los datos ni las funciones, por sí mismas, modelan los objetos del mundo real de un modo eficiente.

La programación estructurada mejora la claridad, fiabilidad y facilidad de mantenimiento de los programas; sin embargo, para programas grandes o a gran escala, presentan retos de difícil solución.

## 2.4. PROGRAMACIÓN ORIENTADA A OBJETOS

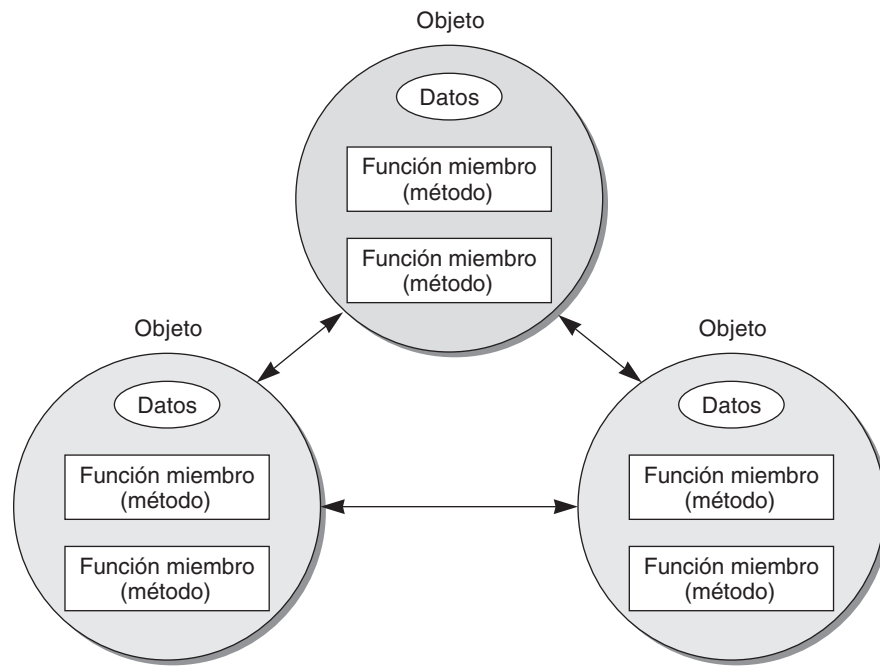
La programación orientada a objetos, tal vez el paradigma de programación más utilizado en el mundo del desarrollo de software y de la ingeniería de software del siglo XXI, trae un nuevo enfoque a los retos que se plantean en la programación estructurada cuando los problemas a resolver son complejos. Al contrario que la programación *procedimental* que enfatiza en los algoritmos, la POO enfatiza en los datos. En lugar de intentar ajustar un problema al enfoque *procedimental* de un lenguaje, POO intenta ajustar el lenguaje al problema. La idea es diseñar formatos de datos que se correspondan con las características esenciales de un problema.

La idea fundamental de los lenguajes orientados a objetos es combinar en una única unidad o módulo, tanto los datos como las funciones que operan sobre esos datos. Tal unidad se llama un **objeto**.

Las funciones de un objeto se llaman *funciones miembro* en C++ o *métodos* (éste es el caso de Smalltalk, uno de los primeros lenguajes orientados a objetos), y son el único medio para acceder a sus datos. Los datos de un objeto, se conocen también como *atributos* o *variables de instancia*. Si se desea leer datos de un objeto, se llama a una función miembro del objeto. Se accede a los datos y se devuelve un valor. No se puede acceder a los datos directamente. Los datos están ocultos, de modo que están protegidos de alteraciones accidentales. Los datos y las funciones se dice que están *encapsulados en una única entidad*. El *encapsulamiento de datos* y la *ocultación* de los datos son términos clave 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 miembros del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración

y mantenimiento del programa. Un programa C++ se compone normalmente de un número de objetos que se comunican unos con otros mediante la llamada a otras funciones miembro. La organización de un programa en C++ se muestra en la Figura 2.9. La llamada a una función miembro de un objeto se denomina *enviar un mensaje* a otro objeto.



**Figura 2.9.** Organización típica de un programa orientado a objetos.

En el paradigma orientado a objetos, el programa se organiza como un conjunto finito de objetos que contiene datos y operaciones (*funciones miembro o métodos*) que llaman a esos datos y que se comunican entre sí mediante mensajes.

### 2.4.1. Propiedades fundamentales de la orientación a objetos

Existen diversas características ligadas a la orientación a objetos. Todas las propiedades que se suelen considerar no son exclusivas de este paradigma, ya que pueden existir en otros paradigmas, pero en su conjunto definen claramente los lenguajes orientados a objetos. Estas propiedades son:

- **Abstracción (tipos abstractos de datos y clases).**
- **Encapsulado de datos.**
- **Ocultación de datos.**
- **Herencia.**
- **Polimorfismo.**

### 2.4.2. Abstracción

La abstracción es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta los restantes aspectos. El término **abstracción** que se suele

utilizar en programación se refiere al hecho de diferenciar entre las propiedades externas de una entidad y los detalles de la composición interna de dicha entidad. Es la abstracción la que permite ignorar los detalles internos de un dispositivo complejo tal como una computadora, un automóvil, una lavadora o un horno de microondas, etc., y usarlo como una única unidad comprensible. Mediante la abstracción se diseñan y fabrican estos sistemas complejos en primer lugar y, posteriormente, los componentes más pequeños de los cuales están compuestos. Cada componente representa un nivel de abstracción en el cual el uso del componente se aísla de los detalles de la composición interna del componente. La abstracción posee diversos grados denominados niveles de abstracción.

En consecuencia, la abstracción posee diversos grados de complejidad que se denominan *niveles de abstracción* que ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. En el modelado orientado a objetos de un sistema esto significa centrarse en *qué* es y *qué hace* un objeto y no en *cómo* debe implementarse. Durante el proceso de abstracción es cuando se decide qué características y comportamiento debe tener el modelo.

Aplicando la abstracción se es capaz de construir, analizar y gestionar sistemas de computadoras complejos y grandes que no se podrían diseñar si se tratara de modelar a un nivel detallado. En cada nivel de abstracción se visualiza el sistema en términos de componentes, denominados **herramientas abstractas**, cuya composición interna se ignora. Esto nos permite concentrarnos en cómo cada componente interactúa con otros componentes y centrarnos en la parte del sistema que es más relevante para la tarea a realizar en lugar de perderse a nivel de detalles menos significativos.

En estructuras o registros, las propiedades individuales de los objetos se pueden almacenar en los miembros. Para los objetos, no sólo es de interés *cómo* están organizados, sino también *qué* se puede hacer con ellos; es decir, las operaciones que forman la interfaz de un objeto son también importantes. El primer concepto en el mundo de la orientación a objetos nació con los tipos abstractos de datos (TAD). Un tipo abstracto de datos describe no sólo los atributos de un objeto, sino también su comportamiento (las operaciones). Esto puede incluir también una descripción de los estados que puede alcanzar un objeto.

Un medio de reducir la complejidad es la abstracción. Las características y los procesos se reducen a las propiedades esenciales, son resumidas o combinadas entre sí. De este modo, las características complejas se hacen más manejables.

---

### EJEMPLO 2.3

*Diferentes modelos de abstracción del término coche (carro).*

- Un coche (carro) es la combinación (o composición) de diferentes partes, tales como motor, carrocería, cuatro ruedas, cinco puertas, etc.
- Un coche (carro) es un concepto común para diferentes tipos de coches. Pueden clasificarse por el nombre del fabricante (Audi, BMW, SEAT, Toyota, Chrisler...), por su categoría (turismo, deportivo, todoterreno...), por el carburante que utilizan (gasolina, gasoil, gas, híbrido...).

La abstracción coche se utilizará siempre que la marca, la categoría o el carburante no sean significativos. Así, un carro (coche) se utilizará para transportar personas o ir de Carchelejo a Cazorla.

---

### 2.4.3. Encapsulación y ocultación de datos

El *encapsulado* o *encapsulación de datos* es el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación. En el caso de los objetos que poseen las mismas características y comportamiento se agrupan en clases, que no son más que unidades o módulos de programación que encapsulan datos y operaciones.

La ocultación de datos permite separar el aspecto de un componente, definido por su *interfaz* con el exterior, de sus detalles internos de implementación. Los términos ocultación de la información (*information hiding*) y encapsulación de datos (*data encapsulation*) se suelen utilizar como sinónimos, pero no siempre es así, y muy al contrario, son términos similares pero distintos. Normalmente, los datos internos están protegidos del exterior y no se puede acceder a ellos más que desde su propio interior y por tanto, no están ocultos. El acceso al objeto está restringido sólo a través de una interfaz bien definida.



El diseño de un programa orientado a objetos contiene, al menos, los siguientes pasos:

1. Identificar los *objetos* del sistema.
2. Agrupar en *clases* a todos objetos que tengan características y comportamiento comunes.
3. Identificar los *datos* y *operaciones* de cada una de las clases.
4. Identificar las *relaciones* que pueden existir entre las clases.

Un **objeto** es un elemento individual con su propia identidad; por ejemplo, un libro, un automóvil... Una **clase** puede describir las propiedades genéricas de un ejecutivo de una empresa (nombre, título, salario, cargo...) mientras que un objeto representará a un ejecutivo específico (Luis Mackoy, director general). En general, una clase define qué datos se utilizan para representar un objeto y las operaciones que se pueden ejecutar sobre esos datos.

Cada clase tiene sus propias características y comportamiento; en general, una clase define los datos que se utilizan y las operaciones que se pueden ejecutar sobre esos datos. Una clase describe un objeto. En el sentido estricto de programación, una clase es un tipo de datos. Diferentes variables se pueden crear de este tipo. En programación orientada a objetos, éstas se llaman *instancias*. Las instancias son, por consiguiente, la realización de los objetos descritos en una clase. Estas instancias constan de datos o atributos descritos en la clase y se pueden manipular con las operaciones definidas dentro de ellas.

Los términos *objeto* e *instancia* se utilizan frecuentemente como sinónimos (especialmente en C++). Si una variable de tipo `Carro` se declara, se crea un objeto `Carro` (una instancia de la clase `Carro`).

Las operaciones definidas en los objetos se llaman *métodos*. Cada operación llamada por un objeto se interpreta como un *mensaje* al objeto, que utiliza un método específico para procesar la operación.

En el diseño de programas orientados a objetos se realiza en primer lugar el diseño de las clases que representan con precisión aquellas cosas que trata el programa. Por ejemplo, un programa de dibujo, puede definir clases que representan rectángulos, líneas, pinceles, colores, etc. Las definiciones de clases incluyen una descripción de operaciones permisibles para cada clase, tales como desplazamiento de un círculo o rotación de una línea. A continuación se prosigue el diseño de un programa utilizando objetos de las clases.

El diseño de clases fiables y útiles puede ser una tarea difícil. Afortunadamente, los lenguajes POO facilitan la tarea ya que incorporan clases existentes en su propia programación. Los fabricantes de software proporcionan numerosas bibliotecas de clases, incluyendo bibliotecas de clases diseñadas para simplificar la creación de programas para entornos tales como Windows, Linux, Macintosh o Unix. Uno de los beneficios reales de C++ es que permite la reutilización y adaptación de códigos existentes y ya bien probados y depurados.

#### 2.4.4. Objetos

El objeto es el centro de la programación orientada a objetos. Un objeto es algo que se visualiza, se utiliza y juega un rol o papel. Si se programa con enfoque orientado a objetos, se intentan descubrir e implementar los objetos que juegan un rol en el dominio del problema y en consecuencia programa. La estructura interna y el comportamiento de un objeto, en una primera fase, no tiene prioridad. Es importante que un objeto tal como un carro o una casa juegan un rol.

Dependiendo del problema, diferentes aspectos de un aspecto son relevantes. Un carro puede ser ensamblado de partes tales como un motor, una carrocería, unas puertas o puede ser descrito utilizando propiedades tales como su velocidad, su kilometraje o su fabricante. Estos atributos indican el objeto. De modo similar, una persona también se puede ver como un objeto, del cual se disponen de diferentes atributos. Dependiendo de la definición del problema, esos atributos pueden ser el nombre, apellido, dirección, número de teléfono, color del cabello, altura, peso, profesión, etc.

Un objeto no necesariamente ha de realizar algo concreto o tangible. Puede ser totalmente abstracto y también puede describir un proceso. Por ejemplo, un partido de baloncesto o de rugby puede ser descrito como un objeto. Los atributos de este objeto pueden ser los jugadores, el entrenador, la puntuación y el tiempo transcurrido de partido.

Cuando se trata de resolver un problema con orientación a objetos, dicho problema no se descompone en funciones como en programación estructurada tradicional, caso de C, sino en objetos. El pensar en términos de objetos tiene una gran ventaja: se asocian los objetos del problema a los objetos del mundo real.

¿Qué tipos de cosas son objetos en los programas orientados a objetos? La respuesta está limitada por su imaginación aunque se pueden agrupar en categorías típicas que facilitarán su búsqueda en la definición del problema de un modo más rápido y sencillo.

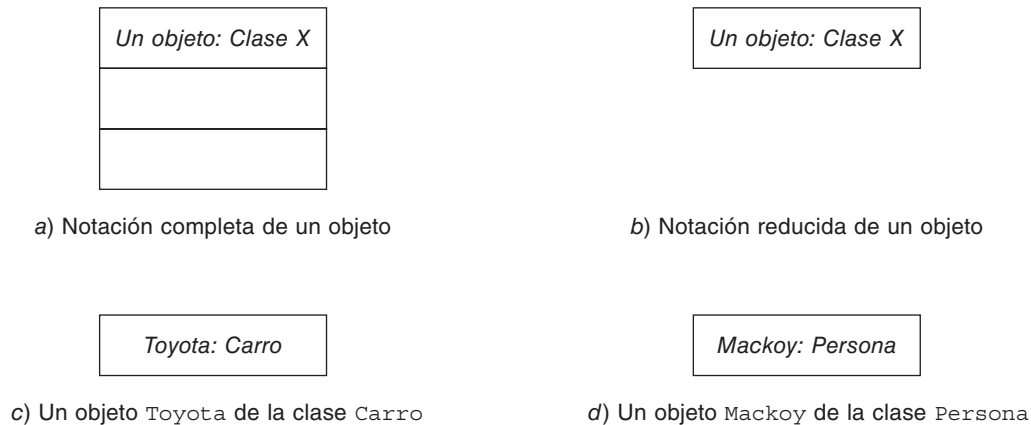
- Recursos Humanos:
  - Empleados.
  - Estudiantes.
  - Clientes.
  - Vendedores.
  - Socios.
- Colecciones de datos:
  - Arrays (arreglos).
  - Listas.
  - Pilas.
  - Árboles.
  - Árboles binarios.
  - Grafos.
- Tipos de datos definidos por usuarios:
  - Hora.
  - Números complejos.
  - Puntos del plano.
  - Puntos del espacio.
  - Ángulos.
  - Lados.
- Elementos de computadoras:
  - Menús.
  - Ventanas.
  - Objetos gráficos (rectángulos, círculos, rectas, puntos...).
  - Ratón (mouse).
  - Teclado.
  - Impresora.
  - USB.
  - Tarjetas de memoria de cámaras fotográficas.
- Objetos físicos:
  - Carros.
  - Aviones.
  - Trenes.
  - Barcos.
  - Motocicletas.
  - Casas.
- Componentes de videojuegos:
  - Consola.
  - Mandos.
  - Volante.
  - Conectores.
  - Memoria.
  - Acceso a Internet.

La correspondencia entre objetos de programación y objetos del mundo real es el resultado eficiente de combinar datos y funciones que manipulan esos datos. Los objetos resultantes ofrecen una mejor solución al diseño del programa que en el caso de los lenguajes orientados a procedimientos.

Un **objeto** se puede definir desde el punto de vista conceptual como una entidad individual de un sistema y que se caracteriza por un estado y un comportamiento. Desde el punto de vista de implementación un **objeto** es una entidad que posee un conjunto de *datos* y un conjunto de *operaciones* (*funciones* o *métodos*).

El estado de un objeto viene determinado por los valores que toman sus datos, cuyos valores pueden tener las restricciones impuestas en la definición del problema. Los datos se denominan también *atributos* y componen la estructura del objeto y las operaciones —también llamadas *métodos*— representan los servicios que proporciona el objeto.

La representación gráfica de un objeto en **UML** se muestra en la Figura 2.10.



**Figura 2.10.** Representación de objetos en UML (Lenguaje Unificado de Modelado).

## 2.4.5. Clases

En POO los objetos son miembros de **clases**. En esencia, una clase es un tipo de datos al igual que cualquier otro tipo de dato definido en un lenguaje de programación. La diferencia reside en que la clase es un tipo de dato que contiene datos y funciones. Una clase contiene muchos objetos y es preciso definirla, aunque su definición no implica creación de objetos.

Una clase es, por consiguiente, una descripción de un número de objetos similares. Madonna, Sting, Prince, Juanes, Carlos Vives o Juan Luis Guerra son miembros u objetos de la clase "músicos". Un objeto concreto, Juanes o Carlos Vives, son *instancias* de la clase "músicos de rock".

Una clase es una descripción general de un conjunto de objetos similares. Por definición todos los objetos de una clase comparten los mismos atributos (datos) y las mismas operaciones (métodos). Una clase encapsula las abstracciones de datos y operaciones necesarias para describir una entidad u objeto del mundo real.

Una clase se representa en **UML** mediante un rectángulo que contiene en una banda con el nombre de la clase y opcionalmente otras dos bandas con el nombre de sus atributos y de sus operaciones o métodos (Figuras 2.11 y 2.12).

## 2.4.6. Generalización y especialización: herencia

La *generalización* es la propiedad que permite compartir información entre dos entidades evitando la redundancia. En el comportamiento de objetos existen con frecuencia propiedades que son comunes en diferentes objetos y esta propiedad se denomina generalización.

Por ejemplo, máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas, etc., son todos electrodomésticos (aparatos del hogar). En el mundo de la orientación a objetos, cada uno de estos aparatos es una **sub-clase** de la clase Electrodoméstico y a su vez Electrodoméstico es una **superclase** de todas las otras clases (máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas...). El proceso inverso de la generalización por el cual se definen nuevas clases a partir de otras ya existentes se denomina *especialización*.

En orientación a objetos, el mecanismo que implementa la propiedad de generalización se denomina **herencia**. La herencia permite definir nuevas clases a partir de otras clases ya existentes, de modo que presentan las mismas características y comportamiento de éstas, así como otras adicionales.

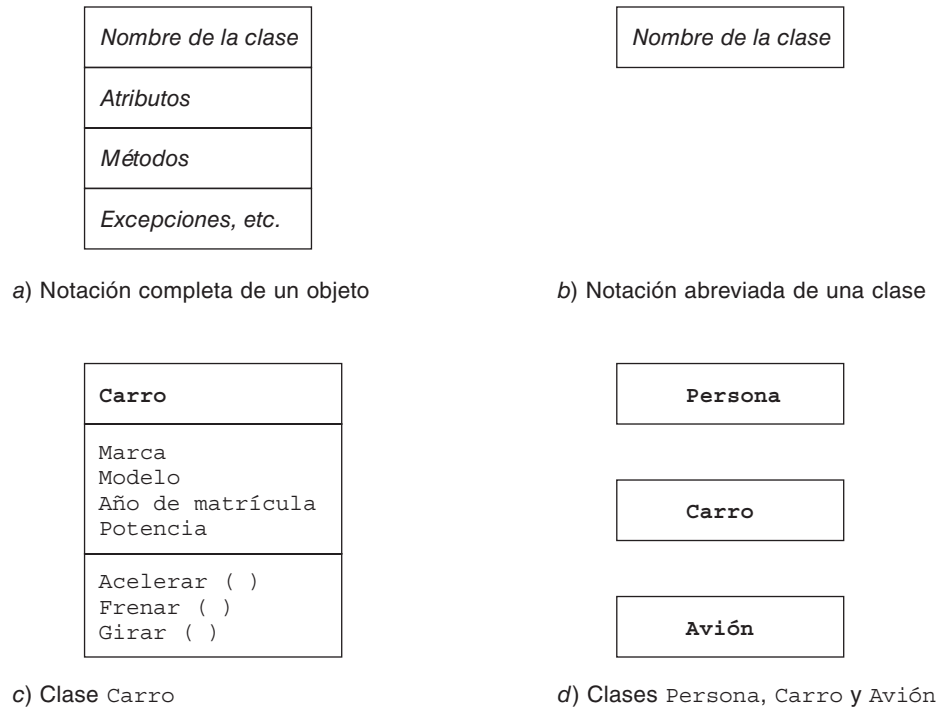


Figura 2.11. Representación de clases en UML.

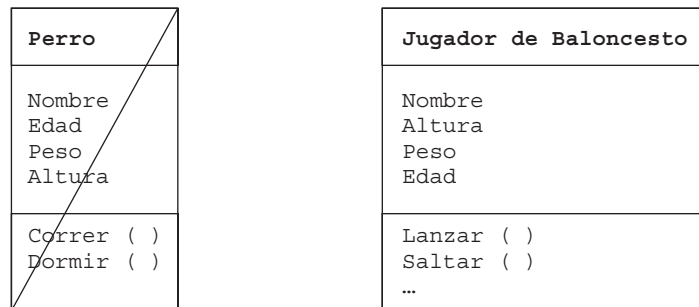


Figura 2.12. Representación de clases en UML con atributos y métodos.

La idea de clases conduce a la idea de herencia. Clases diferentes se pueden conectar unas con otras de modo jerárquico. Como ya se ha comentado anteriormente con las relaciones de generalización y especialización, en nuestras vidas diarias se utiliza el concepto de clases divididas en subclases. La clase animal se divide en anfibios, mamíferos, insectos, pájaros, etc., y la clase vehículo en carros, motos, camiones, buses, etc.

El principio de la división o clasificación es que cada subclase comparte características comunes con la clase de la que procede o se deriva. Los carros, motos, camiones y buses tienen ruedas, motores y carrocerías; son las características que definen a un vehículo. Además de las características comunes con los otros miembros de la clase, cada subclase tiene sus propias características. Por ejemplo los camiones tienen una cabina independiente de la caja que transporta la carga; los buses tienen un gran número de asientos independientes para los viajeros que ha de transportar, etc. En la Figura 2.13 se muestran clases pertenecientes a una jerarquía o herencia de clases.

De modo similar una clase se puede convertir en padre o raíz de otras subclases. En C++ la clase original se denomina *clase base* y las clases que se derivan de ella se denominan *clases derivadas* y siempre son una especialización o *concreción* de su clase base. A la inversa, la clase base es la generalización de la clase derivada. Esto significa que todas las propiedades (atributos y operaciones) de la clase base se heredan por la clase derivada, normalmente suplementada con propiedades adicionales.

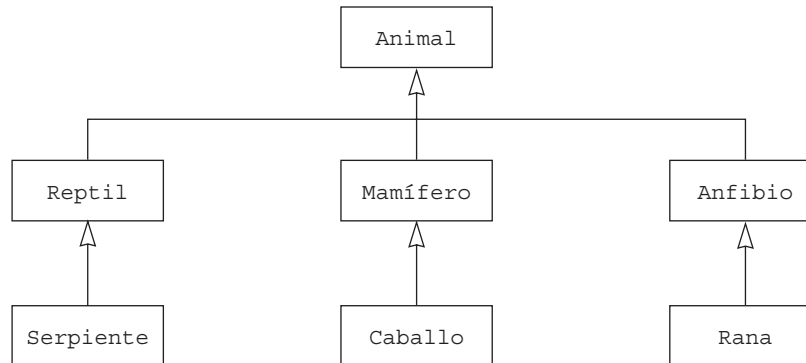


Figura 2.13. Herencia de clases en UML.

### 2.4.7 Reusabilidad

Una vez que una clase ha sido escrita, creada y depurada, se puede distribuir a otros programadores para utilizar en sus propios programas. Esta propiedad se llama *reusabilidad*<sup>4</sup> o *reutilización*. Su concepto es similar a las funciones incluidas en las bibliotecas de funciones de un lenguaje procedimental como C que se pueden incorporar en diferentes programas.

En C++, el concepto de herencia proporciona una extensión o ampliación al concepto de *reusabilidad*. Un programador puede considerar una clase existente y sin modificarla, añadir competencias y propiedades adicionales a ella. Esto se consigue derivando una nueva clase de una ya existente. La nueva clase heredará las características de la clase antigua, pero es libre de añadir nuevas características propias.

La facilidad de reutilizar o reusar el software existente es uno de los grandes beneficios de la POO: muchas empresas consiguen con la reutilización de clase en nuevos proyectos la reducción de los costes de inversión en sus presupuestos de programación. ¿En esencia cuáles son las ventajas de la herencia? Primero, se utiliza para consistencia y reducir código. Las propiedades comunes de varias clases sólo necesitan ser implementadas una vez y sólo necesitan modificarse una vez si es necesario. La otra ventaja es que el concepto de abstracción de la funcionalidad común está soportada.

### 2.4.8. Polimorfismo

Además de las ventajas de consistencia y reducción de código, la herencia, aporta también otra gran ventaja: facilitar el polimorfismo. Polimorfismo es la propiedad de que un operador o una función actúen de modo diferente en función del objeto sobre el que se aplican. En la práctica, el polimorfismo significa la capacidad de una operación de ser interpretada sólo por el propio objeto que lo invoca. Desde un punto de vista práctico de ejecución del programa, el polimorfismo se realiza en tiempo de ejecución ya que durante la compilación no se conoce qué tipo de objeto y por consiguiente qué operación ha sido llamada. En el Capítulo 14 se describirá en profundidad la propiedad de polimorfismo y los diferentes modos de implementación del polimorfismo.

La propiedad de **polimorfismo** es aquella en que una operación tiene el mismo nombre en diferentes clases, pero se ejecuta de diferentes formas en cada clase. Así, por ejemplo, la operación de abrir se puede dar en diferentes clases: abrir una puerta, abrir una ventana, abrir un periódico, abrir un archivo, abrir una cuenta corriente en un banco, abrir un libro, etc. En cada caso se ejecuta una operación diferente aunque tiene el mismo nombre en todos ellos “abrir”. El polimorfismo es la propiedad de una operación de ser interpretada sólo por el objeto al que pertenece. Existen diferentes formas de implementar el polimorfismo y variará dependiendo del lenguaje de programación.

Veamos el concepto con ejemplos de la vida diaria.

En un taller de reparaciones de automóviles existen numerosos carros, de marcas diferentes, de modelos diferentes, de tipos diferentes, potencias diferentes, etc. Constituyen una clase o colección heterogénea de carros (coches). Supongamos que se ha de realizar una operación común “cambiar los frenos del carro”. La operación a realizar es la

<sup>4</sup> El término proviene del concepto inglés *reusability*. La traducción no ha sido aprobada por la RAE, pero se incorpora al texto por su gran uso y difusión entre los profesionales de la informática.

misma, incluye los mismos principios, sin embargo, dependiendo del coche, en particular, la operación será muy diferente, incluirá diferentes acciones en cada caso. Otro ejemplo a considerar y relativo a los operadores “+” y “\*” aplicados a números enteros o números complejos; aunque ambos son números, en un caso la suma y multiplicación son operaciones simples, mientras que en el caso de los números complejos al componerse de parte real y parte imaginaria, será necesario seguir un método específico para tratar ambas partes y obtener un resultado que también será un número complejo.

El uso de operadores o funciones de forma diferente, dependiendo de los objetos sobre los que están actuando se llama polimorfismo (una cosa con diferentes formas). Sin embargo, cuando un operador existente, tal como + o =, se le permite la posibilidad de operar sobre nuevos tipos de datos, se dice entonces que el operador está sobrecargado. La sobrecarga es un tipo de polimorfismo y una característica importante de la POO. En el Capítulo 10 se ampliará, también en profundidad, este nuevo concepto.

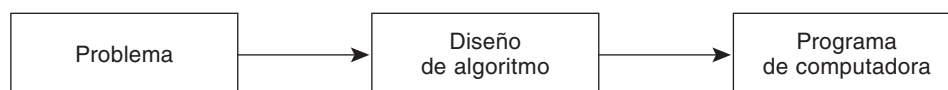
## 2.5. CONCEPTO Y CARACTERÍSTICAS DE ALGORITMOS

El objetivo fundamental de este texto es enseñar a resolver problemas mediante una computadora. El programador de computadora es antes que nada una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático. A lo largo de todo este libro nos referiremos a la *metodología necesaria para resolver problemas mediante programas*, concepto que se denomina **metodología de la programación**. El eje central de esta metodología es el concepto, ya tratado, de algoritmo.

*Un algoritmo es un método para resolver un problema.* Aunque la popularización del término ha llegado con el advenimiento de la era informática, **algoritmo** proviene —como se comentó anteriormente— de *Mohammed al-KhoWârizmi*, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido en la palabra *algorismus* derivó posteriormente en algoritmo. Euclides, el gran matemático griego (del siglo IV a. C.) que inventó un método para encontrar el máximo común divisor de dos números, se considera con Al-Khowârizmi el otro gran padre de la algoritmia (ciencia que trata de los algoritmos).

El profesor Niklaus Wirth —inventor de Pascal, Modula-2 y Oberon— tituló uno de sus más famosos libros, *Algoritmos + Estructuras de datos = Programas*, significándonos que sólo se puede llegar a realizar un buen programa con el diseño de un algoritmo y una correcta estructura de datos. Esta ecuación será una de las hipótesis fundamentales consideradas en esta obra.

La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto.



**Figura 2.14.** Resolución de un problema.

Los pasos para la resolución de un problema son:

1. *Diseño del algoritmo*, que describe la secuencia ordenada de pasos —sin ambigüedades— que conducen a la solución de un problema dado. (*Análisis del problema y desarrollo del algoritmo.*)
2. Expresar el algoritmo como un *programa* en un lenguaje de programación adecuado. (*Fase de codificación.*)
3. *Ejecución y validación* del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será *el diseño de algoritmos*. A la enseñanza y práctica de esta tarea se dedica gran parte de este libro.

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, *la solución de un problema se puede expresar mediante un algoritmo*.

### 2.5.1. Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser *preciso* e indicar el orden de realización de cada paso.
- Un algoritmo debe estar bien *definido*. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser *finito*. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: *Entrada*, *Proceso* y *Salida*. En el algoritmo de receta de cocina citado anteriormente se tendrá:

<i>Entrada:</i>	Ingredientes y utensilios empleados.
<i>Proceso:</i>	Elaboración de la receta en la cocina.
<i>Salida:</i>	Terminación del plato (por ejemplo, cordero).

---

#### EJEMPLO 2.4

*Un cliente ejecuta un pedido a una fábrica. La fábrica examina en su banco de datos la ficha del cliente, si el cliente es solvente entonces la empresa acepta el pedido; en caso contrario, rechazará el pedido. Redactar el algoritmo correspondiente.*

Los pasos del algoritmo son:

1. Inicio.
  2. Leer el pedido.
  3. Examinar la ficha del cliente.
  4. Si el cliente es solvente, aceptar pedido;  
    en caso contrario, rechazar pedido.
  5. Fin.
- 

#### EJEMPLO 2.5

*Se desea diseñar un algoritmo para saber si un número es primo o no.*

Un número es primo si sólo puede dividirse por sí mismo y por la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo, 9, 8, 6, 4, 12, 16, 20, etc., no son primos, ya que son divisibles por números distintos a ellos mismos y a la unidad. Así, 9 es divisible por 3, 8 lo es por 2, etc. El algoritmo de resolución del problema pasa por dividir sucesivamente el número por 2, 3, 4..., etc.

1. Inicio.
2. Poner X igual a 2 ( $X \leftarrow 2$ , X variable que representa a los divisores del número que se busca N).

3. Dividir N por X ( $N/X$ ).
4. Si el resultado de  $N/X$  es entero, entonces N no es un número primo y bifurcar al punto 7; en caso contrario, continuar el proceso.
5. Suma 1 a X ( $X \leftarrow X + 1$ ).
6. Si X es igual a N, entonces N es un número primo; en caso contrario, bifurcar al punto 3.
7. Fin.

Por ejemplo, si N es 131, los pasos anteriores serían:

1. Inicio.
2.  $X = 2$ .
3.  $131/X$ . Como el resultado no es entero, se continúa el proceso.
5.  $X \leftarrow 2 + 1$ , luego  $X = 3$ .
6. Como X no es 131, se continúa el proceso.
3.  $131/X$  resultado no es entero.
5.  $X \leftarrow 3 + 1$ ,  $X = 4$ .
6. Como X no es 131 se continúa el proceso.
3.  $131/X \dots$ , etc.
7. Fin.

## EJEMPLO 2.6

Realizar la suma de todos los números pares entre 2 y 1.000.

El problema consiste en sumar  $2 + 4 + 6 + 8 \dots + 1.000$ . Utilizaremos las palabras SUMA y NÚMERO (*variables*, serán denominadas más tarde) para representar las sumas sucesivas  $(2+4)$ ,  $(2+4+6)$ ,  $(2+4+6+8)$ , etc. La solución se puede escribir con el siguiente algoritmo:

1. Inicio.
2. Establecer SUMA a 0.
3. Establecer NÚMERO a 2.
4. Sumar NÚMERO a SUMA. El resultado será el nuevo valor de la suma (SUMA).
5. Incrementar NÚMERO en 2 unidades.
6. Si NÚMERO  $\leq 1.000$  bifurcar al paso 4; en caso contrario, escribir el último valor de SUMA y terminar el proceso.
7. Fin.

## 2.5.2. Diseño del algoritmo

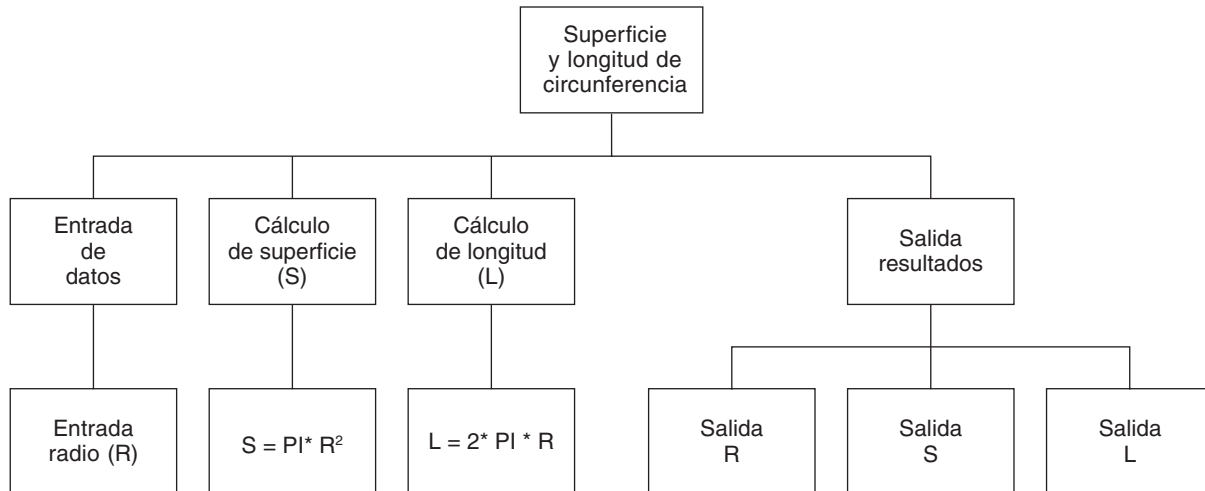
Una computadora no tiene capacidad para solucionar problemas más que cuando se le proporcionan los sucesivos pasos a realizar. Estos pasos sucesivos que indican las instrucciones a ejecutar por la máquina constituyen, como ya conocemos, el *algoritmo*.

La información proporcionada al algoritmo constituye su *entrada* y la información producida por el algoritmo constituye su *salida*.

Los problemas complejos se pueden resolver más eficazmente con la computadora cuando se rompen en subproblemas que sean más fáciles de solucionar que el original. Es el método de *divide y vencerás* (*divide and conquer*), mencionado anteriormente, y que consiste en dividir un problema complejo en otros más simples. Así, el problema de encontrar la superficie y la longitud de un círculo se puede dividir en tres problemas más simples o *subproblemas* (Figura 2.15).

La descomposición del problema original en subproblemas más simples y a continuación la división de estos subproblemas en otros más simples que pueden ser implementados para su solución en la computadora se denomina *diseño descendente* (*top-down design*). Normalmente, los pasos diseñados en el primer esbozo del algoritmo son incompletos e indicarán sólo unos pocos pasos (un máximo de doce aproximadamente). Tras esta primera descripción, éstos se amplían en una descripción más detallada con más pasos específicos. Este proceso se denomina *refinamiento*.





**Figura 2.15.** Refinamiento de un algoritmo.

to del algoritmo (*stepwise refinement*). Para problemas complejos se necesitan con frecuencia diferentes *niveles de refinamiento* antes de que se pueda obtener un algoritmo claro, preciso y completo.

El problema de cálculo de la circunferencia y superficie de un círculo se puede descomponer en subproblemas más simples: 1) leer datos de entrada; 2) calcular superficie y longitud de circunferencia, y 3) escribir resultados (datos de salida).

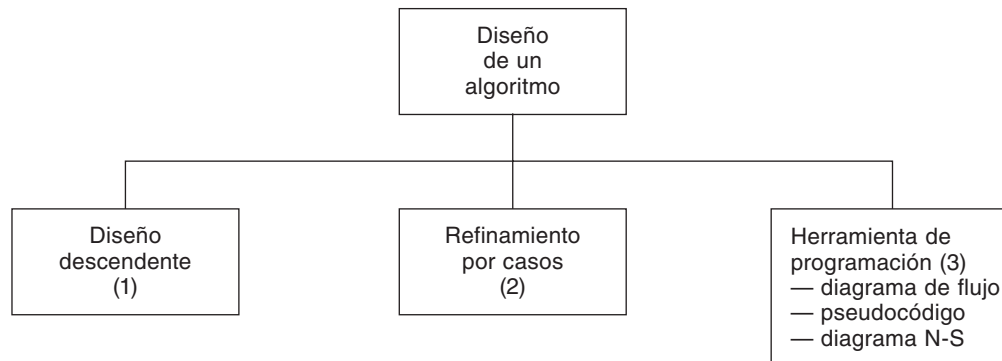
Subproblema	Refinamiento
leer radio	leer radio
calcular superficie	superficie = 3.141592 * radio ^ 2
calcular circunferencia	circunferencia = 2 * 3.141592 * radio
escribir resultados	escribir radio, circunferencia, superficie

Las *ventajas* más importantes del diseño descendente son:

- El problema se comprende más fácilmente al dividirse en partes más simples denominadas *módulos*.
- Las modificaciones en los módulos son más fáciles.
- La comprobación del problema se puede verificar fácilmente.

Tras los pasos anteriores (*diseño descendente* y *refinamiento por pasos*) es preciso representar el algoritmo mediante una determinada herramienta de programación: *diagrama de flujo*, *pseudocódigo* o *diagrama N-S*.

Así pues, el diseño del algoritmo se descompone en las fases recogidas en la Figura 2.16.



**Figura 2.16.** Fases del diseño de un algoritmo.

## 2.6. ESCRITURA DE ALGORITMOS

Como ya se ha comentado anteriormente, el sistema para describir (“escribir”) un algoritmo consiste en realizar una descripción paso a paso con un lenguaje natural del citado algoritmo. Recordemos que un algoritmo es un método o conjunto de reglas para solucionar un problema. En cálculos elementales estas reglas tienen las siguientes propiedades:

- deben ir seguidas de alguna secuencia definida de pasos hasta que se obtenga un resultado coherente,
- sólo puede ejecutarse una operación a la vez.

El flujo de control usual de un algoritmo es secuencial; consideremos el algoritmo que responde a la pregunta:

*¿Qué hacer para ver la película de Harry Potter?*

La respuesta es muy sencilla y puede ser descrita en forma de algoritmo general de modo similar a:

```
ir al cine
comprar una entrada (billete o ticket)
ver la película
regresar a casa
```

El algoritmo consta de cuatro acciones básicas, cada una de las cuales debe ser ejecutada antes de realizar la siguiente. En términos de computadora, cada acción se codificará en una o varias sentencias que ejecutan una tarea particular.

El algoritmo descrito es muy sencillo; sin embargo, como ya se ha indicado en párrafos anteriores, el algoritmo general se descompondrá en pasos más simples en un procedimiento denominado *refinamiento sucesivo*, ya que cada acción puede descomponerse a su vez en otras acciones simples. Así, por ejemplo, un primer refinamiento del algoritmo ir al cine se puede describir de la forma siguiente:

```
1. inicio
2. ver la cartelera de cines en el periódico
3. si no proyectan "Harry Potter" entonces
    3.1. decidir otra actividad
    3.2. bifurcar al paso 7
    si_no
    3.3. ir al cine
    fin_si
4. si hay cola entonces
    4.1. ponerse en ella
    4.2. mientras haya personas delante hacer
        4.2.1. avanzar en la cola
    fin_mientras
    fin_si
5. si hay localidades entonces
    5.1. comprar una entrada
    5.2. pasar a la sala
    5.3. localizar la(s) butaca(s)
    5.4. mientras proyectan la película hacer
        5.4.1. ver la película
    fin_mientras
    5.5. abandonar el cine
    si_no
    5.6. refunfuñar
    fin_si
6. volver a casa
7. fin
```

En el algoritmo anterior existen diferentes aspectos a considerar. En primer lugar, ciertas palabras reservadas se han escrito deliberadamente en negrita (**mientras**, **si\_no**; etc.). Estas palabras describen las estructuras de control fundamentales y procesos de toma de decisión en el algoritmo. Éstas incluyen los conceptos importantes de *selección* (expresadas por **si-entonces-si\_no**, **if-then-else**) y de *repetición* (expresadas con **mientras-hacer** o a veces **repetir-hasta** e **iterar-fin\_iterar**, en inglés, **while-do** y **repeat-until**) que se encuentran en casi todos los algoritmos, especialmente en los de proceso de datos. La capacidad de decisión permite seleccionar alternativas de acciones a seguir o bien la repetición una y otra vez de operaciones básicas.

```

si proyectan la película seleccionada ir al cine
    si_no ver la televisión, ir al fútbol o leer el periódico
mientras haya personas en la cola, ir avanzando repetidamente
    hasta llegar a la taquilla

```

Otro aspecto a considerar es el método elegido para describir los algoritmos: empleo de *indentación* (sangrado o justificación) en escritura de algoritmos. En la actualidad es tan importante la escritura de programa como su posterior lectura. Ello se facilita con la *indentación* de las acciones interiores a las estructuras fundamentales citadas: selectivas y repetitivas. A lo largo de todo el libro la indentación o sangrado de los algoritmos será norma constante.

Para terminar estas consideraciones iniciales sobre algoritmos, describiremos las acciones necesarias para refinar el algoritmo objeto de nuestro estudio; para ello analicemos la acción:

```

Localizar la(s) butaca(s) .

```

Si los números de los asientos están impresos en la entrada, la acción compuesta se resuelve con el siguiente algoritmo:

```

1. inicio //algoritmo para encontrar la butaca del espectador
2. caminar hasta llegar a la primera fila de butacas
3. repetir
    compara número de fila con número impreso en billete
    si son iguales entonces pasar a la siguiente fila fin_si
    hasta_que se localice la fila correcta
4. mientras número de butaca no coincida con número de billete
    hacer avanzar a través de la fila a la siguiente butaca
    fin_mientras
5. sentarse en la butaca
6. fin

```

En este algoritmo la repetición se ha mostrado de dos modos, utilizando ambas notaciones, **repetir... hasta\_que** y **mientras... fin\_mientras**. Se ha considerado también, como ocurre normalmente, que el número del asiento y fila coincide con el número y fila rotulado en el billete.

## 2.7. REPRESENTACIÓN GRÁFICA DE LOS ALGORITMOS

Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Ello permitirá que un algoritmo pueda ser codificado indistintamente en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado gráfica o numéricamente, de modo que las sucesivas acciones no dependan de la sintaxis de ningún lenguaje de programación, sino que la descripción pueda servir fácilmente para su transformación en un programa, es decir, *su codificación*.

Los métodos usuales para representar un algoritmo son:

1. *diagrama de flujo*,
2. *diagrama N-S* (Nassi-Schneiderman),
3. *lenguaje de especificación de algoritmos: pseudocódigo*,
4. *lenguaje español, inglés...*
5. *fórmulas*.

Los métodos 4 y 5 no suelen ser fáciles de transformar en programas. Una descripción en *español narrativo* no es satisfactoria, ya que es demasiado prolija y generalmente ambigua. Una *fórmula*, sin embargo, es un buen sistema de representación. Por ejemplo, las fórmulas para la solución de una ecuación cuadrática (de segundo grado) son un medio sucinto de expresar el procedimiento algorítmico que se debe ejecutar para obtener las raíces de dicha ecuación.

$$x_1 = (-b + \sqrt{b^2 - 4ac})/2a \quad x_2 = (-b - \sqrt{b^2 - 4ac})/2a$$

y significa lo siguiente:

1. *Elevar al cuadrado b.*
2. *Tomar a; multiplicar por c; multiplicar por 4.*
3. *Restar el resultado obtenido de 2 del resultado de 1, etc.*

Sin embargo, no es frecuente que un algoritmo pueda ser expresado por medio de una simple fórmula.

### 2.7.1. Pseudocódigo

El pseudocódigo es *un lenguaje de especificación (descripción) de algoritmos*. El uso de tal lenguaje hace el paso de codificación final (esto es, la traducción a un lenguaje de programación) relativamente fácil. Los lenguajes APL, Pascal y Ada se utilizan a veces como lenguajes de especificación de algoritmos.

El pseudocódigo nació como un lenguaje similar al inglés y era un medio de representar básicamente las estructuras de control de programación estructurada que se verán en capítulos posteriores. Se considera un *primer borrador*, dado que el pseudocódigo tiene que traducirse posteriormente a un lenguaje de programación. El pseudocódigo no puede ser ejecutado por una computadora. La *ventaja del pseudocódigo* es que en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica, una vez que está codificado en un lenguaje de programación. Otra ventaja del pseudocódigo es que puede ser traducido fácilmente a lenguajes estructurados como Pascal, C, FORTRAN 77/90, C++, Java, C#, etc.

El pseudocódigo original utiliza para representar las acciones sucesivas palabras reservadas en inglés —similares a sus homónimas en los lenguajes de programación—, tales como **start**, **end**, **stop**, **if-then-else**, **while-end**, **repeat-until**, etc. La escritura de pseudocódigo exige normalmente la *indentación* (sangría en el margen izquierdo) de diferentes líneas.

Una representación en pseudocódigo —en inglés— de un problema de cálculo del salario neto de un trabajador es la siguiente:

```
start
  //cálculo de impuesto y salarios
  read nombre, horas, precio
  salario ← horas * precio
  tasas ← 0,25 * salario
  salario_netto ← salario - tasas
  write nombre, salario, tasas, salario
end
```

El algoritmo comienza con la palabra **start** y finaliza con la palabra **end**, en inglés (en español, **inicio**, **fin**). Entre estas palabras, sólo se escribe una instrucción o acción por línea.

La línea precedida por // se denomina *comentario*. Es una información al lector del programa y no realiza ninguna instrucción ejecutable, sólo tiene efecto de documentación interna del programa. Algunos autores suelen utilizar corchetes o llaves.

No es recomendable el uso de apóstrofes o simples comillas como representan en algunos lenguajes primitivos los comentarios, ya que este carácter es representativo de apertura o cierre de cadenas de caracteres en lenguajes como Pascal o FORTRAN, y daría lugar a confusión.

Otro ejemplo aclaratorio en el uso del pseudocódigo podría ser un sencillo algoritmo del arranque matinal de un coche.

```

inicio
  //arranque matinal de un coche
  introducir la llave de contacto
  girar la llave de contacto
  pisar el acelerador
  oír el ruido del motor
  pisar de nuevo el acelerador
  esperar unos instantes a que se caliente el motor
fin

```

Por fortuna, aunque el pseudocódigo nació como un sustituto del lenguaje de programación y, por consiguiente, sus palabras reservadas se conservaron o fueron muy similares a las del idioma inglés, el uso del pseudocódigo se ha extendido en la comunidad hispana con términos en español como **inicio**, **fin**, **parada**, **leer**, **escribir**, **si-entonces-si\_no**, **mientras**, **fin\_mientras**, **repetir**, **hasta\_que**, etc. Sin duda, el uso de la terminología del pseudocódigo en español ha facilitado y facilitará considerablemente el aprendizaje y uso diario de la programación. En esta obra, al igual que en otras nuestras, utilizaremos el pseudocódigo en español y daremos en su momento las estructuras equivalentes en inglés, al objeto de facilitar la traducción del pseudocódigo al lenguaje de programación seleccionado.

Así pues, en los pseudocódigos citados anteriormente deberían ser sustituidas las palabras **start**, **end**, **read**, **write**, por **inicio**, **fin**, **leer**, **escribir**, respectivamente.

<b>inicio</b>	<b>start</b>	<b>leer</b>	<b>read</b>
.			
.			
.			
.			
.			
<b>fin</b>	<b>end</b>	<b>escribir</b>	<b>write</b>

### 2.7.2. Diagramas de flujo

Un **diagrama de flujo** (*flowchart*) es una de las técnicas de representación de algoritmos más antigua y a la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobre todo, desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) estándar mostrados en la Tabla 2.1 y que tiene los pasos de algoritmo escritos en esas cajas unidas por flechas, denominadas *líneas de flujo*, que indican la secuencia en que se debe ejecutar.

La Figura 2.17 es un diagrama de flujo básico. Este diagrama representa la resolución de un programa que deduce el salario neto de un trabajador a partir de la lectura del nombre, horas trabajadas, precio de la hora, y sabiendo que los impuestos aplicados son el 25 por 100 sobre el salario bruto.


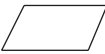

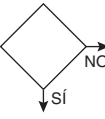
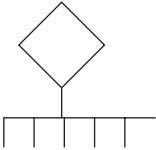








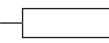
Los símbolos estándar normalizados por **ANSI** (abreviatura de *American National Standards Institute*) son muy variados. En la Figura 2.18 se representa una plantilla de dibujo típica donde se contemplan la mayoría de los símbolos utilizados en el diagrama; sin embargo, los símbolos más utilizados representan:

- **proceso**
- **fin**
- **decisión**
- **entrada/salida**
- **conectores**
- **dirección del flujo**

El diagrama de flujo de la Figura 2.17 resume sus características:

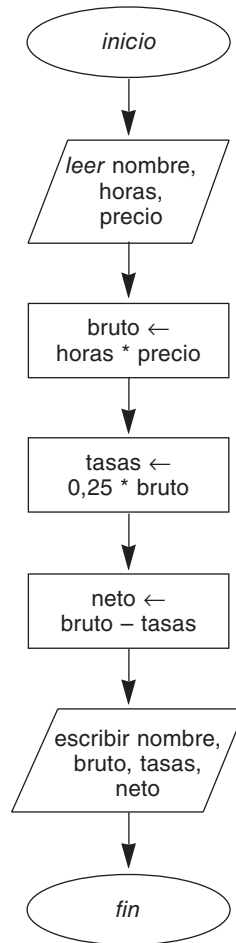
- existe una caja etiquetada “inicio”, que es de tipo elíptico,
- existe una caja etiquetada “fin” de igual forma que la anterior,
- si existen otras cajas, normalmente son rectangulares, tipo rombo o paralelogramo (el resto de las figuras se utilizan sólo en diagramas de flujo generales o de detalle y no siempre son imprescindibles).

Tabla 2.1. Símbolos de diagrama de flujo

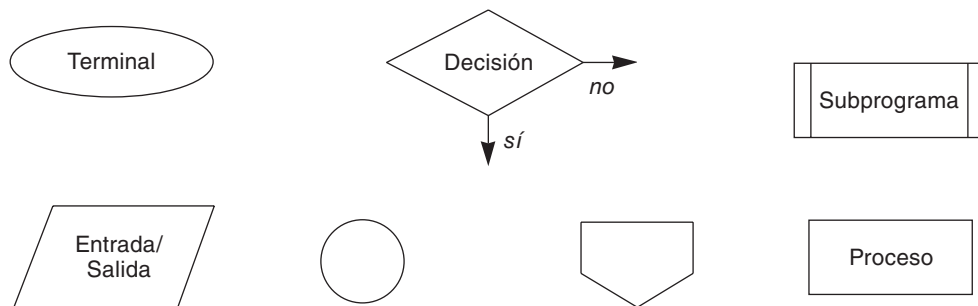
Símbolos principales	Función
	Terminal (representa el comienzo, “inicio”, y el final, “fin” de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa).
	Entrada/Salida (cualquier tipo de introducción de datos en la memoria desde los periféricos, “entrada”, o registro de la información procesada en un periférico, “salida”).
	Proceso (cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etc.).
	Decisión (indica operaciones lógicas o de comparación entre datos —normalmente dos— y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas —respuestas SÍ o NO— pero puede tener tres o más, según los casos).
	Decisión múltiple (en función del resultado de la comparación se seguirá uno de los diferentes caminos de acuerdo con dicho resultado).
	Conector (sirve para enlazar dos partes cualesquiera de un ordinograma a través de un conector en la salida y otro conector en la entrada. Se refiere a la conexión en la misma página del diagrama).
	Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).
	Línea conectora (sirve de unión entre dos símbolos).
	Conector (conexión entre dos puntos del organigrama situado en páginas diferentes).
	Llamada a subrutina o a un proceso predeterminado (una subrutina es un módulo independientemente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).
	Pantalla (se utiliza en ocasiones en lugar del símbolo de E/S).
	Impresora (se utiliza en ocasiones en lugar del símbolo de E/S).
	Teclado (se utiliza en ocasiones en lugar del símbolo de E/S).
	Comentarios (se utiliza para añadir comentarios clasificadores a otros símbolos del diagrama de flujo. Se pueden dibujar a cualquier lado del símbolo).

**Problema:**

Calcular el salario bruto y el salario neto de un trabajador "por horas" conociendo el nombre, número de horas trabajadas, impuestos a pagar y salario neto.



**Figura 2.17.** Diagrama de flujo.



**Figura 2.18.** Plantilla típica para diagramas de flujo.

Se puede escribir más de un paso del algoritmo en una sola caja rectangular. El uso de flechas significa que la caja no necesita ser escrita debajo de su predecesora. Sin embargo, abusar demasiado de esta flexibilidad conduce a diagramas de flujo complicados e ininteligibles.

**EJEMPLO 2.7**

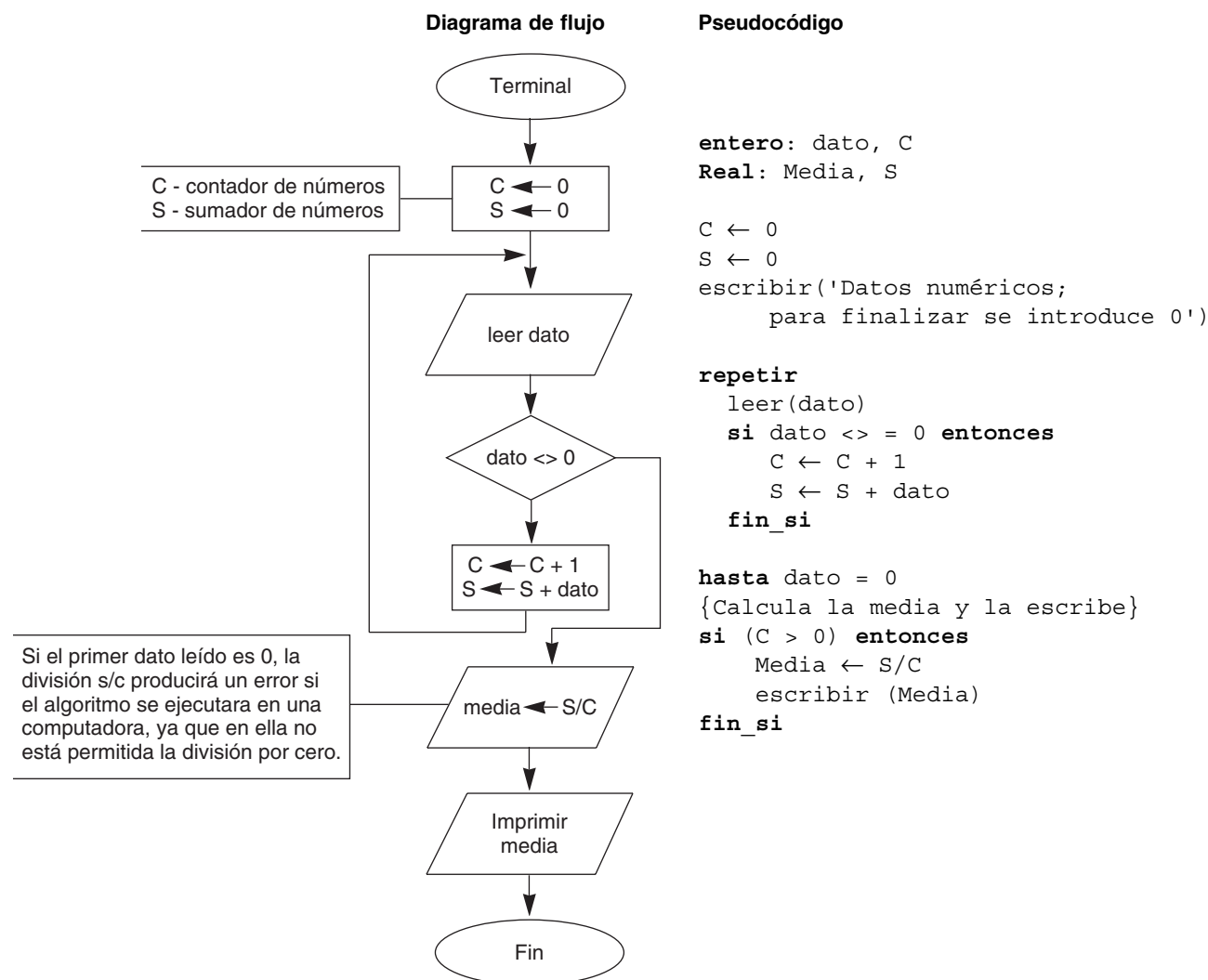
Calcular la media de una serie de números positivos, suponiendo que los datos se leen desde un terminal. Un valor de cero —como entrada— indicará que se ha alcanzado el final de la serie de números positivos.

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, nuestro algoritmo en forma descriptiva sería:

1. Inicializar contador de números C y variable suma S.
2. Leer un número.
3. Si el número leído es cero:
  - calcular la media;
  - imprimir la media;
  - fin del proceso.
 Si el número leído no es cero:
  - calcular la suma;
  - incrementar en uno el contador de números;
  - ir al paso 2.
4. Fin.

El refinamiento del algoritmo conduce a los pasos sucesivos necesarios para realizar las operaciones de lectura, verificación del último dato, suma y media de los datos.

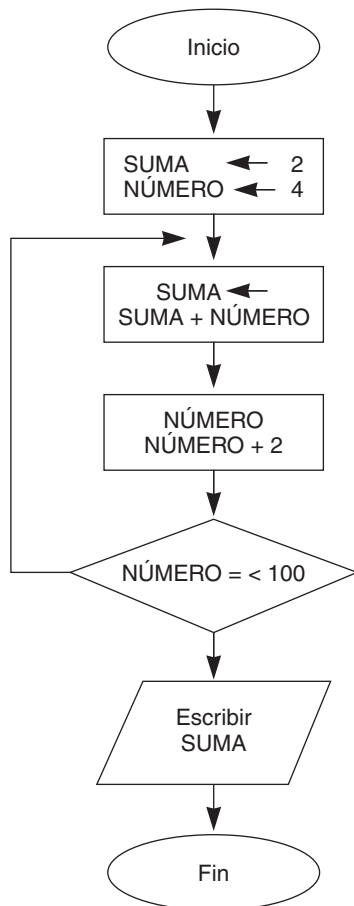
Si el primer dato leído es 0, la división  $S/C$  produciría un error si el algoritmo se ejecutara en una computadora, ya que en ella no está permitida la división por cero.





**EJEMPLO 2.8**

Suma de los números pares comprendidos entre 2 y 100.

**Diagrama de flujo****Pseudocódigo**

```

entero:    numero, Suma
Suma ← 2
numero ← 4
mientras (numero <= 100) hacer
    suma ← suma + numero
    numero ← numero + 2
fin mientras
escribe ('Suma pares entre 2 y 100 =', suma)
  
```

**EJEMPLO 2.9**

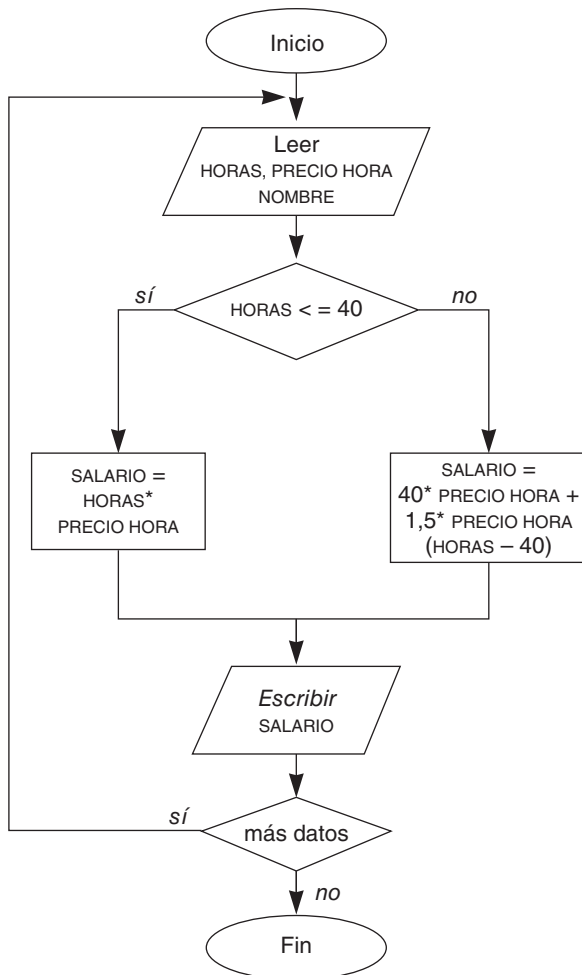
Se desea realizar el algoritmo que resuelva el siguiente problema: Cálculo de los salarios mensuales de los empleados de una empresa, sabiendo que éstos se calculan en base a las horas semanales trabajadas y de acuerdo a un precio especificado por horas. Si se pasan de cuarenta horas semanales, las horas extraordinarias se pagarán a razón de 1,5 veces la hora ordinaria.

Los cálculos son:

1. Leer datos del archivo de la empresa, hasta que se encuentre la ficha final del archivo (HORAS, PRECIO\_HORA, NOMBRE).
2. Si HORAS ≤ 40, entonces SALARIO es el producto de horas por PRECIO\_HORA.
3. Si HORAS > 40, entonces SALARIO es la suma de 40 veces PRECIO\_HORA más 1.5 veces PRECIO\_HORA por (HORAS-40).

El diagrama de flujo completo del algoritmo y la codificación en pseudocódigo se indican a continuación:

### Diagrama de flujo



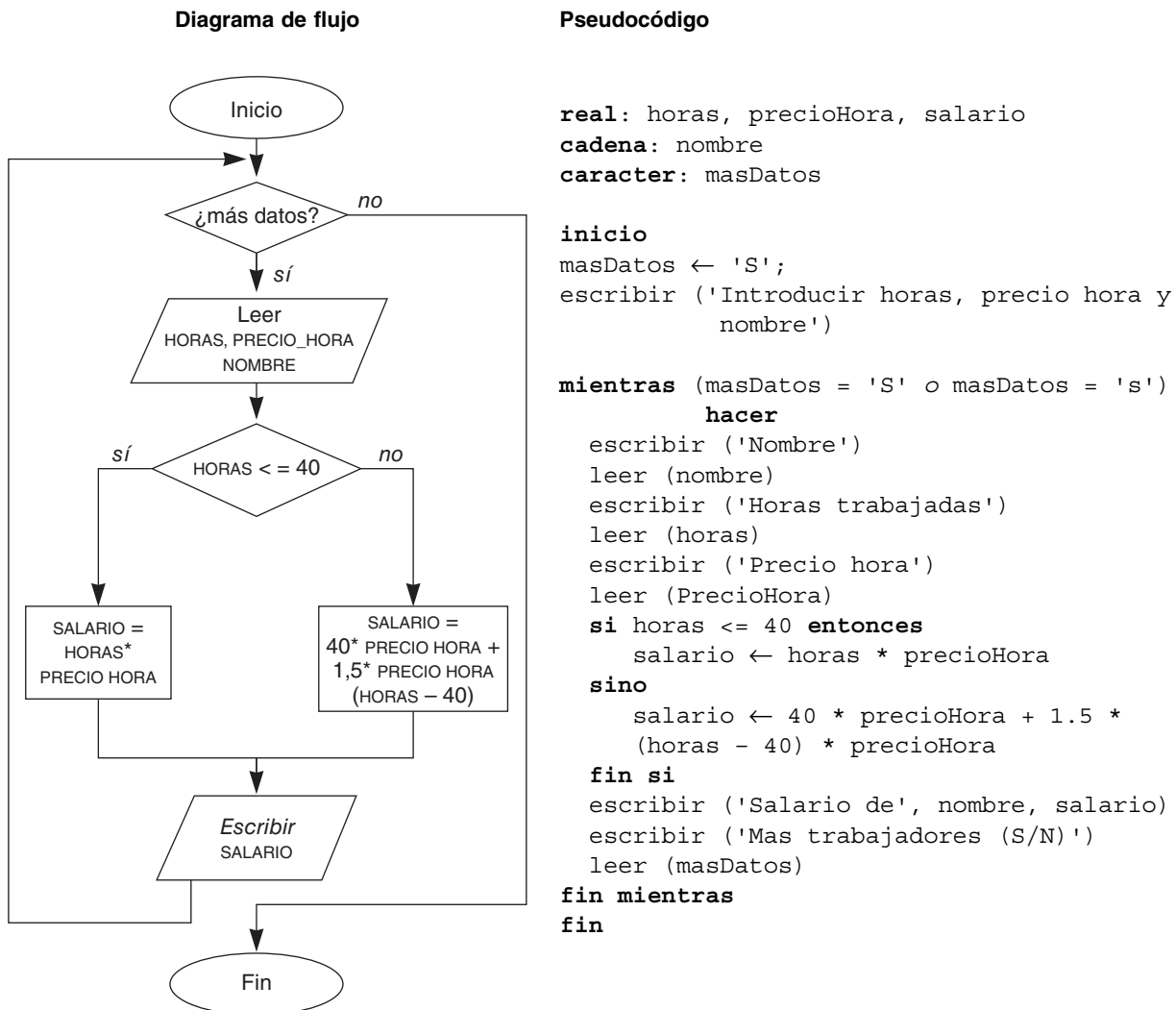
### Pseudocódigo

```

real: horas, precioHora, salario
cadena: nombre
caracter: masDatos

inicio
  escribir('Introducir horas, precio hora
    y nombre')
repetir
  escribir ('Nombre')
  leer (Nombre)
  escribir ('Horas trabajadas')
  leer (horas)
  escribir ('Precio hora')
  leer (precio Hora)
  si (horas <= 40) entonces
    Salario ← horas * precioHora
  sino
    Salario ← 40 * precioHora +
      1.5 * (horas - 40) *
      preciohora
  fin si
  escribir ('Salario de', nombre, salario)
  escribir ('Mas trabajadores S/N')
  leer (masDatos)
hasta masDatos = 'N'
fin
  
```

Una variante también válida del diagrama de flujo anterior es:



### EJEMPLO 2.10

La escritura de algoritmos para realizar operaciones sencillas de conteo es una de las primeras cosas que una computadora puede aprender.

Supongamos que se proporciona una secuencia de números, tales como

5 3 0 2 4 4 0 0 2 3 6 0 2

y desea contar e imprimir el número de ceros de la secuencia.

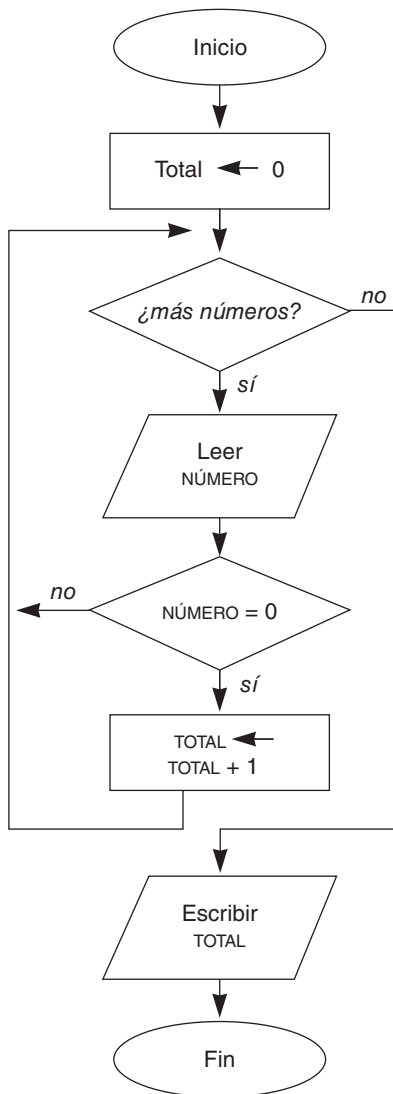
El algoritmo es muy sencillo, ya que sólo basta leer los números de izquierda a derecha, mientras se cuentan los ceros. Utiliza como variable la palabra **NUMERO** para los números que se examinan y **TOTAL** para el número de ceros encontrados. Los pasos a seguir son:

1. Establecer **TOTAL** a cero.
2. ¿Quedan más números a examinar?
3. Si no quedan números, imprimir el valor de **TOTAL** y fin.

4. Si existen mas numeros, ejecutar los pasos 5 a 8.
5. Leer el siguiente numero y dar su valor a la variable NUMERO.
6. Si NUMERO = 0, incrementar TOTAL en 1.
7. Si NUMERO <> 0, no modificar TOTAL.
8. Retornar al paso 2.

El diagrama de flujo y la codificación en pseudocódigo correspondiente es:

#### Diagrama de flujo



#### Pseudocódigo

```

entero: numero, total
caracter: mas Datos;

inicio
  escribir ('Cuenta de ceros leidos del teclado')
  mas Datos ← 'S';
  total ← 0

  mientras (mas Datos = 'S') o (mas Datos = 's') hacer
    leer (numero)
    si (numero = 0)
      total ← total + 1
    fin si
    escribir ('Mas números 'S/N')
    leer (mas Datos)
  fin mientras
  escribir ('total de ceros =', total)
fin

```

**EJEMPLO 2.11**

Dados tres números, determinar si la suma de cualquier pareja de ellos es igual al tercer número. Si se cumple esta condición, escribir "Iguales" y, en caso contrario, escribir "Distintas".

En el caso de que los números sean: 3 9 6

la respuesta es "Iguales", ya que  $3 + 6 = 9$ . Sin embargo, si los números fueran:

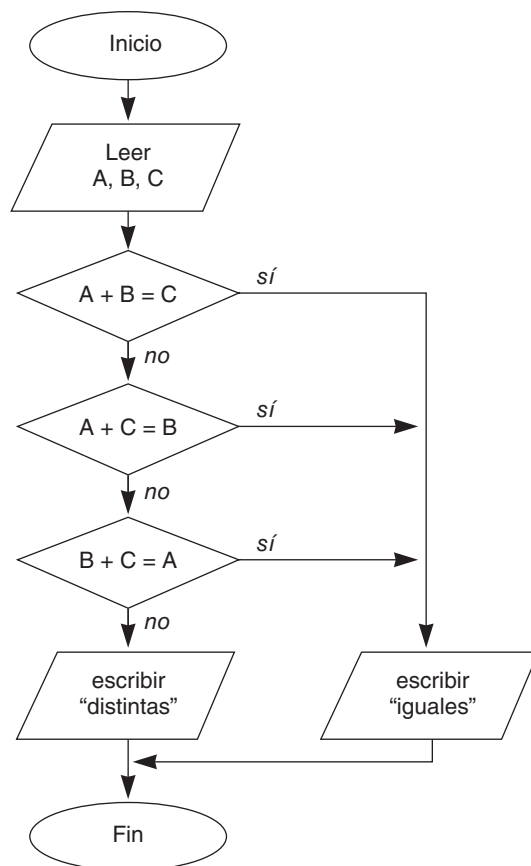
2 3 4

el resultado sería "Distintas".

Para resolver este problema, se puede comparar la suma de cada pareja con el tercer número. Con tres números solamente existen tres parejas distintas y el algoritmo de resolución del problema será fácil.

1. Leer los tres valores, A, B y C.
2. Si  $A + B = C$  escribir "Iguales" y parar.
3. Si  $A + C = B$  escribir "Iguales" y parar.
4. Si  $B + C = A$  escribir "Iguales" y parar.
5. Escribir "Distintas" y parar.

El diagrama de flujo y la codificación en pseudocódigo correspondiente es la Figura 2.19.

**Diagrama de flujo****Pseudocódigo**

**entero:** a, b, c

**inicio**

escribir ('test con tres números:')

leer (a, b, c)

**si** (a + b = c) **entonces**

    escribir ('Son iguales', a, '+', b, '=', c)

**sino si** (a + c = b) **entonces**

        escribir ('Son iguales', a, '+', c, '=', b)

**sino si** (b + c = a) **entonces**

        escribir ('Son iguales', b, '+', c, '=', a)

**sino**

        escribir ('Son distintas')

**fin si**

**fin si**

**fin si**

**fin**

**Figura 2.19.** Diagrama de flujo y codificación en pseudocódigo (Ejemplo 2.11).

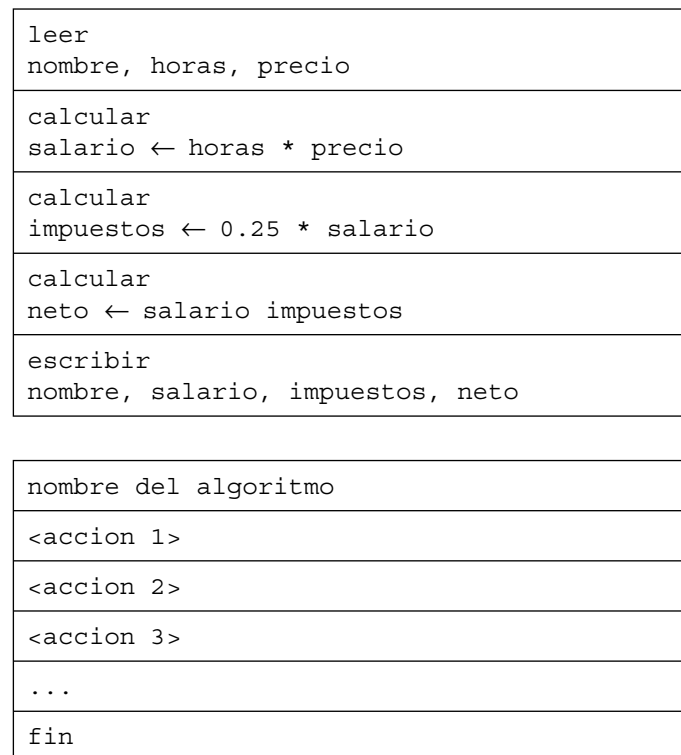
### 2.7.3. Diagramas de Nassi-Schneiderman (N-S)

El diagrama N-S de Nassi Schneiderman —también conocido como diagrama de Chapin— es como un diagrama de flujo en el que se omiten las flechas de unión y las cajas son contiguas. Las acciones sucesivas se escriben en cajas sucesivas y, como en los diagramas de flujo, se pueden escribir diferentes acciones en una caja.

Un algoritmo se representa con un rectángulo en el que cada banda es una acción a realizar.

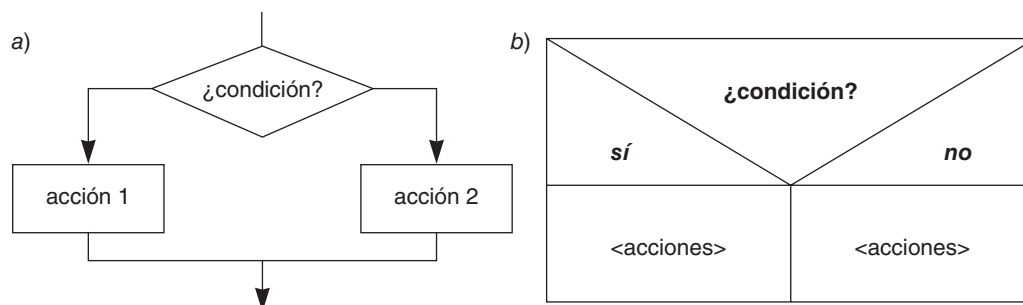
#### EJEMPLO

*Escribir un algoritmo que lea el nombre de un empleado, las horas trabajadas, el precio por hora y calcule los impuestos a pagar (tasa = 25%) y el salario neto.*



**Figura 2.20.** Representación gráfica N-S de un algoritmo.

Otro ejemplo es la representación de la estructura condicional (Figura 2.21).



**Figura 2.21.** Estructura condicional o selectiva: a) diagrama de flujo: b) diagrama N-S.

**EJEMPLO 2.12**

Se desea calcular el salario neto semanal de un trabajador (en dólares o en euros) en función del número de horas trabajadas y la tasa de impuestos:

- las primeras 35 horas se pagan a tarifa normal,
- las horas que pasen de 35 se pagan a 1,5 veces la tarifa normal,
- las tasas de impuestos son:
  - a) los primeros 1.000 dólares son libres de impuestos,
  - b) los siguientes 400 dólares tienen un 25 por 100 de impuestos,
  - c) los restantes, un 45 por 100 de impuestos,
- la tarifa horaria es 15 dólares.

También se desea escribir el nombre, salario bruto, tasas y salario neto (*este ejemplo se deja como ejercicio para el alumno*).

**RESUMEN**

Un método general para la resolución de un problema con computadora tiene las siguientes fases:

1. *Análisis del programa.*
2. *Diseño del algoritmo.*
3. *Codificación.*
4. *Compilación y ejecución.*
5. *Verificación.*
6. *Documentación y mantenimiento.*

El sistema más idóneo para resolver un problema es descomponerlo en módulos más sencillos y luego, median-

te diseños descendentes y refinamiento sucesivo, llegar a módulos fácilmente codificables. Estos módulos se deben codificar con las estructuras de control de programación estructurada.

1. *Secuenciales*: las instrucciones se ejecutan sucesivamente una después de otra.
2. *Repetitivas*: una serie de instrucciones se repiten una y otra vez hasta que se cumple una cierta condición.
3. *Selectivas*: permite elegir entre dos alternativas (dos conjuntos de instrucciones) dependiendo de una condición determinada).

**EJERCICIOS**

**2.1.** Diseñar una solución para resolver cada uno de los siguientes problemas y tratar de refinar sus soluciones mediante algoritmos adecuados:

- a) Realizar una llamada telefónica desde un teléfono público.
- b) Cocinar una tortilla.
- c) Arreglar un pinchazo de una bicicleta.
- d) Freír un huevo.

**2.2.** Escribir un algoritmo para:

- a) Sumar dos números enteros.
- b) Restar dos números enteros.
- c) Multiplicar dos números enteros.
- d) Dividir un número entero por otro.

**2.3.** Escribir un algoritmo para determinar el máximo común divisor de dos números enteros (MCD) por el algoritmo de Euclides:

- Dividir el mayor de los dos enteros positivos por el más pequeño.
- A continuación dividir el divisor por el resto.
- Continuar el proceso de dividir el último divisor por el último resto hasta que la división sea exacta.
- El último divisor es el mcd.

**2.4.** Diseñar un algoritmo que lea y visualice una serie de números distintos de cero. El algoritmo debe terminar con un valor cero que no se debe visualizar. Visualizar el número de valores leídos.

- 2.5.** Diseñar un algoritmo que visualice y sume la serie de números 3, 6, 9, 12..., 99.
- 2.6.** Escribir un algoritmo que lea cuatro números y a continuación visualice el mayor de los cuatro.
- 2.7.** Diseñar un algoritmo que lea tres números y descubra si uno de ellos es la suma de los otros dos.
- 2.8.** Diseñar un algoritmo para calcular la velocidad (en m/s) de los corredores de la carrera de 1.500 metros. La entrada consistirá en parejas de números (minutos, segundos) que dan el tiempo del corredor; por cada corredor, el algoritmo debe visualizar el tiempo en minutos y segundos, así como la velocidad media.
- Ejemplo de entrada de datos: (3,53) (3,40) (3,46) (3,52) (4,0) (0,0); el último par de datos se utilizará como fin de entrada de datos.
- 2.9.** Diseñar un algoritmo para determinar los números primos iguales o menores que  $N$  (leído del teclado). (Un número primo sólo puede ser divisible por él mismo y por la unidad.)
- 2.10.** Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura ( $S = 1/2 \text{ Base} \times \text{Altura}$ ).
- 2.11.** Calcular y visualizar la longitud de la circunferencia y el área de un círculo de radio dado.

- 2.12.** Escribir un algoritmo que encuentre el salario semanal de un trabajador, dada la tarifa horaria y el número de horas trabajadas diariamente.
- 2.13.** Escribir un algoritmo que indique si una palabra leída del teclado es un palíndromo. Un *palíndromo* (capi-cúa) es una palabra que se lee igual en ambos sentidos como “*radar*”.
- 2.14.** Escribir un algoritmo que cuente el número de ocurrencias de cada letra en una palabra leída como entrada. Por ejemplo, “*Mortimer*” contiene dos “m”, una “o”, dos “r”, una “i”, una “t” y una “e”.
- 2.15.** Muchos bancos y cajas de ahorro calculan los intereses de las cantidades depositadas por los clientes diariamente según las premisas siguientes. Un capital de 1.000 euros, con una tasa de interés del 6 por 100, renta un interés en un día de 0,06 multiplicado por 1.000 y dividido por 365. Esta operación producirá 0,16 euros de interés y el capital acumulado será 1.000,16. El interés para el segundo día se calculará multiplicando 0,06 por 1.000 y dividiendo el resultado por 365. Diseñar un algoritmo que reciba tres entradas: el capital a depositar, la tasa de interés y la duración del depósito en semanas, y calcular el capital total acumulado al final del período de tiempo especificado.