CAPÍTULO 1

# Desarrollo de *software*. Tipos abstractos de datos

## Objetivos

Con el estudio de este capítulo usted podra:

- Conocer el concepto de *software* algoritmo y sistema operativo.
- Especificar algoritmos en pseudocódigo.
- Definir los tipos de abstractos de datos.
- Conocer el concepto de objeto.

## Contenido



## Conceptos clave

- Abstracción.
- Algoritmo.
- Clase.
- Depuración.
- Documentación.
- Estructura de datos.
- Herencia.
- Mantenimiento.
- Objetos.
- Polimorfismo.
- Programación estructurada.
- Programación orientada a objetos.
- Sistema operativo.
- *Software*.
- TAD.
- Tipo de dato.

## INTRODUCCIÓN

La principal razón para que las personas aprendan lenguajes y técnicas de programación es utilizar la computadora como una herramienta para resolver problemas. Este capítulo introduce al lector en la metodología a seguir para la resolución de problemas con computadoras y en el diseño de algoritmos examinanando el concepto de *Abstracción de Datos*. La *Abstracción de Datos* es la técnica de inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, facilitar la escritura del programa. La técnica de abstracción de datos es una técnica potente de propósito general que cuando se utiliza adecuadamente, puede producir programas más cortos, más legibles y flexibles.

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como `int`, `char` y `float` en C, C++ y Java. Lenguajes de programación, como C++, tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina *tipo abstracto de dato,* **TAD** *(Abstract Data Type,* **ADT***).* El término abstracto se refiere al medio en que un programador abstrae algunos conceptos de programación creando un nuevo tipo de dato.

La modularización de un programa utiliza la noción de *tipo abstracto de dato* (**TAD**) siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado **TAD**.

Los paradigmas más populares soportados por el lenguaje C++ son: programación estructurada y programación orientada a objetos.

## 1.1. EL *SOFTWARE* (LOS PROGRAMAS)

El *software* de una computadora es un conjunto de instrucciones de programa detalladas que controlan y coordinan los componentes *hardware* de una computadora y controlan las operaciones de un sistema informático. El auge de las computadoras en el siglo pasado y en el actual siglo XXI, se debe esencialmente al desarrollo de sucesivas generaciones de *software* potentes y cada vez más *amistosas* («fáciles de utilizar»)

Las operaciones que debe realizar el *hardware* son especificadas por una lista de instrucciones, llamadas programas, o *software*. Un programa de *software* es un conjunto de **sentencias** o **instrucciones** dadas al computador. El proceso de escritura o codificación de un programa se denomina **programación** y las personas que se especializan es esta actividad se denominan **programadores**. Existen dos tipos importantes de *software*: *software* del sistema y *software* de *aplicaciones*. Cada tipo realiza una función diferente.

***Software* del sistema** es un conjunto generalizado de programas que gestiona los recursos del computador, tal como el procesador central, enlaces de comunicaciones y dispositivos periféricos. Los programadores que escriben *software* del sistema se llaman **programadores de sistemas. *Software* de aplicaciones** el conjunto de programas escritos por empresas o usuarios individuales o en equipo y que instruyen a la computadora para que ejecute una tarea específica. Los programadores que escriben *software* de aplicaciones se llaman **programadores de aplicaciones**.

Los dos tipos de *software* están relacionados entre sí, de modo que los usuarios y los programadores pueden hacer así un uso eficiente del computador. En la Figura 1.1 se muestra una vista organizacional de un computador donde se muestran los diferentes tipos de *software* a modo de capas de la computadora desde su interior (el *hardware*) hacia su exterior (usuario): las dife-


**Figura 1.1.** Relación entre programas de aplicación y programas del sistema.

rentes capas funcionan gracias a las instrucciones específicas (instrucciones máquina) que forman parte del *software* del sistema y llegan al *software* de aplicación, programado por los programadores de aplicaciones, que es utilizado por el usuario y que no requiere ser un especialista.

### 1.1.1. *Software* del sistema

El *software del sistema* coordina las diferentes partes de un sistema de computadora y conecta e interactúa entre el *software* de aplicación y el *hardware* del computador. El *software* del sistema gestiona el *hardware* del computador. Otro tipo de *software* del sistema que gestiona, controla las actividades de la computadora y realizan tareas de proceso comunes, se denomina *utility* o **utilidades** (en algunas partes de Latinoamérica, **utilerías**). El *software* del sistema que gestiona y controla las actividades del computador se denomina sistema operativo. Otro *software* del sistema son los programas traductores o de traducción de lenguajes de computador que convierten los programas escritos en lenguajes de programación, entendibles por los programadores, en lenguaje máquina que entienden las computadoras

El ***software* del sistema** es el conjunto de programas indispensables para que la máquina funcione; se denominan también *programas del sistema*. Estos programas son, básicamente, *el sistema operativo, los editores de texto,* los *compiladores/intérpretes* (lenguajes de programación) y los *programas de utilidad*.

### 1.1.2. *Software* de aplicación

El *software* de aplicación tiene como función principal asistir y ayudar a un usuario de un computador para ejecutar tareas específicas. Los programas de aplicación se pueden desarrollar con diferentes lenguajes y herramientas de *software*. Por ejemplo: una aplicación de procesa-

miento de textos (*word processing)* tal como *Word* de Microsoft o *Writely* de Google que ayuda a crear documentos, una hoja de cálculo tales como *Lotus 1-2-3* o *Excel* que ayudan a automatizar tareas tediosas o repetitivas de cálculos matemáticos o estadísticos, a generar diagramas o gráficos, presentaciones visuales como *PowerPoint*; o a crear bases de datos como *Acces* u *Oracle* que ayudan a crear archivos y registros de datos.

Los usuarios, normalmente, compran el *software* de aplicaciones en discos CDs o DVDs (antiguamente en disquetes) o los descargan (bajan) de la Red Internet y han de instalar el *software* copiando los programas correspondientes de los discos en el disco duro de la computadora. Cuando compre estos programas asegúrese que son compatibles con su computador y con su sistema operativo. Existe una gran diversidad de programas de aplicación para todo tipo de actividades tanto de modo personal, como de negocios, navegación y manipulación en Internet, gráficos y presentaciones visuales, etc.

Los *lenguajes de programación* sirven para escribir programas que permitan la comunicación usuario/máquina. Unos programas especiales llamados *traductores* (**compiladores** o **intérpretes**) convierten las instrucciones escritas en lenguajes de programación en instrucciones escritas en lenguajes máquina (0 y 1, *bits*) que ésta pueda entender.

Los *programas de utilidad*[1] facilitan el uso de la computadora. Un buen ejemplo es un *editor de* textos que permite la escritura y edición de documentos. Este libro ha sido escrito con un editor de textos o *procesador de palabras* ("***word procesor***").

Los programas que realizan tareas concretas, nóminas, contabilidad, análisis estadístico, etc. es decir, los programas que podrá escribir en C++ o Java, se denominan *programas de aplicación*. A lo largo del libro, se verán pequeños programas de aplicación que muestran los principios de una buena programación de una computadora.

### 1.1.3. Sistema operativo

Un sistema operativo **SO** (*Operating System, OS*) es tal vez la parte más importante del *software* del sistema y es el *software* que controla y gestiona los recursos del computador. En la práctica, el sistema operativo es la colección de programas de computador que controla la interacción del usuario y el hardware del computador. El sistema operativo es el administrador principal del computador, y por ello a veces, se le compara con el director de una orquesta ya que este *software* es el responsable de dirigir todas las operaciones del computador y gestionar todos sus recursos.

El sistema operativo asigna recursos, planifica el uso de recursos y tareas del computador, y monitoriza las actividades del sistema informático. Estos recursos incluyen memoria, dispositivos de **E/S** (Entrada/Salida), y la **UCP** (**U**nidad **C**entral de **P**roceso). El sistema operativo proporciona servicios tales como asignar memoria a un programa y manipulación del control de los dispositivos de E/S tales como el monitor el teclado o las unidades de disco. La Tabla 1.1 muestra algunos de los sistemas operativos más populares utilizados en enseñanza y en informática profesional.

Cuando un usuario interactúa con un computador, la interacción está controlada por el sistema operativo. Un usuario se comunica con un sistema operativo a través de una interfaz de usuario de ese sistema operativo. Los sistemas operativos modernos utilizan una interfaz gráfica de usuario, **IGU** (*Graphical User Interface, GUI*) que hace uso masivo de iconos, botones, barras y cuadros de diálogo para realizar tareas que se controlan por el teclado o el ratón (mouse) entre otros dispositivos.

[1] *Utility*: programa de utilidad.

**Tabla 1.1.** Sistemas operativos —actuales y antiguos— utilizados en educación y en la empresa

| Sistema operativo | Características |
|---|---|
| **Windows Vista**[2] | Nuevo sistema operativo de Microsoft presentado a primeros de 2006, pero que se ha lanzado en noviembre de 2006. |
| **Windows XP** | Sistema operativo más utilizado en la actualidad, tanto en el campo de la enseñanza, como en la industria y negocios. Su fabricante es Microsoft. |
| **Windows 98/ME/2000** | Versiones anteriores de Windows pero que todavía hoy son muy utilizados. |
| **UNIX** | Sistema operativo abierto, escrito en C y todavía muy utilizado en el campo profesional. |
| **Linux** | Sistema operativo de *software* abierto, gratuito y de libre distribución, similar a UNIX, y una gran alternativa a Windows. Muy utilizado actualmente en servidores de aplicaciones para Internet. |
| **Mac OS** | Sistema operativo de las computadoras Apple Macintosh. |
| **DOS y OS/2** | Sistemas operativos creados por Microsoft e IBM respectivamente, ya poco utilizados pero que han sido la base de los actuales sistemas operativos. |
| **CP/M** | Sistema operativo de 8 bits para las primeras microcomputadoras nacidas en la década de los setenta. |
| **Symbian** | Sistema operativo para teléfonos móviles apoyado fundamentalmente por el fabricante de teléfonos celulares Nokia. |
| **PalmOS** | Sistema operativo para agendas digitales, PDA, del fabricante Palm. |
| **Windows Mobile, CE** | Sistema operativo para teléfonos móviles (celulares) con arquitectura y apariencias similares a Windows XP; actualmente en su versión 6.0 |

Normalmente, el sistema operativo se almacena de modo permanente en un chip de memoria de sólo lectura (**ROM**) de modo que esté disponible tan pronto el computador se pone en marcha ("*se enciende*" o "*se prende*"). Otra parte del sistema operativo puede residir en disco y se almacena en memoria RAM en la inicialización del sistema por primera vez en una operación que se llama c*arga* del sistema (*booting*).

Uno de los programas más importante es el **sistema operativo**, que sirve, esencialmente, para facilitar la escritura y uso de sus propios programas. El sistema operativo dirige las operaciones globales de la computadora, instruye a la computadora para ejecutar otros programas y controla el almacenamiento y recuperación de archivos (programas y datos) de cintas y discos. Gracias al sistema operativo es posible que el programador pueda introducir y grabar nuevos programas, así como instruir a la computadora para que los ejecute. Los sistemas operativos pueden ser, *monousuarios* (un solo usuario) y *multiusuarios*, o tiempo compartido (diferentes usuarios); atendiendo al número de usuarios y *monocarga* (una sola tarea) o *multitarea* (múltiples tareas) según las tareas (procesos) que puede realizar simul-

[2] Microsoft presentó, a nivel mundial, a finales de noviembre de 2006 y comercializa desde primeros de 2007, un nuevo sistema operativo *Windows Vista*, actualización de Windows XP pero con numerosas funcionalidades, especialmente de Internet y de seguridad, incluyendo en el sistema operativo programas que actualmente se comercializan independientes, tales como programas de reproducción de música, vídeo, y, fundamentalmente, un sistema de representación gráfica muy potente que permitirá construir aplicaciones en tres dimensiones, así como un buscador, un sistema antivirus y otras funcionalidades importantes.

táneamente. C++ corre prácticamente en todos los sistemas operativos, Windows XP, Windows 95, Windows NT, Windows 2000, UNIX, Linux, Vista..., y en casi todas las computadoras personales actuales PC, Mac, Sun, etc.

### 1.1.3.1. *Tipos de sistemas operativos*

Las diferentes características especializadas del sistema operativo permiten a los computadores manejar muchas tareas diferentes así como múltiples usuarios de modo simultáneo o en paralelo o bien de modo secuencial. En base a sus características específicas los sistemas operativos se pueden clasificar en varios grupos.

#### Multiprogramación/Multitarea

La *multiprogramación* permite a múltiples programas compartir recursos de un sistema de computadora en cualquier momento a través del uso concurrente de una UCP. Sólo un programa utiliza realmente la UCP en cualquier momento dado, sin embargo, las necesidades de entrada/salida pueden ser atendidas en el mismo momento. Dos o más programas están activos al mismo tiempo, pero no utilizan los recursos del computador simultáneamente. Con multiprogramación, un grupo de programas se ejecutan alternativamente y se alternan en el uso del procesador. Cuando se utiliza un sistema operativo de un único usuario, la multiprogramación toma el nombre de **multitarea**.

**Multiprogramación**

Método de ejecución de dos o más programas concurrentemente utilizando la misma computadora. La UCP ejecuta sólo un programa pero puede atender los servicios de entrada/salida de los otros al mismo tiempo.

#### Tiempo compartido (múltiples usuarios, *time sharing*)

Un *sistema operativo multiusuario* es un sistema operativo que tiene la capacidad de permitir que muchos usuarios compartan simultáneamente los recursos de proceso de la computadora. Centenas o millares de usuarios se pueden conectar al computador que asigna un tiempo de computador a cada usuario, de modo que a medida que se libera la tarea de un usuario, se realiza la tarea del siguiente, y así sucesivamente. Dada la alta velocidad de transferencia de las operaciones, la sensación es de que todos los usuarios están conectados simultáneamente a la UCP, con cada usuario recibiendo únicamente en un tiempo de máquina determinado.

#### Multiproceso

Un sistema operativo trabaja en multiproceso cuando puede enlazar a dos o más UCP para trabajar en paralelo en un único sistema de computadora. El sistema operativo puede asignar múltiples UCP para ejecutar diferentes instrucciones del mismo programa o de programas diferentes simultáneamente, dividiendo el trabajo entre las diferentes UCP.

La multiprogramación utiliza proceso concurrente con una UCP; el multiproceso utiliza proceso simultáneo con múltiples UCP.

## 1.2. RESOLUCIÓN DE PROBLEMAS Y DESARROLLO DE *SOFTWARE*

En este apartado se estudiará el proceso de desarrollo de *software* y sus fases. Estas fases ocurren en todo el *software* incluyendo los programas pequeños que se suelen utilizar en la etapa de formación de un estudiante o profesional. En los capítulos siguientes se aplicarán las fases de desarrollo de *software* a colecciones organizadas de datos. Estas colecciones organizadas de datos se denominan *estructuras de datos* y la representación y manipulación de tales estructuras de datos constituyen la esencia fundamental de este libro.

La creación de un programa requiere de técnicas similares a la realización de otros proyecto de ciencia e ingeniería, ya que, en la práctica, un programa no es más que una solución desarrollada para resolver un problema concreto. La escritura de un programa es casi la última etapa de un proceso en el que se determina primero ¿cuál es el problema? y el método que se utilizará para resolver el problema. Cada campo de estudio tiene su propio nombre para denominar el método sistemático utilizado para resolver problemas mediante el diseño de soluciones adecuadas. En ciencia y en ingeniería el método se conoce como método científico, mientras que cuando se realiza análisis cuantitativo se suele conocer como enfoque o *método sistemático*.

Las técnicas utilizadas por los desarrolladores profesionales de *software* para llegar a soluciones adecuadas para la resolución de problemas se denomina *proceso de desarrollo de software*. Aunque el número y nombre de las fases puede variar según los modelos, métodos y técnicas utilizadas.

En general, las fases de desarrollo de *software* se suelen considerar las siguientes:

- Análisis y especificación del problema.
- Diseño de una solución.
- Implementación (codificación).
- Pruebas, ejecución, corrección y depuración.
- Documentación.
- Mantenimiento y evaluación.

### 1.2.1. Modelos de proceso de desarrollo de *software*

A lo largo de la historia del desarrollo de *software* desde la década de los cincuenta del siglo pasado se han propuesto muchos modelos. Pressman en la última edición de su conocida obra *Ingeniería del software*, plantea la siguiente división [PRESSMAN 05][3]:

1. Modelos normativos (prescriptivos).
2. Modelos en cascada.
3. Modelos de proceso incremental
   - Modelo incremental.
   - Modelo **DRA** (**D**esarrollo **R**ápido de **A**plicaciones).
4. Modelos de proceso evolutivo
   - Modelo de prototipado (construcción de prototipos).
   - Modelo en espiral.
   - Modelo de desarrollo concurrente.

[3] [PRESSMAN 05] Roger Pressman. *Ingeniería del software. Un enfoque práctico*. México DF: McGraw-Hill, 2005, pp 48-101. Esta obra es una de las mejores referencias para el estudio de ingeniería de *software*.

5. Modelos especializados de proceso.
   - Modelo basado en componentes.
   - Modelo de métodos formales.
   - Desarrollo de *software* orientado a aspectos.
6. El proceso unificado (**RUP** de Booch, Rumbaugh y Jacobson)
7. Métodos Ágiles [ Beck 01]
   - Programación Extrema (**XP**, *Extreme Programming*).

---

En la bibliografía recomendada y en la página *web* oficial del libro puede encontrar amplias referencia para el caso de que se desee estudiar y profundizar en ingeniería de *software*. En este capítulo y siguientes nos centraremos en las fases comunes de desarrollo de *software* como elementos centrales para la resolución de problemas.

### 1.2.2. Análisis y especificación del problema

El análisis de un problema se requiere para asegurarse de que el problema está bien definido y comprendido con claridad. La determinación de que el problema está definido claramente se hace después de que la persona entienda cuáles son las salidas requeridas y qué entradas son necesarias. Para realizar esta tarea, el análista debe tener una comprensión de cómo se pueden utilizar las entradas para producir las salidas deseadas.

Después de un análisis profundo del problema se debe realizar la **especificación** que es una descripción precisa y lo más exacta posible del problema; en realidad, es como un *contrato previo* para solución.

La especificación del problema no suele ser una tarea fácil sobre todo por la complejidad que entrañan la mayoría de los problemas del mundo real. La descripción inicial de un problema no suele ser clara y casi siempre, al principio, suele ser imprecisa y vaga.

La formulación de una especificación del problema requiere una descripción precisa y lo más completa posible de la *entrada* del problema (información disponible para la resolución del problema) y la *salida* requerida. Además, se requiere información adicional tal como: *hardware* y *software* necesarios, tiempo de respuesta, plazos de entrada, facilidad de uso, robustez del *software*, etc.

La persona que realiza el análisis debe tener una perspectiva inicial lo más amplia posible y comprender el propósito principal de los que el problema o sistema pretende conseguir. En sistemas grandes el análisis, lo realiza normalmente un analista de sistemas, y en programas individuales, el análisis se realiza directamente por el programador.

Con independencia de cómo y por quién se realice el análisis, a la conclusión del mismo se debe tener una comprensión muy clara de:

¿Qué debe hacer el sistema o el programa?
¿Qué salidas debe producir?
¿Qué entradas se requieren para obtener las salidas deseadas?

### 1.2.3. Diseño

La fase de diseño de la solución se inicia una vez que se tiene la especificación del problema y consiste en la formulación de los pasos o etapas para resolver el problema. Dos metodologías son las más utilizadas en el diseño: diseño *descendente* o *estructurado* que se apoya en *progra-*

*mación estructurada* y diseño *orientado* a *objetos* que se basa en la *programación orientada a objetos.*

El diseño de una solución requiere el uso de algoritmos. Un **algoritmo** es un conjunto de instrucciones o pasos para resolver un problema Los algoritmos deben procesar los datos necesarios para la resolución del problema. Los datos se organizan en *almacenes* o *estructuras* de datos. Los programas se compondrán de algoritmos que manipulan o procesan las estructuras de datos.

Una buena técnica para el diseño de un algoritmo, como se ha comentado (diseño descendente), es descomponer el problema en subproblemas o subtareas más pequeñas y sencillas, a continuación descomponer cada subproblema o subtarea en otra más pequeña y así sucesivamente, hasta llegar a subtareas que sean fáciles de implantar en C++ o en cualquier otro lenguaje de programación.

Los algoritmos se suelen escribir en *pseudocódigo* o en otras herramientas de programación como *diagramas de flujo* o *diagramas N-S*. Hoy día la herramienta más utilizada es el pseudocódigo o lenguaje algorítmico, consistentete en un conjunto de palabras —en español, inglés, etcétera— que representan tareas a realizar y una sintaxis de uso como cualquier otro lenguaje.

> *Técnica de diseño descendente*. 1) Descomponer una tarea en subtareas; a continuación cada subtarea en tareas más pequeñas. 2) Diseñar el algoritmo que describe cada tarea.

Otra técnica o método de diseño muy utilizado en la actualidad es el *diseño orientado a objetos*. El diseño descendente se basa en la descomposición de un problema en un conjunto de tareas y en la realización de los algoritmos que resuelven esas tareas, mientras que el diseño orientado a objetos se centra en la localización de módulos y objetos del mundo real. Estos objetos del mundo real están formados por datos y operaciones que actúan sobre los datos y modelan, a su vez, a los objetos del mundo real, e interactúan entre sí para resolver el problema concreto.

El diseño orientado a objetos ha conducido a la programación orientada a objetos, como uno de los métodos de programación más populares y más utilizados en el pasado siglo XX y actual XXI.

### Consideraciones prácticas de diseño

Una vez que se ha realizado un análisis y una especificación del problema se puede desarrollar una solución. El programador se encuentra en una situación similar a la de un arquitecto que debe dibujar los planos de una casa; la casa deber cumplir ciertas espeficaciones y cumplir las necesidades de su propietario, pero puede ser diseñada y construida de muchas formas posibles. La misma situación se presenta al programador y al programa a construir.

En programas pequeños, el algoritmo seleccionado suele ser muy simple y consta de unos pocos cálculos que deben realizarse. Sin embargo, lo más normal es que la solución inicial debe ser refinada y organizada en subsistemas más pequeños, con especificaciones de cómo interactuar entre sí y los interfaces correspondientes. Para conseguir este objetivo, la descripción de la solución comienza desde el requisito de nivel más alto y prosigue en modo descendente hacia las partes que deben construir para conseguir este requisito.

Una vez que se ha desarrollado una estructura inicial se refinan las tareas hasta que éstas se encuentren totalmente definidas. El proceso de refinamiento de una solución continúa hasta

que los requisitos más pequeños se incluyan dentro de la solución. Cuando el diseño se ha terminado, el problema se resuelve mediante un *programa* o un *sistema de módulos* (funciones, clases...), *bibliotecas de funciones* o de *clases*, *plantillas*, *patrones*, etc.

### 1.2.4. Implementación (codificación)

La *codificación* o *implementación* implica la traducción de la solución de diseño elegida en un programa de computadora escrito en un lenguaje de programación tal como **C++** o **Java**. Si el análisis y las especificaciones han sido realizadas con corrección y los algoritmos son eficientes, la etapa de codificación normalmente es un proceso mecánico y casi automático de buen uso de las reglas de sintaxis del lenguaje elegido.

El código fuente, sea cual sea el lenguaje de programación en el cual se haya escrito, debe ser legible, comprensible y correcto (fiable). Es preciso seguir buenas prácticas de programación. Los buenos hábitos en la escritura de programas facilita la ejecución y prueba de los mismos. Tenga presente que los programas, subprogramas (funciones) y bibliotecas escritos por estudiantes suelen tener pocas líneas de programa (decenas, centenas...); sin embargo, los programas que resuelven problemas del mundo real contienen centenares, y millares e incluso millones de líneas del código fuente y son escritos por equipos de programadores. Estos programas se usan, normalmente, durante mucho tiempo y requieren un mantenimiento que en muchos casos se realiza por programadores distintos a los que escribieron el programa original. Por estas razones, es muy importante escribir programas que se puedan leer y comprender con facilidad así como seguir hábitos y reglas de lecturas que conduzcan a programas correctos (fiables).

> ***Recuerde***
>
> Escribir un programa sin diseño es como construir una casa sin un plano (proyecto).

### 1.2.5. Pruebas y depuración

La *prueba* y *corrección* de un programa pretende verificar que dicho programa funciona correctamente y cumple realmente todos sus requisitos. En teoría las pruebas revelan todos los errores existentes en el programa. En la práctica, esta etapa requiere la comprobación de todas las combinaciones posibles de ejecución de las sentencias de un programa; sin embargo, las pruebas requieren, en muchas ocasiones, mucho tiempo y esfuerzo, que se traduce a veces en objetivo imposible excepto en programas que son muy sencillos.

Los errores pueden ocurrir en cualquiera de las fases del desarrollo de *software*. Así, puede suceder que las especificaciones no contemplen de modo preciso la información de entrada, o los requisitos dados por el cliente; también puede suceder que los algoritmos no estén bien diseñados y contengan errores lógicos o por el contrario que los módulos o unidades de programa no estén bien codificados o la integración de las mismas en el programa principal no se haya realizado correctamente. La *detección* y *corrección de errores* es una parte importante del desarrollo de *software*, dado que dichos errores pueden aparecer en cualquier fase del proceso.

Debido a que las pruebas exhautivas no suelen ser factibles ni viables en la mayoría de los programas se necesitan diferentes métodos y filosofías de prueba. Una de las responsabilidades

de la ciencia de *ingeniería de software* es la construcción sistemática de un conjunto de pruebas (*test*) de entradas que permitan descubrir errores. Si las pruebas revelan un **error** (*bug*), el proceso de **depuración** —detección, localización, corrección y verificación— se puede iniciar. Es importante advertir que aunque *las pruebas puedan detectar la presencia de un error, no necesariamente implica la ausencia de errores*. Por consiguiente, el hecho de que una prueba o test, revele la existencia de un error no significa que uno indefinible pueda existir en otra parte del programa.

Para atrapar y corregir errores de un programa, es importante desarrollar un conjunto de datos de prueba que se puedan utilizar para determinar si el programa proporciona respuestas correctas. De hecho, una etapa aceptada en el desarrollo formal de *software* es planificar los procedimientos de pruebas y crear pruebas significativas antes de escribir el código. Los procedimientos de prueba de un programa deben examinar cada situación posible bajo la cual se ejecutará el programa. El programa debe ser comprobado con datos en un rango razonable así como en los límites y en las áreas en las que el programa indique al usuario que los datos no son válidos. El desarrollo de buenos procedimientos y datos de prueba en problemas complejos pueden ser más difíciles que la escritura del propio código del programa.

### Verificación y validación

La prueba del *software* es un elemento de un tema más amplio que suele denominarse verificación y validación. **Verificación** es el conjunto de actividades que aseguran que el *software* funciona correctamente con una amplia variedad de datos. **Validación** es el conjunto diferente de actividades que aseguran que el *software* construido se corresponde con los requisitos del cliente [PRESMAN 05][4].

En la práctica, la verificación pretende comprobar que los documentos del programa, los módulos y restantes unidades son correctos, completos y consistentes entre sí y con los de las fases precedentes; la validación, a su vez, se ocupa de comprobar que estos productos se ajustan a la especificación del problema. Boehm [BOEHM 81][5] estableció que la verificación era la respuesta a: "*¿Estamos construyendo el producto correctamente?*" mientras que la validación era la respuesta a: "*¿Estamos construyendo el programa correcto?*".

En la prueba de *software* convencional en posible aplicar diferentes clases de pruebas. Pressman distingue las siguientes:

— *Prueba de unidad:* se centra el esfuerzo de verificación en la unidad más pequeña del diseño de *software*, el componente o módulo (función o subprograma) de *software* que se comprueban individualmente.
— *Prueba de integración:* se comprueba si las distintas unidades del programa se han unido correctamente. Aquí es muy importante descubrir errores asociados con la interfaz.

En el caso de *software* orientado a objetos cambia el concepto de unidad que pasa a ser la **clase** o la **instancia de una clase (objeto)** que empaqueta los *atributos* (datos) y las *operaciones* (funciones) que manipulan estos datos. La prueba de la clase en el *software* orientado a objetos es la equivalente a la prueba de unidad para el *software* convencional. La prueba de integración se realiza sobre clases que colaboran entre sí.

[4] *Ibid*, p. 364.
[5] Boehm.

Otro tipo de pruebas importantes son las pruebas del sistema. Una prueba del sistema comprueba que el sistema global del programa funciona correctamente; es decir las funciones, las clases, las bibliotecas, etc. Las pruebas del sistema abarcan una serie de pruebas diferentes cuyo propósito principal es verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas. Estas pruebas se corresponden con las distintas etapas del desarrollo del *software*.

**Elección de datos de prueba**

Para que los datos de prueba sean buenos, necesitan cumplir dos propiedades [MAINSA 01][6]:

1. Se necesita conocer cuál es la salida que debe producir un programa correcto para cada entrada de prueba.
2. Las entradas de prueba deben incluir aquellas entradas que más probabilidad tengan de producir errores.

Aunque un programa se compile, se ejecute y produzca una salida que parezca correcta no significa que el programa sea correcto. Si la respuesta correcta es 211492 y el programa obtiene 211491, algo está equivocado. A veces, el método más evidente para encontrar el valor de salida correcto es utilizar lápiz y papel utilizando un método distinto al empleado en el programa. Puede ayudarle utilizar valores de entrada más pequeños o simplemente valores de entrada cuya salida sea conocida.

Existen diferentes métodos para encontrar datos de prueba que tengan probabilidad de producir errores. Uno de los más utilizados se denomina *valores de frontera*. Un *valor frontera* de un problema es una entrada que produce un tipo de comportamiento diferente, por ejemplo, la función C++

```
int comprobar_hora (int hora)

//la hora de día está en el rango 0 a 23, tiene dos valores frontera 0
//(referir a 0, no es válido) y 23 (superior a 23 no es válida, ya que 24
//es un nuevo día); de igual modo si se deja contemplar el hecho de
//mañana (AM) o tarde (PM), los valores frontera serán 0 y 11, 12 y 23, //
respectivamente.
```

En general, no existen definiciones de valores frontera y es en las especificaciones del problema donde se pueden obtener dichos valores. Una buena regla suele ser la siguiente: *"Si no puede comprobar todas las entradas posible, al menos compruebe los valores frontera. Por ejemplo, si el rango de entrada va de 0 a 500.000, asegure la prueba de 0 y 500.000, y será buena práctica probar 0, 1 y –1 siempre que sean valores válidos"*.

## 1.2.6. Depuración

La **depuración** es el proceso de fijación o localización de errores. La detección de una entrada de prueba que produce un error es sólo parte del problema de prueba y depuración. Después que se encuentra una entrada de prueba errónea se debe determinar exactamente por qué ocurre

[6] Mainsa.

el error y, a continuación, depurar el programa. Una vez que se ha corregido un error debe volver a ejecutar el programa.

En programas sencillos la depuración se puede realizar con mayor o menor dificultad, pero en programas grandes el *seguimiento* (*traza* o *rastreo*) de errores es casi imposible sin ayuda de una herramienta de *software* denominada *depurador* (*debugger*). Un depurador ejecuta el código del programa línea a línea, o puede ejecutar el código harta que se produzca una cierta condición. El uso de un depurador puede especificar cuáles son las condiciones que originan la ejecución anómala de un programa. Los errores más frecuentes de un programa son:

- *Errores de sintaxis:* faltas gramaticales de la sintaxis del lenguaje de programación.
- *Errores en tiempo de ejecución:* se producen durante la ejecución del programa.
- *Errores lógicos:* normalmente errores de diseño del algoritmo.

**EJEMPLOS**

*Errores de sintaxis*
```
double presupuesto //error, falta el ;
cin presupuesto;   //error, falta >>
```

*Error de ejecución*
```
Cálculo de división por cero, obtener raíces de números
negativos, valores fuera de rango.
```

*Error lógico*
```
Mal planteamiento en el diseño de un algoritmo.
```

## 1.2.7. Documentación

El desarrollo de *software* requiere un gran esfuerzo de documentación de las diferentes etapas. En la práctica, muchos de los documentos clave (críticos) se crean durante las fases de análisis, diseño, codificación y prueba. La documentación completa del *software* presenta todos los documentos en un manual que sea útil a los programadores y a su organización.

Aunque el número de documentos puede variar de un proyecto a otro, y de una organización a otra, esencialmente existen cinco documentos imprescindibles en la documentación final de un programa:

1. Descripción del problema (especificaciones).
2. Cambio y desarrollo de los algoritmos.
3. Listados de programas, bien comentados.
4. Ejecución de las pruebas de muestra.
5. Manual del usuario.

La documentación del *software* comienza en la fase de análisis y especificación del problema y continúa en la fase de mantenimiento y evolución.

## 1.2.8. Mantenimiento

Una vez que el *software* se ha depurado completamente y el conjunto de programas, biblioteca de funciones y clases, etc., se han terminado y funcionan correctamente, el uso de los mismos se puede extender en el tiempo durante grandes períodos (normalmente meses o años).

La fase de mantenimiento del *software* está relacionada con corrección futura de problemas, revisión de especificaciones, adición de nuevas características, etc. El mantenimiento requiere, normalmente, un esfuerzo importante ya que si bien el desarrollo de un programa puede durar días, meses o años, el mantenimiento se puede extender a años e incluso décadas. Un ejemplo típico estudiado en numerosos cursos de ingeniería de *software* fue el esfuerzo realizado para asegurar que los programas existentes funcionan correctamente al terminar el siglo XX conocido como el efecto del año 2000.

Estadísticamente está demostrado que los sistemas de *software*, especialmente los desarrollados para resolver problemas complejos presentan errores que no fueron detectados en las diferentes pruebas y que se detectan cuando el *software* se pone en funcionamiento de modo comercial o profesional. La reparación de estos errores es una etapa importante y muy costosa del mantenimiento de un programa.

Además de estas tareas de mantenimiento existen muchas otras tareas dentro de esta etapa: "mejoras en la eficiencia, añadir nuevas características (por ejemplo, nuevas funcionalidades), cambios en el *hardware*, en el sistema operativo, ampliar número máximo de usuarios...". Otros cambios pueden venir derivados de cambios en normativas legales, en la organización de la empresa, en la difusión del producto a otros países, etc.

Estudios de *ingeniería de software* demuestran que el porcentaje del presupuesto de un proyecto *software* y del tiempo de programador/analista/ingeniero de *software* ha ido creciendo por décadas. Así, en la década de los setenta, se estimaba el porcentaje de mantenimiento entre un 35-40 por 100, en la década de los ochenta, del 40 al 60 por 100, y se estima que en la década actual del siglo XXI, puede llegar en aplicaciones para la web, videojuegos, *software* de inteligencia de negocios, de gestión de relaciones con los clientes (**CRM**), etc., hasta un 80 o un 90 por 100 del presupuesto total del desarrollo de un producto *software*.

Por todo lo anterior, es muy importante que los programadores diseñen programas legibles, bien documentados y bien estructurados, bibliotecas de funciones y de clases con buena documentación, de modo que los programas sean fáciles de comprender y modificar y en consecuencia fáciles de mantener.

## 1.3. CALIDAD DEL *SOFTWARE*

El *software* de calidad debe cumplir las siguientes características:

**Corrección:** Capacidad de los productos *software* de realizar exactamente las tareas definidas por su especificación.

**Legibilidad y comprensibilidad:** Un sistema debe ser fácil de leer y lo más sencillo posible. Estas características se ven favorecidas con el empleo de abstracciones, sangrado y uso de comentarios.

**Extensibilidad:** Facilidad que tienen los productos de adaptarse a cambios en su especificación. Existen dos principios fundamentales para conseguir esto, diseño simple y descentralización.

**Robustez:** Capacidad de los productos *software* de funcionar incluso en situaciones anormales.

**Eficiencia:** La eficiencia de un *software* es su capacidad para hacer un buen uso de los recursos del computador. Un sistema eficiente es aquél cuya velocidad es mayor con el menor espacio de memoria ocupada. Las grandes velocidades de los microprocesadores (unidades centrales de proceso) actuales, junto con el aumento considerable de las memorias centrales (cifras típi-

cas usuales superan siempre 1-4 GB), hacen que disminuya algo la importancia concedida a ésta, debiendo existir un compromiso entre legibilidad, *modificabilidad* y eficiencia.

**Facilidad de uso:** La *utilidad* de un sistema está relacionada con su facilidad de uso. Un *software* es fácil de utilizar cuando el usuario puede comunicarse con él de manera cómoda.

**Transportabilidad (*portabilidad*):** *La transportabilidad o portabilidad* es la facilidad con la que un *software* puede ser transportado sobre diferentes sistemas físicos o lógicos.

***Verificabilidad*:** La *verificabilidad*, "facilidad de verificación" de un *software*, es su capacidad para soportar los procedimientos de validación y de aceptar juegos de test o ensayo de programas.

**Reutilización:** Capacidad de los productos de ser reutilizados, en su totalidad o en parte, en nuevas aplicaciones.

**Integridad:** La integridad es la capacidad de un *software* para proteger sus propios componentes contra los procesos que no tengan el derecho de acceso.

**Compatibilidad:** Facilidad de los productos para ser combinados con otros y usados en diferentes plataformas *hardware* o *software*.

## 1.4. ALGORITMOS

El término **resolución de un problema** se refiere al proceso completo que abarca desde la descripción inicial del problema hasta el desarrollo de un programa de computadora que lo resuelva. La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto. 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 indicando *cómo* hace el algoritmo la tarea solicitada, y eso se traduce en la construcción de un algoritmo. El resultado final del diseño es una solución que debe ser fácil de traducir a estructuras de datos y estructuras de control de un lenguaje de programación específico.

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

### Diagramas de flujo (*flowchart*)

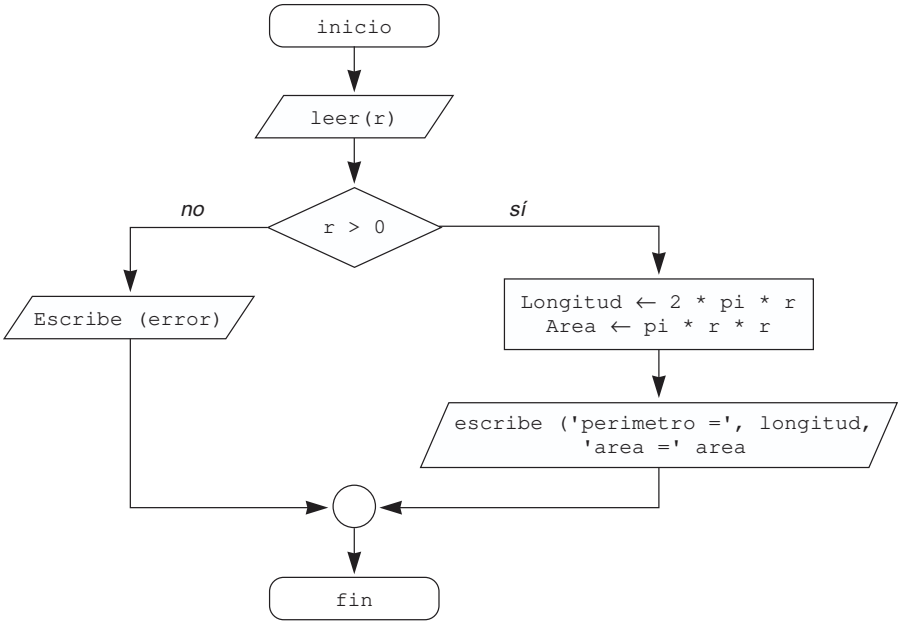
Es una representación gráfica de un algoritmo.

---

**EJEMPLO 1.1.** Diagrama de flujo que representa un algoritmo que lea el radio de un círculo, calcule su perímetro y su área.

Se declaran las variables reales `r`, `longitud` y `area`, así como la constante `pi`

```
constantes pi = 3.14
variables real: r, longitud, area
```




### Pseudocódigo

En esencia el pseudocódigo se puede definir como *un lenguaje de especificación de algoritmos.*

---

**EJEMPLO 1.2.** Realizar un algoritmo que lea tres números; si el primero es positivo calcule el producto de los tres números, y en otro caso calcule la suma.

Se usan tres variables enteras `Numero1, Numero2, Numero3`, en las que se leen los datos, y otras dos variables `Producto` y `Suma` en las que se calcula, o bien el producto, o bien la suma. El algoritmo que resuelve el problema es el siguiente.

```
Entrada Numero1,Numero2 y Numero3
Salida  Suma o el Producto

inicio
1 leer los tres números Numero1, Numero2, Numero3
2 si el Numero1 es positivo
    calcular el producto de los tres números
    escribir el producto
3 si el Numero1 es no positivo
    calcular la suma de los tres números
    escribir la suma
fin
```

El *algoritmo en pseudocódigo* es:

```
algoritmo Producto__Suma
variables
    entero: Numero1, Numero2, Numero3, Producto, Suma
inicio
     Leer(Numero1, Numero2, Numero3)
     si (Numero1 > 0) entonces
         Producto ← Numero1* Numero2 * Numero3
         Escribe('El producto de los números es:', Producto)
     sino
         Suma ← Numero1 + Numero2 + Numero3
         Escribe('La suma de los números es: ', Suma)
     fin si
fin
```

El *algoritmo escrito* en C++ es:

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int Numero1, Numero2, Numero3, Producto, Suma;
    cin >> Numero1 >> Numero2 >>Numero3;
    if(Numero1 > 0)
    {
        Producto = Numero1* Numero2 * Numero3;
        cout <<"El producto de los números es" << Producto;
    }
    else
    {
        Suma = Numero1 + Numero2 + Numero3;
        cout <<" La suma de los números es:" << Suma;
    }
    return 0;
}
```

El **algoritmo** es la especificación concisa del método para resolver un problema con indicación de las acciones a realizar. Un **algoritmo** es un conjunto finito de reglas que dan una secuencia de operaciones para resolver un determinado problema. Es, por consiguiente, *un método para resolver un problema que tiene en general una entrada y una salida*. 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; es decir, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: *Entrada, Proceso* y *Salida*.

**EJEMPLO 1.3.** Diseñar un algoritmo que permita saber si un número entero positivo 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.

*Entrada*: `dato n entero positivo`
*Salida*: `es o no primo.`
*Proceso*:

```
1. Inicio.
2. Hacer x igual a 2 (x = 2, x variable que representa a los divisores del
   número que se buscan).
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 ← x + 1).
6. Si x es igual a n, entonces n es un número primo ; en caso contrario,
   bifurcar al punto 3.
7. Fin.
```

El algoritmo anterior escrito en pseudocódigo y C++ es el siguiente:

| Algoritmo en pseudocódigo | Algoritmo en C++ |
|---|---|
| **algoritmo** primo<br>**inicio**<br>  variables<br>  entero: n, x;<br>  lógico: primo;<br>  leer(n);<br>  x ← 2;<br>  primo ← verdadero;<br>  **mientras** primo **y** (x < n) **hacer**<br>  **si** n mod x <> 0 **entonces**<br>  x ← x+1<br>  **sino**<br>  primo ← falso<br>  **fin si**<br>  **fin mientras**<br>  **si** (primo) **entonces**<br>  escribe('es primo')<br>  **sino**<br>  escribe('no es primo')<br>  **fin si**<br>**fin** | #include <cstdlib><br>#include <iostream><br>using namespace std;<br>int main(int argc, char *argv[])<br>{<br>  int n, x;<br>  bool primo;<br>  cin >> n;<br>  x = 2;<br>  primo = true;<br>  while (primo &&(x < n))<br>  if (n % x != 0)<br>  x = x+1;<br>  else<br>  primo = false;<br>  if (primo)<br>  cout << "es primo";<br>  else<br>  cout << " no es primo";<br>  return 0;<br>} |

**EJEMPLO 1.4.** 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, el algoritmo en forma descriptiva es:

```
inicio
  1. Inicializar contador de numeros C y variable suma S a cero (S ← 0,
     C ← 0).
  2. Leer un numero en la variable N (leer(N))
  3. Si el numero leído es cero: (si (N = 0) entonces)
     3.1 Si se ha leído algún número (Si C>0)
         • calcular la media; (media ← S/C)
         • imprimir la media; (Escribe(media))
     3.2 si no se ha leído ningún número (Si C = 0)
         • escribir no hay datos.
     3.3 fin del proceso.
  4. Si el numero leído no es cero : (Si (N <> 0) entonces
     • calcular la suma; (S ← S+N)
     • incrementar en uno el contador de números; (C←C+1)
     • ir al paso 2.
fin
```

El algoritmo anterior escrito en pseudocódigo y en C++ es el siguiente:

**Algoritmo escrito en pseudocódigo**

```
algoritmo media
 inicio
  variables
    entero: N, C, S;
    real: media;
 C ← 0;
 S ← 0;
 repetir
   leer(N)
  si N <> 0 entonces
   S ← S + N;
   C ← C +1 ;
  fin si
 hasta N = 0
 si C > 0 entonces
  media ← S / C
  escribe( media)
 sino
  escribe("no datos")
 fin si
fin
```

**Algoritmo escrito en C++**

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int N, C, S;
    float media;
    C = 0; S = 0;
    do
    {
        cin >> N;
        if(N != 0)
        {
            S = S + N;
            C = C + 1 ;
        }
    } while (N != 0);
    if(C > 0)
        {media = S / C;
        cout << media;
    }
    else
        cout << "no datos";
    return 0;
}
```

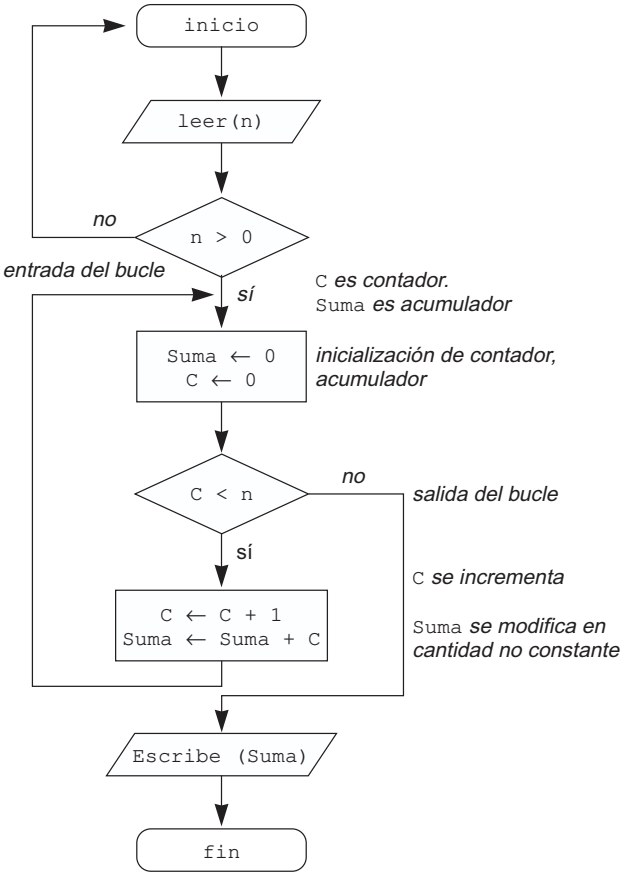
**EJEMPLO 1.5.** Algoritmo que lee un número entero positivo *n*, y suma los *n* primeros número naturales.

Inicialmente se asegura la lectura del número natural `n` positivo. Mediante un contador `C` se cuentan los números naturales que se suman, y en el acumulador `Suma` se van obteniendo las sumas parciales. Además del diagrama de flujo se realiza un seguimiento para el caso de la entrada `n = 5`.

```
Variables Entero: n, Suma, C
```

**seguimiento**

| *paso* | n | C | Suma |
|---|---|---|---|
| 0 | 5 | 0 | 0 |
| 1 | 5 | 1 | 1 |
| 2 | 5 | 2 | 3 |
| 3 | 5 | 3 | 6 |
| 4 | 5 | 4 | 10 |
| 5 | 5 | 5 | 15 |



El algoritmo anterior escrito en pseudocódigo y C++ es el siguiente:

| **Algoritmo escrito en pseudocódigo** | **Algoritmo escrito en C++** |
|---|---|
| **algoritmo** suma_n_naturales<br>**inicio**<br>  variables<br>  entero: Suma, C, n;<br>  **repetir**<br>  leer(n)<br>  **hasta** n>0<br>  C ← 0;<br>  Suma ← 0;<br>  **mientras** C < n **hacer**<br>  C ← C + 1;<br>  Suma ← Suma + C;<br>  **fin mientras**<br>  escribe( Suma)<br>**fin** | #include <cstdlib><br>#include <iostream><br>using namespace std;<br><br>int main(int argc, char *argv[])<br>{<br>  int Suma, C, n;<br>  do<br>  cin >> n;<br>  while (n <= 0);<br>  C = 0; Suma = 0;<br>  while (C < n )<br>  {<br>  C = C + 1;<br>  Suma = Suma + C;<br>  }<br>  cout << Suma;<br>  return 0;<br>} |

## 1.5. ABSTRACCIÓN EN LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación son las herramientas mediante las cuales los diseñadores de programas pueden implementar los modelos abstractos. La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: *abstracción de datos* (perteneciente a los datos) y *abstracción de control* (perteneciente a las estructuras de control).

Desde comienzos del decenio de los sesenta, en que se desarrollaron los primeros lenguajes de programación de alto nivel, ha sido posible utilizar las abstracciones más primitivas de ambas categorías (variables, tipos de datos, funciones, control de bucles (lazos), etc.).

### 1.5.1. Abstracciones de control

Los microprocesadores ofrecen directamente sólo dos mecanismos para controlar el flujo y ejecución de las instrucciones: *secuencia* y *salto*. Los primeros lenguajes de programación de alto nivel introdujeron las estructuras de control: *sentencias de bifurcación* (`if`) y *bucles* (`for`, `while`, `do-loop`, `do-while`, etc.).

Las estructuras de control describen el orden en que se ejecutan las sentencias o grupos de sentencia (*unidades de programa*). Las unidades de programa se utilizan como bloques básicos de la clásica descomposición "descendente". En todos los casos, los subprogramas constituyen una herramienta potente de abstracción ya que durante su implementación, el programador describe en detalle cómo funcionan. Cuando el subprograma se llama, basta con conocer lo que hace y no cómo lo hace. De este modo, se convierten en cajas negras que amplían el lenguaje de programación a utilizar. En general, los subprogramas son los mecanismos más ampliamente utilizados para reutilizar código, a través de colecciones de subprogramas en bibliotecas.

Las abstracciones y estructuras de control se clasifican en estructuras de control a nivel de sentencia y a nivel de unidades. La abstracción de control a nivel de unidad se conoce como *abstracción procedimental*.

**Abstracción procedimental (por procedimientos o funciones)**

Es esencial para diseñar *software* modular y fiable. La *abstracción procedimental* se basa en la utilización de procedimientos o funciones sin preocuparse de cómo se implementan. Esto es posible sólo si conocemos qué hace el procedimiento; esto es, conocemos la sintaxis y semántica que utiliza el procedimiento o función. La abstracción aparece en los subprogramas debido a las siguientes causas:

- Con el nombre de los subprogramas, un programador puede asignar una descripción abstracta que captura el significado global del subprograma. Utilizando el nombre en lugar de escribir el código permite al programador aplicar la acción en términos de su descripción de alto nivel en lugar de sus detalles de bajo nivel.
- Los subprogramas proporcionan ocultación de la información. Las variables locales y cualquier otra definición local se encapsulan en el subprograma, ocultándolas realmente de forma que no se pueden utilizar fuera del subprograma. Por consiguiente, el programador no tiene que preocuparse sobre las definiciones locales.
- Los parámetros de los subprogramas, junto con la ocultación de la información anterior, permiten crear subprogramas que constituyen entidades de *software* propias. Los detalles locales de la implementación pueden estar ocultos mientras que los parámetros se pueden utilizar para establecer la interfaz *público*.

En C++ la abstracción procedimental se establece con los *métodos* o *funciones miembro* de clases.

**Otros mecanismos de abstracción de control**

La evolución de los lenguajes de programación ha permitido la aparición de otros mecanismos para la abstracción de control, tales como *manejo de excepciones, corrutinas, unidades concurrentes* o *plantillas* (*templates*). Estas construcciones las soportan los lenguajes de programación basados y orientados a objetos, tales como C++ C#, Java, Modula-2, Ada, Smalltalk o Eiffel.

## 1.5.2. Abstracciones de datos

Los primeros pasos hacia la abstracción de datos se crearon con lenguajes tales como FORTRAN, COBOL y ALGOL 60, con la introducción de tipos de variables diferentes, que manipulan enteros, números reales, caracteres, valores lógicos, etc. Sin embargo, estos tipos de datos no podían ser modificados y no siempre se ajustaban al tipo necesitado. Por ejemplo, el tratamiento de cadenas es una deficiencia en FORTRAN, mientras que la precisión y fiabilidad para cálculos matemáticos es muy alta.

La siguiente generación de lenguajes, PASCAL, SIMULA-67 y ALGOL 68, ofreció una amplia selección de tipos de datos y permitió al programador modificar y ampliar los tipos de datos existentes mediante construcciones específicas (por ejemplo, *arrays* y registros). Además, SIMULA-67 fue el primer lenguaje que mezcló datos y procedimientos mediante la construcción de clases, que eventualmente se convirtió en la base del desarrollo de programación orientada a objetos.

La *abstracción de datos* es la técnica de programación que permite inventar o definir nuevos tipos de datos (tipos de datos definidos por el usuario) adecuados a la aplicación que se desea realizar. La abstracción de datos es una técnica muy potente que permite diseñar programas más cortos, legibles y flexibles. La esencia de la abstracción es similar a la utilización de un tipo de dato, cuyo uso se realiza sin tener en cuenta cómo está representado o implementado.

Los tipos de datos son abstracciones y el proceso de construir nuevos tipos se llaman abstracciones de datos. Los nuevos tipos de datos definidos por el usuario se llaman *tipos abstractos de datos* (ADT, **A**bstract **D**ata **T**ypes).

El concepto de tipo, tal como se definió en PASCAL y ALGOL 68, ha constituido un hito importante hacia la realización de un lenguaje capaz de soportar programación estructurada. Sin embargo, estos lenguajes no soportan totalmente una metodología orientada a objetos. La abstracción de datos útil para este propósito, no sólo clasifica objetos de acuerdo a su estructura de representación, sino que se clasifican de acuerdo al comportamiento esperado. Tal comportamiento es expresable en términos de operaciones que son significativas sobre esos datos, y las operaciones son el único medio para crear, modificar y acceder a los objetos.

En términos más precisos, Ghezzi indica que un tipo de dato definido por el usuario se denomina *tipo abstracto de dato* (**TAD**) si:

- Existe una construcción del lenguaje que le permite asociar la representación de los datos con las operaciones que lo manipulan;
- La representación del nuevo tipo de dato está oculta de las unidades de programa que lo utilizan [GHEZZI 87][7].

Las **clases** de C++, C# y Java cumplen las dos condiciones: agrupa los datos junto a las operaciones, y su representación queda oculta de otras clases.

Los tipos abstractos de datos proporcionan un mecanismo adicional mediante el cual se realiza una separación clara entre la *interfaz* y la *implementación* del tipo de dato. La **implementación de un tipo abstracto de dato** consta de:

1. *Representación*: elección de las estructuras de datos.
2. *Operaciones*: elección de los algoritmos.

La interfaz del tipo abstracto de dato se asocia con las operaciones y datos *visibles* al exterior del **TAD**.

## 1.6. TIPOS ABSTRACTOS DE DATOS

Algunos lenguajes de programación tienen características que nos permiten ampliar el lenguaje añadiendo sus propios tipos de datos. Un tipo de dato definido por el programador se denomina *tipo abstracto de datos* (**TAD**) para diferenciarlo del tipo fundamental (predefinido) de datos. Por ejemplo, en C++ el tipo `Punto`, que representa a las coordenadas `x` e `y` de un sistema de coordenadas rectangulares, no existe. Sin embargo, es posible implementar el tipo abstracto de datos, considerando los valores que se almacenan en las variables y qué operaciones están disponibles para manipular estas variables. En esencia un tipo abstracto de datos es un tipo que consta de datos (estructuras de datos propias) y operaciones que se pueden realizar sobre esos datos. Un **TAD** se compone de *estructuras de datos* y los *procedimientos* o *funciones* que manipulan esas estructuras de datos.

[7] GHEZZI 87.

Un tipo abstracto de datos puede definirse mediante la ecuación:

**TAD** = Representación (datos) + Operaciones (funciones y procedimientos)

La estructura de un tipo abstracto de dato, desde un punto de vista global, se compone de la interfaz y de la implementación (Figura 1.2).

Las estructuras de datos reales elegidas para almacenar la representación de un tipo abstracto de datos son invisibles a los usuarios o clientes. Los algoritmos utilizados para implementar cada una de las operaciones de los **TAD** están encapsuladas dentro de los propios **TAD**. La característica de ocultamiento de la información significa que los objetos tienen *interfaces públicos*. Sin embargo, las representaciones e implementaciones de esos *interfaces* son *privados*.

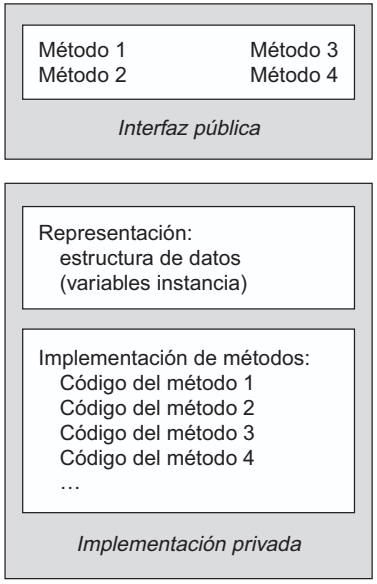



**Figura 1.2.** Estructura de un tipo abstracto de datos (**TAD**).

### 1.6.1. Ventajas de los tipos abstractos de datos

Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

1. Permite una mejor conceptualización y *modelización* (modelado) del mundo real. Mejora la representación y la comprensibilidad. Clasifica los objetos basados en estructuras y comportamientos comunes.
2. Mejora la robustez del sistema. Permiten la especificación del tipo de cada variable, de tal forma que se facilita la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.

3. Mejora el rendimiento (prestaciones). Para sistemas tipificados, el conocimiento de los objetos permite la optimización de tiempo de compilación.
4. Separa la implementación de la especificación. Permite la modificación y mejora de la implementación sin afectar al interfaz público del tipo abstracto de dato.
5. Permite la extensibilidad del sistema. Los componentes de *software* reutilizables son más fáciles de crear y mantener.
6. Recoge mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

Un programa que maneja un **TAD** lo hace teniendo en cuenta las operaciones o funcionalidad que tiene, sin interesarse por la representación física de los datos. Es decir, los *usuarios* de un **TAD** se comunican con éste a partir de la interfaz que ofrece el **TAD** mediante funciones de acceso. Podría cambiarse la implementación de tipo de datos sin afectar al programa que usa el **TAD** ya que para el programa está *oculta*.

Las unidades de programación de lenguajes que pueden implementar un **TAD** reciben distintos nombres:

| | |
|---|---|
| Modula-2 | *módulo* |
| Ada | *paquete* |
| C++ | *clase* |
| *C#* | *clase* |
| Java | *clase* |

En estos lenguajes se definen la *especificación* del TAD, que declara las operaciones y los datos, y la *implementación*, que muestra el código fuente de las operaciones y que permanece oculto al exterior del módulo.

---

**EJEMPLO 1.6.** Clase `hora` que tiene datos separados de tipo int para horas, minutos y segundos. Un constructor inicializará este dato a 0, y otro lo inicializará a valores fijos. Una función miembro deberá visualizar la hora en formato 11:59:59. Otra función miembro sumará dos objetos de tipo hora pasados como argumentos. Una función principal `main()` creará dos objetos inicializados y otro que no está inicializado. Se suman los dos valores inicializados y se deja el resultado en el objeto no inicializado. Por último, se muestra el valor resultante.

```
#include <cstdlib>
#include <iostream>
using namespace std;

class hora
{
private:
    int horas, minutos, segundos;
public:
    hora(){ horas = 0; minutos = 0; segundos = 0; }
    hora(int h, int m, int s){horas = h; minutos = m; segundos = s;}
    void visualizar();
    void sumar(hora h1, hora h2 );
};
```

```
void hora::visualizar()
{
    cout << horas << ":" << minutos << ":" << segundos << endl;
}

void hora::sumar(hora h1, hora h2 )
{
    segundos = h2.segundos + h1.segundos;
    minutos = h2.minutos + h1.minutos + segundos / 60;
    segundos = segundos % 60;
    horas = h2.horas + h1.horas + minutos / 60;
    minutos = minutos % 60;
}

int main(int argc, char *argv[])
{
    hora h1(10,40,50), h2(12,35,40), h;
    h1.visualizar();
    h2.visualizar();
    h.sumar(h1,h2);
    h.visualizar();
    return 0;
}
```

## 1.6.2. Especificación de los TAD

El objetivo de la especificación es describir el comportamiento del **TAD**; consta de dos partes, la descripción matemática del conjunto de datos, y de las operaciones definidas en ciertos elementos de ese conjunto de datos.

La especificación del **TAD** puede tener un enfoque *informal*, éste describe los datos y las operaciones relacionadas en *lenguaje natural*. Otro enfoque más riguroso, *especificación formal*, supone suministrar un conjunto de *axiomas* que describen las operaciones en su aspecto *sintáctico* y *semántico*.

### 1.6.2.1. *Especificación informal de un TAD*

Consta de dos partes:

1. Detallar en los datos del tipo, los valores que pueden tomar.
2. Describir las operaciones, relacionándolas con los datos.

El formato que generalmente se emplea, primero especifica el nombre del **TAD** y los datos:

```
TAD nombre del tipo (valores y su descripción)
```

A continuación, cada una de las operaciones con sus argumentos, y una descripción funcional en lenguaje natural, con este formato:

```
Operación(argumentos).
```

**Descripción funcional**

A continuación se especifica, siguiendo esos pasos, el tipo abstracto de datos `Conjunto`:

***TAD Conjunto*** `(colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).`

Las operaciones básicas sobre conjuntos son las siguientes:

| | |
|---|---|
| *Conjuntovacio* | Crea un conjunto sin elementos. |
| *Añadir(Conjunto,elemento)* | Comprueba si el elemento forma parte del conjunto, en caso negativo se añade. La operación modifica al conjunto. |
| *Retirar(Conjunto, elemento)* | Si el elemento pertenezca al conjunto se retira. La operación modifica al conjunto. |
| *Pertenece(Conjunto,elemento)* | Verifica si el elemento forma parte del conjunto, en cuyo caso devuelve *cierto*. |
| *Esvacio(Conjunto)* | Verifica si el conjunto no tiene elementos, en cuyo caso devuelve *cierto*. |
| *Cardinal(Conjunto)* | Devuelve el número de elementos del conjunto. |
| *Union (Conjunto, Conjunto)* | Realiza la operación matemática unión de dos conjuntos. La operación devuelve un conjunto con los elementos comunes y no comunes de ambos. |

Se puede especificar más operaciones sobre conjuntos, todo dependerá de la aplicación que se quiera dar al **TAD**.

### *Norma*

La especificación informal de un **TAD** tiene como objetivo describir los datos del tipo y las operaciones según la funcionalidad que tienen. No sigue nor mas rígidas, simplemente indica, de forma comprensible, la acción que realiza cada operación.

#### 1.6.2.2. *Especificación formal de un TAD*

La especificación formal proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones. La descripción ha de incluir una parte de sintaxis, en cuanto a los tipos de los argumentos y el tipo del resultado, y una parte de semántica que detalla para unos valores particulares de los argumentos la expresión del resultado que se obtiene. La especificación formal ha de ser lo bastante *potente* para que cumpla el objetivo de verificar la corrección de la implementación del **TAD**.

El esquema que se sigue consta de una cabecera con el nombre del **TAD** y los datos:

```
TAD nombre del tipo (valores que toma los datos del tipo)
```

A continuación, la sintaxis de las operaciones que lista las operaciones mostrando los tipos de los argumentos y el tipo del resultado:

```
Operación(Tipo argumento, ...)  ->  Tipo resultado
```

Se continúa con la *semántica* de las operaciones. Ésta se construye dando unos valores particulares a los argumentos, a partir de los cuales se obtiene una expresión resultado. Éste puede tener referencias a tipos ya definidos, valores de tipo lógico o referencias a otras operaciones del propio **TAD**.

```
Operación(valores particulares argumentos) ⇒ expresión resultado
```

Al realizar la especificación formal siempre hay operaciones definidas por sí mismas, se denominan *constructores* del **TAD**. Mediante los constructores se generan todos los posibles valores del **TAD**. Normalmente, se elige como constructor la operación que inicializa (por ejemplo, `Conjuntovacío` en el `TAD Conjunto`), y la operación que añade un dato o elemento (operación común a la mayoría de los tipos abstractos de datos). Se acostumbra a marcar con un asterisco a las operaciones que son constructores.

A continuación, se especifica formalmente el `TAD Conjunto`. Para formar la expresión resultado se hace uso, si es necesario, de la sentencia alternativa **`si-entonces-sino`**.

```
TAD Conjunto(colección de elementos sin duplicidades, pueden estar en
cualquier orden, se usa para representar los conjuntos matemáticos con sus
operaciones).
```

### *Sintaxis*

```
*Conjuntovacio[8]                  ->   Conjunto
*Añadir(Conjunto, Elemento)[8]     ->   Conjunto
Retirar(Conjunto, Elemento)        ->   Conjunto
Pertenece(Conjunto, Elemento)      ->   boolean
Esvacio(Conjunto)                  ->   boolean
Cardinal(Conjunto)                 ->   entero
Union(Conjunto, Conjunto)          ->   Conjunto
```

### *Semántica* ∀ e1,e2 ∈ Elemento y ∀ C,D ∈ Conjunto

```
Añadir(Añadir(C, e1), e1)          ⇒  Añadir(C, e1)
Añadir(Añadir(C, e1), e2)          ⇒  Añadir(Añadir(C, e2), e1)
Retirar(Conjuntovacio, e1)         ⇒  Conjuntovacio
Retirar(Añadir(C, e1), e2)         ⇒  si e1 = e2 entonces
                                      C sino
                                      Añadir(Retirar(C, e2), e1)
Pertenece(Conjuntovacio, e1)       ⇒  falso
Pertenece(Añadir(C, e2), e1)       ⇒  si e1 = e2 entonces cierto sino
                                      Pertenece(C, e1)
Esvacio(Conjuntovacio)             ⇒  cierto
```

[8] El asterisco representa a un constructor del TAD. En este caso Conjuntovacio y Añadir que crean un conjunto vacío y añaden un elemento a un conjunto.

```
Esvacio(Añadir(C, e1))                ⇒  falso
Cardinal(Conjuntovacio)               ⇒  Cero
Cardinal(Añadir(C, e1))               ⇒  si Pertenece(C, e1) entonces
                                         Cardinal(C) sino 1 + Cardinal(C)
Union(Conjuntovacio, Conjuntovacio)   ⇒  Conjuntovacio
Union(Conjuntovacio, Añadir(C, e1))   ⇒  Añadir(C, e1)
Union(Añadir(C, e1), D)               ⇒  Añadir(Union(C, D), e1)
```

## 1.7. PROGRAMACIÓN ESTRUCTURADA

La programación orientada a objetos se desarrolló para tratar de paliar diversas limitaciones que se encontraban en anteriores enfoques de programación. Para apreciar las ventajas de la POO, es preciso constatar las limitaciones citadas y cómo se producen con los lenguajes de programación tradicionales.

C, Pascal y 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 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 un 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 resulta 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.

### 1.7.1. Datos locales y datos globales

En un programa procedimental, por ejemplo escrito en C, existen dos tipos de datos. *Datos locales* que son 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.

Otros tipos 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 éstos sean globales. En la Figura 1.3 se muestra la disposición de variables locales y globales en un programa procedimental.

Un programa grande (Figura 1.4) 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 diseñar la estructura del programa. En segundo lugar, el programa es difícil de modificar ya qué 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.

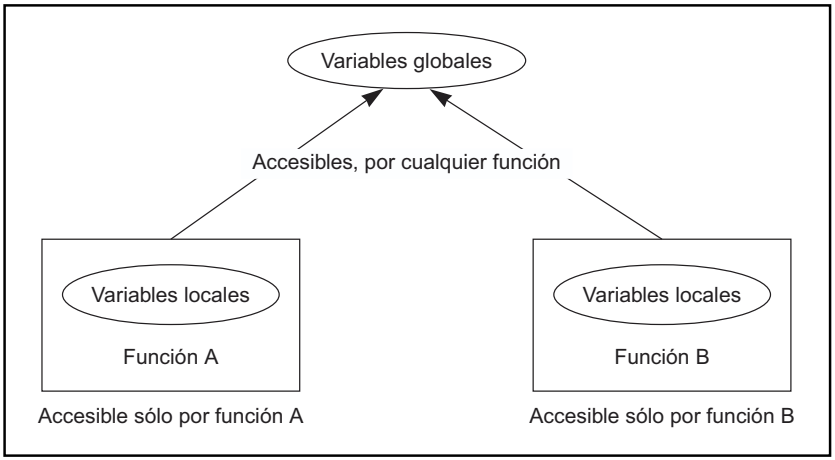



**Figura 1.3.** Datos locales y globales.

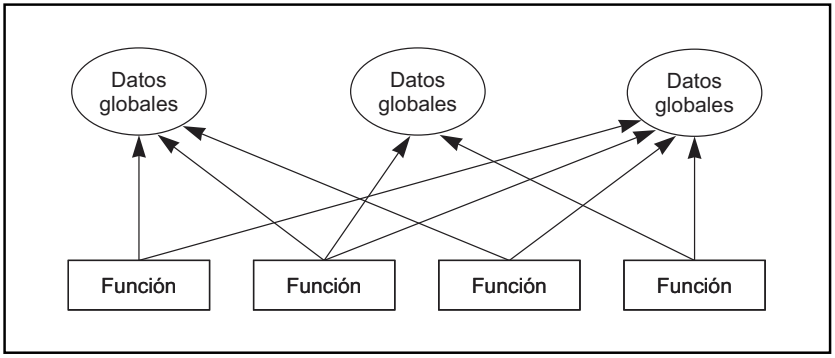



**Figura 1.4.** Un programa procedimental.

### 1.7.2. Modelado del mundo real

Otro 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, etcétera. En realidad, los atributos del mundo real tienen su equivalente en los datos de un programa; toman un valor específico, tal como 200 metros cuadrados, 20.000 dólares, cinco puertas, etc.

El **comportamiento** son las acciones 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, etc. 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.

## 1.8. 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 **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 las funciones miembro 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 1.5. La llamada a una función miembro de un objeto se denomina *enviar un mensaje* a otro objeto.

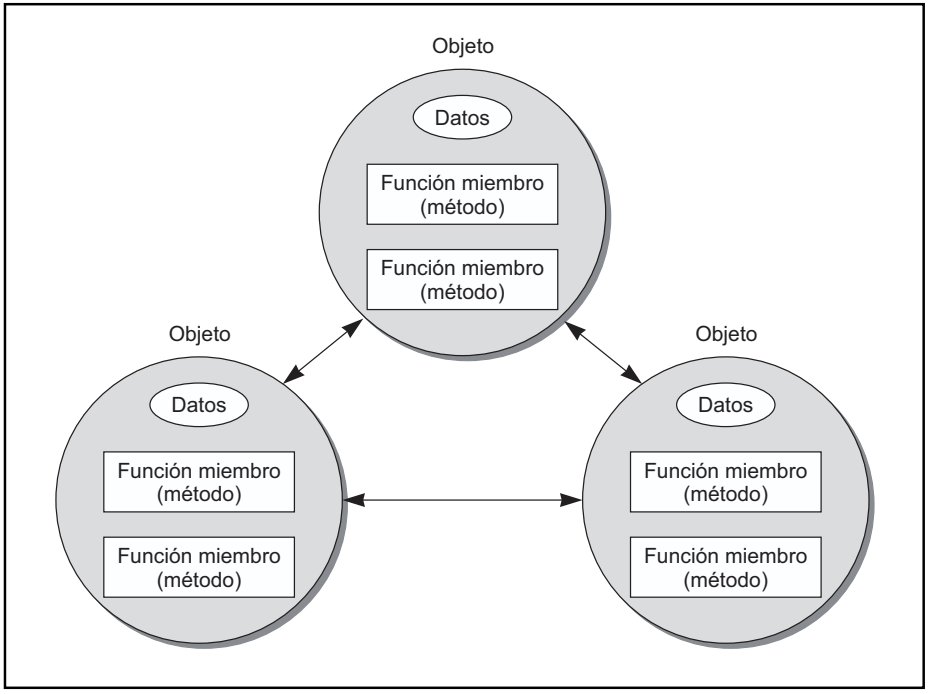



**Figura 1.5.** Organización típica de un programa orientado a objetos.

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

## 1.8.1. 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* o *encapsulamiento* de datos.
- *Ocultación* de datos.
- *Herencia*.
- *Polimorfismo*.

C++ soporta todas las características anteriores que definen la orientación a objetos, aunque hay numerosas discusiones en torno a la consideración de C++ como lenguaje orientado a objetos. La razón es que, en contraste con lenguajes tales como Smalltalk, Java o C#, C++ no es un lenguaje orientado a objetos puro. C++ soporta orientación a objetos pero es compatible con C y permite que programas C++ se escriban sin utilizar características orientadas a objetos. De hecho, C++ es un lenguaje *multiparadigma* que permite *programación estructurada*, *procedimental*, *orientada a objetos* y *genérica*.

### 1.8.2. 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 propiedad 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 es de interés *cómo* están organizados sino también *qué* se puede hacer con ellos. Es decir, las operaciones que forman la composición interna 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 1.7.** 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`.

### 1.8.3. 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, a veces se utilizan en contextos diferentes. En C++ no es lo mismo, los datos internos están protegidos del exterior y no se pueden 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 los 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.

En C++, 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 conjunto de objetos. 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, etcétera. 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, Unix o Vista. Uno de los beneficios reales de C++ es que permite la reutilización y adaptación de códigos existentes y ya bien probados y depurados.

### 1.8.4. 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 intenta descubrir e implementar los objetos que juegan un rol en el dominio del problema y en consecuencia del 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 dominio son relevantes. Un carro puede ser ensamblado por 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 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 1.6.

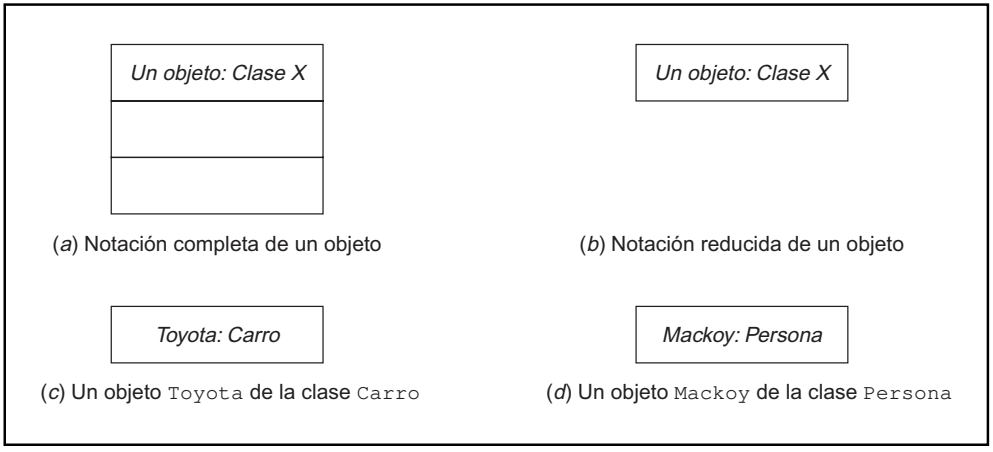



**Figura 1.6.** Representación de objetos en UML (Lenguaje Unificado de Modelado).

### 1.8.5. Clases

En POO los objetos son miembros o instancias 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 describe muchos objetos y es preciso definirla, aunque su definición no implica creación de objetos (Figura 1.7).

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 de rock"`. Un objeto concreto, `Juanes` o `Carlos Vives`, son *instancias* de la clase `"músicos de rock"`.

En **C++** una clase es una estructura de dato o tipo de dato que contiene funciones (métodos) como miembros y datos. 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.

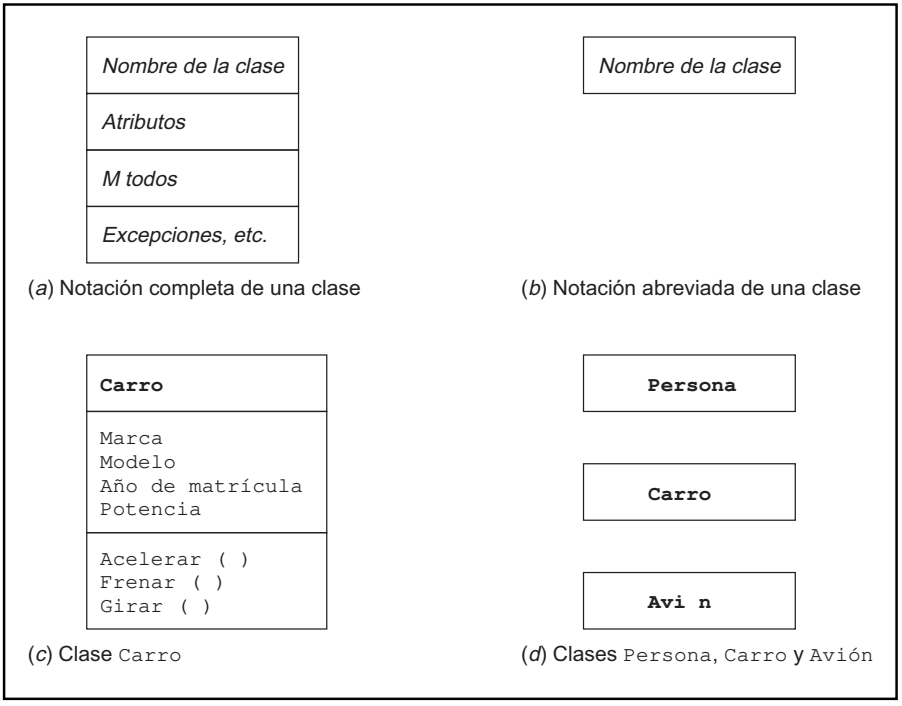



**Figura 1.7.** Representación de clases en UML.

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 (Figura 1.8).

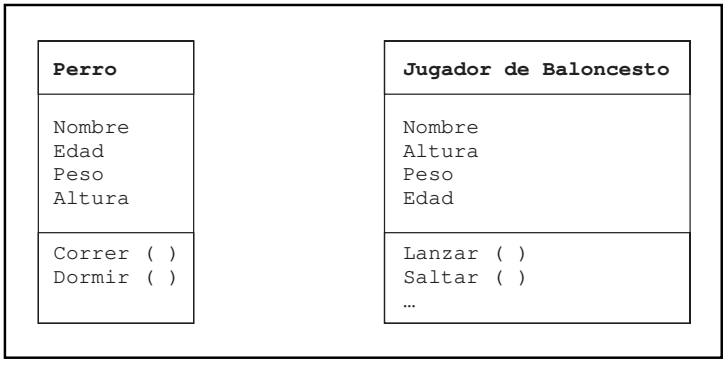



**Figura 1.8.** Representación de clases en **UML** con atributos y métodos.

### 1.8.6. 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 un **`subclase`** 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.

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 1.9 se muestran clases pertenecientes a una jerarquía o herencia de clases.

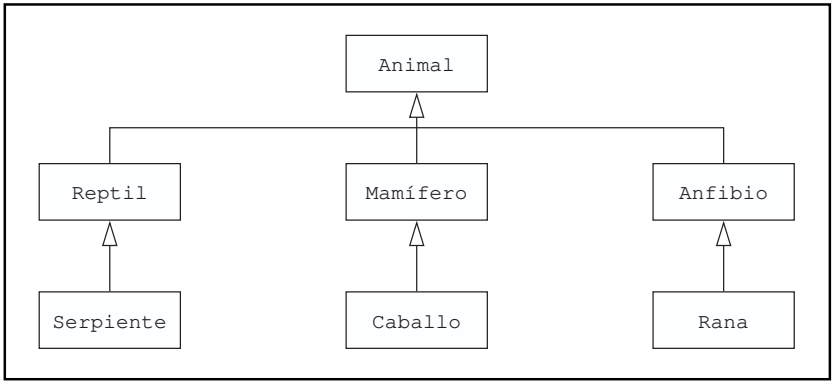



**Figura 1.9.** Herencia de clases en UML.

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.

### 1.8.7. 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*[9] 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 clases 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, propiedades comunes de varias clases sólo necesitan ser implementadas una vez y sólo necesitan modificarse una vez si es necesario. Tercero, el concepto de abstracción de la funcionalidad común está soportada.

### 1.8.8. 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 practica, 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.

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 (coches), de marcas diferentes, de modelos diferentes, de tipos diferentes, potencias diferentes, etc. Constituyen una

[9] El término proviene del concepto ingles *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.

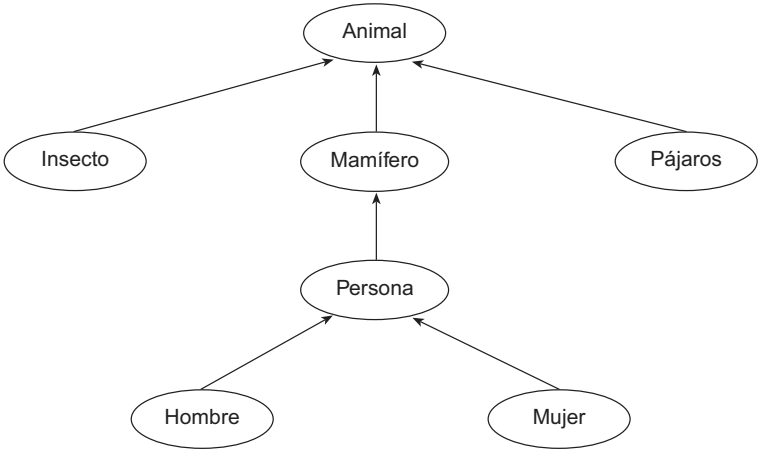
clase o colección heterogénea de carros (coches). Supongamos que se ha de realizar una operación común `"cambiar los frenos del carro"`. La operación a realizar es la misma, incluye los mismos principios; sin embargo, dependiendo del coche, en particular, la operación será muy diferente, incluirá diferentes 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.

---

**EJEMPLO 1.8.** Definir una jerarquía de clases para: animal, insecto, mamíferos, pájaros, persona hombre y mujer. Realizar un definición en pseudocódigo de las clases.

Las clases de objeto `Mamífero`, `Pájaro` e `Insecto` se definen como *subclases* de `Animal`; la clase de objeto `Persona`, como una subclase de `Mamífero`, y un `Hombre` y una `Mujer` son subclases de `Persona`.




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

```
clase Animal
  atributos (propiedades)
    string: tipo;
    real: peso;
    (...algun tipo de habitat...):habitat;
  operaciones
```

```
    crear()→ criatura;
    predadores(criatura)→ fijar(criatura);
    esperanza_vida(criatura) → entero;
    ...
fin criatura

clase Insecto hereda Animal
    atributos (propiedades)
      cadena: nombre
    operaciones
    ...
fin Insecto

clase Mamifero hereda Animal;
    atributos (propiedades)
      real: periodo_gestacion;
    operaciones
      ...
fin Mamifero

clase Pajaro hereda Animal
    atributos (propiedades)
      cadena: nombre
      entero: Alas
    operaciones
    ...
fin Pajaro

clase Persona hereda Mamifero;
    atributos (propiedades)
      string: apellido, nombre;
      date: fecha_nacimiento;
      pais: origen;
fin Persona

clase Hombre hereda Persona;
    atributos (propiedades)
      mujer: esposa;
      ...
    operaciones
      ...
fin Hombre

clase Mujer hereda Persona;
    propiedades
      esposo: hombre;
      string: nombre;
      ...
fin Mujer
```

## 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*.

El sistema más idóneo para resolver un problema es descomponerlo en módulos más sencillos y luego, mediante diseños descendentes y refinamiento sucesivo, llegar a módulos fácilmente codificables. Estos módulos se deben codificar con las estructuras de control de programación estructurada.

Los tipos abstractos de datos (**TAD**) describen un conjunto de objetos con la misma representación y comportamiento. Los tipos abstractos de datos presentan una separación clara entre la interfaz externa de un tipo de datos y su implementación interna. La implementación de un tipo abstracto de datos está oculta. Por consiguiente, se pueden utilizar implementaciones alternativas para el mismo tipo abstracto de dato sin cambiar su interfaz.

La especificación de un tipo abstracto de datos se puede hacer de manera *informal*, o bien, de forma más rigurosa, una especificación *formal*. En la especificación *informal* se describen literalmente los datos y la funcionalidad de las operaciones. La especificación formal describe los datos, la sintaxis y la semántica de las operaciones considerando ciertas operaciones como axiomas, que son los constructores de nuevos datos. Una buena especificación formal de un tipo abstracto de datos debe poder verificar la *bondad* de la implementación.

Las características más importantes de la programación orientada a objetos son: abstracción, encapsulamiento, herencia, polimorfismo, clases y objetos.

Una **clase** es un tipo de dato definido por el usuario que sirve para representar objetos del mundo real. Un objeto de una clase tiene dos componentes: un conjunto de atributos y un conjunto de operaciones. Un objeto es una instancia de una clase. Herencia es la relación entre clases que se produce cuando una nueva clase se crea utilizando las propiedades de una clase ya existente. Polimorfismo es la propiedad por la que un mensaje puede significar cosas diferentes dependiendo del objeto que lo recibe.

## EJERCICIOS

**1.1.** Diseñar el diagrama de flujo de algoritmo que visualice y sume la serie de números 4, 8, 12, 16, ..., 400.

**1.2.** Diseñar un diagrama de flujo y 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 imprimir 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.

**1.3.** Escribir un algoritmo que encuentre el salario semanal de un trabajador, dada la tarifa horaria y el número de horas trabajadas diariamente.

**1.4.** Escribir un diagrama de flujo y 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 "`y`", una "`t`" y una "`e`".

**1.5.** Dibujar un diagrama jerárquico de objetos que represente la estructura de un coche (carro).

**1.6.** Dibujar diagramas de objetos que representen la jerarquía de objetos del modelo `Figura` geométrica.

**1.7.** Construir el **TAD** Natural para representar a los números naturales, con las operaciones: `Cero`, `Sucesor`, `EsCero`, `Igual`, `Suma`, `Antecesor`, `Diferencia` y `Menor`. Realizar la especificación informal y formal considerando como constructores las operaciones `Cero` y `Sucesor`.

**1.8.** Diseñar el **TAD** Bolsa como una colección de elementos no ordenados y que pueden estar repetidos. Las operaciones del tipo abstracto: `CrearBolsa`, `Añadir` (un elemento), `BolsaVacia` (verifica si tiene elementos), `Dentro` (verifica si un elementos pertenece a la bolsa), `Cuantos` (determina el número de veces que se encuentra un elemento), `Union` y `Total`. Realizar la especificación informal y formal considerando como constructores las operaciones `CrearBolsa` y `Añadir`.

**1.9.** Diseñar el **TAD** `Complejo` para representar a los números complejos. Las operaciones que se deben definir: `AsignaReal` (asigna un valor a la parte real), `AsignaImaginaria` (asigna un valor a la parte imaginaria), `ParteReal` (devuelve la parte real de un complejo), `ParteImaginaria` (devuelve la parte imaginaria de un complejo), `Modulo` de un complejo y `Suma` de dos números complejos. Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

**1.10 .** Diseñar el tipo abstracto de datos Matriz con la finalidad de representar matrices matemáticas. Las operaciones a definir: `CrearMatriz` (crea una matriz, sin elementos, de `m` filas por `n` columnas), `Asignar` (asigna un elemento en la fila *i* columna *j*), `ObtenerElemento` (obtiene el elemento de la fila *i* y columna *j*), `Sumar` (realiza la suma de dos matrices de las mismas dimensiones), `ProductoEscalar` (obtiene la matriz resultante de multiplicar cada elemento de la matriz por un valor). Realizar la especificación informal y formal considerando como constructores las operaciones que desee.