Capítulo 1

IDEAS GENERALES SOBRE PROGRAMACIÓN

1. INTRODUCCIÓN

En la vida diaria se realizan diferentes actos que ponen en contacto a las personas con el mundo de los ordenadores. Actividades como usar un cajero automático, pagar con tarjeta de crédito, reservar billetes de tren o avión, utilizar el teléfono, enviar una carta, hacer la compra por Internet, hacen un uso cada vez mayor de las tecnologías de la información y la comunicación.

Los ordenadores se encuentran plenamente integrados en la sociedad actual y cada vez, en mayor medida. Tal es así, que algunos autores apuestan por llamar a esta época la *"Era de la Información y la comunicación"*.

En el cine y la literatura de ciencia-ficción se ha utilizado mucho el argumento del ordenador que piensa por sí mismo, o el que se rebela contra sus amos humanos, recuérdese *2001: Una odisea del espacio*, primera película que trata el tema. Sin embargo, con la tecnología actual, los ordenadores se limitan a llevar a cabo las instrucciones que se les dan, sin tener ningún tipo de iniciativa propia.

En este capítulo, se van a desarrollar conceptos básicos que le introducirán en el fascinante mundo de la informática a través de la construcción de programas que hagan posible que los ordenadores realicen los trabajos que se deseen.

2. SISTEMA INFORMÁTICO

No es necesario saber mucho sobre ordenadores para poder utilizarlos como una eficaz herramienta de trabajo. Sin embargo, al conocer los elementos que los integran se puede comprender mejor lo que ocurre cuando se utilizan.

Un ordenador es un **dispositivo electrónico programable capaz de almacenar y procesar información**. Pero por sí sólo, no es capaz de hacerlo, necesita todo un sistema a su alrededor para realizar estas tareas.

Un **sistema informático**, término utilizado para referirse al **conjunto de recursos que son necesarios para la elaboración y el uso de aplicaciones informáticas**, está sostenido por los tres elementos básicos siguientes:

- El elemento **físico,** conocido con el nombre de **hardware.**
- El elemento **lógico,** conocido con el nombre de **software.**
- El elemento **humano**, conocido como **personal informático**.

El hardware queda fijado cuando se diseña y fabrica, por tanto, no se modifica hasta que lo decida el interesado y siempre que el diseño lo permita, pero el software es fácilmente modificable por el usuario, característica que hace a los ordenadores o computadoras tan versátiles y poderosos.

Es evidente que entre todos estos elementos, o entre cualquier tipo de sistemas independientes o autónomos, deben existir medios que permitan la conexión y comunicación entre ellos. A estas vías de enlace entre elementos se las conoce como **interfaz**. Por ejemplo, un usuario y un cajero automático pueden considerarse "sistemas independientes", de manera que necesitan un protocolo de comunicación entre ambos. La interfaz entre el usuario y el cajero se suministra mediante dispositivos como el teclado y la pantalla de dicho cajero y las instrucciones de uso de estos dispositivos, que permitirán una adecuada comunicación entre ambos sistemas.

2.1. El elemento físico (Hardware)

El **hardware** engloba a todos aquellos elementos con entidad física que forman parte del sistema informático, es decir, son palpables, materiales. Son objetos tales como los componentes del propio ordenador, dispositivos externos (por ejemplo, la impresora, el teclado, los cables de conexión entre elementos o las unidades donde se guarda la información, etc.)

Del ordenador se desea que almacene información y la procese. Para ello, se necesitan los elementos capaces de obtener esta información, los que están preparados para guardarla, aquellos dedicados a manipularla y, por último, otros cuya misión sea hacerla llegar al usuario una vez elaborada. Se necesitan mecanismos que aseguren la comunicación entre la persona y la máquina.

La figura 1.1 representa un esquema básico de los componentes elementales del hardware de la mayoría de los ordenadores.

- **Unidad Central de Proceso**, conocida en el argot informático como **CPU (C**entral **P**rocess **U**nit**)** o **procesador**. Es el principal elemento del ordenador, su cerebro. Su trabajo consiste en coordinar y ejecutar todas las instrucciones que se leen de la memoria RAM. Sus elementos principales son:

- **UAL** (**Unidad Aritmético-Lógica**) o **ALU** (**A**rithmetic-**L**ogical **U**nit). Componente de la **CPU** cuya misión consiste en realizar todas las operaciones aritméticas elementales (suma, resta, multiplicación, división) y las operaciones lógicas (por ejemplo, la comparación de dos valores).

- **Unidad de Control.** Componente de la **CPU** encargado de controlar las acciones del resto de las unidades, interpretando y ejecutando las instrucciones en la secuencia adecuada.

- **Registros del microprocesador.** Desde el punto de vista del programador, es interesante conocer la existencia de bancos de memoria de alta velocidad muy especializados, conocidos como registros del microprocesador, donde se almacenan algunos datos e instrucciones de un programa mientras se ejecuta.

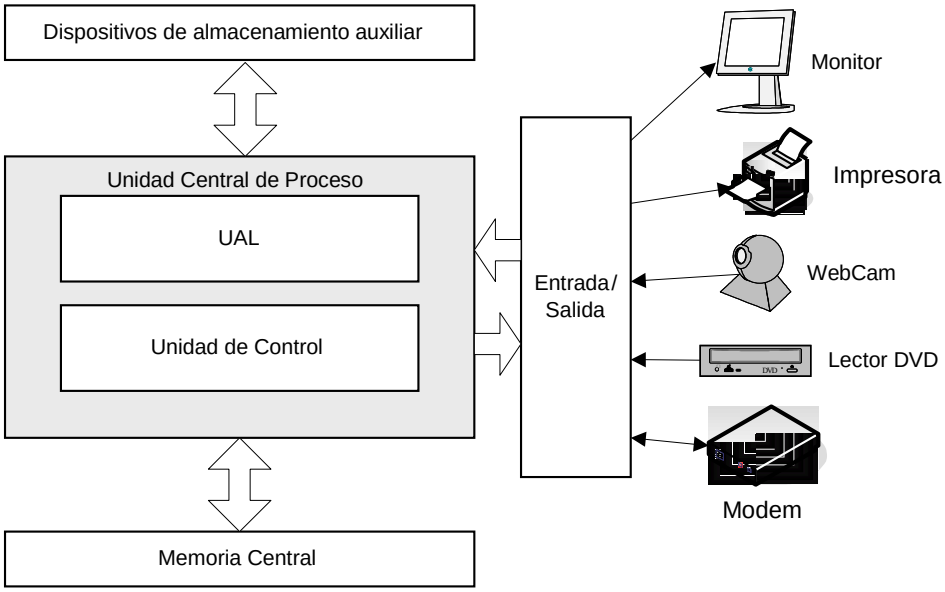


Figura 1.1. Componentes del hardware

Generalmente, la ALU dispone de los registros de propósito general, que sirven para tareas como procesos de acumulación, contador de índices de bucles, transferencias de datos, o manipulación de bits.

La UC también dispone de registros como: **Contador de programa** que contiene la dirección de la siguiente instrucción que debe ejecutarse, **Registro de Estado** que guarda información sobre el estado actual de las operaciones que se están realizando, **Registro de instrucción** que contiene la instrucción que se está ejecutando y **Registro de puntero de pila** que mantiene la dirección necesaria para las operaciones de pila.

Existen otros registros que guardan las direcciones de los distintos segmentos de memoria RAM (tipo de memoria central. Se hablará de ella en el capítulo dos) utilizados durante el proceso de ejecución de los programas: segmento de código, de datos, de pila y segmento extra. Los **segmentos** son áreas de memoria utilizadas para almacenar unos datos concretos.

- **Memoria Central**, también llamada **memoria principal** o **memoria interna.** Es un dispositivo de almacenamiento de información. Existen dos tipos principales de memoria central: la memoria ROM, de sólo lectura, donde se almacena software del sistema de forma permanente y la memoria RAM para almacenamiento temporal de información. Desde el punto de vista del programador, esta última es la más interesante. En ella se guardan todos los datos, tanto de entrada como resultados intermedios y definitivos de las operaciones realizadas durante la ejecución de los programas, así como las instrucciones que forman los propios programas, utilizando para ello los distintos segmentos ya nombrados.

- El ordenador necesita obtener la información con la que trabajará de algún sitio y, además, poder comunicar los resultados de sus operaciones. Para ello dispone de las **Unidades de Entrada/Salida**, a las que se conectan todos los demás aparatos que transmiten la información entre el ordenador y su entorno.

- **Dispositivos o periféricos de entrada**. Son los componentes hardware encargados de introducir la información desde el exterior para su posterior proceso. Un ejemplo de dispositivo que se utiliza para la entrada es el teclado, conocido comúnmente como *dispositivo estándar de entrada*.

- **Dispositivos o periféricos de salida**. Son los componentes hardware encargados de hacer llegar al exterior los resultados procedentes de los procesos realizados en el sistema informático. Un ejemplo de dispositivo utilizado para la salida es el monitor, conocido comúnmente como *dispositivo estándar de salida*.

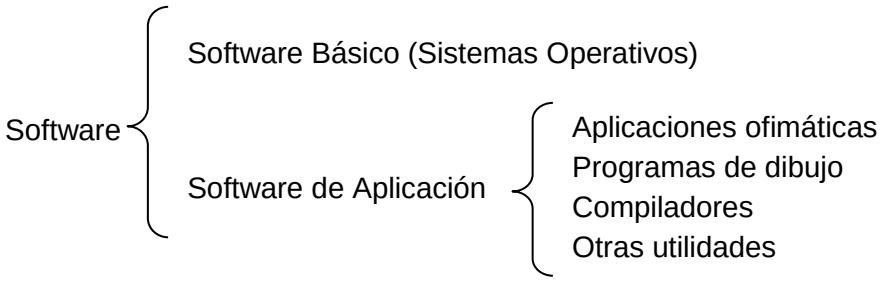
- Existen, además, dispositivos que permiten la comunicación en ambos sentidos, y conocidos como **dispositivos de entrada y salida**, por ejemplo el módem.

- **Dispositivo de almacenamiento auxiliar**, a veces llamado **dispositivo de almacenamiento secundario** o **memoria auxiliar**. Son unidades de almacenamiento masivo de información mucho más lentas que la memoria central y con una capacidad mayor. Estas memorias son utilizadas para guardar datos y programas de forma permanente a diferencia de la memoria RAM, que se borra cuando se apaga el ordenador. Cuando la CPU requiera estos datos, deberán transferirse desde el dispositivo de almacenamiento auxiliar a la memoria central para su proceso. El principal elemento de almacenamiento auxiliar de información es el disco duro.

2.2. El elemento lógico (Software)

El software de un sistema informático es el conjunto de elementos lógicos, programas, datos, información, etc., que hacen posible el uso y funcionamiento de los ordenadores.

Se puede decir que los elementos básicos del software son los datos y las órdenes o instrucciones. Si el software forma parte del sistema informático, deberá almacenarse en un soporte físico tal como la memoria central o la memoria secundaria.



Se puede clasificar el elemento lógico como **software básico** y **software de aplicación**.

El software básico, o software de sistema, es el conjunto de programas imprescindibles para gestionar el hardware del sistema informático. Este conjunto de programas es conocido con el nombre de **Sistema Operativo**.

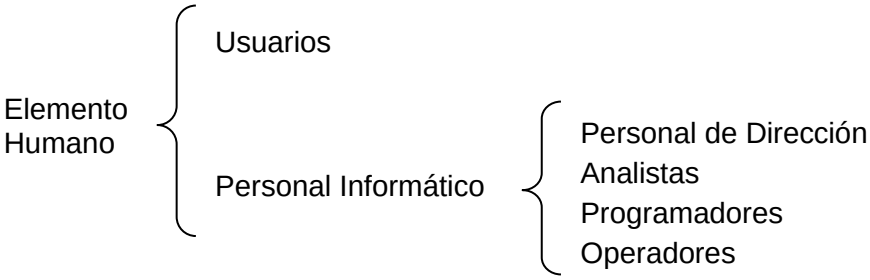
Sistema Operativo es la herramienta lógica del sistema informático que controla el funcionamiento del equipo físico y gestiona todos los recursos haciendo transparente al usuario las características físicas de la máquina, facilitando de este modo su uso y mejorando su eficacia. Son ejemplos de sistemas operativos: MS-DOS, UNIX, Linux, OS/2, OS-400, Windows XP, Mac OS-X.

El software de aplicación está formado por un conjunto de programas diseñados con el objetivo de que los ordenadores realicen trabajos específicos, facilitando al usuario la realización de sus actividades. Son aplicaciones tales como herramientas ofimáticas (Microsoft Office, Open Office), programas de dibujo (Corel Draw, Microsoft Visio), programas para la realización de nóminas, o para llevar la contabilidad de la empresa (Contaplus, Contawin). Pertenecen también a este grupo de software de aplicación las herramientas de programación para los distintos lenguajes, necesarias para la realización de programas, es decir, para crear nuevo software, que serán comentados en este mismo capítulo.

2.3. El elemento humano

Se puede decir que éste es el elemento más importante del sistema informático y la misma razón de ser de la informática: hacer la vida más cómoda y agradable a los seres humanos. Sin las personas, los ordenadores serían total y absolutamente inútiles.

En la sociedad actual, todos en algún momento del día nos vemos relacionados con algún sistema de proceso de información, por ejemplo, usar el cajero automático, o mandar un correo electrónico. En general, se llama **usuario** a este grupo de personas que utilizan los ordenadores en última instancia, usando programas de utilidades más o menos complejos creados por otras personas, con el objetivo de ayudarse en alguna actividad. Y se conoce como **personal informático** al conjunto de personas que trabajan para garantizar el correcto funcionamiento de los ordenadores, es decir, en lugar, o además, de utilizarlos como herramienta, son el objeto de su trabajo.



Este conjunto humano se puede clasificarse según el trabajo que realizan en:

- **Personal de dirección.** Tiene a su cargo la responsabilidad de dirigir y coordinar el Servicio de Informática de una empresa, conocido tradicionalmente como **Centro de Proceso de Datos (CPD)** y, más actualmente, como **Departamento de Servicios de Información.**

- **Analistas**. Hacen tareas de análisis, buscando e identificando las necesidades y requerimientos de información y realizando el diseño general de las aplicaciones.

- **Programadores**. Realizan aplicaciones y algoritmos y traducen a un lenguaje de programación los diseños elaborados por los analistas.

- **Operadores**. Realiza determinados trabajos de puesta en marcha y control de los equipos para garantizar su correcto funcionamiento.

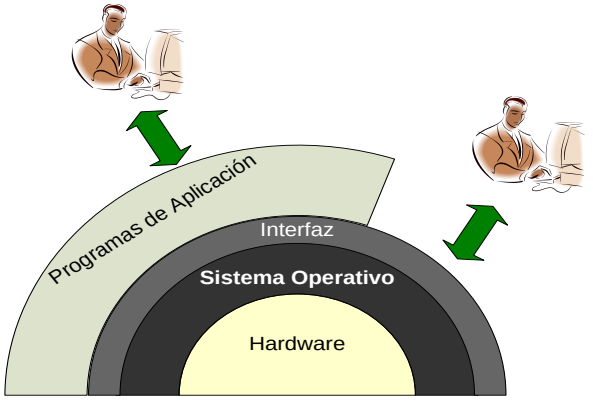


Figura 1.2. Sistema informático

3. ¿QUÉ SIGNIFICA PROGRAMAR?

El cerebro, ordenador del pensamiento y el comportamiento humano, es el encargado de que todos los actos se efectúen en secuencias lógicas. Desde pequeños se va aprendiendo a realizar tareas y a reaccionar ante distintas situaciones: aprendemos a andar, a comer, a observar e, incluso, a relacionarnos con los demás, a esperar de ellos ciertas conductas que también ellos esperan de nosotros. Nuestras vidas están organizadas obedeciendo a un determinado orden. Que todo ocurra como esperamos nos hace sentir seguros, tranquilos y confiados.

Muchos de los actos que se realizan diariamente, se hacen de un modo absolutamente automático, casi inconscientemente, pero lejos de ser éste un hecho desafortunado, permite al cerebro poder evadirse de tareas rutinarias y tener sus recursos disponibles para la realización de otras tareas compatibles. Por ejemplo, no es necesario que pensemos conscientemente en cada movimiento a realizar para subir una escalera:

```
levantar un pie hasta la altura del primer escalón,
adelantarlo hasta apoyarlo en dicho escalón,
levantar el otro pie hasta la altura del escalón siguiente,
repetir los dos pasos anteriores mientras queden escalones.
```

Este proceso, que se realiza mecánicamente, pone en funcionamiento una gran cantidad de recursos del organismo: los ojos observan la altura del escalón, advierten si quedan escalones por subir y mandan la información al cerebro que, a su vez, envía impulsos a través del sistema nervioso a los músculos para que reaccionen de la forma esperada. Sin embargo, toda esta agitación no impide realizar a la vez otras tareas compatibles. No podríamos bajar la escalera simultáneamente a la subida, pero sí podríamos transportar en nuestros brazos una pila de libros y, además, ocupar nuestra mente en la ardua tarea de pensar dónde vamos a colocarlos.

Todas estas funciones que ahora se realizan de manera automática han tenido en todos nosotros un proceso de aprendizaje basado en la repetición de secuencias lógicas, es decir, hemos programado nuestro cerebro para que las realice. También las acciones realizadas de manera consciente se desarrollan obedeciendo a un proceso ordenado de pasos: conducir un vehículo, cantar, estudiar, hablar con los demás. Nuestra jornada se desarrolla en un casi perfecto orden, básico para la convivencia. Se podría decir que la sociedad se fundamenta en cierta ordenación adquirida a través de procesos programados o programables.

Programar significa **planificar y concretar la secuencia de órdenes concisas para la realización de una tarea**. Cuando informáticamente se habla de programar, se está pensando en las órdenes que se deben dar al ordenador para que lleve a cabo una tarea concreta. Esto se hace a través de lo que se denomina **programa**, que no es más que el conjunto de secuencias lógicas o conjunto de instrucciones, en un orden específico, que se dan al ordenador para que realice una determinada tarea.

4. PARADIGMAS DE PROGRAMACIÓN

Paradigma es una palabra derivada de las griegas "ejemplo" y "mostrar", que significa "un ejemplo que sirve como norma" (DRAE). A lo largo de la corta historia de la informática, poco más de medio siglo, se han establecido diferentes modelos para el desarrollo informático de problemas del mundo real.

Un **paradigma de programación** es un modelo básico para el diseño y la implementación de programas. Un modelo que con unas normas específicas tales como: estructura modular, alta reusabilidad, fuerte cohesión, etc., determina el proceso de diseño y la estructura final de un programa.

La mayoría de los programadores están familiarizados con el paradigma mas difundido, el de la **programación procedimental**, cuya característica fundamental es la secuencia de acciones realizadas paso a paso que deben seguirse para la resolución de un problema, es decir, indica cómo obtener la solución, Sin embargo, existen una gran variedad de ellos en razón de alguna particularidad metodológica o funcional, como por ejemplo:

- **Programación lógica**. Basada en reglas lógicas y asertos. Define un entorno de programación de tipo conversacional, deductivo y no determinista.
- **Programación heurística**. Aplica reglas de "buena lógica" para resolver el problema.
- **Paradigmas basados en el flujo de datos**, **basados en restricciones**, etc.
- **Paradigmas procedimentales**. Describen paso a paso cómo se obtiene la solución del problema.

En este curso, se va a estudiar el paradigma procedimental **orientado a objetos**, denominado también con efectos laterales que incorpora tipos abstractos de datos.

5. FASES DE LA PROGRAMACIÓN

Desde que se piensa en informatizar una tarea hasta que pueda ser realizada por un ordenador, deben darse determinados pasos que se resuelven aplicando un método.

Se puede definir metodología como filosofía que engloba un conjunto de reglas, procedimientos, técnicas y herramientas que usan los creadores de software para elaborar programas de una manera sistemática. Dividiremos este proceso de creación en diferentes fases: fase de resolución del problema, fase de implementación y fase de explotación y mantenimiento.

Realmente las fases que describimos a continuación son las etapas típicas en el ciclo de vida del software, pero son extrapolables cuando se trata de solucionar problemas sencillos, como es habitual mientras se aprende a programar.

5.1. Fase 1: Resolución del problema

Consiste en determinar claramente el problema que se desea mecanizar y buscar una buena solución que permita resolverlo.

Se puede subdividir en las siguientes etapas o subfases:

- **Análisis**. El análisis indica la especificación de requisitos del cliente (el profesor, o el enunciado del problema). Se trata de comprender, determinar y definir perfectamente el problema.

- **Solución General o Diseño**. En esta fase se convierte la especificación de requisitos establecida en la fase de análisis en un diseño, más o menos detallado, estableciendo el comportamiento paso a paso, o la secuencia lógica de instrucciones, capaz de resolver el problema planteado en un tiempo finito. Esto es lo que se denomina "realizar un algoritmo". Estilos distintos, de distintos programadores a la hora de obtener la resolución del problema, darán lugar a algoritmos diferentes igualmente válidos.

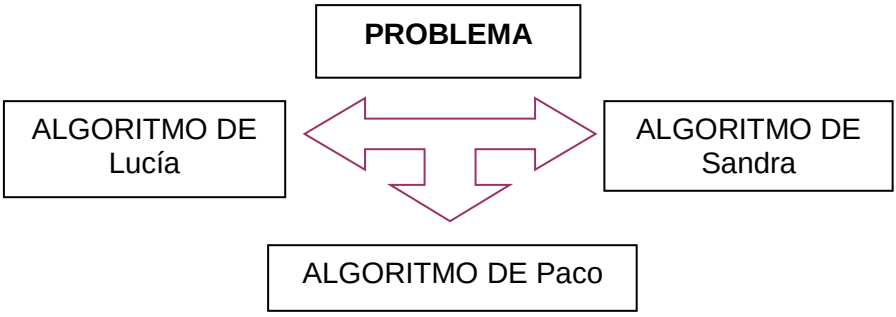


Figura 1.3. Solución General

Diariamente, se usan numerosos algoritmos hablados o escritos que no son más que la descripción de la secuencia lógica de acciones para la realización de una determinada tarea: instrucciones para poner en marcha la lavadora, usar el vídeo doméstico, enviar un fax o realizar un sabroso plato, son algunos de ellos.

Por ejemplo, cuando se prepara una tortilla de patatas, se realiza un proceso paso a paso que podría desarrollarse con el siguiente algoritmo:

Suponemos que:

- Disponemos de todos los ingredientes: patatas, huevos, aceite y sal
- Disponemos de todos los utensilios: sartén, plato, espumadera y tenedor.
- Sabemos: pelar, cortar, freír, batir, Añadir,...

Realización:

1. Pelar las patatas.
2. Cortarlas a cuadraditos.
3. Freírlas, no demasiado, en abundante aceite (de oliva).
4. Batir una cantidad de huevos suficiente.
5. Añadir las patatas fritas y escurridas.
6. Poner sal al gusto.
7. Echar la mezcla en una sartén untada de aceite.
8. Dejar que se cuaje unos minutos.
9. Dar la vuelta a la mezcla con ayuda de un plato.
10. **Repetir desde el paso 8 hasta que la tortilla haya cuajado a nuestro gusto.**

Debe observar que sin la frase "hasta que haya cuajado a nuestro gusto" en el paso 10, estaríamos dejando cuajar y dando la vuelta a la tortilla indefinidamente, se quemaría y ocurriría un auténtico desastre. Esta situación que nunca termina es conocida como bucle infinito. En este caso, nuestro algoritmo incumpliría el requisito establecido anteriormente "resolver un problema en un tiempo finito". Esto puede parecer una cuestión trivial en la vida cotidiana, porque el que va a llevar a cabo el algoritmo es un ser inteligente (al menos, en la mayoría de los casos). Sin embargo, tiene una gran importancia cuando se trata de programar una máquina, ya que como se ha dicho antes, el ordenador carece de iniciativa y realizará las operaciones de forma automática, exactamente tal como se le ordene.

- **Prueba o Traza:** Realizar un buen diseño es fundamental antes de pasar a la fase de implementación del algoritmo, por lo que es imprescindible asegurarse que realmente tenemos una buena solución. Esto se puede llevar a cabo realizando la traza del diseño convenientemente. Consiste en seguir paso a paso las instrucciones del algoritmo con datos concretos, bien sea con inspección ocular del código escrito, a mano o con ayuda de alguna herramienta informática. Su objetivo es comprobar si realmente la solución adoptada como general, en la fase de diseño, resuelve el problema. Si se detectan errores, volveremos a la fase de análisis, modificando el algoritmo e, incluso, adoptando una nueva alternativa. Sólo cuando el algoritmo satisfaga completamente los requerimientos y objetivos especificados en la fase de análisis se pasará a la fase siguiente.

5.2. Fase 2: Implementación

Consiste en construir mediante un lenguaje de programación, una solución específica libre de errores a partir de la solución general de la etapa anterior.

También, se puede dividir esta fase en los siguientes pasos:

- **Solución específica**. Cuando el algoritmo, descrito en la fase anterior, satisface totalmente el problema, deberá escribirse con un lenguaje de programación, dando lugar a la solución específica o programa. La traducción del algoritmo a un lenguaje específico de programación se conoce como codificación. La codificación en lenguajes distintos (PASCAL, C, JAVA) dará lugar a soluciones o programas diferentes. Cada traducción diferente del algoritmo produce implementaciones diferentes igualmente válidas.

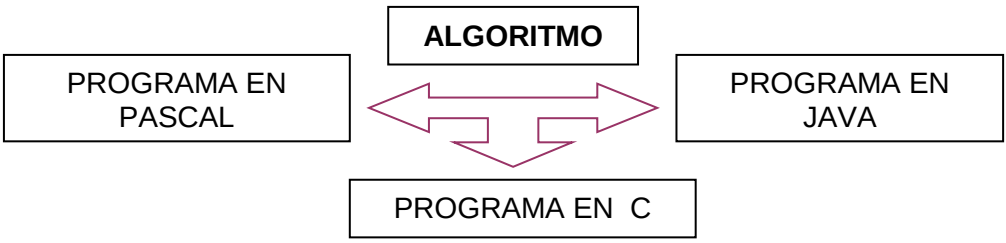


Figura 1.4. Solución dependiente del lenguaje

Puede incluso que varias personas traduzcan el mismo algoritmo al mismo lenguaje de programación y resulten implementaciones diferentes; puesto que los lenguajes de programación, de forma similar al lenguaje natural, permiten cierta flexibilidad en su uso. Así, al igual que las personas tenemos un estilo particular en la escritura en lenguaje natural, también los programadores desarrollamos con la práctica un estilo diferente en la escritura de algoritmos en lenguajes de programación.

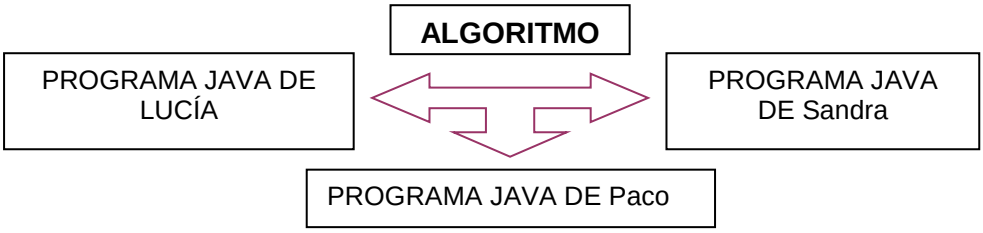


Figura 1.5. Solución dependiente del lenguaje y del programador

La experiencia en diseñar y codificar programas nos hará desarrollar un estilo propio. A lo largo de este libro, se ofrecerán recomendaciones sobre los buenos estilos de programación.

- **Prueba de ejecución y validación**. En esta fase, se ejecuta realmente el programa usando distintos datos de prueba, para verificar que el programa responde a los requerimientos establecidos o se detectan nuevas incorrecciones, si se manejan bien las situaciones de error, si la interfaz es amigable. En definitiva, se trata de poner en "aprietos" al programa para ver su respuesta en los casos más difíciles. Mientras los resultados no sean los deseados no se pasará a la fase siguiente. Existen dos tipos de pruebas en programación modular y en la orientada a

objetos: pruebas unitarias y pruebas de integración. Es decir, pruebas de funcionalidad de cada módulo o cada clase, y pruebas para observar que los módulos o las clases funcionan correctamente cuando se usan juntos en un programa de aplicación en concreto.

5.5. Fase 3: Explotación

Una vez instalado el programa en el sistema informático, los usuarios podrán usarlo mientras sea útil; se dice entonces que el programa está en explotación.

Durante el tiempo que el programa permanece en uso, deberá someterse a evaluaciones periódicas, modificándolo si fuera aconsejable para adaptarlo o actualizarlo de acuerdo con las necesidades surgidas en ese tiempo, o bien para corregir fallos no detectados anteriormente. Este proceso se conoce como fase de mantenimiento. Precisamente, porque los programas se hacen para ser usados muchas veces y habrá que mantenerlos a lo largo de su vida, es importante añadir una documentación adecuada, que haga fácil al programador la comprensión, uso y modificación de dichos programas.

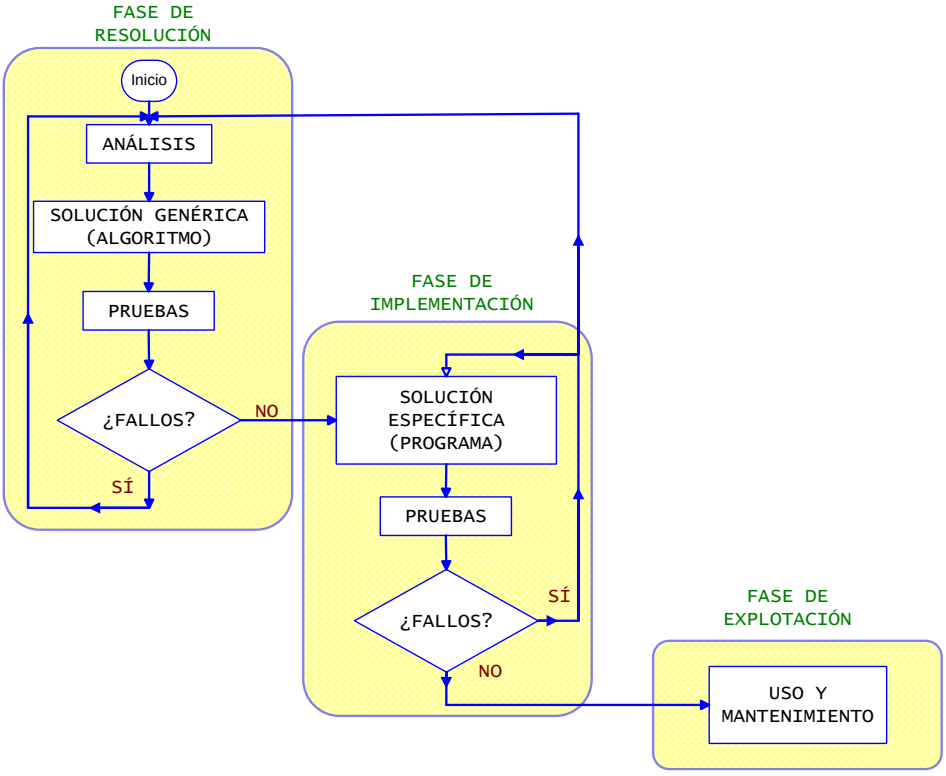


Figura 1.6. Fases del desarrollo de software

Algunos principiantes e, incluso, programadores con cierta experiencia, tratan de hacer más corto el trabajo de programación pasando directamente desde la definición del problema a la codificación del mismo en el lenguaje elegido. Esto está

totalmente desaconsejado porque aunque, en principio, parece una forma de ahorrar tiempo y esfuerzo se traduce precisamente en lo contrario. A medida que se vaya avanzando en la materia, se irán encontrando razones suficientes para asegurar que mientras más tiempo inicial se dedique a pensar, definir y pulir el algoritmo que resuelve un problema, menos tiempo y esfuerzo se dedicará a corregir errores. Así que ¡primero pensar y analizar, y lo último codificar!

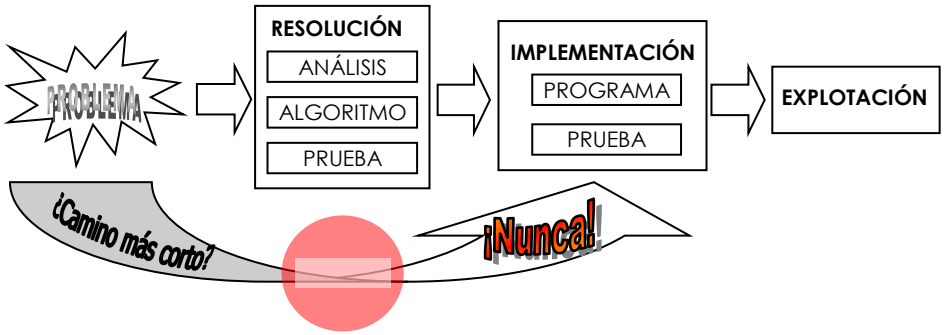


Figura 1.7. El camino equivocado

6. LENGUAJES DE PROGRAMACIÓN

Un **lenguaje de programación** es una notación para escribir algoritmos para resolver un problema concreto en un ordenador. Se puede definir como el conjunto de reglas, símbolos y palabras especiales establecidas para la construcción de programas. Se trata, como en un lenguaje de comunicación entre personas, de establecer una gramática, es decir, de definir las reglas aplicables a un conjunto cuyos elementos son los símbolos y palabras especiales definidos con anterioridad. Las reglas son de origen sintáctico, que ordenan la construcción de secuencias válidas del lenguaje, y semántico, que dan significado a esta construcción. Por ejemplo, me calzo el bolígrafo para ir al teatro, es una frase sintácticamente bien construida sobre elementos válidos del lenguaje, en este caso el español, pero, semánticamente incorrecta.

Los lenguajes de programación pueden clasificarse dependiendo de su proximidad al lenguaje "que entiende" la máquina. Se va a hacer un pequeño repaso de la historia y evolución de los lenguajes de programación.

6.1. Lenguaje Máquina

Es el lenguaje usado directamente por el procesador y está formado por un conjunto de instrucciones codificadas en binario.

La tecnología de la máquina tan sólo le permite entender en 0 y 1 (bits) y es, de esta forma, como se codificaban las órdenes en los comienzos de la informática. Cada tipo de máquina tiene su propio conjunto de instrucciones, formadas por secuencias de ceros y unos, cada una con un significado específico que permite a la unidad de control reconocer una operación básica y ejecutarla. El lenguaje máquina lo incorpora

el fabricante del procesador. Los algoritmos así escritos se dice que están en **código máquina**.

Esta forma de programar causaba graves problemas:

- **Programas totalmente dependientes de la máquina.** Los lenguajes máquina eran específicos para cada procesador. Esto requería hacer programas también específicos para cada tipo de ordenador, de forma que, si el mismo algoritmo debía ejecutarse en máquinas distintas, había que codificarlo de maneras diferentes.

- **Los programas en código máquina eran muy difíciles de leer e interpretar**, por tanto, también resultaban difíciles de modificar. Esto los hacía depender totalmente de sus diseñadores. Todo esto provocaba la realización de nuevo código cada vez que se necesitaban cambios en los algoritmos.

- **Los programadores de la época se veían obligados a recordar largas secuencias de ceros y unos**, correspondientes a las operaciones o instrucciones disponibles para los diferentes tipos de procesadores existentes en el mercado con los que trabajar.

- Eran los mismos **programadores los encargados de introducir los códigos binarios** correspondientes a las distintas operaciones que debía realizar la computadora, lo cual, además de constituir un tedioso trabajo, es un método propenso a errores.

Por ejemplo, instrucciones como sumar, restar o mover, podían tener el siguiente código:

OPERACIONES	LENGUAJE MÁQUINA
SUMAR	00101101
RESTAR	00010011
MOVER	00111010

6.2. Lenguaje Ensamblador

Para paliar los problemas inherentes al uso del lenguaje máquina y como evolución de los mismos aparecen los **lenguajes ensambladores**. La idea surge del uso de palabras mnemotécnicas, en lugar de las largas secuencias de ceros y unos, para referirse a las distintas operaciones disponibles en el juego de instrucciones que soporta cada máquina en particular. En este lenguaje, cada instrucción equivale a una instrucción en código máquina. Supone un primer intento por acercar el lenguaje de los procesadores al lenguaje natural.

El lenguaje máquina y el lenguaje ensamblador se conocen como **lenguajes de bajo nivel** por ser dependientes de la arquitectura del procesador que los soporta.

El ordenador no puede ejecutar directamente un programa escrito en lenguaje ensamblador, por lo que es indispensable un programa capaz de realizar la traducción al lenguaje máquina. Este programa se conoce como **Ensamblador** siendo también específico para cada tipo de procesador.

El lenguaje ensamblador presenta casi los mismos inconvenientes que el lenguaje máquina:

- De nuevo **programas totalmente dependientes de la máquina**. Los lenguajes ensambladores también son específicos para cada microprocesador, por lo tanto, un programa tan sólo podrá ser utilizado por la computadora para la que ha sido diseñado u otra equivalente. Por ello, si un algoritmo debía ejecutarse en máquinas con lenguajes ensambladores distintos había que codificarlo de maneras diferentes.

- Los programadores estaban obligados a conocer perfectamente la máquina, ya que estos lenguajes **manejan directamente recursos del sistema**: memoria, registros del microprocesador, etc.

Las operaciones sumar, restar o mover pueden tener, por ejemplo, los siguientes mnemotécnicos:

OPERACIONES	LENGUAJE ENSAMBLADOR
SUMAR	ADD
RESTAR	SUB
MOVER	MOV

6.3. Lenguajes Compilados

Los programadores de la etapa anterior seguían obligados a pensar a la hora de diseñar los algoritmos en términos de instrucciones máquina básicas, aún muy alejados de la manera natural en que nos comunicamos las personas. Siguiendo en el camino del acercamiento hombre-máquina y en el intento de paliar los problemas derivados del uso de ensambladores, se desarrollaron los que se llaman **lenguajes compilados** (Pascal, C, C++, Modula). A estos y a los siguientes, se les conoce como **lenguajes de alto nivel** por ser lenguajes más cercanos al lenguaje natural.

También, los algoritmos escritos en un lenguaje de alto nivel necesitan para su ejecución un programa llamado compilador, capaz de realizar la traducción al lenguaje máquina (Turbo Pascal, Turbo C, Borland C++, Visual C++). La traducción del programa se efectúa de manera que **cada instrucción escrita en lenguaje de alto nivel se transforma en una o más instrucciones de lenguaje máquina**.

El **compilador** traduce completamente el texto escrito en lenguaje de alto nivel y una vez acabada la traducción, informa de los posibles errores. El programador deberá corregir dichos errores y, sólo entonces, se generará la traducción lista para ejecutar.

El uso de lenguajes de alto nivel para programación supone una serie de ventajas respecto a los lenguajes anteriormente descritos:

- Logran una **mayor independencia de la máquina**, pudiéndose traducir los programas sobre cualquier equipo, con relativo poco esfuerzo y con el único requisito de disponer del compilador adecuado.
- El programador no necesita conocer el hardware específico de la máquina sobre la que se ejecutarán los programas, puesto que no manipula directamente los recursos del sistema.
- Es un lenguaje, como se ha señalado, más cercano al natural, por tanto, los programas son **más fáciles de leer y modificar**, y menos susceptibles de errores.

Los programas resultantes de la traducción desde un lenguaje de alto nivel ocupan más recursos del sistema informático, memoria y tiempo con el consiguiente aumento del precio del programa en marcha. Pero esto que al principio era un problema, ha dejado de serlo en la actualidad al tener las memorias cada vez mejores tiempos de acceso y ser más baratas, y los procesadores mucho más rápidos que cuando aparecieron los primeros compiladores. Sin embargo, en ciertas aplicaciones que requieren acceder directamente a elementos del hardware o necesitan unos tiempos de respuesta muy cortos, se realizan en ensamblador las partes del programa más críticas y en un lenguaje de alto nivel, el resto. Por ejemplo, juegos o programas de control de maquinaria.

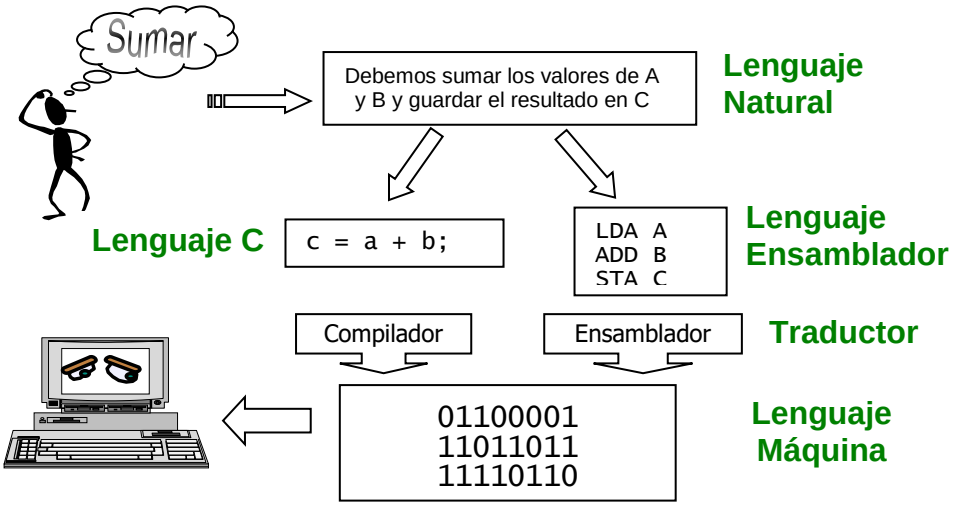


Figura 1.8. Uso de los lenguajes de programación

6.4. Lenguajes Interpretados

Existen lenguajes de alto nivel cuyos traductores, llamados **Intérpretes,** realizan lo que se podría llamar traducción simultánea, es decir, cada instrucción escrita en un lenguaje interpretado (por ejemplo, Smalltalk) es analizada, traducida y ejecutada tras su verificación. La principal diferencia con los lenguajes compilados, es que una vez traducida cada instrucción, no se guarda en la memoria, sino que se ejecuta y, a continuación, se elimina.

Los lenguajes interpretados presentan algunas ventajas frente a los compilados: los programas que originan ocupan menos memoria que los originados por los compiladores, ya que no guardan una versión completa del programa traducido a código máquina. También presentan algunas desventajas: los programas interpretados son mucho más lentos, ya que han de ser traducidos mientras que se ejecutan y pueden quedar suspendidos si se producen fallos durante la ejecución. Además, para poder ejecutar el programa es necesario tener el intérprete ejecutándose en la máquina, cosa que no es necesaria en el caso de los compiladores: bastará con tener una copia del programa ya traducido a código máquina para que el procesador sea capaz de ejecutarlo.

Una buena aplicación de los intérpretes consiste en utilizarlos en la fase de prueba del programa. Una vez finalizada la fase de compilación y con el objetivo de detectar errores que pudieran aparecer durante la ejecución, se ejecuta paso a paso, observando con ello la secuencia de órdenes que se siguen en el algoritmo y los valores que van tomando los distintos objetos que lo componen. Esta tarea de buscar y corregir errores se conoce como **depuración del programa**.

Actualmente, hay lenguajes en el mercado llamados pseudo-compilados o pseudo-interpretados, por ejemplo, **Java**. Desde un punto de vista amplio, se puede decir que Java es un lenguaje compilado e interpretado a la vez. El código fuente se compila para obtener un código binario en forma de *bytecodes*, que son estructuras similares a las instrucciones máquina, con la característica de no ser específicas de ninguna máquina en particular. Este código es interpretado por la MVJ (Máquina Virtual Java, JVE en inglés), que lo interpreta y ejecuta. La MVJ es, en definitiva, el programa que lo traduce al código máquina del procesador en particular. Algo parecido ocurre con la plataforma .NET de Microsoft.

7. FASES EN LA CREACIÓN DE UN PROGRAMA

A partir del diseño correcto del algoritmo específico que resuelve un problema, ¿qué pasos se deben seguir hasta obtener un programa igualmente correcto? Se va a dividir el camino en las siguientes fases:

- **Fase de edición**. El algoritmo escrito en un lenguaje de alto nivel mediante un editor de texto, se conoce como **programa** o **código fuente**, que tendrá un nombre asignado por el programador o en su defecto asignado por el propio compilador. Este nombre obedecerá a las reglas que para ello tenga previsto el sistema operativo instalado en la máquina donde estamos creando el programa.

Generalmente, el programa fuente tiene como nombre el que le asigna el programador y, como extensión, una secuencia de caracteres típica del lenguaje de programación usado. En la mayoría de los sistemas se sigue el siguiente formato:

```
NOMBRE_DEL_PROGRAMA.EXTENSIÓN
```

siendo los propios lenguajes los que traen implícitas las reglas para su construcción. Por ejemplo, Primero.c, Raiz.cpp, Ordenar.pas, según sean programas codificados en C, C++ o Pascal, respectivamente.

- **Fase de compilación**. Una vez codificado el algoritmo, hay que asegurarse de que no contiene errores, para ello, el programa fuente se traduce utilizando y ejecutando el programa compilador que tendrá como entrada el propio código fuente. El compilador comprueba que el programa fuente se ha escrito cumpliendo las reglas gramaticales del lenguaje. Si aparecen errores, deberá volverse al editor de texto para corregirlos y, hecho esto, ejecutar de nuevo el compilador. Si el código está libre de errores, se generará el programa traducido a código máquina, conocido con el nombre de **programa** o **código objeto**. Este programa aún no se puede ejecutar ya que falta por añadir los módulos de enlace.

  Generalmente, el código generado tiene como nombre el mismo que le asignó el programador al programa fuente y como extensión *obj*. Por ejemplo, Primero.obj, Raiz.obj, Ordenar.obj.

- **Fase de enlazado (linkage)**. Si se trabaja en un entorno que disponga de subprogramas enlazables o con compiladores que incluyan **bibliotecas** (llamadas también **librerías**), el código objeto resultante de la etapa anterior, deberá ser sometido a un proceso de montaje, donde se añaden los llamados módulos de enlace. Este proceso se conoce como **montaje**, **enlazado** o **linkage** y el programa encargado de realizarlo **montador**, **enlazador** o **linker**. Consiste en la inclusión de determinados módulos (bibliotecas) que son necesarios para que el programa pueda realizar ciertas tareas como, por ejemplo, manejar los dispositivos de entrada/salida de la máquina. En esta fase, también pueden producirse errores, que podrán ser corregidos volviendo a la fase de edición y repitiendo todo el proceso. Si no se producen errores, se generará el **programa** o **código ejecutable**.

- **Fase de ejecución:** El programa resultante ya, definitivamente, en código máquina listo para su carga y ejecución. El **programa ejecutable** suele tener el mismo nombre que su correspondiente programa fuente y con extensión, por ejemplo, *exe*. En el ejemplo, Primero.exe, Raiz.exe, Ordenar.exe.

  Con la carga de la versión ejecutable, se producirá la entrada de datos al programa y se generarán unos resultados de acuerdo con el código ejecutado que podrán o no contener errores. Si se detectaran fallos deberá

volverse a la primera etapa (edición) para corregirlos y repetir todo el proceso.

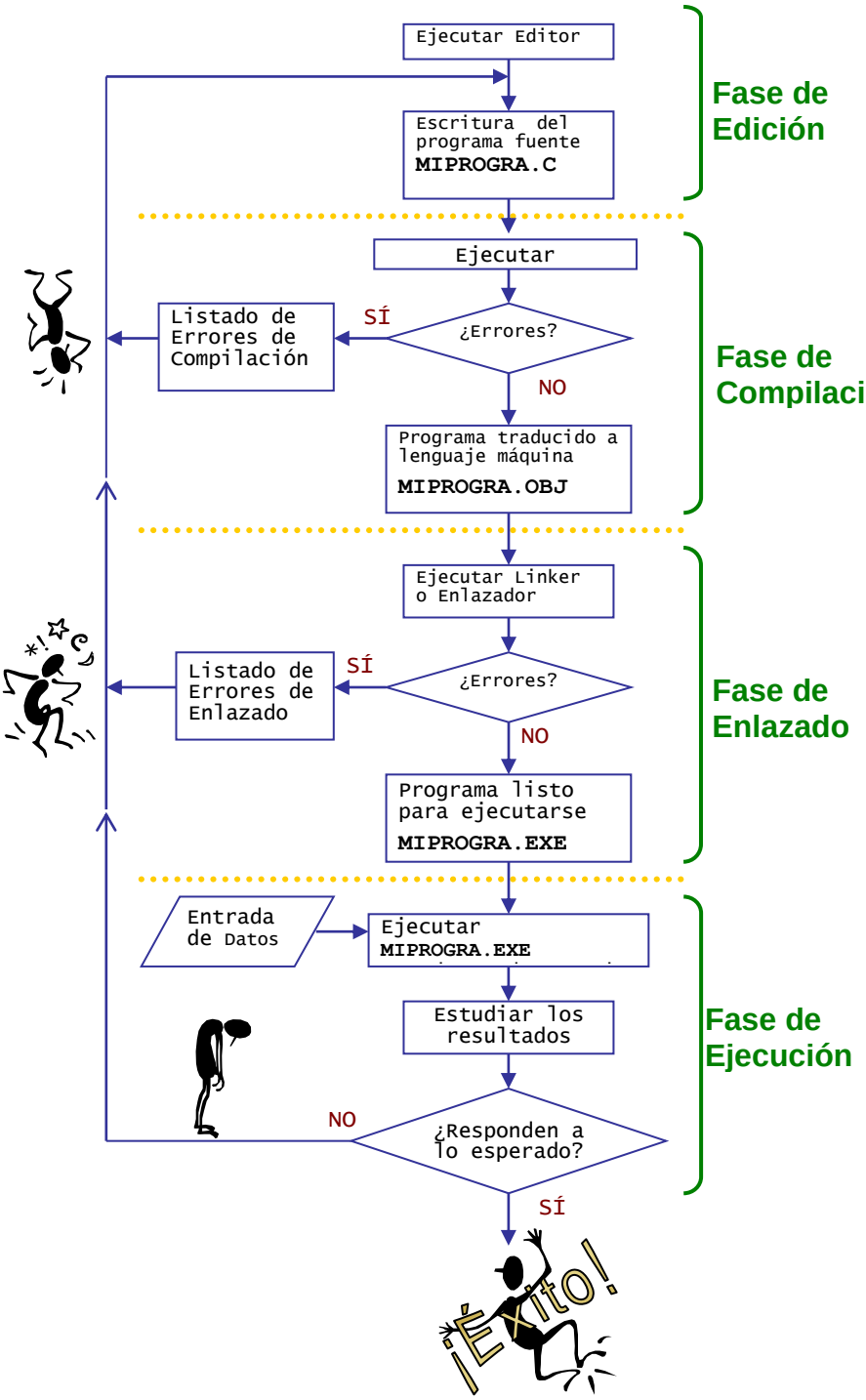


Figura 1.9. Fases en la creación de un programa compilado

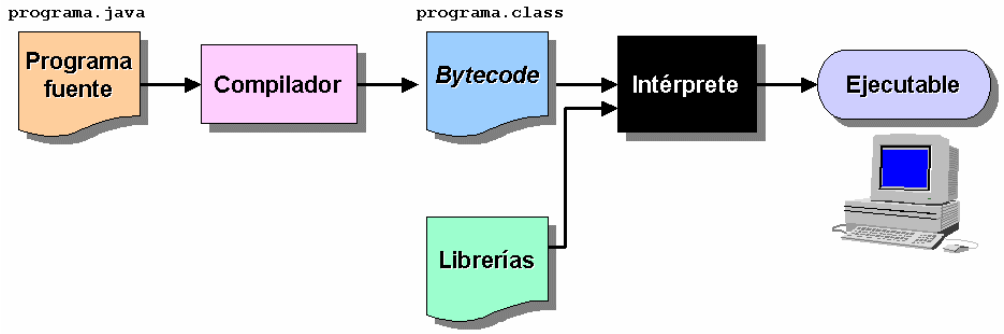


Figura 1.1. Esquema del proceso de creación de un programa con Java

7. VERIFICACIÓN, DEPURACIÓN Y PRUEBA DE PROGRAMAS

Después de haber comprendido absolutamente las especificaciones y requerimientos del problema planteado, y haber seguido todas las fases de creación del programa, por fin se puede ejecutar el código resultante para observar los resultados. Si no funciona correctamente, es decir, si no produce los resultados deseados, se deduce que debe haber algún error en alguna de las fases del proceso y hay que pasar a realizar la compleja, tediosa y difícil tarea de detectar el error y corregirlo.

Este proceso, llamado **depuración del programa**, suele ocupar la mayor parte del tiempo de los programadores. Su duración es mayor mientras menos tiempo se haya dedicado a la fase de análisis y a la de diseño.

La prueba del programa, que consiste en la ejecución del código con un conjunto de valores llamado **juego de ensayo**, puede descubrir errores, pero no puede asegurar la ausencia de ellos. Tan sólo se puede decir que el programa funciona para los valores probados y con las condiciones de la prueba, pero ¿se puede asegurar lo mismo para valores o condiciones no probados? Se podría ampliar el campo de posibilidades de acierto aumentando la cantidad y diversidad de juegos de ensayo, mas no parece esta una técnica demasiado robusta.

Sería aconsejable que la depuración del programa no se estableciera tan sólo en la última fase del desarrollo del software. Es bueno que los errores se detecten lo antes posible, no hay que olvidar que esto se traduce en ahorro de trabajo y coste del producto: mientras más pronto se detecten los fallos, más fáciles son de corregir. Las actividades de verificación y depuración deben establecerse en todo el proceso de producción del software, incluyendo:

- Buen diseño para que el programa sea correcto y de calidad.
- Recorridos e inspecciones de diseño y de código.

- Métodos para la depuración del programa.
- Previsión de planes de prueba.
- Elección de los datos de prueba que formarán el juego de ensayo.

7.1. Errores

La detección, corrección y prevención de errores suponen tareas muy importantes desarrolladas en todas las fases del proceso de desarrollo de programas correctos.

En el desarrollo de un programa, según la etapa donde se producen, se pueden considerar diferentes tipos de errores.

ERRORES DE ESPECIFICACIÓN Y DISEÑO

Quizá sea el peor de los tipos de fallos. Se producen por el uso de algoritmos diseñados a partir de especificaciones incorrectas del problema, surgidas en la fase de análisis del mismo y tienen como consecuencia que el programa no haga lo que debe hacer. Son errores difíciles y costosos de corregir puesto que suponen repetir una gran parte del proceso de desarrollo del software. El esfuerzo conceptual de diseño y la codificación no han dado el fruto esperado, y hay que replantear de nuevo las especificaciones, modificar el diseño de manera que cumpla las funciones específicas que se pretende debe hacer, codificar el nuevo algoritmo y probar de nuevo el programa. Por otro lado, todo este proceso supone un coste adicional del producto, puesto que deberán sumarse horas de trabajo y de uso de recursos por parte del equipo de desarrollo.

Generalmente, los programadores no son expertos en las áreas de aplicación a las que van destinados los programas que realizan, por ejemplo, realizar un programa para una empresa que analiza restos arqueológicos o desarrolla productos fitosanitarios. Por este motivo, las especificaciones deben ser meticulosamente estudiadas, si se quiere asegurar el éxito en el desarrollo de los algoritmos. Para conseguirlo, es importante una buena comunicación entre el analista (programador, alumno) y las personas que desean informatizar una tarea (cliente, profesor). Con el objetivo de asegurar las respuestas a todas aquellas preguntas que ayuden a comprender con precisión qué se espera que haga el programa, es decir, establecer correctamente las especificaciones y requisitos del problema.

ERRORES EN TIEMPO DE COMPILACIÓN

Son errores detectados por el compilador cuando intenta traducir el programa fuente. Se pueden dividirlos en los siguientes tipos:

- **Errores lexicográficos**. Son producidos por la aparición de elementos no reconocibles por el compilador. Los más comunes suelen ser la aparición de caracteres que no pertenecen al lenguaje o palabras del lenguaje mal

escritas. Por ejemplo, en español pograma en vez de programa, o en C maim en lugar de main.

- **Errores sintácticos**. Son los provocados por el incumplimiento de las reglas de sintaxis del lenguaje y detectados por el compilador al no reconocer una tira de caracteres como una secuencia con formato válido de instrucción. Son típicos errores como expresiones incompletas

```
(S = A + B/)
```

o caracteres no permitidos en una instrucción (• no está permitido en C como símbolo para multiplicar).

```
A • B = C
```

- **Errores semánticos**. Estos errores no interrumpen el proceso, pero el compilador avisa de una incorrección por falta de significado. Por ejemplo, la advertencia del uso en el programa de alguna variable no declarada. Por establecer una analogía con el lenguaje natural, diremos que sería una frase bien construida, sin faltas de ortografía, pero sin significado alguno (por ejemplo, la mesa vio la película y le gustó mucho su sabor). En C se conocen como **warnings** (advertencias). Por ejemplo: usar un operador inadecuado

```
if (a = b)  //uso de = como símbolo de comparación en vez de ==
```

Si se tiene un buen conocimiento del lenguaje de programación y se presta especial atención en la codificación del algoritmo, se pueden evitar la mayoría de estos errores detectados en tiempo de compilación. Es una mala técnica el no ser cuidadosos en esta etapa en la confianza de que será el compilador quién se encargue de avisar de los errores.

ERRORES DE ENLAZADO

Son debidos, principalmente, a que el compilador no encuentra los módulos de enlace porque no se le ha indicado correctamente la ruta donde se encuentran o porque, si se le ha indicado correctamente dicha ruta, realmente los módulos no se encuentran definidos en ese lugar. Estos errores son relativamente fáciles de detectar y su corrección pasa por configurar el compilador para que encuentre correctamente dichos módulos o por colocar los módulos creados por el programador en el lugar adecuado.

ERRORES EN TIEMPO DE EJECUCIÓN

Son fallos del programa más difíciles de detectar y solucionar que los errores hallados en tiempo de compilación. Algunos de ellos provocan la terminación anormal del programa o su caída (en lenguaje coloquial se dice que el sistema se ha colgado). Generalmente, están relacionados con desbordamientos o con operaciones indeterminadas, por ejemplo, bucles infinitos, división por cero, cálculo de la raíz cuadrada de un número negativo, etc. Por ejemplo:

```
COCIENTE = DIVIDENDO / DIVISOR
```

Será una instrucción válida sólo en el caso en que DIVISOR sea distinto de 0, puesto que si toma este valor, la operación no es posible y el programa provocaría un error al intentar realizar la operación en tiempo de ejecución.

Frecuentemente, se producen errores de ejecución por no haber prevenido en el programa los posibles fallos que pudieran cometer las personas que realizan la entrada de datos para dicho programa. Por ejemplo, si el programa espera un dato numérico y el usuario introduce una secuencia de caracteres que, por tanto, no se corresponde con un número, se provocará un conflicto entre tipos de datos y un error en tiempo de ejecución.

Un buen método para la prevención de errores provocados en la entrada de datos, en especial si esta se produce desde teclado, consiste en comprobar explícitamente en el programa que los datos se encuentran en los rangos correctos. Será el propio programa quién tenga mecanismos para actuar si no fuera así, en vez de dejar que el sistema responda erróneamente en tiempo de ejecución. Este proceso se conoce como **validación de la entrada de datos**.

Existen muchas situaciones en los programas donde pueden y deben prevenirse errores potenciales. Corresponde a los programadores añadir a los programas el código necesario para el manejo y prevención de estos errores, evitando que se produzcan condiciones que provoquen la parada del programa durante su ejecución. Se pueden añadir las verificaciones oportunas de estas condiciones, así como una apropiada estrategia para actuar en caso de error evitando la terminación anormal del programa. En el argot informático se conoce como muerte elegante del programa. Por ejemplo, todos conocemos el mensaje de Windows “se ha detectado un error. El programa finalizará y se reiniciará su equipo”, esto es más elegante que hacerlo de forma brusca sin dar ningún aviso de ello.

A la capacidad de un programa para recuperarse cuando se ha producido una situación de error se le llama **robustez**. Existen algunos campos de aplicación de la informática donde la robustez de los algoritmos no es una característica sólo deseable, sino absolutamente imprescindible e, incluso, crítica. Por ejemplo, en sistemas de control en naves espaciales, de mísiles, pilotaje automático de aviones, control de procesos industriales, monitorización de pacientes en el campo médico, son todas ellas situaciones donde los programas de ningún modo pueden terminar de manera anormal o inesperada.

Otros errores producidos en tiempo de ejecución no provocan la terminación anormal, ni la caída del sistema, simplemente producen resultados erróneos. Pueden ser debidos a incorrecciones en el diseño del algoritmo, por lo que debe volverse a la fase anterior para corregirlos. Por ejemplo, puede haberse usado una variable que no tiene asignado un valor, etc. Estos errores de lógica son más difíciles de prevenir y corregir, por lo que el programador debe ser cuidadoso con ellos.

7.2. Aprender a diseñar programas correctos

Una prueba concienzuda de un programa con juegos de ensayos exhaustivos ayuda a detectar errores, pero no puede concluir su inexistencia y, por tanto, la corrección del programa. Por esta razón, la validación y verificación de programas es un tema importante en el área de investigación de la Informática Teórica.

Para llevar a cabo la validación y verificación, y localizar las partes incorrectas de un programa, se disponen de las técnicas de verificación formal, aplicables a mano y se cuenta con la ayuda de herramientas automáticas como auditores de código, generadores de referencias cruzadas, analizadores de flujo de control, etc. No es objetivo de este libro entrar en detalle en la explicación y uso de estas técnicas, pero sí se usarán algunos conceptos en los que se basan las técnicas de verificación que ayudarán en el proceso de diseño de programas.

ASERTOS

Un **aserto** es una afirmación sobre el estado del algoritmo. Por ejemplo, tras ejecutarse la instrucción

```
RESTA = A - 1
```

siendo `RESTA` y `A` números enteros, se está en condiciones de afirmar «el valor de `RESTA` es menor que el valor de `A` » ya sea `A` positivo o negativo.

Definir asertos es, sin duda, una herramienta muy útil en el proceso de construcción de programas correctos. El objetivo de la verificación formal es que se puedan establecer asertos sobre lo que se pretende que realice el programa basándose en las especificaciones y requerimientos, y demostrar ,mediante argumentos razonados, que un determinado diseño cumple los asertos preestablecidos en lugar de esperar a la ejecución del programa. Por ejemplo, en el algoritmo para realizar la tortilla de patatas tenemos como aserto de que disponemos de sartén, si eso es así, el algoritmo no puede empezar comprando una satén, si así lo hace, sería un error.

PRECONDICIONES Y POSTCONDICIONES

Durante el desarrollo de cualquier tarea que forma parte de un trabajo más amplio, sea o no informático, se plantean situaciones que se deben establecer y comprobar, para asegurarse que, realmente, esta tarea encaja como una pieza en el trabajo completo. Se trata de asegurar qué ocurre en sus límites, es decir, qué circunstancias deben darse al comienzo o entrada de la tarea en cuestión y cuáles se esperan deben ser ciertas al finalizar, es decir, a la salida.

A la entrada de cualquier operación, instrucción, tarea o módulo (se llama **módulo** a una parte de un programa que realiza una tarea específica) se producen una serie de asertos que deben ser verdad para que funcione correctamente. Estos asertos reciben el nombre de **precondiciones**, de forma que si la operación, instrucción o módulo se ejecuta sin que se cumplan las precondiciones, no tendremos garantías de los resultados obtenidos.

De igual forma, cuando se realizan operaciones o módulos se establecen asertos sobre los resultados que deben esperarse a la salida. A estos asertos que describen el resultado de operaciones se los conoce como **postcondiciones**.

Pongamos un ejemplo sencillo para definir cuales serían las precondiciones y postcondiciones: un algoritmo para buscar a un alumno dentro de una clase y hacerlo salir para que atienda una llamada urgente de teléfono.

Diseñaremos un módulo llamado *BuscarAlumno* y vamos a expresarlo del siguiente modo:

```
BuscarAlumno (Clase, AlumnoBuscado)
```

Donde `Clase` es el aula donde debemos buscar al alumno y `AlumnoBuscado` el nombre del alumno que deseamos buscar.

Proceso que realiza: Buscar un alumno de la clase y hacerlo salir si está.

Entrada: ¿qué necesito para realizar el proceso?: la `Clase` y el `AlumnoBuscado`.

Precondiciones: La clase debe no estar vacía. Debe ser, por tanto, horario lectivo y que los alumnos no se encuentren en laboratorios, salas de vídeo, campos de deporte, etc. Si todo esto es un requisito para buscar al alumno, deberá ser comprobado con anterioridad a poner en marcha BuscarAlumno por otro algoritmo que podemos llamar `MirarSiClaseVacía`.

Salida: ¿qué se devuelve tras haber realizado la tarea deseada?: `Clase` cambiada y `AlumnoBuscado` para que atienda la llamada de teléfono, caso de encontrarse en clase o, por ejemplo, un **mensaje "**el alumno buscado no está en el aula**",** caso de no encontrarse en clase**.**

Postcondiciones: Estamos en condiciones de asegurar que `Clase` queda con un alumno menos si hubiera salido o igual si el alumno no se encuentra en el aula.

Como se puede observar, precondiciones y postcondiciones están relacionadas con la verificación de programas. Haciendo afirmaciones explícitas de las situaciones que se deben dar en las comunicaciones entre programas, podremos evitar errores de tipo lógico, cometidos por falta de comprensión del problema. Por ejemplo, una vez definida la precondición "La clase debe no estar vacía", debemos comprobar fuera de la tarea `BuscarAlumno` la condición de no vacía en los términos convenientemente establecidos. El algoritmo que nos ocupa presupone que los alumnos están en clase. La postcondición nos asegura el estado de la clase. Esto es importante para los usuarios de dicho algoritmo. Por ejemplo, supongamos que un profesor entra en clase con el objetivo de repartir a todos los alumnos un formulario que deben cumplimentar, o desea que realicen todos ellos un examen. Todos los alumnos deben permanecer en el aula, es decir, la clase debe estar llena. En este caso, si hubiésemos utilizado el algoritmo `BuscarAlumno`**,** no podemos realizar el formulario. O si por el contrario, ya están los alumnos realizando el formulario, no podemos utilizar el módulo

`BuscarAlumno` durante el intervalo en que los alumnos rellenen el cuestionario o realicen el examen, puesto que usarlo supone hacer salir al `AlumnoBuscado` del aula.

El uso de las precondiciones y postcondiciones en las especificaciones de los módulos es una herramienta importante. La información que proporcionan ayudan a clarificar ideas y a diseñar los programas de una manera más sencilla y en la confianza de que no se producirán errores lógicos de conocimiento y comprensión de especificaciones.

INVARIANTES DE BUCLE

Una de las estructuras de control para el diseño son los bucles, además de las estructuras para bifurcaciones y bloques de sentencias simples. El diseño de los bucles es especialmente dificultoso, a veces se repiten más de lo deseado, otras menos, a veces no acaban y otras, simplemente, no realizan lo que se pretende. Existe un aserto llamado invariante del bucle que nos ayudará a construir correctamente un bucle y que consiste en determinar qué condiciones deben ser verdad al comienzo de cada iteración del bucle. El invariante da información sobre las veces que debe repetirse el bucle, el propósito e, incluso, la semántica de dicho bucle.

Sobre este tema, se hablará extensamente en el capítulo tres.

7.3. Recorridos e inspecciones de diseño y de código

Cuando se diseñan e implementan programas, ya sea individualmente o en equipo, se pueden detectar muchos errores con tan sólo usar lápiz y papel antes de utilizar el ordenador. Esta comprobación manual del diseño de un programa prestando especial atención a aquellos puntos que se consideran problemáticos, incluso marcando los cambios sobre el mismo papel, es un método útil y conveniente para la verificación de programas y la detección precoz de errores. Con esta filosofía, es frecuente incluso que varios programadores que trabajen en equipo, intercambien sus respectivos códigos aplicando en ellos la verificación manual.

Extensiones de la comprobación manual son el **recorrido** y las **inspecciones del diseño y del código** de los programas, con objeto de hallar errores antes de comenzar la fase de prueba.

En el recorrido, los datos de prueba se “arrastran” por el programa simulando la ejecución manualmente, con lápiz y papel, e inspeccionando concienzudamente diseño y código en un intento de descubrir errores. A esto se llama seguir la traza del algoritmo.

Conviene aplicar este método en diferentes momentos del proceso de creación de software:

- En las etapas de análisis y de diseño de la solución general, dentro de la fase de resolución del problema, para asegurarse que se cumplen todos las especificaciones, que han sido incluidas todas las tareas requeridas y que todas se comunican y encajan perfectamente en el diseño general.

- Dentro de la fase de implementación, en la etapa de diseño de la solución específica, una vez acabada la codificación, es conveniente hacer una inspección de los listados del código fuente.

Al término de esta última inspección, se puede comenzar con un alto índice de éxito la fase de prueba del programa.

7.4. Planes para depuración

Cuando se plantea la solución general del problema, estaría bien identificar los puntos más conflictivos para dedicar a ellos especial interés a la hora de verificar y depurar errores. De esta forma, puede incluso establecerse planes específicos para la depuración antes de ejecutar un programa.

Una de las técnicas usadas consiste en insertar en los puntos conflictivos del programa sentencias (*printf* en C) que permitirán ir observando la evolución de los valores de aquellos objetos que interesa verificar. Estas líneas está bien que las vea el programador pero no el usuario del programa, por lo que una vez depurado deben eliminarse.

Algunos compiladores incorporan programas para depuración en línea que facilitan salidas de la traza, establecer puntos de parada (*breakpoints*) y otras facilidades por lo que se puedan observar los valores que van tomando los objetos que interesan, haciendo bastante más sencillo el proceso de depuración de programas.

También, se hablará de estas técnicas cuando en capítulos posteriores se hayan adquirido conocimientos suficientes.

7.5. Prueba de programas

Una vez acabadas las verificaciones, comprobaciones manuales e inspecciones de diseño y de código, se está en óptimas condiciones para ejecutar el programa. Sin embargo, todavía es posible que se presenten errores en ejecución, por lo que se ha de comenzar la fase de prueba del programa.

Como método más fiable en la prueba se sugiere la verificación paso a paso de fragmentos significativos de programa, en lugar de hacerlo con todo el programa de una vez, en especial si se trata de programas suficientemente largos y complejos.

Si se utiliza un conjunto de datos y unos casos de prueba correctos, evaluados y verificados anteriormente, la prueba permitirá asegurar con ciertas garantías que un programa funciona correctamente.

¿Qué casos de prueba se pueden establecer en nuestros programas?. Los objetivos de los casos de prueba consisten en verificar que un programa funcionan perfectamente. Se puede establecer casos de prueba para comprobar que un programa gestiona correctamente los errores cometidos a la entrada de datos o errores por desbordamiento en operaciones.

Ejemplo: Supongamos un programa para realizar operaciones de dividir. ¿Bastaría con la instrucción?

```
Resultado = Dividendo /Divisor
```

Podemos incluir los siguientes casos de prueba: uno en el que el Divisor sea distinto de cero, y otro en el que el Divisor sea cero y el Dividendo puede tener cualquier valor.

Los datos de prueba del primer caso pueden ser Dividendo = 6, Divisor = 3 y obtendríamos Resultado = 2.

Para la prueba del segundo caso, podemos elegir el mismo valor para el Dividendo y Divisor = 0. ¿Qué pasa ahora con el valor de Resultado?, el ordenador no puede realizar esta operación y daría un error de ejecución.

Incluiremos en nuestro algoritmo los mecanismos suficientes para controlar que procesos parecidos a este tengan respuesta por parte del sistema informático. Esto podemos llevarlo a cabo, impidiendo que se pueda introducir en este caso el valor 0 como entrada (esta es la mejor opción) o impidiendo que la operación de dividir se realice en el caso de haberlo introducido.

En el primer caso, podríamos mandar un mensaje antes de dar valor a Divisor e impedir que se pase a la línea siguiente si le damos el 0.

```
"De a Divisor un valor distinto de 0, es imposible
dividir por 0"

Saldrá siempre el anterior mensaje si da el valor 0

Resultado = Dividendo /Divisor
```

En el segundo caso, podríamos dar el valor 0, mandar un mensaje en este caso y realizar la operación sólo en el caso de no ser 0 .

```
Si Divisor  es cero

      "Es imposible dividir por 0"

Si Divisor no es cero

      Resultado = Dividendo /Divisor
```

¿Cómo se pueden determinar los casos y datos de prueba convenientes para los programas? La primera tarea consistirá en establecer cuáles son los requerimientos de prueba, por ejemplo, ¿se deben verificar todas las entradas?, ¿se debe comprobar el rango de todas las operaciones?, ¿es necesario verificar la naturaleza de todos los valores numéricos?.

En aquellos casos en que el código del programa o módulo sea suficientemente pequeño, es posible realizar una prueba exhaustiva de los elementos de entrada, es decir, verificar el programa para todos los casos posibles con lo que se puede asegurar que el software cumple todas las especificaciones. Pero, en la mayoría de las ocasiones no será posible realizar la prueba exhaustiva. Para estos casos, existen estrategias que permiten establecer casos y datos de prueba de forma sistemática.

Uno de estos métodos consiste en probar casos generales de datos, se conoce como **cubrimiento de datos**. Debe probarse, al menos, un ejemplo de cada categoría de datos de entrada, así como los límites en que deben moverse e, incluso, los casos especiales. Por ejemplo, si los datos de entrada son fechas, para observar como responde el programa se debe intentar dar valores menores que 1 y mayores que 12 para el mes y para el día, menores que 1 y mayores que 28, 30 o 31 dependiendo del mes en cuestión (o 29, si se tratara de año bisiesto).

Cuando la prueba no puede ser exhaustiva, existen también métodos formales de verificación de programas. Esta materia pertenece al área de la informática teórica donde se trata de establecer programas, análogos al método de demostración de teoremas matemáticos, que permitan asegurar la corrección de algoritmos.

Todo lo expresado anteriormente, hace aconsejable el establecimiento documental de un plan de pruebas formal, que determine los objetivos de la prueba, los casos de prueba junto con los detalles de cada uno de ellos, las entradas, salidas esperadas y condiciones de éxito.

Conviene planificar una estrategia general para la prueba de programas antes de comenzar dicha fase, antes incluso de la fase de implementación. Lo más adecuado sería incluir los requerimientos de prueba en la etapa de análisis, y la planificación y métodos de prueba en la etapa de diseño.

Para mayor profundidad se recomienda consultar [PIATTINI et al. 2003], que contiene un capítulo dedicado a pruebas del software.

7.6. Algunas consideraciones

Aplicar las técnicas de verificación de programas supone un alto consumo de tiempo y coste, pero no todos los programas requieren el mismo esfuerzo, esto dependerá de la envergadura de dicho programa. Un programa cuya ejecución sea crítica para la vida del ser humano, es obvio que tendrá un alto nivel de verificación, por ejemplo, los programas para controlar ciertas operaciones quirúrgicas. Por el contrario, un juego de ordenador tendrá una verificación más sencilla.

En clase, los requerimientos de verificación serán sencillos generalmente y especificados por el profesor como parte del proceso de aprendizaje. Sin embargo, lo habitual en el entorno de trabajo es que se especifiquen expresamente a la firma del contrato entre el cliente y la empresa desarrolladora.

Desde otro punto de vista, la verificación estará recomendada para ciertas secciones o módulos del programa especialmente complicadas y propensas a errores.

Es oportuno recordar las palabras de C. A. R. Hoare: “Hay dos formas de construir un diseño de software: Una es hacer un diseño tan simple que sean obvios los defectos y la otra, es hacer un diseño tan complicado que no haya defectos obvios. La primera forma es mucho más difícil”

La verificación y la prueba de programas son actividades que se aplican a lo largo de la vida del software. El siguiente cuadro resume los distintos tipos de actividades de verificación y prueba en las distintas fases del desarrollo de programas.

FASES	ACTIVIDADES DE VERIFICACIÓN Y PRUEBA
Análisis	▪ Estudiar los requisitos hasta comprender completamente las especificaciones. ▪ Estudiar los requerimientos de prueba.
Diseño	▪ Asegurar el diseño de programas correctos. ▪ Realizar todos los tipos de inspecciones de código. ▪ Verificar el diseño. ▪ Realizar un plan de pruebas.
Codificación	▪ Dominar el lenguaje de programación. ▪ Realizar inspecciones del código fuente. ▪ Añadir sentencias específicas al código para la depuración del programa. ▪ Realizar un plan de pruebas. ▪ Realizar la verificación del código.
Prueba	▪ Realizar la prueba de acuerdo al plan de pruebas establecido. ▪ Depurar los segmentos de código que sea necesario. ▪ Si se realizaron subprogramas y se depuraron por separado, integrarlos para su prueba conjunta. ▪ Ejecutar nuevas pruebas después de las correcciones realizadas. ▪ Realizar pruebas de aceptación del producto completo a la entrega.
Mantenimiento	▪ Realizar un plan de pruebas para todo el producto cada vez que se haya cambiado o añadido nuevas funcionalidades, o bien se hayan corregido problemas detectados en la fase de explotación.

8. UNAS PALABRAS SOBRE CALIDAD DEL SOFTWARE

No es suficiente que un programa funcione para que sea bueno, esto es sólo necesario. Para que sea bueno, además, debe estar hecho con calidad. M.A. Jackson, uno de los padres del paradigma estructurado, dijo: “El comienzo de la sabiduría para un ingeniero de software es reconocer la diferencia entre hacer que un programa funcione y conseguir que lo haga correctamente” [PRESSMAN, 2003]. Esta frase debería exhibirse en todas las aula donde se enseñe a programar.

La definición oficial del estándar IEEE Std.610-1991[IEEE, 1991a] expone que calidad es el grado con el que un sistema, componente o proceso cumple los requisitos establecidos y las necesidades o expectativas del cliente o usuario.

El software considerado de calidad debe reunir una serie de características que serán objetivos a la hora de realizar programas. Estas características son las siguientes:

- El programa funciona de acuerdo a los requisitos establecidos.
- Legibilidad y comprensión.
- Modificable sin necesidad de invertir excesivo tiempo y esfuerzo.
- Su realización debe atenerse al tiempo y presupuesto establecidos.

Se va a pasar a detallar un poco cada uno de estas características.

- **Funciona**. Que el software de calidad funciona, significa:
  - que realiza **completa** y **exactamente** lo que se estableció en los requisitos.
  - y de forma **correcta**, lo hace bien.
  - y **eficientemente**, emplea poco tiempo y recursos.

  Para saber lo que debe realizar, se detalla una lista de definición de **requerimientos**.

  Para los especialistas informáticos, los requerimientos se definirán tras el estudio detallado de lo que quiere el cliente o los futuros usuarios del software. Para los alumnos de informática se deducen a partir del enunciado de un problema. Por ejemplo, ”Realizar un programa que calcule el área de ciertas figuras geométricas: Círculo, triángulo, cuadrado y rectángulo ”.

  Para realizar un programa que cumpla los requisitos definidos se usan las especificaciones del software, que deben detallar la función que se realiza, los formatos de la entrada y de la salida deseada, y aspectos especiales a tener en cuenta para el diseño e implementación del programa.

  A veces, en los enunciados se detallarán las especificaciones, otras habrá que deducirlas basándose en la definición de los requerimientos y en otras ocasiones, habrá libertad para decidir con qué especificaciones se han de diseñar los programas, siempre y cuando aseguren su corrección y completitud. Cuando el programador toma cualquier decisión en este sentido, deberá documentarla.

- **Legibilidad y comprensión**. Significa que los programas deben poder leerse y entenderse con facilidad. Para ello, se aplicarán metodologías de diseño: estructurada, modular, etc., a la que se añadirá documentación para facilitar lectura y comprensión.

- **Modificable**. Para que el software sea de calidad, deberá poder modificarse con cierta facilidad en el menor tiempo posible y con el mínimo esfuerzo. Las modificaciones pueden ocurrir en cualquier fase a lo largo de la vida del software porquen aparecerán nuevos requerimientos o porque aparecerán errores en alguna de las fases. Es posible que el software sea modificado por su creador, pero lo más habitual es que lo sea por otro desarrollador. En cualquier caso, es

evidente que si se desea que sea fácilmente modificable, deberá cumplir primero el objetivo del apartado anterior, es decir, deberá ser legible y comprensible.

La metodología de diseño que se estudiará en capítulos posteriores ayudará a crear programas que tengan esta característica.

- **Su realización debe atenerse al tiempo y presupuestos establecidos.** En el entorno académico el desarrollo de software en un tiempo por encima del establecido sólo es significativo a la hora de conseguir una evaluación positiva del programa. Sin embargo, en el mundo laboral alargar el tiempo de desarrollo de un producto repercute directamente en la subida de costes, con la consiguiente insatisfacción del cliente. Evitar esto es una de las mayores preocupaciones de las empresas desarrolladoras. Existen técnicas y herramientas en el mercado que implementan metodologías para gestionar y monitorizar adecuadamente todo el proceso de construcción del software, de manera que haga posible el cumplimiento de tiempos y costes establecidos. Como no son objetivo de la materia de este libro no se hablará de ellas, pero se estudiarán en el módulo de Análisis y Diseño de Aplicaciones Informáticas [PIATTINI et al. 2003].