Jorge Vasconcelos Santillán

MANUAL DE CONSTRUCCION DE PROGRAMAS

México, 2000

Construcción de Programas
Jorge Vasconcelos Santillán
México, D.F.

Presentación

Esta obra pretende ser un enlace entre los textos de introducción a la programación y los de ingeniería de software.

Fue elaborada de modo sencillo, con abundantes ejemplos y sugerencias, para que resultara atractiva a los programadores prácticos (que tienden a detestar los libros teóricos y pesados), y los motive a mejorar la calidad de sus programas.

De igual modo puede ser útil para aclarar algunos conceptos sobre Análisis y Diseño de Sistemas e Ingeniería de Software, que suelen resultar confusos al estudiarse por primera vez.

La obra reune ideas de muchos textos, profesores y programadores expertos, así como experiencias del propio autor. No obstante, para que pueda completarse, el lector deberá agregarle sus propios conocimientos.

Es importante advertir que la obra sólo comprende técnicas para la construcción de programas procedimentales; aunque actualmente hay otras técnicas con mucho éxito, como la programación y diseño orientado a objetos, la programación visual o las técnicas de inteligencia artificial.

Una recomendación final: para que este libro le resulte más útil conviene que ya haya pasado muchas horas programando y conocer bien algún lenguaje, aunque no lo domine.

El Autor

Prefacio

"La buena programación no se aprende de generalidades, sino viendo cómo los programas significativos pueden hacerse claros, fáciles de leer, fáciles de mantener y modificar, pensados para los humanos, eficientes y confiables; aplicando el sentido común y buenas prácticas de programación. El estudio cuidadoso y la imitación de buenos programas dirige hacia una mejor escritura."

SOFTWARE TOOLS IN PASCAL
Kernigham y Plauger

La aparición de las computadoras personales acarreó innumerables ventajas, y una de las más notables fue permitir que cualquier persona pudiese convertirse con relativa facilidad en un programador.

Sin embargo, al evitarse la escrupulosidad que se requería para llegar a ser un programador de computadoras, se incurrieron en vicios nuevos. Desafortunadamente se ha producido una gran cantidad de *software basura*: programas de poca calidad, hechos al vapor; y que se entregan al consumidor -a precios bajos e inclusive altos-, pero sin soporte futuro. Por el contrario el software de calidad requiere no sólo de muchas líneas de código, sino también de muchas horas de trabajo para revisar cuidadosamente el proyecto, desde los borradores hasta el producto final.

Debido a las labores que envuelve el desarollo de un buen proyecto, los especialistas en cómputo se percataron de la necesidad de métodos que facilitasen la construcción de programas (aún mucho antes del advenimiento de las microcomputadoras). Tales métodos deberían cumplir tres funciones:

1º. Ayudar en la detección y corrección de errores,
2º. Asegurar la corrección, tanto de algoritmos como de programas, y
3º. Ayudar a desarrollar programas fáciles de comprender y modificar.

Esta obra, Manual de Construcción de Programas, presenta algunas de estas técnicas, y las pone a disposición del programador que se ha formado durante esta generación de computadoras.

La filosofía seguida está plasmada en la frase inicial de Kernigham y Plauger, pues se ejemplifica con muchos programas para que el lector aprecie aciertos y errores, y aprenda de la imitación. Los elementos teóricos son los mínimos para evitar caer en generalidades que resultan tan ilegibles como inaplicables. No está saturada de ejemplos que resulten difíciles de entender o que deban imitarse, sino por el contrario, sólo se presentan elementos básicos (ideas, programas o sugerencias), para que el programador reflexione y le surjan valiosas ideas.

Aun cuando resulte doloroso para muchos programadores jóvenes (generalmente formados durante el bachillerato), la práctica de la programación no es suficiente; se requiere de una pequeña dosis de teoría sólida; pero no en forma aislada, sino complementada con la práctica; de este modo las experiencias prácticas permitirán la asimilación o rectificación de los conceptos, al tiempo que la teoría ayudará a tomar decisiones y mejorar el desempeño práctico.

Este libro es básicamente un manual de consulta. Consta de siete secciones que pueden ser estudiadas independientemente. A lo largo del libro hay múltiples ejemplos, que conviene analízar con cuidado.

El Capítulo 1 menciona algunos conceptos de los debe estar consciente un programador que se interese por desempeñar un buen trabajo.

El Capítulo 2 es una introducción al Análisis y Diseño de Sistemas. Fue escrita para que los programadores especializados tengan nociones de las actividades que engloban al desarrollo de los sistemas de cómputo. Como no tiene el rigor de los textos especializados en la materia, puede ser utilizado como apoyo en los primeros cursos de programación avanzada a nivel bachillerato e incluso licenciatura.

El Capítulo 3 es un recordatorio sobre el importantísimo tema de los algoritmos, su necesidad y sus ventajas. Recuerda también los diagramas de flujo y profundiza en los conceptos sobre programación estructurada y programación modular, que suelen utilizarse sin conocerse.

El Capítulo 4 presenta el paso posterior al algoritmo, la creación de programas. Se basa en el artículo de Kernigham "Estilos de Programación", publicado en la revista *Computing Surveys* en 1974, cuya validez actual es indudable; además se presentan varias sugerencias para mejorar el desarrollo de programas, así como algunos criterios para medir su calidad.

El Capítulo 5 está dedicado al más tedioso trabajo que involucra la construcción de programas: la documentación. Describe los tres principales niveles de documentación: interna, técnica y para el usuario. Para facilitar esta fase, se proveen numerosas sugerencias y formatos básicos útiles para los programadores que documentan por primera vez.

El Capítulo 6 muestra ideas para depurar los programas, técnicas, describe algunos de los errores comunes, e incluso contiene un pequeño catálogo de programas con errores poco evidentes.

Finalmente, el capítulo 7 hace una breve introducción al tema de optimización de programas.

Reconocimientos

Esta obra es producto de años de experiencia, de ideas de muchos especialistas y de abundante material bibliográfico; y este espacio está dedicado precisamente a presentar un reconocimiento a ellos. De antemano ofrezco disculpas por las omisiones en que haya incurrido.

Las siguientes personas influyeron decisivamente en el contenido global de la obra:

Eduardo González y **Salvador Castro**, del Banco de México, ambos profesores de la Fundación Arturo Rosenblueth especialistas en Análisis y Diseño de Sistemas. Las sugerencias para construir Diagramas de Flujos de Datos se basan en los apuntes de clase de Eduardo González.

Peter Juliff, cuyas ideas fueron la base para construir los cuadros sobre las diversas formas de los módulos, los diagramas de estructura y de Nassi-Schneiderman.

Luis Daniel Soto Maldonado, que, como profesor de la Fundación Arturo Rosenblueth, aportó muchos conceptos para el mejoramiento de los programas y en especial sobre la Notación Húngara.

Brian Kernigham y a **Plauger**, cuyo artículo "Programming Style Examples and Counterexamples" es la base del capçitulo 4; espero que perdonen la adaptación que se hizo.

Nora Vasconcelos que elaboró el dibujo del sistema biológico del capítulo 2.

Leticia Mijangos, que elaboró varios documentos cuya estuctura y estilo sirvieron de base para elaborar las plantillas para los Manuales Técnico y del Usuario.

Los siguientes temas se inspiraron en otras obras:

El tema de acoplamiento y cohesión se basa principalmente en el libro de **Fairley**, Ingeniería de Software de Mc Graw Hill.

El cuadro Seis formas de decir: "Hola Mundo" se basa en un documento obtenido a través de **Internet**.

Los criterios para mejorar la calidad de un programa y las malas prácticas de programación se basan en una sección del libro de Paul M. Embree y Bruce Kimble, ***C Language Algorithms for Digital Signal Processing***. Prentice Hall, USA, 1991.
La apología sobre el goto fue inspirada por **Herbert Schildt**.

Los siguientes documentos aportaron ideas fundamentales para la elaboración de toda la obra:

Artículo de Prado Angeles, Ernesto. **Tips para Análisis y Diseño de Programas**. *Pc / Tips Byte*. México, Marzo 1992.

Apuntes del profesor Eduardo González sobre **Análisis y Diseño de Sistemas**.

Enciclopedia Práctica de Informática Algar. Barcelona, España, 1986.

Libro de texto de Bores, Rosario y Rosales, Román. ***Computación, Metodología, Lógica Computacional y Programación***. Mc Graw Hill, México, 1993.

Libros de **Luis Joyanes** sobre programación, publicados por Mc Graw Hill.

Kruse, Robert L. ***Estructura de Datos y Diseño de Programas***. Prentice Hall, México, 1988.

Yourdon, Edward. ***Análisis Estructurado Moderno***. Prentice hall. México, 1993.

Y finalmente, un especial reconocimiento al excelente libro de Peter Juliff, ***Program Design***, 3rd Ed. Prentice Hall, Australia 1990; lamentablemente agotado.

Contenido

Índice de cuadros y figuras

Capítulo 1

❖ *Lo que todo programador debe saber*

"Programar es asignar lógica al caos,
vida a lo inerte e inteligencia a lo ininteligible."

J. Galicia.

1.1 ¿Qué es un programador?

Un **programador** es un especialista en computación, que conoce las características y funcionamiento de la computadora y tiene la capacidad para describir la solución a un problema mediante una secuencia de pasos elementales, y posteriormente puede transformarlos en acciones que la máquina pueda efectuar.

Los programadores, junto con los analistas y los diseñadores, forman el equipo de especialistas encargados del desarrollo de sistemas útiles para resolver problemas mediante la computadora.

El analista estudia el panorama del problema, el diseñador propone una estrategia de solución y el programador se encarga de convertir esa estrategia en programas que la computadora pueda ejecutar.

Cuando el proyecto es muy pequeño, el programador mismo se convierte también en analista, diseñador y consumidor.

De cualquier manera, todo programador debe tener los conocimientos suficientes para comprender la labor que desempeñan los otros especialistas y en su caso para realizarla.

1.2 El trabajo del programador

Un programador tiene a su cargo varias responsabilidades, de entre las que destacan:

1. Construir algoritmos útiles y elegantes, o aplicar prudentemente los algoritmos y programas existentes que ya resuelvan el problema.

2. Construir una estructura de datos apropiada al problema.

3. Tener conocimiento de los procesos que ocurren detrás de las rutinas provistas por los lenguajes.

Es muy importante que un programador sea capaz de construir una aplicación compleja a partir de instrucciones muy elementales.

4. Construir soluciones a la medida y no adaptar los problemas a soluciones preestablecidas.

5. Usar las herramientas disponibles, pero evitando que una herramienta detenga o altere la solución a un problema.

6. Investigar aquellos conceptos involucrados en un problema que ignore. Si ocupará una programa ya existente, no debe limitarse a *capturar* los algoritmos; debe entender un programa antes de poder modificarlo o extraer partes de él.

Un compromiso de ética personal debería ser:
"No desarrollar un programa sin conocer la teoría que lo fundamenta"

7. Reconocer cuando es realmente útil usar la computadora y cuando se convierte en abuso. No debe delegar la responsabilidad personal a las facilidades que ofrezca la computadora.

Cuadro 1-1

LAS VIRTUDES DE UN PROGRAMADOR

I
Ser capaz de analizar un problema hasta comprenderlo completamente.

II
Ser capaz de diseñar una estrategia que describa los pasos requeridos para obtener la solución del problema.

III
Conocer el funcionamiento, capacidades y limitaciones de las computadoras.

IV
Dominar un lenguaje de programación en particular, y además conocer algún lenguaje adecuado a la solución del problema.

V
Ser capaz de evitar el perfeccionismo y equilibrar entre lo ideal y lo real.

1.3 Ventajas de una planeación

Nunca encienda la computadora si desconoce lo que va a hacer.

Planear significa pensar un poco antes de dedicarse a construir. Es una labor que implica *trabajo de escritorio*.

La planeación es especialmente importante para desarrollar programas. Si el problema es sencillo, la planeación será ligera (una simple inspección al problema); si el problema es complejo, la planeación será más estricta (pasar por las fases de análisis, diseño, implementación).

Si bien es cierto que elaborar un programa directamente sobre la computadora es fácil y cómodo, también es cierto que suelen resultar con algoritmos y estructuras de datos demasiado complicadas (excepto si el problema original es sencillo). Esto se debe a que es una tarea dispersa: hay que pensar el algoritmo, codificar y mecanografiar al mismo tiempo; esta convergencia de tareas aumenta las posibilidades de error y produce código difícil de depurar. El cuadro 1-2 (a) muestra un programa que se escribió directamente sobre el teclado, sin planearse.

Cuadro 1-2 (a)

Claro y obscuro (a)

```
/* Programa para leer un dividendo y un divisor y
  escribir el residuo de su división.
*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>

int Dividendo, Divisor;
int Modulo(int, int);

void main()
{
 clrscr();
 printf("Este programa devuelve el residuo de la division de dos números \n");
 printf("(Los números deben ser enteros y el dividendo mayor que el divisor) \n");
 printf("Presione [ENTER] para continuar o cualquier otra tecla para salir");

 if (getch() != 13)
    exit(0);
 delline();  printf("\n");

 printf("¿Cuál es el valor de el dividendo? ");
 scanf("%d",&Dividendo);

 printf("¿Cuál es valor del divisor? ");
 scanf("%d",&Divisor);
```

```
 if (Modulo(Dividendo,Divisor) != 0)
    printf("El residuo de %d entre %d es igual a %d \n", Dividendo, Divisor,
             Modulo(Dividendo,Divisor));
 else
    printf("La división de %d entre %d es exacta, el residuo es igual a 0 \n",
              Dividendo, Divisor);

 do { } while (kbhit() == 0);
 clrscr();
}

int Modulo(divndo, divsor)
int divndo, divsor;
{
 int Factor;
 Factor = 1;
 while (divsor <= divndo) {
   if ((divsor * Factor) == divndo)
    return(0);
    if ((divsor * (Factor+1)) > divndo)
    return(divndo - (divsor * Factor));
    Factor ++;
 }
 return 0;
}
```

Aun cuando pueda parecer tedioso, aplicar un método de planeación ahorra tiempo durante la codificación y la depuracion del programa, ya que se simplifican los algoritmos y las estructuras.

Si ya tiene muy arraigada la costumbre de programar directamente sobre el teclado, basta que siga algunas de estas técnicas para que mejore la calidad de su software. Pero no olvide que para problemas difíciles deberá dedicarle algunos minutos al trabajo de escritorio.

El cuadro 1-2 (b) muestra otro programa, que realiza la misma tarea que el anterior, para el cual se tomaron algunos minutos de reflexión, antes de sentarse a programarlo.

cuadro 1-2 (b)

Claro y obscuro (b)

```
/*
  Dados un dividendo y un divisor,
  escribir el residuo de su división.
*/

main ()
{
  int dividendo, divisor;
  int cociente, residuo;

  printf("\nValor del dividendo: ");
  scanf("%d",&dividendo);

  printf("\nValor del divisor: ");
  scanf("%d",&Divisor);

  /* Aplicar una división entera */
  cociente = dividendo / divisor;
  residuo  = dividendo - cociente*divisor;

  printf ("El residuo es %d\n",residuo);
  return (0);
}
```

1.4 Cómo planear las actividades

Para elaborar un programa lo primero que debe hacerse es definir sus objetivos y delimitar con precisión sus alcances; y enseguida planear todas las tareas que deberán desarrollarse para construirlo. Esto incluye tanto los elementos teóricos que deberán investigarse, como las opciones que ofrecerá el programa. Si le es útil, puede seguir este esquema:

1. Enlistar todas las actividades y opciones que el programa deberá efectuar, sin detallarlas, ni ordenarlas.
2. Jerarquizar la lista, considerando complejidad y tiempo disponible.
3. Detallar cada actividad, siguiendo este mismo esquema.
4. Asignar tiempos.

Al desarrollar las actividades trate de cumplir con los tiempos marcados. Si llegara a excederse en el tiempo asignado a un paso, deberá reconsiderar el tiempo disponible para el resto de las actividades.

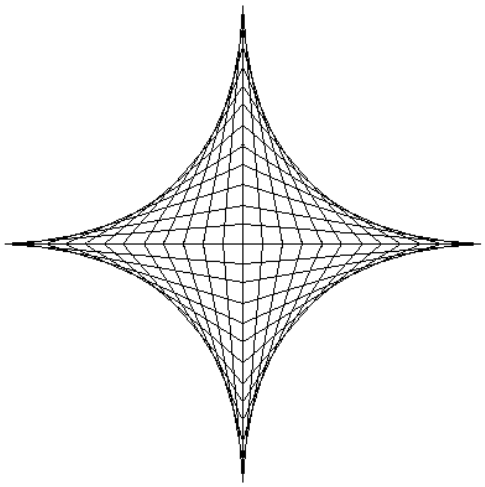
No importa que su planeación sea muy sencilla o poco profunda, siempre le será de mucha ayuda.

cuadro 1-3

Un ejercicio de planeación

Hay programas que muy fácilmente se pueden desarrollar directamente sobre la computadora, pero también hay otros que sin un análisis previo, no salen. Aquí hay un par de ejercicios que muestran ambos casos.

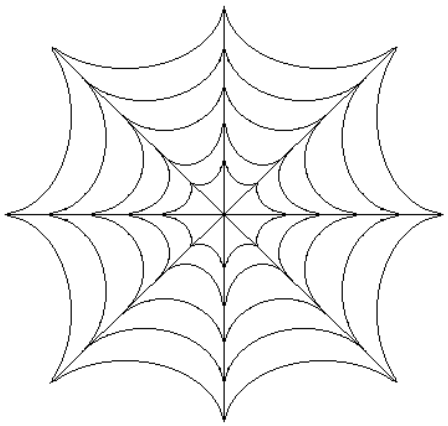
I. En el lenguaje de su predilección elabore un programa para dibujar en la pantalla esta figura :



(Si tarda más de quince minutos, entonces todavía le falta mucha práctica en la programación.)

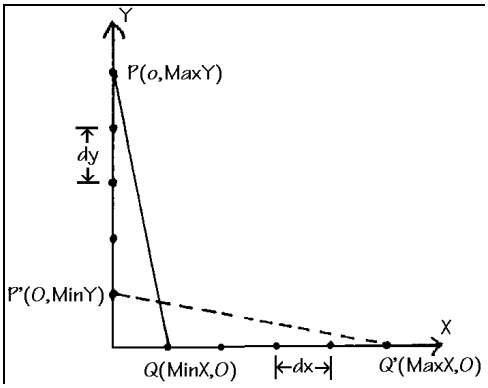
Si ya logró resolver este problema, pase al siguiente:

II. Elabore un programa para dibujar una telaraña similar a esta:



A continuación se muestra una estrategia para solucionar ambos problemas, pero no vaya a verla hasta no haberlo intentado usted mismo.

cuadro 1-3 (solución)



Para resolver el primer problema, considerése una recta **L**, que une los puntos ***P*** y ***Q***, como se muestra en el dibujo de la izquierda.

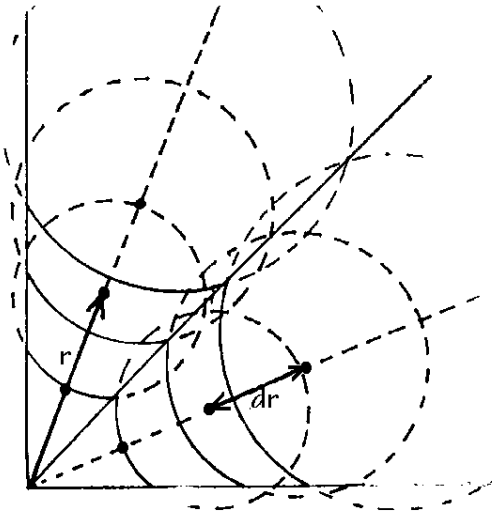
La figura se forma por el desplazamiento de la recta **L**, de tal modo que los puntos ***P*** y ***Q*** se muevan sobre los ejes. Para lograrlo, el punto inicial ***P*** siempre tendrá las coordenadas **(0,Y)**, y el punto final ***Q*** será **(X,0)**.

X y **Y** se moverán a intervalos regulares, desde un valor mínimo hasta un valor máximo. El algoritmo básico para hacer un cuadrante de la figura es:

```
Y = MaxY
X = MinX;
Para i desde 0 hasta 50 Hacer
    Inicio
    DibujaLínea (0,Y,X,0)
    y = y - 10;
    x = x +10
    Fn
```

Para generar la segunda figura efectuamos un análisis de su forma; posee ocho líneas rectas unidas por un mismo vértice y separadas por un ángulo de 45°. Entre cada línea se encuentran curvas separadas una distancia constante.

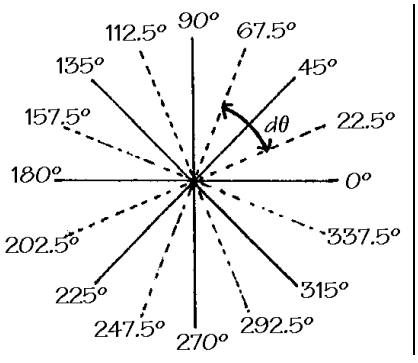
Para facilitar la construcción, consideremos que cada curva es el arco de un círculo, cuyo centro se encuentra en la bisectriz de las rectas que la sostienen (véase la figura de la derecha). Ese centro se desplaza, una longitud constante, sobre la bisectriz para generar cada nuevo arco.



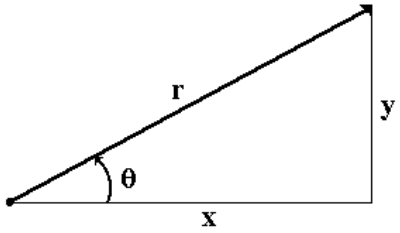
Dado que tenemos la distancia a la que se halla el centro, y sólo puede encontrarse en alguna de las rectas cuyos angulos conocemos, el centro se expresará como un vector con longitud y ángulo:

centro = (***r***,***q)***

El conjunto de rectas y sus ángulos se muestran a la derecha.



Sin embargo, para poder dibujar el arco requerimos que el centro se exprese en coordenadas cartesianas, para ello aplicamos las funciones trigonométricas:



$$y = r \cdot Sen(q)$$
$$x = r \cdot Cos(q)$$

Por lo tanto, para desplazar el centro, el ángulo ***q*** deberá tomar alguno de los valores 22.5, 67.5, 112.5,... y la longitud ***r*** deberá incrementarse en alguna cantidad constante.

El algoritmo básico será:

```
dr = 10
dθ = 45

r = 5
θ = 22.5
Para i desde 1 hasta 8 Hacer
   Inicio
   y = r*Sen(θ)
   x = r*Cos(θ)
   dibujar arco con centro en x,y y radio aproximado r/2
   r = r + dr
   θ = θ + dθ
   Fin
```

ahora que tenemos el algoritmo, ya podemos sentarnos a probarlo en la computadora. En su estado actual requerirá algunos ajustes que sólo apreciaremos una vez que haya aparecido algo en la pantalla; sin embargo, la parte más dificil ya fue salvada: conceptualizar una solución.

Evite trabajar en varios proyectos a la vez, pues las ideas suelen mezclarse y confundirse.

Capítulo 2

❖ *Análisis y diseño de sistemas*

2.1 Conceptos sobre sistemas

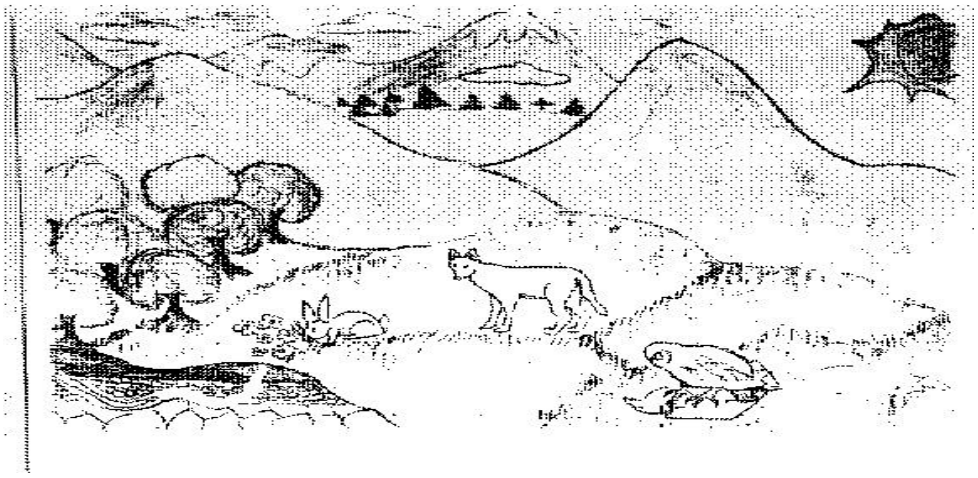
Un **sistema** es un grupo de elementos u objetos, y las relaciones que existen entre ellos. La figura 2-1 muestra como ejemplo un sistema biológico (ecosistema).

Un **sistema de información**[1] es aquél que tiene como elemento principal la información y considera las transformaciones que sufre.

Un sistema de información debe contar con los siguientes elementos:

1. Información (datos).
2. Transformadores de información.
3. Fuentes de información.
4. Receptores de información.
5. Almacenes de datos.

Figura 2-1: Un sistema biológico



La Tierra es un complejo sistema donde intervienen muchos actores: el Sol, las plantas, la atmósfera, el suelo y los animales. Las plantas producen el oxigeno que hay en el aire, para ello requieren de sustancias alimenticias alojadas en el suelo, de agua y de los rayos solares. En la atmósfera también se encuentran las nubes, producto de la evaporación del agua (depositada en mares y lagos); al enfriarse las nubes se deshacen como lluvia que aprovecharán todos los seres vivos y llenarán nuevamente los depósitos de agua. Las plantas también sirven de alimento a los animales herbívoros; éstos a su vez, son el alimento de los carnívoros; y todos ellos al morir enriquecen el suelo. Esta es una compleja red de intercambios de la que dependemos todos.

[1] El concepto de sistema de información no considera cuáles son los componentes físicos del sistema (es decir, no hace referencia a computadoras).

Un **sistema computacional** es un sistema de información que se apoya en las computadoras.

Las fuentes y los receptores de información son los usuarios. Las transformaciones se efectúan mediante programas en la computadora y los almacenes suelen ser archivos en discos o cintas (e incluso la memoria RAM).

Al estudiar sistemas destaca el concepto de ***modelo***. Básicamente un modelo es una representación abstracta de la realidad. Los modelos permiten estudiar los fenómenos, objetos y sistemas, sin considerar detalles superfluos. Cuando se ha completado el estudio y se tienen conclusiones, éstas pueden aplicarse a la realidad con cierta confiabilidad.

Cuadro 2-1

Diferencia entre sistema y programa

La forma más sencilla de explicar la diferencia entre sistema y programa es recordar un antiguo refrán popular: *"una golondrina no hace un verano"*; con esto queremos hacer notar que un programa asilado (por largo que sea) no es un sistema. Desafortunadamente hay una tendencia popular a llamar "sistema" a todo programa y también a considerar que todo sistema involucra computadoras.

Un programa es simplemente un grupo de instrucciones que la computadora debe seguir para transformar datos, mientras que un sistema computacional debe integrar varios componentes, desde varios programas hasta varias computadoras.

2.2 Desarrollo de sistemas

Por *desarrollo de sistemas* se conoce a una disciplina dedicada a la construcción de sistemas donde las transformaciones y flujos se efectúen de la mejor manera.

Para desarrollar sistemas no necesariamente se requiere de equipo de cómputo (una organización es un sistema), pero en esta obra nos concentraremos en los sistemas computacionales.

Un sistema computacional puede constar desde un pequeño grupo de programas para operar en pocas computadoras (como es el caso del sistema operativo), hasta los requerimientos para control de vuelos espaciales o transmisión vía satélite.

Cuando se requiere resolver ciertas necesidades y el sistema no existe o el existente se ha vuelto inadecuado, entonces es necesario construir un sistema de cómputo nuevo.

Las fases que se siguen para desarrollar sistemas computacionales se conoce como *ciclo de vida del software* y esencialmente consta de cinco etapas:

1. Especificación de Requerimientos.
2. Análisis.
3. Diseño.
4. Implementación y Pruebas.
5. Mantenimiento.

De acuerdo a como se suceden las fases del ciclo de vida del software, han sido propuestos dos modelos para el desarrollo de sistemas: el **modelo cascada** y el **modelo espiral** (figura 2-2).

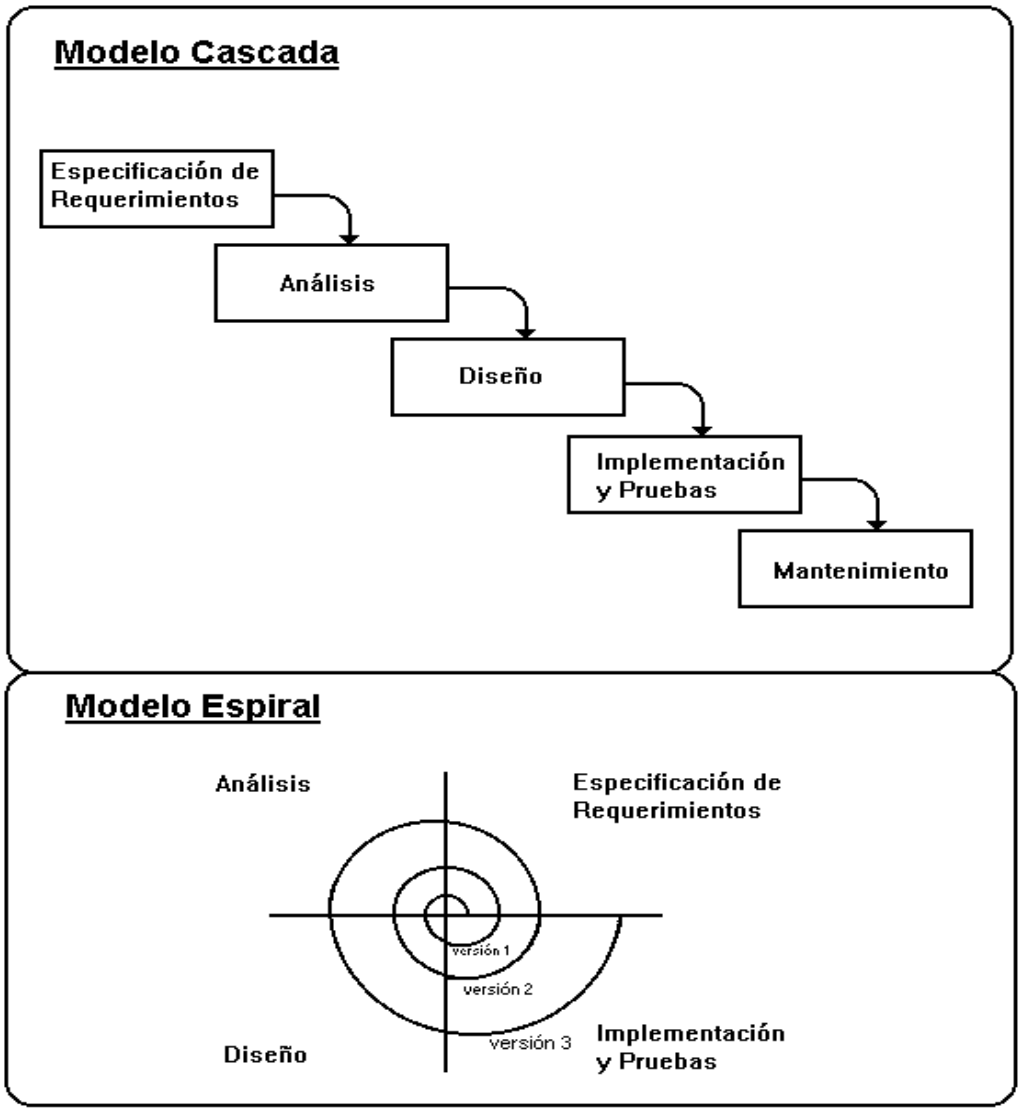


Figura 2-2. Modelos del Ciclo de Vida del Software

En el modelo cascada, el resultado de cada fase es *alimentado* a la siguiente, terminándose el proceso con el mantenimiento. En esta última fase se efectúan los cambios necesarios para que el sistema siga funcionando ante nuevas necesidades. El software termina su vida cuando deja de cumplir con el propósito para el que fue creado, y ya no es posible seguir actualizándolo.

El modelo espiral elimina la fase de mantenimiento, por lo que cualquier modificación al sistema deberá cumplir con todas las fases nuevamente. Esto obliga a que cada fase sea muy bien hecha y esté documentada.

2.3 Las fases del desarrollo de sistemas

La figura 2-3 muestra el desarrollo de sistemas (o programas) como un proceso de caja negra, entra una serie de requerimientos y a la salida se obtiene el sistema.

En esta sección se explicará cada fase.

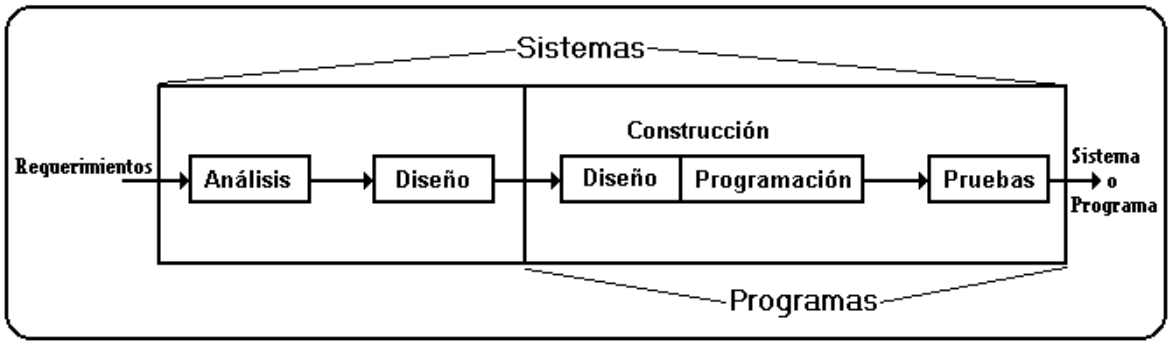


Figura 2-3. Desarrollo de Sistemas o Programas

Especificación de Requerimientos

Este es el primer paso para poder construir el sistema. Consiste en elaborar una lista que indique* :

1. ¿Qué necesito?
2. ¿Cuál es el esbozo del problema?
3. ¿Qué espero lograr con el sistema?
4. ¿Con qué recursos cuento actualmente?
5. ¿Qué posibilidades tengo de obtener más recursos?

No se puede desarrollar un buen sistema sin antes haber comprendido el problema y la teoría asociada.

* Si desea ser muy formal incluya también una exposición de motivos, descripción de antecedentes y del estado actual, organigrama de la institución, ventajas esperadas, etcétera.

La Fase de Análisis

El análisis nos dice: *"las cosas están así"*.

La fase de análisis consiste en hacer un estudio del estado existente de las cosas. Esta etapa es muy importante, pues delimitará con precisión al problema y los alcances de la solución.

El análisis permite construir un modelo del problema, que pueda utilizarse para elaborar el modelo de la solución.

El análisis permite detectar las características y problemas en el sistema.

El resultado del análisis debe ser una descripción completa y detallada del funcionamiento del sistema bajo estudio, ya sea que exista o que sólo sea una propuesta.

cuadro 2-2

Cómo analizar un proyecto

1. **Sumergirse en el problema**
2. **Hacer una descripción de lo que está siendo requerido.**
3. **Separar las salidas deseadas de los datos que pueden ser tomados como entradas.**
5. **Identificar las restricciones al proyecto.**

La Fase de Diseño de Sistemas

El diseño nos dice: *"ésta es la solución"*

La fase de diseño se encarga de la preparación de un sistema de información, independientemente de cómo se vaya a implementar.

El resultado del diseño es un modelo lógico, que describe el funcionamiento y especifica detalladamente cada parte del sistema.

El Diseño de Programas

La etapa de diseño de programas relaciona (o enlaza) el modelo lógico (elaborado al diseñar el sistema) con las características específicas del hardware.

Esta fase no suele aparecer en los modelos de desarrollo de software; sin embargo si un sistema de cómputo se desarrolla correctamente, hay un paso intermedio entre el diseño del sistema y su implementación.

El diseño de programas se encarga de construir conceptualmente la solución

Esta fase debe presentarse siempre antes de sentarse a programar.

Durante el diseño de programas no se proponen soluciones alternativas, sólo se construye.

La Implementación y las Pruebas

La implementación de programas se encarga de construir *físicamente* la solución

Durante esta fase se construye y corrige el sistema requerido.

En algunas ocasiones, para construir la solución se recurre a **prototipos**, que son programas *huecos* que sólo muestran algunas de las opciones del sistema. El prototipo permite que el usuario defina aún más los requerimientos. Para facilitar modificaciones al prototipo se debe programar modularmente.

cuadro 2-3

Sugerencias para Análisis y Diseño

1. Se considera que el desarrollo de un proyecto es tan grande o complejo que no se pueden recordar todos los detalles, por lo que hay que usar alguna metodología.

2. Documentar cada etapa del proyecto (instalación, uso, desarrollo y mantenimiento).

3. Dedicar todo el tiempo necesario para establecer una definición exacta acorde a las necesidades reales.

4. Elaborar un documento de requisitos, que consiste en una especificación completa y consistente (sin contradicciones) de lo que se tiene que hacer, pero sin especificar cómo deberá hacerse.

   Introducción: Necesidad del sistema, contexto, breve descripción de funcionamiento, estructura del documento y notación.
   Requerimientos de Hardware, configuración mínima y óptima.
   Descripción simple de servicios al usuario.
   Glosario
   Indice

   Siempre cerciórese de que entiende el problema por completo.

5. Enfoque evolutivo basado en construcción de prototipos (para dar alguna idea al usuario, experimentación). Modelo que incluya sólo algunas de las opciones que se tendrán en el sistema final.

6. Adoptar una metodología de programación y ser consistente durante todo el desarrollo.

7. Cuando tenga que desarrollar programas para otras personas tenga en cuenta lo siguiente:

   Es muy raro que los usuarios estén completamente conscientes de la verdadera naturaleza de sus problemas y suelen esperar que el experto les diga cuál es el problema y cómo resolverlo.

   Los usuarios no están interesados en los detalles de funcionamiento, sólo desean respuestas del programa.

8. Para poder determinar la función del programa deseado por el usuario: Empezar con una vaga enunciación de intenciones, que se afine con persuación e interrogación para obtener las especificación de requerimientos.

   ¿ Cómo se utilizará ?
   ¿ Qué aspecto tendrá ?
   ¿ Cómo se organizará ?
   ¿ Qué deberá hacer (procesos y cálculos)?

   Al finalizar se deberá poder enunciar con claridad las necesidades del usuario. Señalar algunas de las tareas específicas del programa.

   En proyectos personales por lo general el programador es también diseñador, analista y consumidor. Aun así haga un esfuerzo de explicarse a sí mismo los problemas, las soluciones y las necesidades con la misma facilidad que si estuviera hablando con otra persona.

cuadro 2-4

Quién y cuándo del Análisis de Sistemas.

1960 - Edsger Dijkstra de la Universidad de Eindhoven alertó a los programadores sobre el peligro de usar indiscriminadamente el "*goto*".

1966 - Böhm y Jacopini demostraron que para construir programas sólo se requieren tres estructuras de control (teorema de la estructura).

70's - Se desarrolla la programación estructurada, como una técnica que incorpora y formaliza el teorema de la estructura y la consecuente eliminación del "*goto*".

Las aportaciones académicas de Niklaus Wirth y David Parnas se convertirían más tarde en la Ingeniería de Software.

80's - Se desarrollan las metodologías estructuradas. Eduard Yourdon, Tom de Marco, Gerry Weinberg y Michael Jacson trasladaron los conceptos teóricos en técnicas prácticas y heurísticas que los programadores pudieran entender y seguir.

2.4 Metodologías estructuradas: el análisis

Se llaman metodologías estructuradas a un conjunto de técnicas y convenciones que sirven para hacer un modelo del problema que pueda evolucionar suavemente hasta obtenerse el modelo de la solución[2].

Seguir una metodología ayudará a ordenar y planear el trabajo.

Usar metodologías estructuradas ofrece dos beneficios principales:

1. Permite documentar el sistema al mismo tiempo que se le construye.

2. Establece un mapa (o plan de acción) que le permitirá al programador visualizar claramente los módulos integrantes y los parámetros requeridos.

"Es mejor usar una metodología que no usar nada."

Salvador Castro

Dentro de las metodologías estructuradas hay análisis estructurado, diseño estruturado, revisiones estructuradas y algunas otras. En este libro sólo se tratará el primero.

El análisis estructurado es una técnica que facilita el estudio de un problema para construir o modificar un sistema.

Esta metodología se integra de tres herramientas:

1. **Diagrama de Flujo de Datos** (DFD)
2. **Diccionario de Datos** (DD)
3. **Especificaciones Estructuradas de Proceso** (minispec).

1. Diagramas de Flujo de Datos

Un **diagrama de flujo de datos** es una representación atemporal (no depende del tiempo, es decir, puede usarse en cualquier momento) en forma de red de las partes componentes de un sistema y de las trayectorias seguidas por los datos. El diagrama de flujo de datos es totalmente diferente de los Diagramas de Flujo usados para representar algoritmos (figura 2-4).

[2]Las metodologías estructuradas se estudian con profundidad en textos de Análisis y Diseño de Sistemas e Ingeniería de Software.

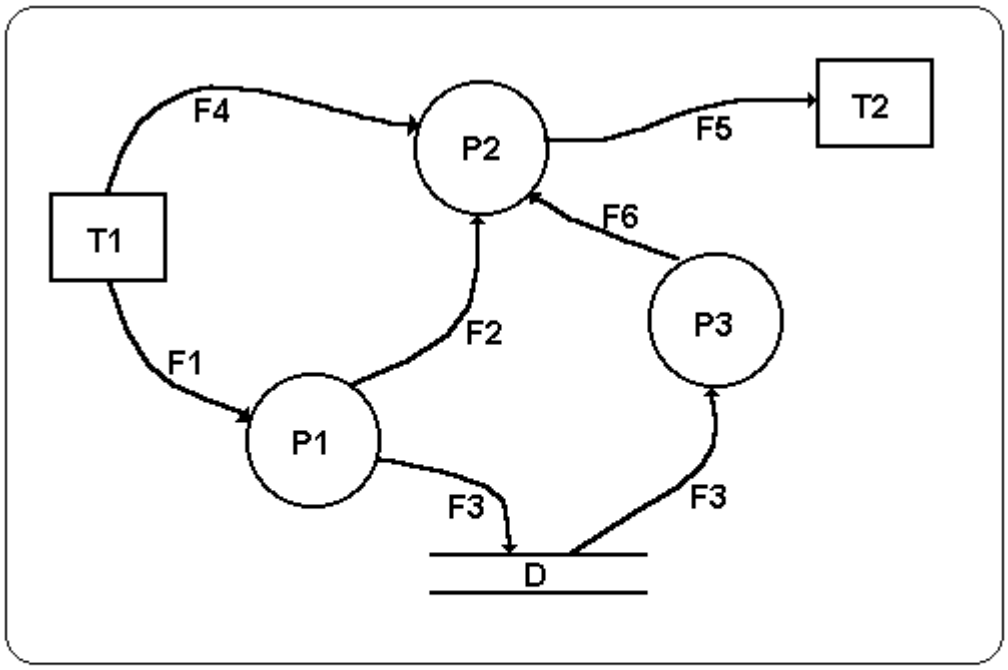


Figura 2-4. Un diagrama de flujo de datos: T1 emite los datos F1 y F4. P1 transforma F1 en los datos F2 (y lo envía a P2) y F3 (que guarda en el depósito D). P3 recupera F3 y lo transforma en F6. Finalmente el proceso P2 toma los datos F4, F2 y F6; y los transforma en F5 para enviárselo al receptor T2.

Esta herramienta facilita la subdivisión de un sistema en sus componentes lógicos (conceptuales).

La figura 2-5 muestra los dibujos usados en los diagramas de flujo de datos y el cuadro 2-7 ilustra su uso con un ejemplo.

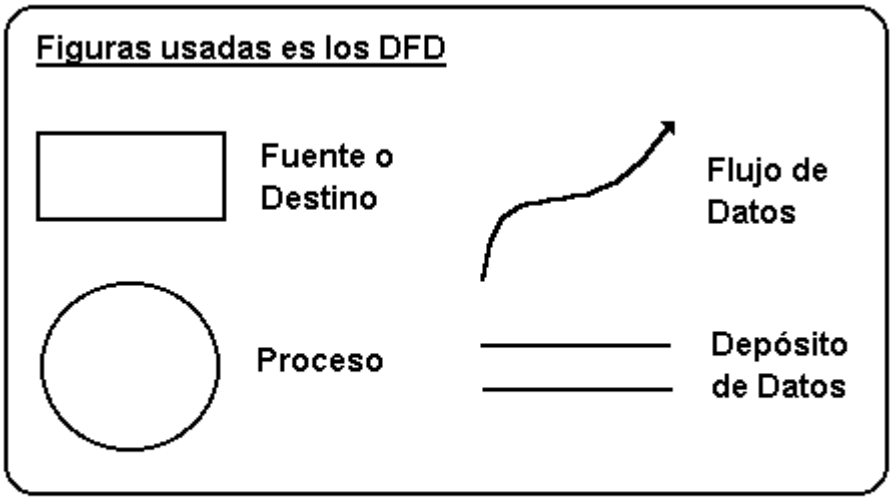


Figura 2-5. Figuras empleadas en los Diagramas de Flujo de Datos.

FLUJO DE DATOS

Es una *tubería* imaginaria por la cual fluyen *paquetes de información* de estructura conocida. Esta tubería permite el desplazamiento de los paquetes en cualquier momento; es decir, no indica el momento preciso en el que se moverá información.

El flujo de datos se dibuja como una flecha que conecta dos procesos. El sentido de la flecha indica la dirección en que viajan los paquetes de datos. A la flecha se le asigna un nombre (relacionado con el nombre del paquete) que se registra en el diccionario de datos.

Los paquetes diferentes deben fluir por tuberías diferentes. Una analogía práctica: el agua fría y el agua caliente fluyen por tuberías diferentes.

Si al momento de nombrar un flujo de datos observa que representa órdenes (describe un dato que no es procesado sino interpretado), debe eliminarse ese flujo.

Notación para nombrar flujos de datos (ver cuadro 2-7):

1. Nombrar al flujo con el nombre que represente al dato o a lo que se sabe de ellos.
2. Usar mayúsculas o guiones para separar las palabras. Suprimir artículos, conjunciones y preposiciones.
3. Dos flujos distintos no deben tener el mismo nombre.
4. Los flujos que se mueven entre depósitos no requieren etiquetarse.

PROCESO

Es un elemento del sistema que se encarga de transformar datos. Los flujos de datos llegan y salen del proceso.

Se dibuja como un círculo, llamado *burbuja*, con un nombre y un número en su interior.

Una burbuja indica que se debe efectuar un trabajo, pero no debe mencionar cómo hacerlo.

Es muy importante considerar que los procesos no destruyen ni crean datos, sólo los transforman.

Notación para nombrar procesos (ver cuadro 2-7):

1. El nombre lo tomará según los flujos de E/S. Si no lo puede tomar significa que es un proceso complejo que deberá descomponerse.
2. Generalmente deberá ser una oración imperativa de verbo activo.

DEPOSITOS DE DATOS

Es un elemento que almacena datos para su uso posterior.

Se dibuja como dos líneas paralelas con el nombre de los datos que guarda, y se registra en el diccionario de datos.

Físicamente un depósito de datos puede ser un archivo, una arreglo, una variable, un archivero, una hoja de papel, etcétera.

Los depósitos actúan como *buffer* cuando se requiere de interfaces en los sistemas (las interfaces son elementos que conectan dos partes de diferente naturaleza, por ejemplo dos computadoras diferentes o la computadora y el usuario).

FUENTES O DESTINOS

Son elementos externos que envían información al sistema o requieren de los resultados. Pueden ser personas u otros sistemas.

Tanto las fuentes como los destinos se dibujan con un rectángulo en cuyo interior está el nombre del elemento externo.

Se requiere de una fuente/destino cuando no importa saber cómo se produjeron los datos o qué uso se les dará.

cuadro 2-5

Sugerencias para dibujar Diagramas de Flujos de Datos

1. Concentrarse primero en visualizar los flujos de datos: cuáles son, de dónde vienen y a dónde van.
2. Etiquetar y anexar en el diccionario de datos todos los flujos de datos.
3. Tratar de visualizar cómo se enlazan o separan los flujos de datos para obtener las salidas.
4. Poner burbujas en blanco donde se requiera transformar datos. La convergencia o divergencia de flujos indica la existencia de un proceso (transformación). Para etiquetarlas, tomar como base los nombres de los flujos que entran y salen. Las burbujas no deben crear ni destruir datos.
5. Considerar la situación normal del sistema, ignorando la inicialización y terminación.
6. Omitir el manejo de errores.
7. No mostrar flujos de control.
8. No considerar restricciones de hardware ni de recursos.

10. Estar preparado para comenzar de nuevo.

CONSTRUCCION DE DIAGRAMAS DE FLUJOS DE DATOS

1. Dibujar un diagrama de contexto del sistema: representar con un círculo a todo el sistema e indicar las entradas al sistema con flechas que se dirigen hacia la burbuja, y las salidas con flechas que salen de la burbuja. Tanto las entradas como las salidas deben registrarse en el diccionario de datos.

Si se ignoran datos en el diagrama de contexto, deberán ignorarse en lo sucesivo.

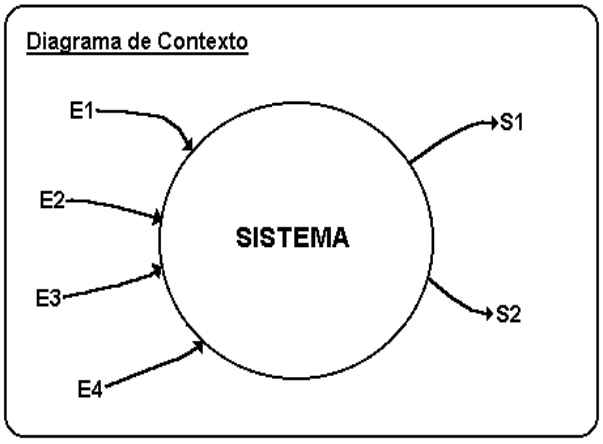


Figura 2-12: Diagrama de Contexto

El diagrama de contexto deberá descomponerse en diagramas de niveles inferiores, cada vez más detallados.

2. Según las características propias del sistema, hacer un diagrama de conectividad o pasar directamente al diagrama de nivel 0.

Un diagrama de conectividad consiste en trazar, sobre el diagrama de contexto, las líneas que unen los flujos de entrada para formar los flujos de salida. Terminado este diagrama se dibujan burbujas en todos los puntos donde los flujos convergen o divergen, obteniéndose el diagrama de nivel cero.

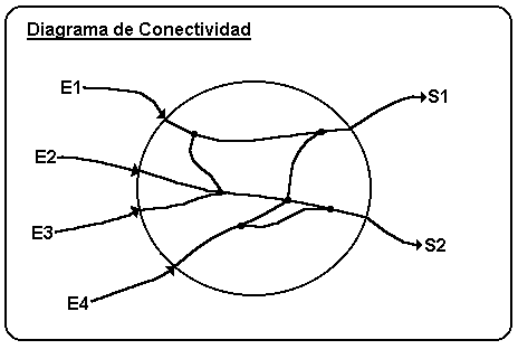


Figura 2-13: Diagrama de Conectividad

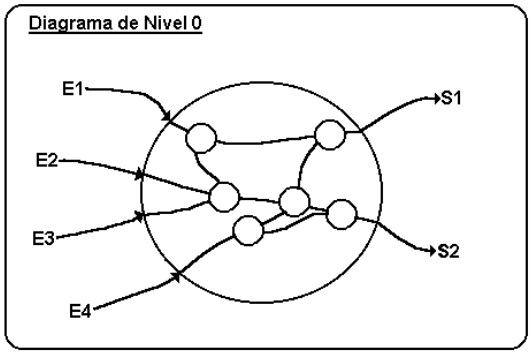


Figura 2-14: Diagrama de Nivel 0

El diagrama de nivel cero representa las principales actividades del sistema que se está estudiando, e indica la complejidad del mismo.

3. Así como se subdividió el diagrama de contexto, se sigue subdividiendo cada una de las burbujas en los diagramas de flujo de datos, utilizando como referencia el número de burbuja del cual provienen (burbuja padre).

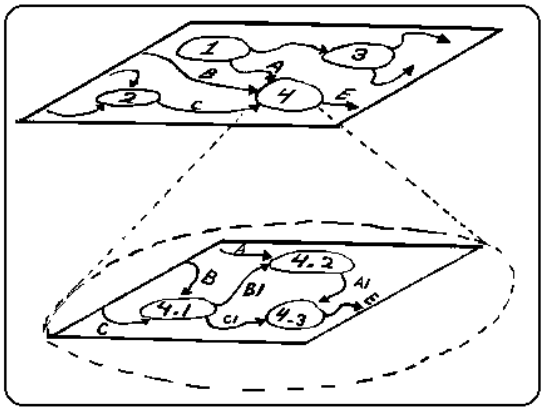


Figura 2-15: Diagramado por niveles

Los flujos que entran o salen de la burbuja padre, deben entrar y salir del diagrama hijo.

La descomposición debe continuar hasta que todas las burbujas sean *primitivas funcionales*, es decir, procesos que cumplen alguna de las siguientes condiciones:

a) Tienen una sola entrada y una salida.
b) No existen flujos de datos internos por lo que no se puede fragmentar.
c) Se puede escribir fácilmente la especificación de proceso.

Es muy importante que cada diagrama de flujo de datos contenga entre 7 y 9 burbujas (excepto el diagrama de nivel cero). Un número mayor de burbujas indica la necesidad de descomponer mejor el diagrama.

2. Diccionario de datos

Un diccionario de datos es un compendio que describe los datos mostrados en los diagramas de flujo de datos. Debe enlistar alfabéticamente todos los datos del sistema.

Los datos anotados deben cumplir lo siguiente:

a) Describir el significado de los flujos y depósitos mostrados en los diagramas de flujo de datos.
b) Indicar los rangos y unidades de los datos elementales.
c) Indicar cómo se conforman los datos compuestos.

En un diccionario de datos se utilizan los siguientes símbolos:

= descripción del dato
+ enlace entre datos
() dato opcional
{} repetición de datos
[] datos alternativos
| separador de opciones

** comentario
@ dato llave

3. Especificaciones de Proceso

Una especificación estructurada de proceso o miniespecificación es un texto que describe lo que sucede en el interior de un proceso, indicando qué debe hacerse para transformar las entradas en salidas.

Dicho en otras palabras es un algoritmo y por lo tanto puede describirse mediante pseudocódigo (lenguaje estructurado) o diagramas de flujo.

cuadro 2-6

Las ventajas del análisis estructurado para el programador

Aun cuando el análisis estructurado es una herramienta dirigida propiamente al analista, el programador puede beneficiarse directamente con ella.

I. Las burbujas del diagrama de flujo de datos permiten reconocer los módulos que integrarán al programa, y los flujos de datos muestran los parámetros que deberán pasarse.

II. El diccionario de datos permite determinar fácilmente los tipos de datos que deberán definirse en el programa.

III. Las miniespecificaciones serán las base del algoritmo que se programará en cada módulo.

cuadro 2-7

Una sencilla aplicación del análisis estructurado

En este ejemplo consideramos la operación básica de un videoclub:

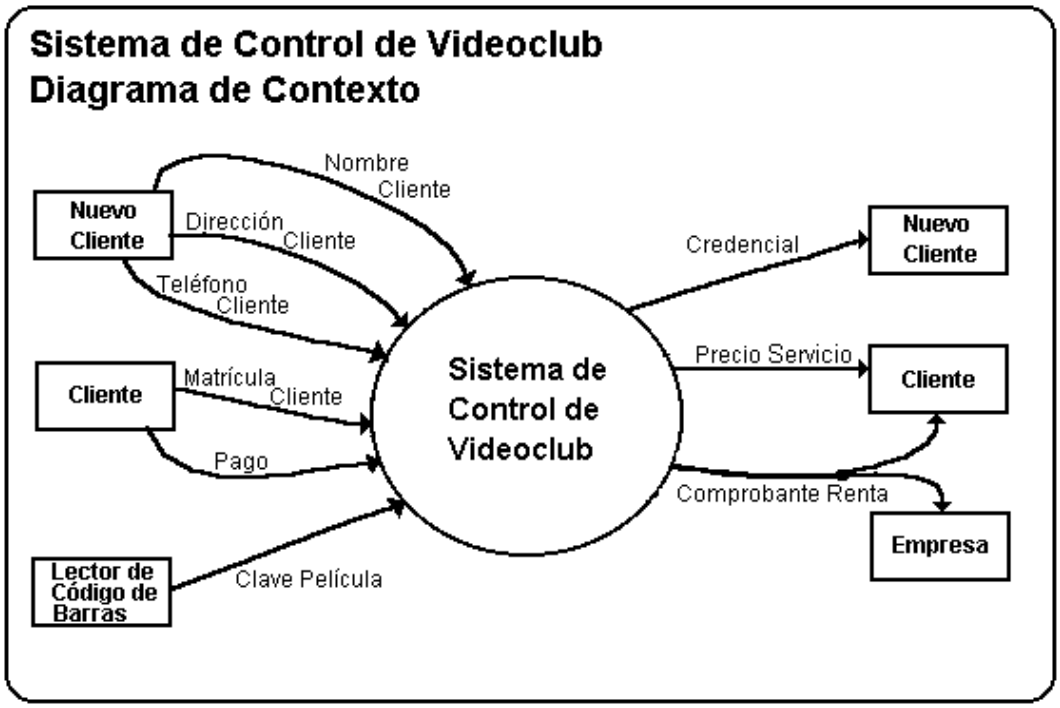
1. dar de alta nuevos clientes,
2. rentar películas y
3. recibir la devolución de los videocassettes.

Los procedimientos son: (a) a los nuevos clientes se les asigna una credencial con su nombre y matrícula; (b) para rentar películas basta con mostrar la credencial, pasar la película deseada por una lectora de código de barras y efectuar el pago correspondiente; y (c) para la devolución basta entregar la película (que vuelve a pasar por la lectora).

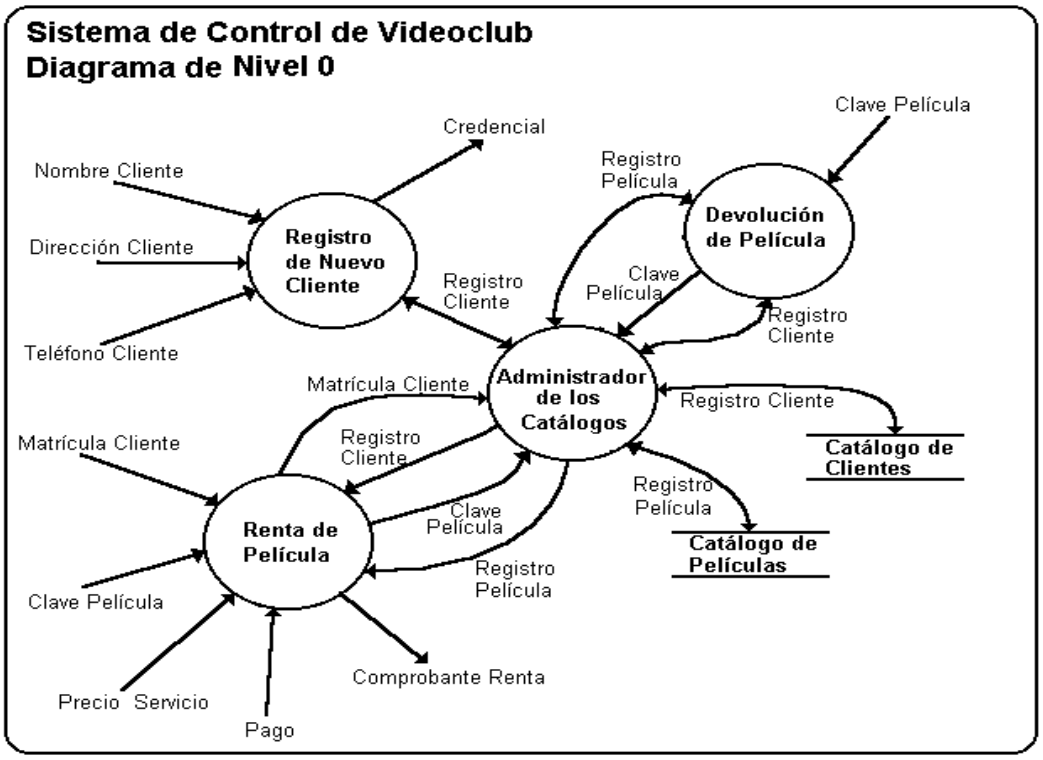
Para mostrar cómo se aplican las herramientas del análisis estructurado, primero veremos los diagramas de flujos de datos, luego las especificaciones de proceso y finalmente el diccionario de datos, sin embargo estas tres se construyen de manera paralela y cada componente está referido en las otras.

I. Diagramas de flujo de datos

Paso 1: Representar al sistema en su conjunto mediante un diagrama de contexto



Paso 2: Descomponer la burbuja de contexto en el diagrama de nivel 0.



Este ejemplo es muy sencillo, por lo que una vez que hemos alcanzado este nivel las burbujas ya son primitivas, por lo que podemos pasar a la especificación de proceso. (Si el sistema fuese más complejo, cada una de las burbujas del nivel cero deberá subdividirse, tal como se hizo con el diagrama de contexto).

Observe que el diagrama de flujo de datos resulta útil para determinar los módulos que deberá tener el programa o sistema y los parámetros que deberán pasarse entre ellos.

II. Especificaciones de proceso

1. Registro de Nuevo Cliente

Solicitar **Nombre_Cliente**
Solicitar **Dirección_Cliente**
Solicitar **Teléfono_Cliente**
Asignar **Matrícula_Cliente**
Integrar **Registro_Cliente**
Actualizar **Catálogo de Clientes**
Entregar **Credencial**

2. Renta de Película

Por cada **Clave_Película**
Solicitar **Matrícula_Cliente**
Actualizar servicios en **Catálogo de Clientes**
Obtener **Registro_Película**
Actualizar existencias en **Catálogo de Películas**
Calcular **Precio_Servicio**
Informar **Precio_Servicio**
Recibir **Pago**

Entregar **Comprobante_Renta**

3. Devolución de Película
Por cada **Clave_Película**
Actualizar existencias en **Catálogo de Películas**

(Los procesos que implican movimientos en los catálogos se efectúan a traves del administrador de catálogos, que es en realidad un manejador de bases de datos, por lo que no se le detallará aquí.)

Observe que las especificaciones de proceso resultan útiles para diseñar y construir los algoritmos que se usarán en los programas.

III. Diccionario de datos

Clave_Película = * Clave asignada al título grabado en videocassette *

Catálogo_de_Clientes = * Archivo(s) que contienen los datos de los usuarios *
= { Registro_Cliente }

Catálogo_de_Películas = * Archivo(s) que contienen los datos de las películas *
= { Registro_Película }

Comprobante_Renta = * Recibo del servicio prestado *
Nombre_Película + Precio_Servicio + Nombre_Usuario

Credencial = * Identificación de un usuario del videoclub *
Nombre_Cliente + Matrícula_Cliente

Dirección_Cliente = * Domicilio particular del cliente *
Calle + Número + [Colonia|Municipio] + Ciudad + Código_Postal

Matrícula_Cliente = * Número de registro asignado al cliente *

Nombre_Cliente = * Nombre propio del cliente *
Apellido_Paterno + Apellido_Materno + Nombre

Pago = * Cantidad monetaria requerida por el servicio *

Precio_Servicio = * Total a pagar por la renta *

Registro_Cliente = * Configuración de los registros del catálogo de clientes *
Nombre_Cliente + Matrícula_Cliente + Teléfono_Cliente +
Dirección_ Cliente + [Observaciones]

Registro_Película = * Configuración de los registros del catálogo de películas *
Nombre_Película + Clave_Película + Existencias

Teléfono_Cliente = * Teléfono particular del cliente *

Observe que el diccionario resulta útil para determinar las variables y estructura de los registros que se usarán en los programas.

Una advertencia, este ejemplo sólo es una propuesta del autor, diferentes analistas pueden tener opiniones diferentes, incluso usted mismo puede encontrar posibles mejoras.

Capítulo 3

❖ *Diseño de algoritmos*

3.1 Etapas del desarrollo de programas

Un programa se elabora para resolver un problema mediante la computadora. En computación un **problema** consiste en obtener ciertos resultados a partir de unos datos previos.

Para desarrollar un programa y tener mayor confianza sobre su correcto funcionamiento es conveniente seguir las etapas presentadas en la figura 3-1.

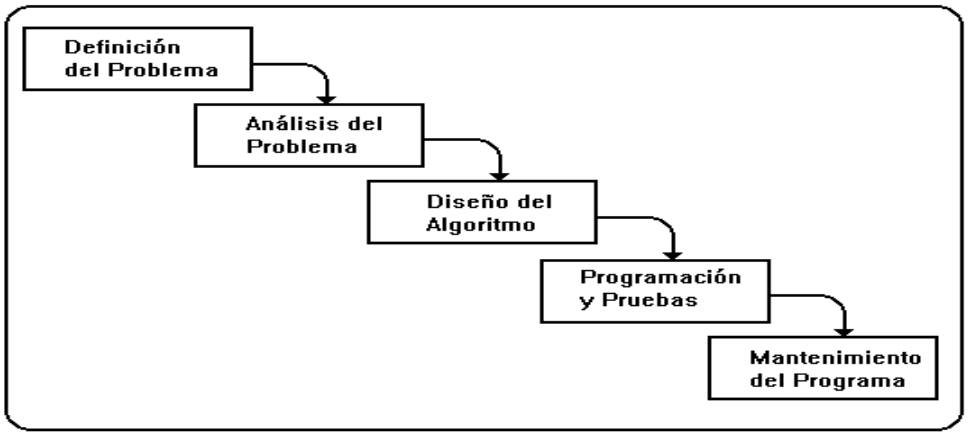


Figura 3-1: Etapas del Desarrollo de Programas

- **Definición del problema**

Consiste en establecer cuáles son los datos disponibles (para resolver el problema) y a qué resultados se desea llegar.

Para superar esta fase se debe redactar un texto que indique qué es lo que se pretende solucionar y contenga toda la información útil al respecto.

Algunas veces la definición del problema ya está elaborada (como en los libros de física). Sin embargo llega a suceder que no toda la información proporcionada es útil.

Para mejorar la comprensión del problema, pregunte todo lo necesario a las personas que están relacionadas con él.

- **Análisis del problema**

Consiste en hacer un examen profundo del problema y su situación, con el fin de entenderlo por completo y poder proponer una solución.

Durante esta fase se debe definir con precisión qué información es útil y cuál no. Si se descubre información faltante, deberá investigarse.

Como resultado de esta fase se propondrá una solución, pero sin indicar cómo obtenerla.

> Es imposible especificar un problema que no se ha entendido. Si usted es capaz de explicarlo a otra persona, sin dar por supuesto que ella sabe lo que quiere comunicarle, entonces ya ha entendido el problema.

- **Diseño del algoritmo**

Consiste en planear y especificar la estrategia que se seguirá para alcanzar la solución de un problema.

El producto de esta fase es un documento que detalle los pasos que posteriormente se convertirán en programas.

- **Programación y pruebas**

Consiste en codificar el algoritmo diseñado en un lenguaje de programación y probar su funcionamiento para corregir errores.

- **Mantenimiento**

Consiste en modificar el programa según vayan apareciendo nuevas necesidades.

El cuadro 3-2 muestra un ejemplo muy sencillo donde se aplican cada una de estas fases.

> No siempre puede (ni conviene) seguirse el orden estricto del diseño de programas. Hay que efectuar experimentos para ir modificando los planes iniciales.

Cuadro 3-2

Centrar un letrero*

● ANALISIS

1o. **Enunciado del problema**: Centrar un letrero cualquiera de manera similar a como se haría con una máquina de escribir.

2o. **Solución del problema**: Para centrar un letrero es necesario dejar espacios antes de escribirlo, entonces hay que desarrollar un método para calcular esa cantidad de espacios y aplicarlo.

● DISEÑO

1o. **Descripción sencilla del algoritmo**

Se cuentan las letras (incluyendo los espacios) en el letrero y se resta del total de columnas en la hoja ("golpes por página"). El número que resulta corresponde al total de espacios en blanco que quedarían si se escribe la línea desde el margen izquierdo. Por lo tanto, para centrar se deben repartir los espacios en dos porciones iguales, una quedará antes y otra después del letrero.

2o. **Consideraciones iniciales**

a) Como el letrero puede ser cualquiera, se le deberá preguntar al usuario cuál es, y almacenarlo en una variable (que llamaremos **LETRERO**).

b) El ancho es el total de columnas o golpes por página; se fija al principio del trabajo y no se varía. Como el ancho puede ser de 80, 70, 65, o cualquier otro, lo sustituiremos por **ANCHO**.

c) Se requerirán de dos variables más: **LONG**, donde se guarde el total de letras (caracteres) contenidos en el letrero y **NUM_ESPACIOS** que tendrá el valor de espacios en blanco necesarios.

3o. Algoritmo

(1) Inicio
(2) **ANCHO** = 80
(3) Leer **LETRERO**
(4) **LONG** = total de caracteres en **LETRERO**
(5) **NUM_ESPACIOS** = **ANCHO - LONG** / 2
(6) Imprimir **NUM_ESPACIOS** de blancos
(7) Imprimir **LETRERO**
(8) Fin

Comentarios:

- En la línea (2) fijamos el ancho en 80, según sean las necesidades puede fijarse en cualquier otro valor.
- La línea (4) indica el total de caracteres, manualmente se hace contando las letras, en los lenguajes de programación hay funciones especiales que hacen esto precisamente.

* Adaptado de Vasconcelos Santillán, Jorge. *Introducción a la Computación*. Publicaciones Cultural. México, 1997.

- **Programación**

```
Program CentrarLetrero;
Var
  i,
  Ancho,
  Long,
  NumEspacios : Integer;
  Letrero : String;
Begin
  Ancho := 80;
  Write ('Escriba algún mensaje ');
  Read  (Letrero);
  Long := Length(Letrero);
  NumEspacios := (Ancho - Long) Div 2;
  For  i := 1 To NumEspacios Do
       Write (' ');
  Write (Letrero);
End.
```

- **Mantenimiento**

Las necesidades cambia y el ancho de la pantalla será de 132. Tambien deberá aparecer el letrero entre asteriscos.

```
Program CentrarLetrero;
Var
  i,
  Ancho,
  Long,
  NumEspacios : Integer;
  Letrero : String;
Begin
  Ancho := 132;                                  Modificación
  Write ('Escriba algún mensaje ');
  Read  (Letrero);
  Long := Length(Letrero);
  NumEspacios := (Ancho - Long) Div 2;
  For  i := 1 To NumEspacios Do
       Write (' ');
  Write ('*',Letrero,'*');                       Modificación
End.
```

Que una idea (algoritmo, programa o lenguaje) sea antiguo, no implica que sea obsoleto; por el contrario, le puede aportar valiosos elementos que sólo requieran adaptarse a su trabajo propio.

cuadro 3-3

UN ALGORITMO DE SEIS PASOS PARA HACER PROGRAMAS
1. Especifique con precisión todo el problema.
2. Divida el problema en subproblemas. Diseñe un algoritmo que solucione cada subproblema. Hágalos tan sencillos que su corrección sea evidente.
3. Verifique que el algoritmo sea lógicamente correcto y que cumpla con las especificaciones.
4. Codifique usando el lenguaje apropiado, estructuras de datos conocidas y otras rutinas útiles. Aplique técnicas de programación estructurada y de documentación.
5. Compruebe el programa con algunos datos cuidadosamente seleccionados y elimine los errores que pueda haber. Nunca deje errores sin corregir, o en su caso documentar.
6. Efectue estos pasos con cada subproblema.

Los buenos algoritmos son
elegantes entidades algebráicas.

Richard E. Blahut

3.2 Diferencia entre algoritmo y programa

El algoritmo define la lógica con que fluyen las instrucciones de un programa.

Un **algoritmo** es una fórmula o una secuencia de pasos que resultan útiles para darle solución a un problema. Debe ser independiente de la sintaxis de un lenguaje de programación en particular puesto que es la concreción de un plan y es un medio para comunicar ideas entre personas (no obstante un algoritmo puede tomar en consideración a otros algoritmos).

Un **programa** es un grupo de instrucciones, escritas en un lenguaje especializado, cuya finalidad es indicarle a una máquina cómo efectuar un trabajo.

A cada línea de un algoritmo le corresponden una o más líneas de código.

En el cuadro 3-4 se compara un algoritmo con el correspondiente programa en un lenguaje de programación.

Si usted programa en Pascal, C, o algún lenguaje similar, reconocerá que entre el algoritmo y el programa no hay mucha diferencia. Si el algoritmo se redacta usando un lenguaje más formal (**pseudocódigo**) se parecerá aún más al programa:

```
inicio
    leer A, B, C
    X1 = (-B + √(B² - 4·A·C)) / (2·A)
    X2 = (-B - √(B² - 4·A·C)) / (2·A)
    escribir X1, X2
fin
```

y esta semejanza será aún mayor si la redacción se hace en inglés:

```
begin
    read A, B, C
    X1 = (-B + √(B² - 4·A·C)) / (2·A)
    X2 = (-B - √(B² - 4·A·C)) / (2·A)
    write X1, X2
end
```

cuadro 3-4

- **Algoritmo para obtener las raíces de la ecuación cuadrática**

Paso 1: Obtener los coeficientes de la ecuación cuadrática
Paso 2: Aplicar la fórmula $X_{1,2} = (-B \pm \sqrt{B^2 - 4 \cdot A \cdot C}) / (2 \cdot A)$
Paso 3: Entregar los resultados de la fórmula
Paso 4: Fin

- **Programa en Turbo Pascal para calcular las raíces de ecuación cuadrática**

```
Program Obtener_Raices;
Var
   A,B,C,
   X1,X2  : Real;
Begin
   Write ('Escriba el coeficiente cuadrático: ');
   Read  (A);
   Write ('Escriba el coeficiente lineal: ');
   Read  (B);
   Write ('Escriba el término independiente: ');
   Read  (C);

   X1 := (-B + SQRT (B*B - 4*A*C)) / (2*A);
   X2 := (-B - SQRT (B*B - 4*A*C)) / (2*A);
```

```
    WriteLn ('Las raíces de la ecuación son: ',X1:0:2,' y ',X2:0:2);
End.
```

cuadro 3-5

Suma de Matrices

Hay varias formas que describen el algoritmo de la suma de dos matrices:

1. "*Sumarn entrada a entrada*"
2. Esquemáticamente:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} z & y & x \\ y & v & u \\ t & s & r \end{bmatrix} = \begin{bmatrix} a+z & b+y & c+x \\ d+y & e+v & f+u \\ g+t & h+s & i+r \end{bmatrix}$$

3. De manera general

Sean
A, **B** y **C** matrices de n x m,
a_{ij} un elemento cualquiera de **A**
b_{ij} un elemento cualquiera de **B**
c_{ij} un elemento cualquiera de **C**

$$c_{ij} = a_{ij} + b_{ij}, \quad 1 \le i \le n \text{ y } 1 \le j \le m$$

Esta última forma permite indicar con mayor facilidad otros procedimietnos, como calcular la suma de las diagonales de una matriz

$$Diagonal = \sum_{i=1}^{n} a_{i\,i} \qquad Antidiagonal = \sum_{i=1}^{n} a_{i\ (n-i+1)}$$

Así como éste, hay una gran cantidad de casos en los que hacer el algoritmo resulta en la construcción del programa[3] (si se programa en un lenguaje de alto nivel). Sin embargo también existen otros casos en los que la diferencia entre algoritmo y programa es enorme (véase el cuadro 3-5).

Un programa puede constar de uno o más algoritmos.

[3]Esta facilidad ha motivado la construcción de programas directamente sobre el teclado.

Dos algoritmos son compatibles si producen las mismas salidas ante iguales entradas.

Obsérve que en el ejemplo del cuadro 3-6 resulta más fácil entender el algoritmo que el programa. Si usted tiene oportunidad de estudiar inteligencia artificial (campo donde suelen utilizarse búsquedas, esto es, ubicar la mejor posibilidad entre una gran gama situaciones, como en el ajedrez) observará que este algoritmo es fácilmente implementable en cualquier lenguaje. Si sólo tuviese el programa, antes de intentar hacerlo con otro lenguaje, tendría que aprender Lisp.

cuadro 3-6

- **Algoritmo para realizar una búsqueda de "primero en profundidad".**

```
1. Inicializar la lista OPEN con el nodo INICIO
2. Mientras haya nodos en OPEN hacer
2.1 Apuntar TMP al primer elemento de OPEN y retirarlo de la lista.
2.2 Si TMP es el nodo BUSCADO entonces terminar con éxito
   en otro caso expandir los sucesores de TMP y colocarlos
   al principio de OPEN.
3. Si OPEN está vacío entonces terminar con fracaso.
```

- **Programa en Lisp para efectuar una búsqueda de "primero en profundidad".**

```
(defun DFS (inicio buscado sucesores)
    (setq open (list inicio) )
    (loop ((null open))
        (setq tmp (car open))
        (setq open (cdr open))
        ((equal tmp buscado))
        (setq l (expand tmp sucesores))
        (setq open (append l open) )
    )
)
```

3.3 Bases para el diseño de algoritmos

El diseño de algoritmos es una actividad que consiste en especificar los pasos necesarios para alcanzar la solución de un problema. Debe considerar que algunos de esos pasos serán otros algoritmos (que ya estarán diseñados o deberán planearse después).

cuadro 3-7

Características de un algoritmo

1) **Alcanzar la solución (correcta) en un tiempo finito.**
2) **Constar de pasos claros, precisos y no ambiguos.**

3) Mostrar claramente cuáles son los datos iniciales y cuáles son los resultados.

Haga algoritmos tan sencillos y claros que su lógica sea evidentemente correcta.

Para establecer esa serie de pasos se requiere reflexionar sobre el problema. Si el problema es grande entonces conviene dividirlo en otros más pequeños que puedan entenderse con más detalle y atender cada uno de los subproblemas por separado, sin preocuparse por los demás.

Por cada problema se debe considerar lo siguiente:

No espere encontrar el algoritmo adecuado al primer intento.

1. Definir con precisión qué datos se utilizarán como entradas.
2. Definir con precisión qué datos se requerirán como salidas.
3. Si ya existen algoritmos adecuados, aprovecharlos prudentemente.
4. Determinar qué acciones se deben efectuar sobre las entradas hasta convertirlas en resultados y describir cada una con frases no ambiguas.

Si le resulta difícil diseñar algún algoritmo considere aplazarlo un poco. A veces posponer los problemas simplifica su solución; y siempre está dispuesto a empezar desde el principio, es posible que el segundo intento sea más breve y fácil.

Por lo general las acciones que pueden integrarse en un algoritmo son:[4]

a) **Pedir datos**
b) **Desplegar datos**
c) **Evaluar condiciones**
d) **Ejecutar operaciones matemáticas**

Estas acciones pueden describirse mediante dibujos (diagramas de flujo, figura 3-2), lenguaje natural, pseudocódigo (español o inglés estructurado), e incluso con las palabras propias de un lenguaje de programación.

[4]Para controlar el orden en que se efectúan estas acciones se requiere de estructuras de control o de saltos (*goto*).

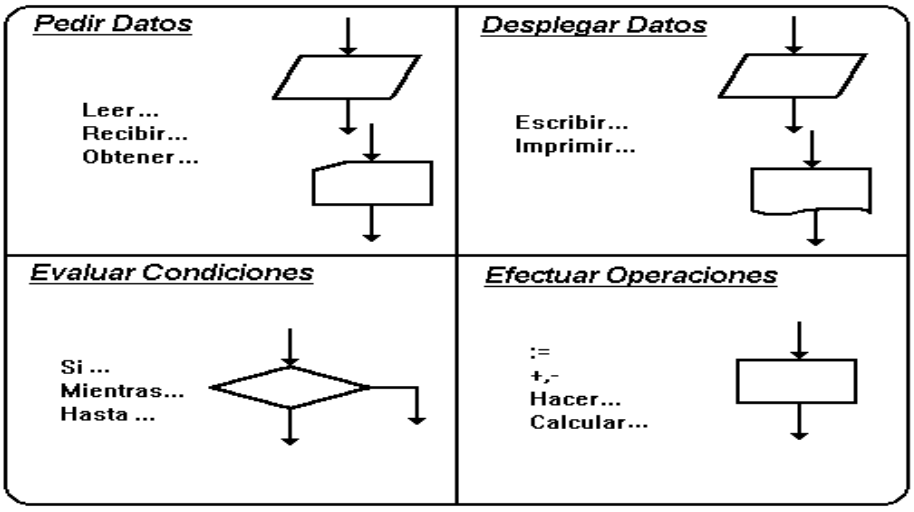


Figura 3-2. Figuras usadas en los Diagramas de Flujo

Observe que el ejemplo siguiente sólo usa los elementos antes mencionados.

Problema: El salario diario de un empleado se calcula en base a los siguientes datos: se pagan a N$ 15 cada una de las primeras 8 horas, y a $20 cada hora extra. Calcular el salario mensual, considerando que se trabajan sólo cinco días a la semana.

Algoritmo:

```
Inicio
MAX_HORAS = (8 horas)*(5 días)*(4 semanas);
Leer HORAS_TRABAJADAS
Si HORAS_TRABAJADAS > MAX_HORAS Entonces
    Inicio
    HORAS_EXTRAS = HORAS_TRABAJADAS - MAX_HORAS
    HORAS_TRABAJADAS = MAX_HORAS
    SUELDO = HORAS_TRABAJADAS * 15 + HORAS_EXTRAS * 20
    Fin
En otro caso
    SUELDO = HORAS_TRABAJADAS * 15
Imprimir SUELDO
Fin
```

No intente diseñar hasta el último detalle, muchas de las características finales del programa van surgiendo mientras se le desarrolla.

Para que los algoritmos se puedan convertir con facilidad en programas elegantes y de calidad conviene seguir las técnicas de programación estructuración y programación modular.

Existen situaciones en que puede ser difícil diseñar, entonces conviene lograr que primero funcione, adquirir experiencia sobre el comportamiento de las entradas y salidas, y la siguiente vez hacerlos bien desde el principio.

cuadro 3-8

Un lenguaje mínimio

Los procesos más complejos pueden ser construídos a partir de un lenguaje muy reducido que no contenga estructuras de programación. Este tipo de problemas son estudiados por la Computabilidad. Enseguida vemos un pequeño conjunto de instrucciones a partir del cual se construyen las demás estructuras de programación.

- **Instrucciones Básicas:**

1o. Inc *variable* → Incrementar en 1 el valor de la variable

2o. Dec *variable* → Decrementar en 1 el valor de la variable

3o. While *variable* ≠ 0 Do → Repetir un proceso si el
... valor de la variable es
End While diferente de cero

- **Inicializar una variable en cero** { *variable* := 0; }

```
Procedure Clear variable
    While variable ≠ 0 Do
        Dec Variable
    End While
```

- **Asignar un valor a una variable** { nom1 := nom2; }

```
Procedure asigna variable
    Clear Aux
    Clear Nom1
    While Nom2 ≠ 0 Do
        Dec Nom2
        Inc Aux
    End While

    While Aux ≠ 0 Do
        Inc Nom1
        Inc Nom2
        Dec Aux
    End While
```

- **Evaluar una condición** { If X = 0 Then S1 Else S2 }

```
Procedure If aux
    Inc Flag
    Clear Aux
    While Aux ≠ 0 Do   { else }
        S2
        Clear Aux
        Clear Flag
    End While
```

```
While Flag ≠ 0 Do
    S1
    Clear Flag
End While
```

- **Ciclo Repeat** { Repeat S Until X = 0 }

```
Procedure Repeat Until X
    Inc X
    While X ≠ 0 Do
        Dec X
        S
    End While
```

3.4 Diagramas de flujo

Los diagramas de flujo son representaciones del flujo que siguen las instrucciones de un programa.

Los diagramas de flujo fueron muy utilizados durante la década de los sesentas, pero resultaron inadecuados como herramienta de modelado de sistemas, debido a que facilitan la creación de código no estructurado y complejo. Frente a ellos, el pseudocódigo fue una alternativa realmente mejor.

Pero si bien estos diagramas han caído en desuso, aún pueden resultar útiles. Conviene recurrir a los diagramas de flujo en los siguientes casos:

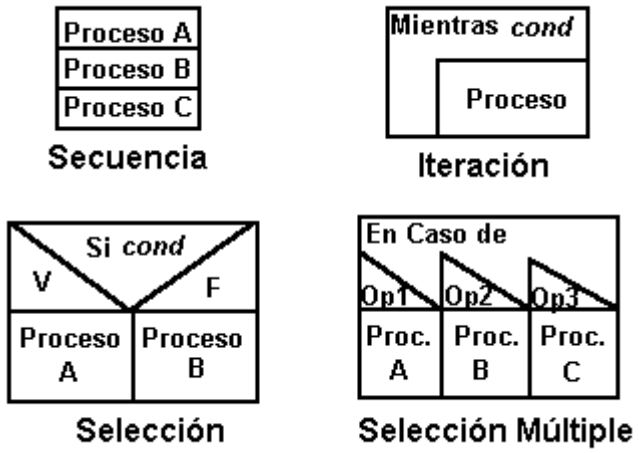
1) Para seguir la secuencia lógica de procesos complejos.
2) Para intercambiar entre estructuras de control
3) Para redireccionar el flujo, eliminando *goto's*, o sustituyéndolos por estructuras de control.
4) Para cambiar un programa de un lenguaje a otro.
5) Para esbozar fácilmente un procedimiento complejo.

cuadro 3-9

Los diagramas de Nassi-Schneiderman

Una alternativa a los diagramas de flujo son los diagramas de Nassi-Schneiderman, desarrollados hacia 1973.

Son diagramas estructurados y pueden representar algoritmos sin utilizar flechas para indicar el flujo.



3.5 Programación estructurada

La **programación estructurada** es una técnica de diseño de programas que requiere la aplicación de tres conceptos[5]:

1. Uso de las estructuras de control: Secuencia, Selección e Iteración (cuadro 3-9).
2. Eliminación del uso del *goto*.
3. Diseño de algoritmos y programas con un solo flujo de entrada y uno solo de salida.

Mediante esta técnica se puede leer y entender la lógica de los programas secuencialmente, de principio a fin; y puede aplicarse aún a nivel de pseudocódigo.

Si está acostumbrado a programar en C o Pascal le resultará natural aplicar las estructuras, no usar el *goto* y tener módulos con una sola entrada y una sola salida.

Estas técnicas pueden seguirse aun cuando el lenguaje no posea las implementaciones de las estructuras, como se muestra a continuación con un ejemplo en lenguaje ensamblador.

[5]En realidad los dos últimos son consecuencia de la aplicación del primero.

```
    mov cx,10                   ; emulacion de repeat-until
REPEAT:                         ; Repeat
    nop  ;call cuerpo del módulo ;
    dec cx                      ;
    jnz REPEAT                  ; Until CX = 0

    mov cx,10                   ; emulacion de while-do
WHILE: jcxz NEXT                ; While CX <> 0 Do
    nop  ;call cuerpo del módulo ;
    dec cx                      ;
    jmp WHILE                   ; End While

                                ; emulacion de For-Next
NEXT: mov cl, 10                ; limite inferior
        mov ch, 20              ; limite superior
FOR: cmp cl,ch                  ; For X := CL To CH Do
    je CONTINUE
    nop   ;call cuerpo del módulo
    inc cl
    jmp FOR                     ; Next X
CONTINUE: ;fin                  ; Salida del FOR-Next
```

Para finalizar esta sección, presentamos un algoritmo no estructurado (con su diagrama de flujo) y un fragmento de programa con dos puntos de salida.

1) Hacer proceso A

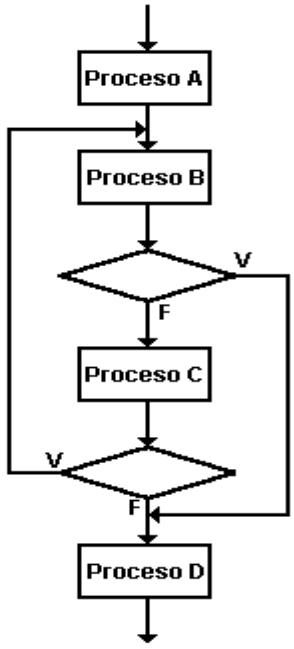
2) Hacer proceso B

3) Si *cond* Entonces ir a 6

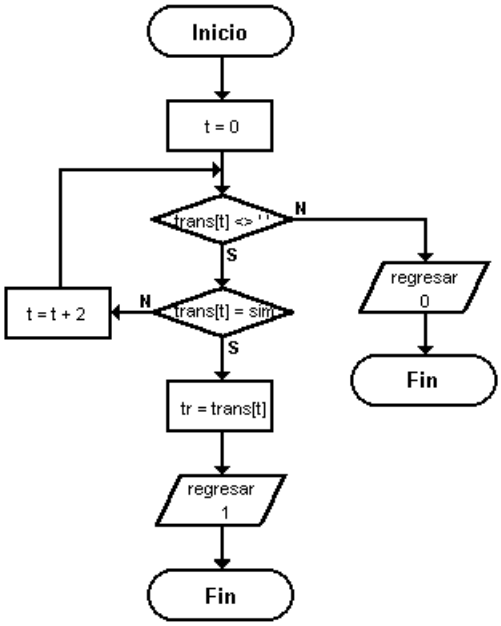
4) Hacer proceso C

5) Si *cond* Entonces ir a 2

6) Hacer proceso D



```
int traslada (simbolo,
tr)
char *simbolo, *tr;
{
    int t;
    t = 0;
    while (*trans[t] {
        if
(!strcmp(trans[t],simb
olo)) {

    strcpy(tr,trans[t+1]
;
            return 1;
            }
        }
    t += 2;
}
```



Aplicar programación estructurada produce código fácil de leer, fácil de mantener e incluso fácil de escribir.

Cuadro 3-10: Estructuras de Programación

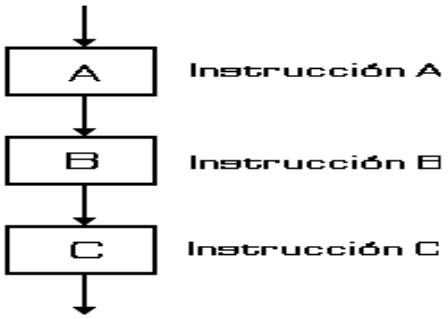
La programación estructurada se basa en el uso de las estructuras de control:

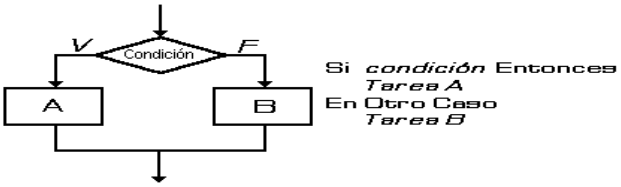
- **Secuencia**
- **Selección**
- **Iteración**

Estas estructuras controlan el flujo que siguen las instrucciones de un programa.

SECUENCIA

Estructura utilizada en operaciones que deberán ejecutarse una sola vez, siguiendo un orden determinado, de inicio a fin.





SELECCION

Estructura utilizada para elegir entre acciones diferentes.

ITERACION

Estructura utilizada para realizar varias veces una misma acción.

3.6 Programación modular

La programación **modular** es una técnica de construcción de programas que consiste en descomponer un gran proceso en pequeñas rutinas independientes, que realicen tareas concretas y sean fáciles de comprender y mantener.

cuadro 3-11

LAS DIVERSAS FORMAS DE LOS MODULOS

MACRO: Es una sola instrucción, que se incluye en el programa fuente, y se expande en varias instrucciones del mismo nivel del lenguaje.

SUBRUTINA (**procedimiento**): Es una sección de código que lleva una tarea autocontenida dentro de un programa. Se ejecuta mediante una llamada desde uno o más puntos dentro del programa, y después de completarse regresa a la instrucción siguiente de su llamada.

FUNCIONES: Se ejecutan como subrutinas. Efectúan operaciones matemáticas. Se les pasa un parámetro y su llamada regresa el resultado del cálculo.

CORRUTINAS: Módulos que efectúan su operación de modo paralelo o interactivo.

SUBRUTINA RECURSIVA: Procedimiento que tiene la capacidad para llamarse a sí mismo. No todos los lenguajes tienen esta facilidad.

Un programa escrito modularmente facilita el mantenimiento, pues es más fácil encontrar y aislar la parte a modificar y sólo es necesario alterar ese módulo.

Es fácil visualizar un programa como una construcción modular si se considera que todos los programas siguen los siguientes patrones de lógica:

1. Patrón básico:

a) Principio - Rutinas de inicialización, lectura de entradas, apertura de archivos, activación de modos gráficos.
b) Mitad - Procesamiento de datos.
c) Fin - Emisión de resultados, cierre de archivos, restauración de modos de video, liberación de memoria.

3. Ciclo determinístico

Los procesos se repiten un predeterminado número de veces.

4. Ciclo indeterminado

Los procesos se repiten hasta que una condición específica es detectada.

● Sugerencias para construcción de módulos

1. Antes de desarrollar un módulo se debe tener claro para qué servirá.

2. Cada subprograma deberá ejecutar una sola tarea, pero hacerla bien.

3. Ningún módulo deberá afectar a algún otro procedimiento. Use variables locales tanto como le sea posible ya que su margen de acción está limitado naturalmente al módulo que las define.

4. Cada módulo deberá comportarse como una caja negra: recibir entradas, efectuar un proceso desconocido y entregar resultados.

5. La comunicación entre módulos debe efectuarse mediante parámetros.

6. Si al tratar de describir la tarea realizada por un módulo se extiende o se exageran los detalles, deberá considerarse la división del módulo.

7. Cualquier tarea que se efectúa más de una vez en un programa, debe ser separada en un módulo. Todas las secciones que actúen como interfaces de entrada y salida de datos hacia el usuario también deberán ser módulos especiales.

8. Escribir los módulos de modo general. Colocarlos en librerías para que puedan ser incorporados en programas diferentes.

9. Evitar que la implementación de un módulo complicado retarde la construcción del resto del programa.

10. Construir módulos con pocos ciclos y pocas opciones para facilitar su prueba y depuración. El tamaño máximo conveniente para un módulo es de unas 23 líneas (una pantalla).

● El acoplamiento y la cohesión

Los términos ***Acoplamiento y cohesión*** fueron acuñados por Stevens, Constantine y Myers para describir la calidad de los módulos de un programa. Mientras mejores sean los módulos, más fácil será la depuración.

Acoplamiento significa dependencia de un módulo respecto de otro. Visto de otra forma, indica qué tan intersectados están los módulos (figura 3-3a). Hay varios niveles de acoplamiento, el más alto es acoplamiento de contenido y el más bajo es el acoplamiento de datos.

1. **Acoplamiento de contenido**: las instrucciones de un módulo o sus datos locales pueden ser alterados por otro módulo. Por ejemplo, un virus altera el contenido de otros módulos (archivos o programas); o también ciertos programas en BASIC (intérprete) o en lenguaje máquina sí pueden alterar su propio código porque cada instrucción ocupa una zona específica de memoria que puede ser alterada al ejecutarse la instrucción que le antecede (esto mismo es difícil de lograr si los módulos están escritos en C o en Pascal ya que ambos lenguajes requieren compilador).

2. **Acoplamiento de zonas**: varios módulos utilizan áreas de datos comunes (variables globales). Esto es prácticamente inevitable cuando se usan archivos puesto que no puede haber archivos locales para cada módulo.

El acoplamiento de datos es el más recomendable.

3. **Acoplamiento de control**: el flujo seguido por las instrucciones de un módulo puede ser controlado desde otro módulo, mediante parámetros de control. Este tipo de aclopamiento es característico de los módulos manejadores de menús.

4. **Acoplamiento de datos**: los datos requeridos por un módulo se obtienen mediante parámetros.

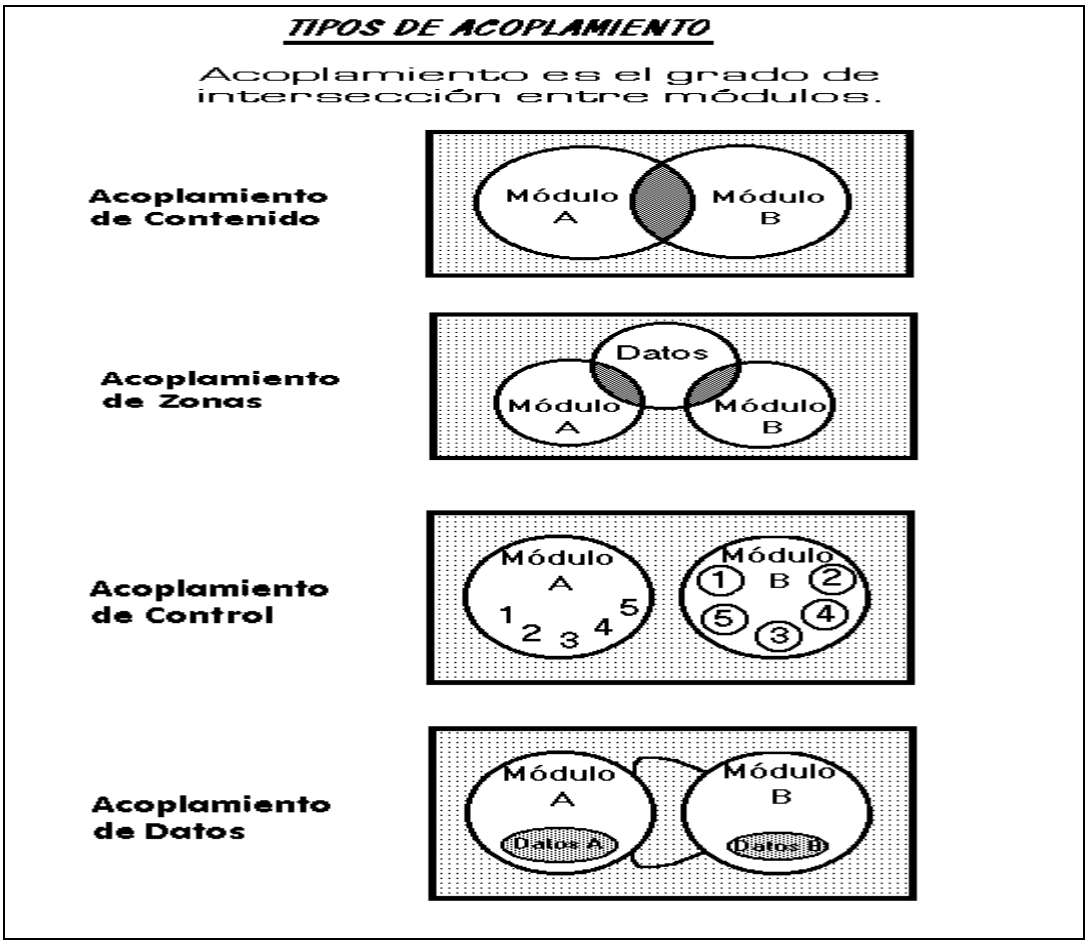


Figura 3-3 (a)

Cohesión es el grado de interrelación entre las instrucciones de un mismo módulo. Indica qué tanto pertenecen las instrucciones a un módulo, para alcanzar un mismo objetivo (figura 3-3b). La cohesión nos indica cuántas tareas hace a la vez un mismo módulo.Hay varios niveles de cohesión, el más bajo es la cohesión coincidental y el más alto es cohesión funcional.

1. **Cohesión coincidental**: un módulo tiene instrucciones muy poco relacionadas; es decir, hace varias cosas a la vez. Por ejemplo un módulo de inicialización (abre archivos, ajusta modo de video, etcétera) o una mala modularización.

2. **Cohesión lógica o temporal**: un mismo módulo realiza varias funciones. Pueden requerir de parámetros de control, o ser secuencial. Este tipo de cohesión se presenta en un módulo manejador de menú.

3. **Cohesión en la comunicación**: un mismo módulo realiza actividades diferentes pero utilizando los mismos datos.

4. **Cohesión secuencial**: la salida de una instrucción es la entrada de la siguiente.

5. **Cohesión funcional**: todas las instrucciones se enfocan a desarrollar una sola tarea.

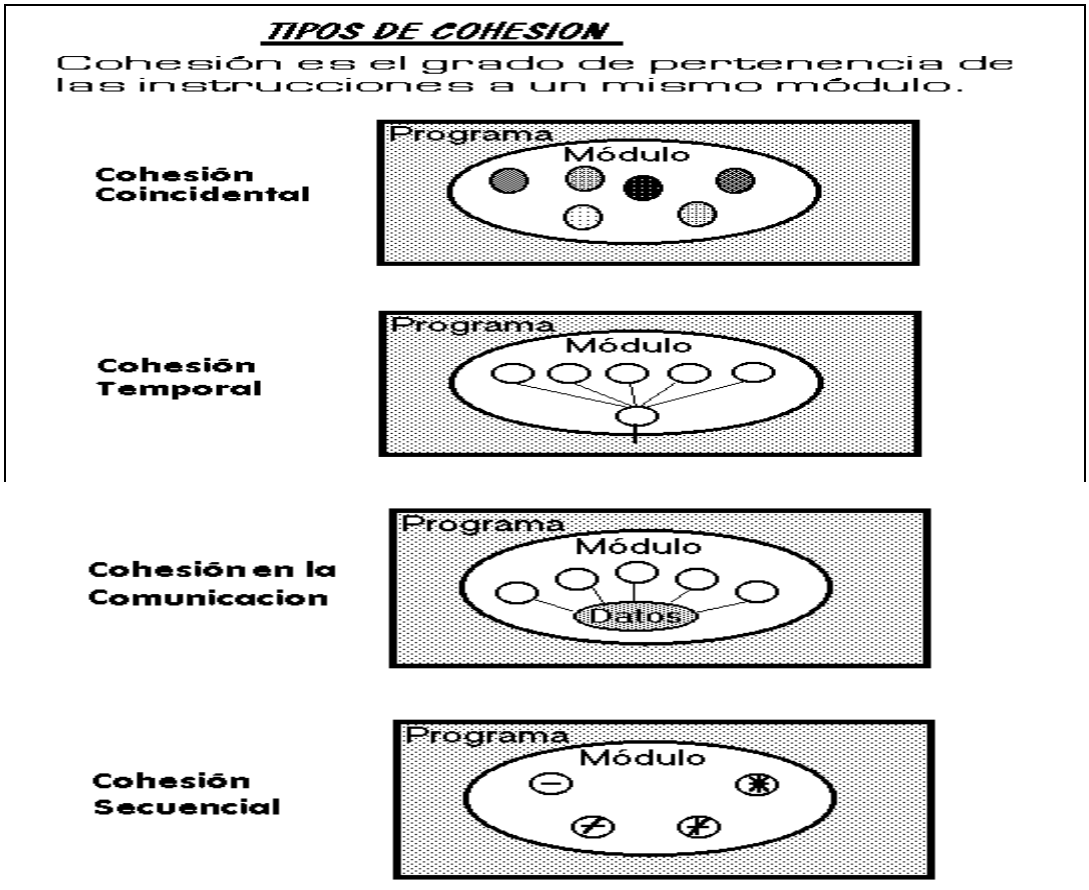


Figura 3-3 (b)

Un buen programa tiene módulos con una alta cohesión y un bajo acoplamiento, así todos sus módulos colaboran con el propósito global del programa y para hacer modificaciones basta afectar en un módulo específico sin preocuparse del resto.

El cuadro 3-11 presenta alguno ejemplos de código con diferentes grados de cohesión y acoplamiento.

cuadro 3-12

♦ Acoplamiento de contenido

☞ *En este programa hay acoplamiento de contenido entre los módulos **ajustar** e **imprimir**.*

```
title SALUDO
.model small
.stack 10h
.data
    CR = 0Dh
    LF  = 0Ah
    MESSAGE db CR,LF,'Hola Mundo','$'

.code
inicio proc                         imprime cinco veces el mensaje
      mov cx, 5
ciclo: call imprimir
      loop ciclo

      call ajustar                  esta rutina altera el código del módulo imprimir
      mov cx, 5
ciclo2:call imprimir
      loop ciclo2

fin:   mov ax, 4C00h
      int 21h
      ret
inicio endp

imprimir proc                             despliega en pantalla el mensaje
      push cx
      push ds
      mov ax, @data
uno:   mov ds, ax
dos:   mov dx, offset MESSAGE
tres:  mov ah, 9
      int 21h
      pop ds
      pop cx
      ret
imprimir endp

ajustar proc                              altera las instrucciones marcadas con las banderas
      mov bx, offset uno
      mov cs:[bx], 0D18Bh
```

```
      mov bx, offset dos
      mov cs:[bx], 0C280h
      mov cs:[bx+1], 30C2h
      mov bx, offset tres
      mov cs:[bx], 02B4h
      ret
ajustar endp
End Inicio
```

Resultado de la ejecución:

♦ Acoplamiento de contenido

☞ *dado que este módulo es recursivo, hay acoplamiento consigo mismo, debido a que se está haciendo un paso de parámetros por referencia.*

```
Function Factorial (var i : Word):Word;
    Begin
        If i = 0 Then
            Factorial := 1
         Else
            Begin
            i := i - 1;
            Factorial := (i+1) * Factorial(i);
            End;
    End;
```

♦ Acoplamiento de zonas

☞ *Todos los módulos hacen referencia al mismo arreglo. Si uno de ellos modifica el contenido, se modifica para todos..*

```
Const
    MAX_STACK = 20;

Var
    Stack : Record;
    arr  : Array[1..MAX_STACK] Of String;
     tope : Byte;
    End;

Procedure Push (st : String);
    Begin
        With stack Do
        Begin
        Inc(tope);
        If tope <= MAX_STACK Then
            arr[tope] := st;
         End;
```

```
    End;



Function Pop : String;
    Begin
        With Stack Do
         Begin
        If tope > 0 Then
            Begin
            Pop := arr[tope];
             arr[tope] := '';
            Dec(tope);
              End
        End;
    End;


Function Peek : String;
    Begin
        With Stack Do
            If tope > 0 Then
                Peek := arr[tope]
             Else
                Peek := '';
    End;
```

♦ Acoplamiento de control y cohesión tempora

☞ *La sección del código que se ejecutará, depende del valor de* ***Opcion****.*

```
Procedure CalcularAreas (Opcion : Char);
    Var
        b,h,
        l,r,
        area : Real;
    Begin
    Case Opcion Of
      '1' : Begin
          Write  ('Escriba la Base y la altura: ');
           Readln (b,h);
           area  := b * h;
           Write ('El área es: ',area);
          End;
      '2' : Begin
          Write  ('Escriba el lado: ');
           Readln (l);
           area  := l * l;
           Write ('El área es: ',area);
          End;
      '3' : Begin
          Write  ('Escriba el radio: ');
           Readln (r);
           area  := PI * r * r;
           Write ('El área es: ',area);
          End;
      End;
   End;
```

♦ Acoplamiento de control

☞ *En este fragmento de código se observa claramente un acoplamiento de control entre los módulos, debido a que para que **Accion2** pueda ejecutar sus instrucciones, depende de un dato que está fuera de él, **Resp**. El valor de **Resp** se obtiene de otro módulo ajeno, **Accion**1.*

```
Var
    Resp : Char;

Procedure  Accion1;
    Begin

        { Instrucciones...}

        WriteLn ('Desea Continual');
        ReadLn (Resp);
    End;

Procedure Acción2;
    Begin
        If Resp = 'S' Then
            Begin

            { Instrucciones... }

            End;
    End;

Begin
    Accion1;
    accion2;
End.
```

☞ *Para liberar este acoplamiento de control, podemos reescribir parte del código.*

```
Var
    Resp : Char;

Procedure  Accion1;
    Begin
        { Instrucciones...}
    End;

Procedure Acción2;
    Begin
        { Instrucciones... }
    End;

Begin
    Accion1;
    WriteLn ('Desea Continual');
    ReadLn (Resp);
    If Resp = 'S' Then Accion2;
End.
```

☞ *Este reacomodo dejó a los módulos **Accion1** y **Accion2** desacoplados, ya que se delego el control del flujo a un módulo superior.*

♦ Acoplamiento de datos

☞ *Cada módulo recibe datos externos mediante parámetros.*

```
Const
    MAX_STACK = 20;
Type
    t_Stack = Record
        arr  : Array[1..MAX_STACK] Of String;
        tope : Byte;
    End;

Procedure Push (var stack : t_Stack; st : String);
    Begin
        With stack Do
        Begin
        Inc(tope);
        If tope <= MAX_STACK Then
            arr[tope] := st;
        End;
    End;

Function Pop(var Stack : t_Stack) : String;
    Begin
        With Stack Do
        Begin
        If tope > 0 Then
            Begin
            Pop := arr[tope];
            arr[tope] := '';
            Dec(tope);
            End
        End;
    End;

Function Peek (Stack : t_Stack) : String;
    Begin
        With Stack Do
        If tope > 0 Then
            Peek := arr[tope]
        Else
            Peek := '';
    End;
```

♦ Cohesión coincidental

☞ *Este es el programa original que fue hecho de un solo bloque.*

```
PROGRAM ASTERISCO;{VERSION ORIGINAL}
    USES CRT;
    VAR
```

```
      C,C2,C3,CONT,CONT1:INTEGER;
     BEGIN
        CLRSCR;
        CONT1:=1;
        CONT:=7;
        REPEAT
        GOTOXY (CONT,CONT1);
        WRITELN ('*');
        CONT:=CONT-1;
        CONT1:=CONT1+1;
        UNTIL CONT =1;
        BEGIN
        C2:=1;
        C:=7;
        REPEAT
        GOTOXY (C,C2);
        WRITELN ('*');
        C:=C+1;
        C2:=C2+1;
        UNTIL C2= 7;
        BEGIN
        C3:=7;
        C:=1;
        REPEAT
        GOTOXY (C,C3);
        WRITELN ('*');
        C:=C+1;
        C3:=7;
        UNTIL C=14;
        END;
        END;
        READLN;
     END.
```

☞ *Al modularizar el programa anterior sólo se le fragmenta. Observe el programa principal, donde se aprecia con claridad cómo la relación entre sus instrucciones es sólo coincidencia.*

```
PROGRAM ASTERISCO;{VERSION MODULAR}
    USES CRT;
     VAR
       C,C2,C3,CONT,CONT1:INTEGER;

    Procedure Primero;
      BEGIN
        CLRSCR;
        CONT1:=1;
        CONT:=7;
     End;
    Procedure Segundo;
    Begin
        REPEAT
        GOTOXY (CONT,CONT1);
        WRITELN ('*');
        CONT:=CONT-1;
        CONT1:=CONT1+1;
        UNTIL CONT =1;
      End;
    Procedure Tercero;
```

```
        BEGIN
        C2:=1;
        C:=7;
        REPEAT
        GOTOXY (C,C2);
        WRITELN ('*');
        C:=C+1;
        C2:=C2+1;
        UNTIL C2= 7;
      End;
    Procedure Cuarto;
        BEGIN
        C3:=7;
        C:=1;
        REPEAT
        GOTOXY (C,C3);
        WRITELN ('*');
        C:=C+1;
        C3:=7;
        UNTIL C=14;
        END;

    BEGIN
      Primero;
      Segundo;
      Tercero;
      Cuarto;
    END.
```

♦ Cohesión en la comunicación

☞ *La misma variable **Empleado** se usa en cada módulo.*

```
Procedure Actualizar (Empleado : TipoPersona);
    Begin
        MostrarInformacion(Empleado);
        ActualizarArchivo   (Empleado);
        ImprimiCheque       (Empleado);
    End;
```

♦ Cohesión Secuencial

☞ *La salida de cada etapa es utilizada por la siguiente.*

```
Procedure RaícesCuadráticas;
    Var
        A,B,C,•
        X1,X2,
        Disc   : Real
        flag   : Boolean;
    Begin
        A := ObtenerCoeficiente;
        B := ObtenerCoeficiente;
        C := ObtenerCoeficiente;
        Disc := CalcularDiscriminante (A,B,C);
        If Disc >= 0 Then
            Begin
```

```
        X1 := CalcularRaiz1 (A,B,Disc);
        X2 := CalcularRaiz2 (A,B,Disc);
        Imprimir X1,X2;
        End;
End;
```

3.7 Diagramas de estructura

Los **diagramas de estructura** son representaciones gráficas de la jerarquía existente entre los módulos de un programa, y las estructuras de programación utilizadas para controlar la operación de esos módulos.

Cada módulo se representa como un rectángulo (figura 3-4). Los módulos a su vez se pueden componer de otros, o ser módulos primitivos.

Un módulo primitivo se encarga de desarrollar el procesamiento de los datos, mientras que un módulo no primitivo se encarga de *administrar* a otros módulos.

Para elaborar un diagrama de estructura considere lo siguiente:

1. Descomponer el problema en una jerarquía de módulos (cada módulo debe desarrollar una sola tarea).

2. Indique las estructuras de control del siguiente modo:

- La ejecución de varios módulos en secuencia se indica de derecha a izquierda.
- La ejecución condicionada de varios módulos se indica mediante el rombo, señalándose de derecha a izquierda el orden en que se evalúan las condiciones.
- La iteración se indica mediante uan flecha que indica los módulos a repetir.

El diagrama de estructura se puede traducir a pseudocódigo (y viceversa), debido a que existe una correspondencia directa de estructuras de control y los nombres de los módulos.

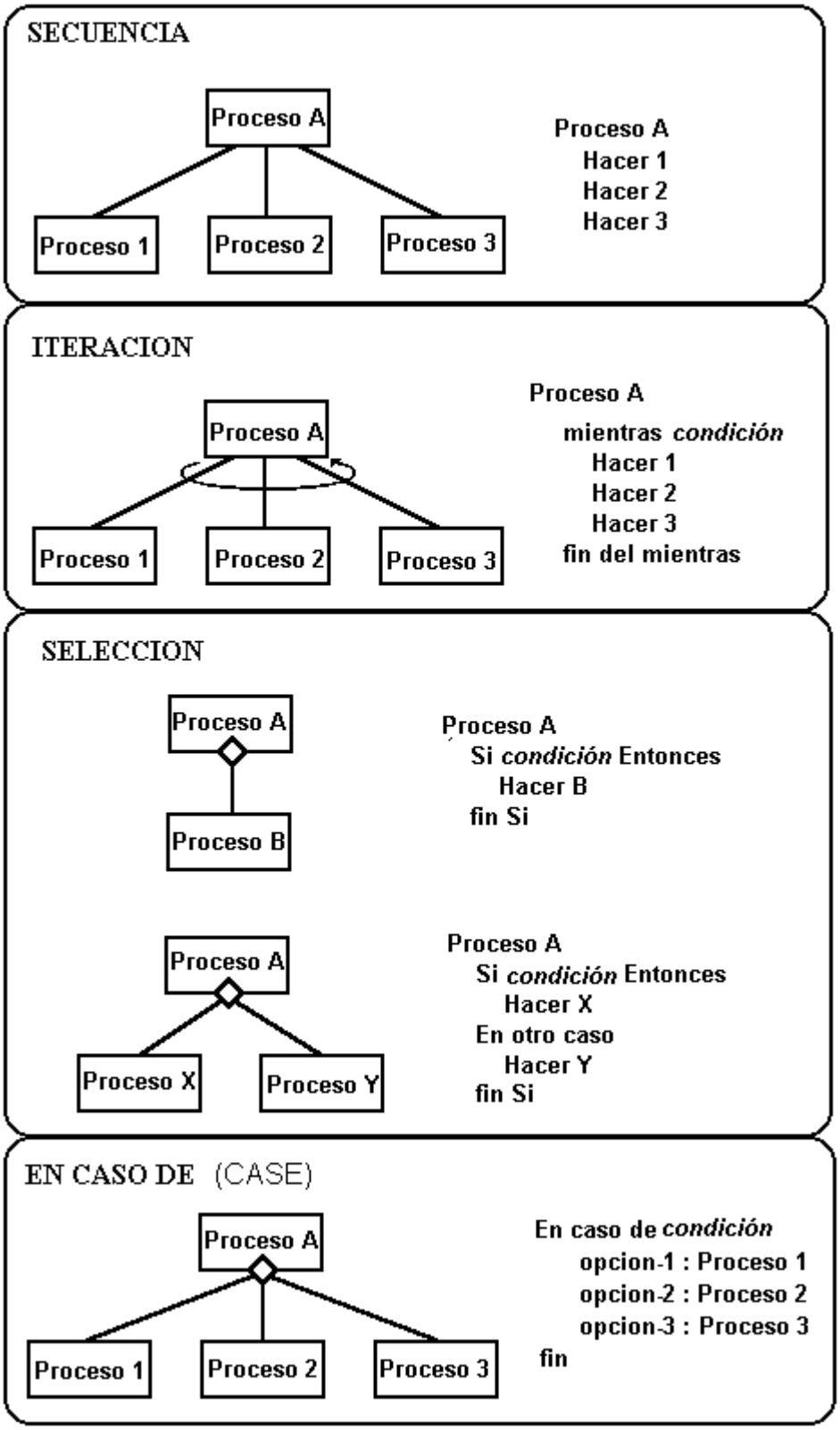
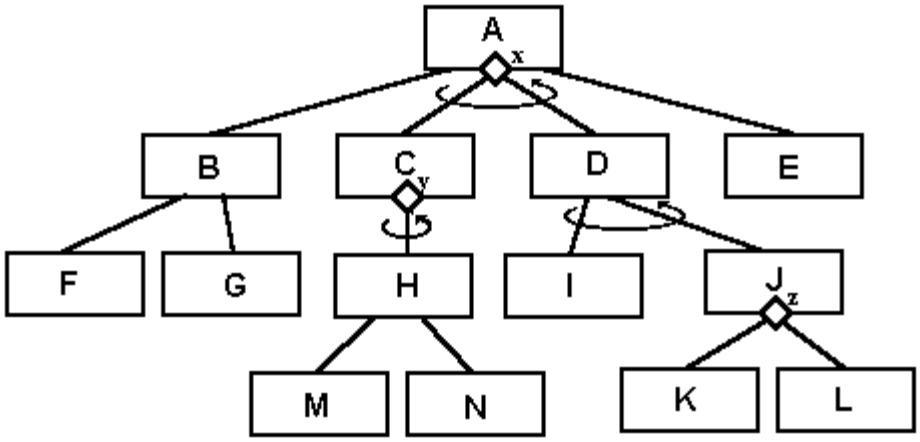


Figura 3-4: Figuras para Diagramas de Estructura

Por ejemplo, para el diagrama de estructura de la figura, corresponde el siguiente pseudocódigo.

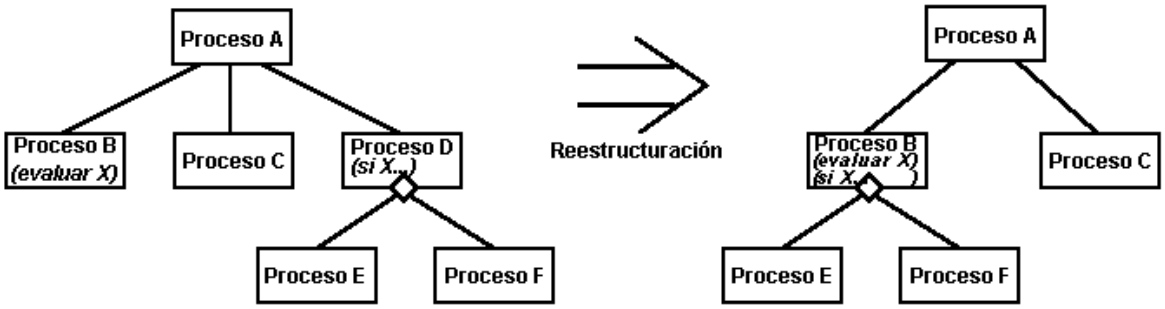



```
Programa A
    Hacer B
    Mientras condición
        Si x entonces
            Hacer C
        En otro caso
            Hacer D
    Fin del mientras
    Hacer E
Fin del programa

Procedimiento B
    Hacer F
    Hacer G
Fin del procedimiento

Procedimiento C
    Si y entonces
        Mientras condición
            Hacer H
        Fin del mientras
Fin del procedimiento

Procedimiento D
    Mientras condición
        Hacer I
        Hacer J
    Fin del mientras
Fin del procedimiento

Procedimiento H
    Hacer M
    Hacer N
Fin del procedimiento

Procedimiento J
    Si z entonces
        Hacer K
    En otro caso
        Hacer L
Fin del procedimiento
```

Los procedimientos E, F, G, I, M, N, K y L no aparecen puesto que son módulos terminales. Estos módulos son los que efectúan el "*trabajo pesado*", es decir, es la parte del programa que efectúa el procesamiento de los datos en sí. Los procedimientos B, C, D, H, J y el programa A, sí se describen, pues se encargan de *administrar* a los módulos de procesamiento.

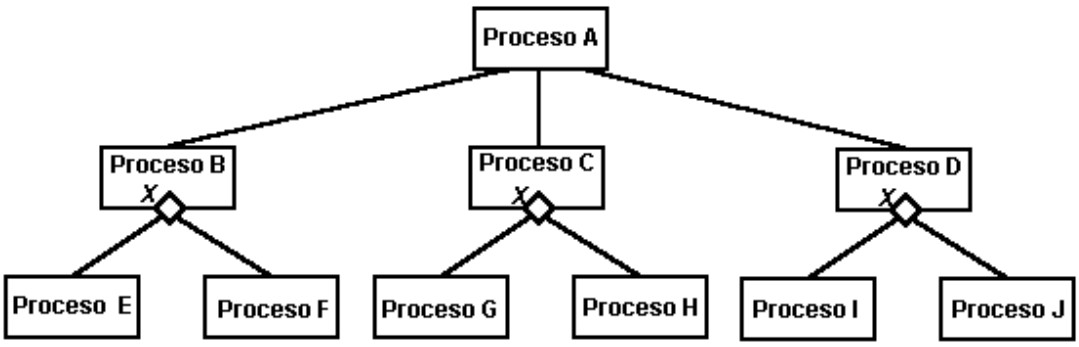
Estos diagramas permiten analizar las decisiones y el control de los módulos. Esta facilidad permite redibujar la estructura para mejorar la eficiencia del programa (en caso de ser necesario).

Obsérvese en el diagrama de la izquierda que los módulos B y D están acoplados, ya que uno es el que calcula el valor de ***X*** y otro es el que lo usa. A la derecha se ve el mismo programa una vez reestructurado.

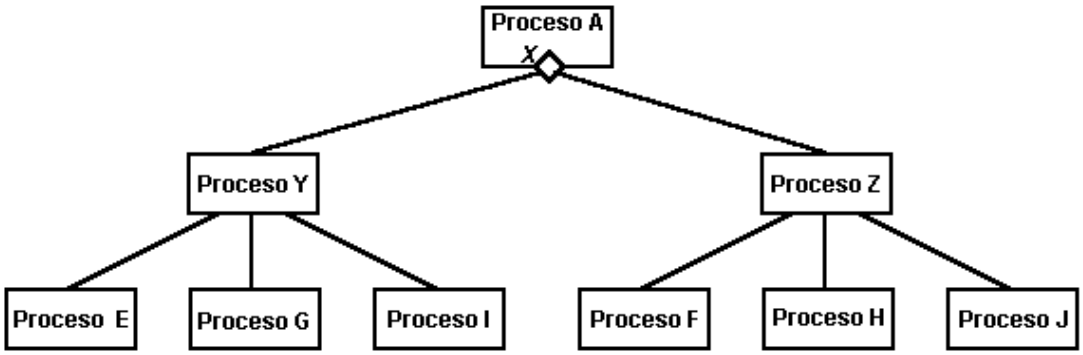


Se debe estar preparado para reorganizar la jerarquía y secuencia; y para reagrupar o dividir funciones.

Como último ejemplo vea el siguiente diagrama. La variable ***X*** es evaluada tres veces, y de cualquier modo, si el resultado de la evaluación es **verdadero** se efectuarán los módulos E, G, I y de resultar **falso**, se efecturán F, H, J.



Esta es la versión del diagrama una vez reestructurado, de tal modo que ***X*** sólo se evalúa una sola vez.



Aun si tiene que programar en un lenguaje que no facilite la construcción de módulos (como el BASIC o el FORTRAN originales) puede aplicar las técnicas de construcción de módulos delimitando con claridad las secciones del código; sólo tenga mucho cuidado al manejar las variables (ya que carecerá de variables locales), pues fácilmente puede caer en acoplamiento de zonas.

Capítulo 4

❖ *Construcción de programas*

cuadro 4-1

Niveles de construcción de programas

Programas de bajo nivel: Son programas que explotan las características propias de la computadora (normalmente son *herramientas* dependientes del equipo). Por ejemplo, las rutinas gráficas, el manejo de puertos, las interfaces con el usuario, etcétera. En este tipo de programas no puede seguirse estrictamente las fases del diseño y generalmente son rutinas que deben ser optimizadas, aun sacrificando su claridad (suelen ser rutinas que sólo estudiarán especialistas).

Programas de alto nivel: Son programas que entregan resultados independientemente de la máquina. Para construírlos hay que enlazar herramientas ya disponibles. En este tipo de programas conviene seguir estrictamente las fases del diseño y generalmente no deben ser optimizadas, ya que se requiere claridad para que las entienda un no especialista.

Para facilitar el proceso de mantenimiento, hay que evitar mezclar niveles al construir las rutinas.

4.1 Estilo de Programación: ejemplos y contraejemplos

Estilo indica las características en cuanto a la forma en que se usan las instrucciones para elaborar un programa.

Esta sección está basada en un documento publicado en 1974 por Kernigham y Plauger[6]. Contiene conceptos aún muy válidos que conviene recordar y aplicar. (No se hizo sólo una traducción, se actualizó y adaptó a la realidad actual).

Estilo es alguna disciplina más allá de un simple conjunto re restricciones sobre qué tipo de instrucciones usar.

Por buen estilo de programación hay que considerar la expresividad, la estructura, la robustez y la documentación de un programa.

- **Expresividad**

Codificar con trucos

Un truco es una solución que no es evidente algorítmicamente; es decir, resulta difícil saber qué hace o porqué funciona.

[6]Kernigham, Brian W. and Plauger, P.J. "Programming Style: Examples and Counterexamples." *Computing Surveys*, Vol. 6, No. 4. December 1974.

Antes de continuar con la lectura, observe la siguiente rutina y trate de decir qué hace.

```
#define N 4

main ()
{
int i,j;
int x[N][N];

for (i=1;i<=N;i++)
   for (j=1;j<=N;j++)
      x[i-1][j-1] = (i/j) * (j/i);
}
```

Este programa se basa en el truncamiento de la división de enteros en C: si **i** es menor que **j**, **i/j** es cero; de igual manera, si **j** es menor que **i**, **j/i** es cero. Sólo cuando **i** es igual a **j**, se obtiene un cociente diferente de cero. De este modo el código coloca unos en la diagonal de **x** y ceros en cualquier otra parte.

Este programa no es bueno porque es virtualmente ilegible, demasiado ingenioso (tramposo) para su importancia. Aun cuando el truco mejore la velocidad del programa es más importante que el código sea claro, así la gente puede depurarlo, mantenerlo y modificarlo.

Un principio de estilo en el idioma inglés dice: "*Di lo que quieras decir, tan simple y directamente como puedas*", se aplica también a la programación.

Esta rutina efectua el mismo trabajo y es más clara:

```
#define N 4

main ()
{
int i,j;
int x[N][N];

for (i=1; i<=N; i++)
   for (j=1; j<=N; j++)
      if (i == j)
         x[i-1][j-1] = 1.0;
      else
         x[i-1][j-1] = 0.0;
}
```

Si esta versión resulta ineficiente, puede modificarse para incrementar la velocidad, pero resulta menos clara:

```
#define N 4

main ()
{
int i,j;
int x[N][N];
```

```
for (i=1; i<=N; i++){
  for (j=1; j<=N; j++)
      x[i-1][j-1] = 0.0;

  x[i-1][i-1] = 1.0;
  }
}
```

Codificar demasiado complicado

Un ejemplo de lo que ocurre cuando se programa sobre la marcha:

```
10 IF X < Y THEN GOTO 50
20 IF Y < Z THEN GOTO 50
30 SMALL = Z
40 GOTO 70
50 IF X < 2 THEN GOTO 100
60 SMALL = Z
70 GOTO 110
80 SMALL = Y
90 GOTO 110
100 SMALL = X
110 ...
```

En sólo diez líneas hay seis *goto*. Antes de continuar leyendo, trate de decir qué hace este programa?.

Esta secuencia encuentra el menor de los tres números **X**, **Y**, **Z**; y lo guarda en **SMALL**.

Este mismo programa pudo ser construído así:

```
10 SMALL = X
20 IF Y < SMALL THEN SMALL = Y
30 IF Z < SMALL THEN SMALL = Z
```

Sólo tres instrucciones, sin *goto* y claramente correcto. Su generalización para muchas variables es obvia.

Reescribiendo el código

Observe el siguiente programa

```
PROGRAM TRAPECIO;

CONST
   MSG1 : STRING[20] = 'AREA BAJO LA CURVA';
   MSG2 : STRING[23] = 'POR LA REGLA DEL TRAPECIO';
   MSG3 : STRING[16] = 'PARA DELTA X = 1/';

VAR
   I : INTEGER;
   J : INTEGER;
```

```
   K : INTEGER;
   L : REAL;
   M : REAL;
   N : INTEGER;
   AREA1 : REAL;
   AREA : REAL;
   LMTS : REAL;

PROCEDURE OUT;
BEGIN
      AREA := AREA + LMTS;
     WRITELN (MSG3,K,AREA);
   AREA := 0;
END;

BEGIN
     WRITELN (MSG1);
     WRITELN (MSG2);
     WRITELN ;
   AREA := 0;
         FOR K := 4 TO 10 DO
           BEGIN
           M := 1 / K;
           N := K - 1;
      LMTS := 0.5 * M;
           I := 1;
         FOR J := 1 TO 10 DO
           BEGIN
           L := SQR(1 / K);
     AREA1 := 0.5 * M * (2 * L);
     AREA := AREA + AREA1;
       IF I = N THEN OUT
        ELSE I := I + 1;
      END;
     END;
END.
```

En este programa se hace derroche de palabras. Los mensajes de salida son declarados y asignados sin necesidad. Hay demasiadas variables temporales y declaraciones de las mismas. Ademas la estructura aparenta estar rota.

Para modificarlo: hacer una cosa a la vez. Poner los mensajes en sentencias **WRITE** donde pertenecen. Eliminar las variables intermedias innecesarias. Simplificar las inicializaciones y borrar el procedimiento innecesario. Combinar las declaraciones restantes. El código se encoje ante sus ojos, revelando un algoritmo simple.

Alinear el código según el bloque y estructura a que pertenecen.

```
PROGRAM TRAPECIO;

VAR
   J,K  : INTEGER;
   AREA : REAL;

BEGIN
   WRITELN ('EL AREA BAJO LA CURVA');
   WRITELN ('POR LA REGLA DEL TRAPECIO');
   WRITELN ;
```

```
    FOR K := 4 TO 10 DO
       BEGIN
       AREA := 0.5 / K;
       FOR J :=  1 TO K-1 DO
          AREA = AREA + (SQR(J/K)/K);

       WRITELN ('PARA DELTA X = 1/',K,AREA);
       END;
END.
```

- **Estructuras de control de flujo**

Los programas pueden ser escritos usando solamente

1) Selección: **IF-THEN-ELSE**, donde la parte **ELSE** es opcional.

2) Iteración: **WHILE**, **FOR**, con diferentes formas de iniciar y terminar el ciclo.

3) Secuencia: llamadas a procedimiento y bloques **BEGIN-END**.

Aun cuando esto es suficiente, es conveniente añadir:

4) Do Case que equivale a múltiples **IF**.

5) **BREAK** e **ITERATE**, que salen de un ciclo o saltan una parte del mismo.

Hay una tendencia a creer que sólo por usarlas (y sólo con ellas) se pueden evitar todos los problemas. Esto es falso, no son la panacea. Un buen estilo, cuidado e inteligencia, son necesarios. A continuación mostramos algunos vicios en que se puede incurrir, aun con las estructuras de programación.

Entonces nulo

Se evalua una condición, que de resultar verdadera, no ejecuta acción alguna. Por ejemplo, la siguiente rutina debería ordenar un arreglo de ocho números en orden ascendente según su valor absoluto:

```
For i := 1 to 8 Do
    If Abs(A[i]) < ABS (A[i+1]) Then ;
    Else
      Begin
      Tmp := A[i];
      A[i] := A[i+1];
      A[i+1] := A[i];
      End;
```

El corazon de esta secuencia es una instrucción *"hacer nada"*: si la condición es veradera, entonces *hace nada*, de otra manera hace algo.

Esta rutina no puede ordenar correctamente porque:

1) Sólo se hace una pasada sobre el arreglo, y un ordenamiento simple requiere unas **N** pasadas.

2) Se hace una referencia fuera de los límites del arreglo cuando **A[I+1]** es accedido en la última iteración, con **I = 8**;

Hay varias formas de hacer correctamente un ordenamiento simple. Podemos hacer **N-1** pasadas sobre el arreglo, o encender una bandera cada vez que sea necesario intercambiar dos elementos, así sabríamos que se requiere una pasada adicional.

```
For m .= 1 To N-1 Do
  For j := 1 to N-1 Do
    If Abs(A[j]) < ABS (A[j+1]) Then
      Begin
      Tmp := A[i];
      A[i] := A[i+1];
      A[i+1] := A[i];
      End;
```

Entonces casi nulo

La construcción **IF-THEN-ELSE** está correctamente, pero la condición que nunca se cumple.

```
Var
   i : Integer;
Begin
   Read (i);
   If i > 40000 Then  WriteLn ('Número muy grande');
   ....
End
```

En el programa anterior se observa que el letrero nunca se imprimirá dado que el máximo entero es 36767 (en una implementación de dos bytes por entero).

En otro caso salir

En caso de fallar la evaluación de la condición del **If**, entonces se *"brinca"* (sale de) esa sección del código.

Por ejemplo, este código corresponde a una rutina recursiva de búsqueda en un árbol binario.

```
Procedure Inorder (nodo : apuntador);
   Begin
      If nodo = NIL Then
         Exit;
      Else
         Begin
         Inorder (nodo^.izq);
         Write   (nodo^.info);
         Inorder (nodo^.der);
         End;
      End;
```

Esta misma rutina ha sido reescrita sin el **Else-Exit**.

```
Procedure Inorder (nodo : apuntador);
  Begin
     If nodo <> NIL Then
        Begin
        Inorder (nodo^.izq);
        Write   (nodo^.info);
        Inorder (nodo^.der);
        End;
  End;
```

ENTONCES - SI

Intente decir qué hace este código cuando QTY = 350, y si QTY = 150.

```
IF QTY > 10 Then
     IF QTY > 200 Then
          IF QTY >= 500 Then
               Tarifa = Tarifa  + 1.00
          Else
               Tarifa = Tarifa  + 0.50
          Else;
     Else
          Tarifa = 0.0;
```

Puesto que sólo un conjunto de acciones es efectuado en cada caso, se puede reconstruir de esta forma

If *cond1* **Then** primer caso
else If *cond2* **Then** segundo caso
...
else If *condn* **Then** n-avo caso
else caso por omisión; Esta línea es optativo

Con lo que se puede reescribir el código:

```
    If QTY >= 500 Then Tarifa = Tarifa + 1.00
Else If QTY >  200 Then Tarifa = Tarifa + 0.50
Else If QTY <= 10  Then Tarifa = 0.0;
```

El **Entonces-Si** puede ser la única forma de asegurarse que las evaluaciones (y sus consecuencias) sean ejecutadas en el orden adecuado, por ejemplo:

```
If I > 0 Then
    If A(I) = B(I) Then...
```

Lo cual asegura que **I** está dentro del rango antes de usarla como índice. Lenguajes como Pascal o C proveen de conectores booleanos que garantizan la evaluación (de derecha a izquierda) y la salida tan pronto como el valor de verdad de la expresión sea determinado, por ejemplo:

```
If (I>0) And (A[I] = B[I]) Then...  (Pascal)

if (I>0 && A[i] == B[I]) ... (C)
```

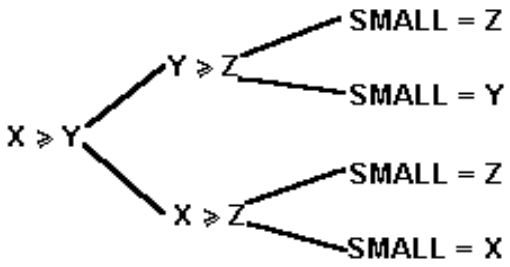
Pero si se topase con un lenguaje que no provee tales facilidades, puede recurrir a las construcción **Then-If.**

Arboles frondosos

El uso de **If-Then** anidados favorece la construcción de grandes árboles de posibilidades. Sin embargo, un programa es una construcción unidimensional, lo cual oscurece la conectividad de las dos dimensiones de los árboles.

Este pequeño programa produce un árbol bastante amplio para los objetivos del mismo.

```
If X >= Y Then
    If Y >= Z Then
        SMALL = Z
    Else
        SMALL = Y
Else
    If X >= Z Then
        SMALL = Z
    Else
SMALL = X
```



Este es un *árbol frondoso*, innecesariamente complejo en cualquier caso, difícil de leer porque es conceptualmente corto y gordo.

La secuencia **ELSE-IF** es larga y débil con los árboles* , es reflejo de cómo pensamos.

* Existen algunos algoritmos, como las búsquedas empleadas en juegos como ajedrez o gato; cuya base de funcionamiento es expandir el árbol de posibilidades del juego y elegir la mejor rama para ganar.

Es más fácil leer descendentemente una lista de objetos, considerándolos como uno a la vez; que recordar la ruta completa hacia el interior de un árbol, aun si las rutas tienen sólo dos o tres ligas.

● Resumen de If-Then-Else

1) Evitar árboles de decisión frondosos

```
If...
   Then If.....
      Else....
Else If....
   Then If.....
      Else....
```

Reorganizándolos en una estructura tipo **CASE**

```
   If... Then....      Case....
Else If... Then....    ------
```

Un **Entonces-Si** es una primera advertencia de que un árbol de decisión está creciendo de modo erróneo. Un **ELSE** nulo indica que el programador sabe que el problema descansa adelante y está trantando de defenderse contra él. Un else -break puede dejar al lector sin comprender cómo se alcanza la próxima parte del código.

Un **Entonces nulo** o un **Entonces *Goto*** suele indicar que una prueba relacional necesita hacerse y alguna de las instrucciones dentro del bloque la hacen.

La regla general es: **después de tomar una decisión hacer algo**. No solamente ir hacia algún lado (goto) o tomar otra decisión. Si sigue cada decisión con una acción que vaya con esto, se puede ver a simple vista lo que cada decisión implica.

● Ciclos

Construir ciclos es relativamente sencillo, deben seguir este esquema:

```
Inicializar condición
While condición (razón para ciclar)
   cuerpo del ciclo
   actualizar condición
Fin del ciclo
```

Un error común en la estructura anterior es olvidar inicializar o actualizar la condición.

● Conclusiones

Una buena programación no es sinónimo de programación sin **goto**.

Cuando un programa no puede hablar por sí mismo, rara es el caso que la confiabilidad o la comprensión resulte de incluir documentación aislada entre el código y el lector.

La mala programación no puede ser explicada, debe ser reescrita.

Mucha gente trata de excusarse por escribir malos programas mediante la maldición de los inconvenientes del lenguaje que deben usar. Hemos visto repretidamente que aun Fortran puede ser domesticado con la disciplina adecuada. La presencia de malas características no es una invitación a usarlos, no es la ausencia de las buenas características una excusa para evitar simularlas tan claramente como sea posible. Los buenos lenguajes son gradables, pero no vitales.

cuadro 4-2

Seis formas de decir: "HOLA MUNDO"

BASIC básico

```
10 PRINT "HOLA MUNDO"
20 END
```

Pascal básico

```
program Hola(input, output)
begin
writeln('Hola Mundo')
end.
```

Lisp básico

```
(defun hola
(print
(cons 'Hola (list 'Mundo))))
```

C con apuntadores

```
#include <stdio.h>
void main(void)
{
char *message[] = {"Hola ", "Mundo"};
int i;

for(i = 0; i < 2; ++i)
printf("%s", message[i]);
printf("\n");
}
```

C con trucos

```
#include <stdio.h>
#define S "Hola, Mundo\n"
main(){exit(printf(S) == strlen(S) ? 0 : 1);}
```

C orientado a objetos

```
#include <iostream.h>
#include <string.h>

class string
{
private:
int size;
char *ptr;

public:
string() : size(0), ptr(new char('\0')) {}

string(const string &s) : size(s.size)
{
ptr = new char[size + 1];
strcpy(ptr, s.ptr);
}

~string()
{
delete [] ptr;
}
friend ostream &operator <<(ostream &, const string &);
string &operator=(const char *);
};
ostream &operator<<(ostream &stream, const string &s)
{
return(stream << s.ptr);
}

string &string::operator=(const char *chrs)
{
if (this != &chrs)
{
delete [] ptr;
size = strlen(chrs);
ptr = new char[size + 1];
strcpy(ptr, chrs);
}
return(*this);
}

int main()
{
string str;

str = "Hola Mundo";
cout << str << endl;
return(0);
}
```

cuadro 4-3

Sugerencias para programar

1. Al programar puede serle útil construir primero una interfaz básica (opciones principales que ejecuten procedimientos vacíos), luego archivos de datos y finalmente los procesos (*rellenar* los procedimientos que han quedado vacíos).
2. Si es un problema que aborda por primera vez, encárguese de que primero funcione el programa, familiarícese con su comportamiento, documente inmediatamente y finalmente arréglelo adecuadamente.
3. Construya las herramientas (acceso a disco, acceso a video, etc.) según las vaya necesitando. Si por las características del problema se explotarán características del hardware, construya módulos interfaz que aislen la rutina controladora de hardware de la rutina de procesamiento.
4. No se puede desarrollar un buen software sin antes haber comprendido el problema y la teoría asociada.
5. Si usará una característica desconocida del lenguaje, habrá que experimentar antes, pues pueden ocurrir efectos secundarios indeseables.

4.2 Criterios para medir la calidad de un programa

● **Confiabilidad**

Un programa confiable es aquél que rara vez falla. Esto es de especial importancia en programas que manejen gran cantidad de datos en una misma corrida, puesto que si una secuencia de datos provoca la caída del programa es muy difícil detectar qué dato en particular lo provocó.

Considere el siguiente fragmento de programa:

```
x := -3.14;
While x < 3.14 Do
    Begin
    y := Sen(x) / Cos(x);
    x := x + 0.1;
    End;
```

Tal como está funcionará sin problemas, y lo hará para la mayoría de incrementos de **x**; sin embargo si cambia el incremento de 0.1 a 0.0628 el programa fallará, ya que en un momento dado, **x** toma el valor de cero lo que resulta en una división por cero.

Un programa puede considerarse realmente confiable sólo después de mucho tiempo de uso sin problemas.

- **Mantenibilidad**

Un programa mantenible es aquél que puede ser modificado y transportado fácilmente por alguien diferente al programador original.

Por tanto es muy importante procurar la *facilidad de comprensión* del código, ya que su propósito no es solamente instruir a un dispositivo electrónico sobre cómo solucionar un problema, sino informar a los futuros programadores sobre cómo fue resuelto el problema.

Para facilitar el mantenimiento se pueden seguir estas cuatro técnicas:

1) Escribir el programa modularmente, así sólo es necesario alterar un módulo en particular.

2) Incluir documentación adecuada y útil, comentando lo que no es obvio: para qué es la rutina, parámetros requeridos y zonas críticas.

3) Utilizar identificadores cuyos nombres sean significativos, y se relacionen con el problema a resolver.

4) Simplificar al máximo las expresiones booleanas utilizadas en ciclos y selecciones.

- **Transportabilidad**

Un programa transportable es aquél que no depende de las características de un equipo en particular, por lo que puede ejecutarse con facilidad en diferentes máquinas.

La transportabilidad se puede facilitar siguiendo estas cuatro técnicas:

1) Utilizar variables y constantes en lugar de números concretos. Aplique esta técnica al referirse a las dimensiones de la pantalla, que cambian de un equipo a otro.

2) Programe con lenguajes y librerías estándares, por ejemplo use versiones aceptadas por ANSI.

3) Aislar lo más posible aquellas rutinas que exploten características propias del hardware y documentarlas adecuadamente. Así podrán ser modificadas sin tener que reconstruir todo el código.

4) Los programas ejecutables no suelen ser transportables, por lo tanto conserve siempre el código fuente, así podrá recompilar.

No es posible hacer programas que funcionen para todas las máquinas. Aun así conviene programar de la manera más transportable posible.

Tambien hay programas para los que la transportabilidad no es una virtud: aquéllos que explotan al máximo las características de la máquina (por ejemplo en el despliegue gráfico), o que se construyen como herramientas para la máquina (por ejemplo los programas de diagnóstico o las utilerías de manejo de discos).

- **Extensibilidad**

Un programa extensible es aquél que puede ser fácilmente modificado para desarrollar una tarea más compleja sin tener que desechar los algoritmos originales.

Suele requerirse de extensiones en las siguientes situaciones:

1) Se requieren cambios
2) Deben añadirse nuevas funciones
3) Se deben explotar nuevas características del hardware.

Por ejemplo, este procedimiento ordena un arreglo de enteros:

```
Const
    N = 10000;
Var
    A : Array [1..N] Of Integer:

Procedure OrdenarEnteros;
    Begin
    For m .= 1 To N-1 Do
      For j := 1 to N-1 Do
        If Abs(A[j]) < ABS (A[j+1]) Then
          Begin
          Tmp := A[i];
          A[i] := A[i+1];
          A[i+1] := A[i];
          End;
    End;
```

Utilizando como base este mismo procedimiento, podemos hacer un procedimiento para ordenar registros de una agenda:

```
Const
    N = 10000;
Type
    Persona = Record
        Nombre,
        Dirección   : String;
        Edad        :  Integer;
        Telefono    : String;
    End;
Var
    A : Array [1..N] Of Persona:

Procedure OrdenarRegistros;
    Begin
    For m .= 1 To N-1 Do
      For j := 1 to N-1 Do
        If Abs(A[j].Nombre) < ABS (A[j+1].Nombre) Then
          Begin
          Tmp := A[i];
          A[i] := A[i+1];
          A[i+1] := A[i];
          End;
    End;
```

- **Generalidad**

Un mismo programa puede resolver diversas situaciones de un mismo problema. Por ejemplo, esta función transforma el valor **Yr** en el valor **Yp**; el algoritmo básico es el mismo, pero adecuado para poder operar con varios tipos de datos.

```
Function Yp (var Yr) : Word;
    Var
        rY  : Real      Absolute Yr;
        iY  : Integer   Absolute Yr;
        siY : ShortInt Absolute Yr;
        byY : Byte      Absolute Yr;
    Begin
        Case DataTypeDefa Of
        _INTEGER:   Yp := (OrigenY+AltVent)-Trunc(SY*(iY-Yrmin));
        _SHORTINT:Yp := (OrigenY+AltVent)-Trunc(SY*(siY-Yrmin));
        _BYTE:      Yp := (OrigenY+AltVent)-Trunc(SY*(byY-Yrmin));
        _REAL:      Yp := (OrigenY+AltVent)-Trunc(SY*(rY-Yrmin));
        End;
    End;
```

Esta rutina se encarga de dibujar un punto en la pantalla. Aun cuando su función es muy sencilla, está diseñada para dibujar puntos delgados o gruesos.

```
Procedure Punto (X,Y : Word);
    Begin
        SetLineStyle (SolidLn,00,NormWidth);
        SetFillStyle (SolidFill,DataCol);
        Case LineaDef Of
        DELGADA : PutPixel (X,Y,DataCol);
        ANCHA   : FillEllipse (X,Y,2,2);
        End;
    End;
```

Sin embargo observe que mientras más general es una rutina, es más difícil su lectura ya que abarca gran cantidad de posibilidades que opacan la claridad del algoritmo.

- **Código reutilizable**

Las tareas que se utilizan muchas veces dentro de un mismo programa suelen ser también muy utilizadas en varios programas diferentes.

Tal es el caso de las rutinas que inicializan gráficos, manejan archivos, las interfaces con el usuario, los módulos de entradas y salidas, los ordenamientos y muchas más. Es realmente un desperdicio de tiempo reescribir estas rutinas cada que se hace un programa nuevo.

- **Eficiencia**

Un programa eficiente hace un uso adecuado del tiempo o de la memoria.

Un programa demasiado eficiente complica la claridad del código, pero una eficiencia aceptable consiste en evitar el derroche de tiempo o memoria.

```
Const
     TOTAL_ESTRELLAS = 10000;
Var
     i,X,Y,col,
     MaxX,
     MaxY,
     MaxCol,
     NumEstrellas: Integer;

 Begin
   Randomize;

   { Limpiar pantalla }
   ClearDevice;
   NumEstrellas := TOTAL_ESTRELLAS;
   For i := 1 To NumEstrellas Do
        Begin
        MaxX := GetMaxX;
        MaxY := GetMaxY
        MaxCol := GetMaxColor;
       { Calcular coordenadas }
       X := Random (MaxX);
       Y := Random (MaxY);

        { Calcular color }
        col := Random (MaxCol);

        { Poner punto según coordenadas y color }
        PutPixel (X,Y,col);
        End;
End.
```

- **Elegancia**

Elegancia es una cualidad ambigua y muy relativa, sin embargo podemos considerar que un programa es elegante cuando tiene un buen estilo de programación.

Resulta también elegante hacer programas cuyo código sea tan simple y claro que su lógica sea evidentemente correcta.

- **Interfaz amigable**

De acuerdo a las necesidades particulares del usuario, la interfaz debe ser lo más sencilla y agradable.

Una buena interfaz le debe permitir al usuario interactuar fácilmente con el programa, requiriendo un mínimo de entrenamiento.

Como parte de la interfaz se encuentran las "*ayudas"*, que permiten mostrar en pantalla explicaciones en cualquier momento de la operación del programa.

Actualmente tienen gran popularidad las interfaces gráficas, sin embargo, agregar íconos y ventanas a un producto mal diseñada no lo convertirá en uno bueno; se requiere que tanto el programa como su interfaz sean cuidadosamente elaborados.

cuadro 4-4

¿Hice un buen programa?
Mi programa...
1o. ¿Resuelve el problema para el que fue hecho?
2o. ¿Funciona bien en todas las circunstancias?
3o. ¿Está escrito con claridad y en forma lógica?
4o. ¿Tiene módulos cortos y concretos?
5o. ¿No desperdicia tiempo y ni memoria?
6o. ¿Lo comprenden fácilmente otros programadores?
7o. ¿Tiene una interfaz que muestra suficiente información para que el usuario sepa cómo usarlo?

4.3 Algunas malas prácticas de programación

El sólo hecho de aplicar técnicas de programación estructurada no asegura que un programa sea bueno.A continuación se hace mención de varias de las costumbres del programador que pueden conducir a un mal programa.

A. Extenderse demasiado en la interfaz, antes de terminar el problema principal.

B. Construir funciones muy grandes (que exceden de dos páginas, por ejemplo). Los programas o módulos muy grandes son difíciles de entender y mantener

C. Construir funciones con varios propósitos, pues se empobrece la cohesión y se limita la utilidad.

D. Un programa principal que no use módulos. Esto provoca programas largos y difíciles de entender. Operaciones complicadas pueden ser probadas independientemente si se colocan en funciones pequeñas, lo que hace el programa más fácilmente de entender.

La primera mala práctica de programación es programar sin diseñar antes.

E. Funciones muy ligadas al programa principal (usando muchas variables globales, por ejemplo). Muchas funciones acopladas vuelven imposible el mantenimiento, porque un cambio en una causa cambios indeseados e inesperados en el resto de las funciones.

F. Abusar de t**rucos de programación o documentarlos pobremente**. Los Muchos trucos de programación no son confiables y son difíciles de comprender para las demás personas (aun con comentarios).

G. Usar identificadores poco significativos y pocos comentarios.

4.4 El *goto*

Goto es una palabra que suele ser odiada y repudiada ante su sola mención. Sin embargo, quien aspire a convertirse en un buen programador debe conocer el concepto y aprender a manejarlo, pues así podrá recurrir a varios lenguajes, transportar códigos y estudiar libros *antiguos* de programación.

El *goto* fue el resultado de una necesidad histórica, las computadoras que se construyeron siguiendo el modelo Von Neumman ejecutaban las instrucciones de modod secuencial, las órdenes fluían una después de otra. Para lograr repetir secciones (o evitar segmentos de código) se recurrió a una instrucción capaz de cambiar el flujo seguido por las instrucciones, el ***goto***.

Desafortunadamente el *goto* permite hacer programas monstruosos, como este (intente seguirlo):

```
10 A = 0
20 B = 0
30 C = 0
100 Print "M= "
103 Input A
106 Goto 120
110 Print 2
113 A = A / 2
116 If A = 1 Then Goto 190
120 If (Frac(A/2)) = 0 Then Goto 1
125 B = 3
130 C = Sqr (A) + 1
140 If B >= C Then Goto 180
145 If (Frac(A/2)) = 0 Then Goto 160
150 B = B + 2
155 Goto 140
160 If (A/B*B-A) = 0 Then Goto 170
165 Goto 150
170 Print B
173 A = A/B
176 Goto 130
180 Print A
190 Print "Fin"
200 Goto 100
```

En la década de los sesentas, Edsger Dijkstra se encargó de alertar a los programadores sobre lo peligroso que puede resultar el *goto* mal empleado.

El movimiento de programación estructurada trató de abstraer los elementos necesarios para controlar el flujo de las instrucciones y eliminar la necesidad del *goto*.

> Aun cuando odie el *goto* considere aprender a usarlo, pues existen lenguajes en los que no tiene otro elemento de control.

Pero ¿qué ocurre con la gran cantidad de código y algoritmos que fueron escritos aplicando este primer elemento de control?, ¿o si el lenguaje no posee las estructuras de programación, como ocurre con el lenguaje máquina?, ¿y si el algoritmo que necesita sólo está disponible en diagrama de flujo?

cuadro 4-5

El *goto* enmascarado

Usted puede estar usando el goto sin saberlo:

Tanto las estructuras **If-Then**, **Repeat** y **For**; como las llamadas a procedimientos se traducen finalmente en saltos de lenguaje máquina (instrucciones **jump**).

Mas aún, ciertas instrucciones de los lenguajes de alto nivel son un *goto* casi evidente:

en C: **return**/**continue**/**break**
en Pascal: **exit**, **halt, case** (puede sustituir saltos de manera estruturada)
en ASM: **jmp**, **jne**, **jz**,...

Las esctructuras de iteración y las llamadas a procedimientos se traducen como *goto* a nivel de lenguaje máquina.

Los lenguajes de alto nivel suelen proveer las estructuras de control necesarias para evitar al goto, pero los lenguajes de bajo nivel normalmente no las tienen (no considere sólo lenguajes para computadoras, incluso las calculadoras científicas tienen facilidades de programación e incluso en los procedimientos administrativos para organizaciones se usa el goto). Y aun cuando crea que no lo usa (véase el cuadro 4-5), puede abusar de él.

El *goto* usado en exceso hace el programa difícil de seguir, dificulta la depuración y las modificaciones. Por el contrario, usado con cuidado puede simplificar la lógica del programa. Por ejemplo,

```
For .....
    For ....
        While ...
            Begin
            If  cond Then Goto Stop;
            ....
            End
Stop:  WriteLn ('Se ha producido un error');
```

Eliminar el *goto* requeriría utilizar una serie de banderas y complicar la lógica del programa.

No conviene eliminar el **goto** si complica la lógica del programa.
Pero úselo en condiciones excepcionales.

Se puede usar un martillo para clavar un tornillo en una tabla.
Siempre será rápido, pero difícilmente será tan adecuado
como utilizar un destornillador.

Thomas Murphy.

4.5 Lenguajes de programación

Actualmente hay una gran cantidad de lenguajes para las más diversas aplicaciones; muchos son sólo productos *de laboratorio,* otros ya cayeron en desuso y otros son muy utilizados.

Al elegir un lenguaje hay que tener en cuenta que algunas veces resulta más rápido construir la función deseada, que buscarla entre cientos que posea el lenguaje.

La selección del lenguaje debe hacerse de acuerdo con las facilidades que posea (eficiencia de los compiladores, herramientas de apoyo al desarrollo, editores, librerías, etcétera), su disponibilidad en el mercado, e incluso las preferencias del programador.

cuadro 4-6

Lenguajes procedurales

Si ya conoce un lenguaje procedural le será fácil y útil aprender un poco de otros.

Pascal: Es un lenguaje útil para aprender a programar estructural y modularmente. También es útil para "comunicar" algoritmos.

BASIC: Es útil para conocer algo de programación sin profundizar. Conocer BASIC permite comprender textos antiguos y facilita la comunicación con los programadores no especialistas.

C: Es muy útil para construir sistemas transportables y para "comunicar" algoritmos.

Ensamblador: El ensamblador permite desarrollar segmentos de código rápidos y poderosos. Aprender un poco ensamblador permite entender secciones que aparecen en los libros, así como comprender el funcionamiento de los microprocesadores.

Fortran: Es muy útil para Aplicaciones numéricas fuera del mundo de las PC; y facilita la comunicación con científicos.

En la medida de lo posible use el lenguaje que tenga a su disposición, sin embargo que la herramienta no detenga o altere la solución de un problema; pero si tiene la fortuna de elegir un lenguaje, opte por aquél que facilite la codificación del algoritmo diseñado, haga más legible el programa y facilite su mantenimiento.

No hay lenguajes realmente útiles para todo tipo de aplicaciones.

Es muy recomendable mantenerse al tanto del "mundo" de los lenguajes a través de las revistas especializadas y conseguir libros que compendien de manera comparativa varios lenguajes. Así usted se mantendrá actualizado en esta materia y podrá elegir el mejor camino para resolver un problema en particular.

Capítulo 5

❖ *Documentación de programas*

5.1 ¿Por qué documentar?

La documentación se integra de todos los elementos que explican las características de un programa o sistema, y son necesarios para poder utilizarlo, operarlo o modificarlo.

Documentar es una tarea tan necesaria e importante como escribir el código: el código indica cómo funciona el programa, y la documentación indica porqué lo hace.

El responsable de un proyecto no debe permitir que se entreguen sistemas sin documentación.

Cuando un programa es pequeño el programador generalmente puede retener en la mente todos los detalles (al menos por algún tiempo), por lo cual no necesita de documentación. En el caso de programas amplios se vuelve imposible recordar cómo se relaciona cada detalle con los demás.

Si además se considera que posteriormente el programa tendrá que ser modificado, se apreciará que el propósito del código de un programador no es solamente instruir a un dispositivo electrónico sobre cómo solucionar un problema, sino informar a los futuros programadores sobre cómo fue resuelto el problema.

Pero la documentación no sólo es para los programadores, también los usuarios de los programas necesitan de ayuda para operarlo; e incluso el comprador requiere de material que le permita evaluarla compra.

Hacer que la documentación sea concisa y descriptiva a la vez.
Ser congruente en todos los programas.

El tiempo de lectura de los programas es mucho más largo que el de escritura.

Si el proyecto es pequeño, puede ser suficiente con recopilar toda la documentación en un archivo e integrarlo con los demás archivos del programa.

Cuadro 5-1

Documentación

MANUAL TECNICO O DE MANTENIMIENTO: Documentación destinada a los programadores y sirve de referencia para darle mantenimiento a un sistema durante su vida útil. Tiene dos niveles: Interna y Externa.

Documentación interna: Son los comentarios que se encuentran dentro del programa fuente, y que describen detalles significativas para un programador.

Documentación externa: Es aquella que se encuentra independiente del programa fuente. Suele encontrarse en un manual (impreso) que se proporciona con el programa y describe con profundidad sus características técnicas y funcionamiento.

Manual del usuario: Es un documento que le indica al usuario (u operador) conocer, utilizar y operar correctamente el sistema. Describe los objetivos y opciones del programa, así como sus características externas.

La documentación básica para un programa (o un sistema pequeño) se muestra en el cuadro 5-1. Sin embargo, si se desarrolla un sistema grande o complejo, cada etapa del proceso de desarrollo deberá ser documentada. A continuación mencionamos algunos de los puntos que conviene incluir, pero sin profundizar en ellos.

A. Documentación de la etapa de análisis.

1. Documento de Necesidades y requerimientos.
2. Cronograma
3. Reporte de factibilidad
4. Propuesta de funcionamiento
5. Diagramas de Flujos de Datos, Diccionario de Datos y Miniespecificaciones.
6. Especificar en lenguaje natural para el usuario.
7. Reportes de avance.

B. Documentación de la estapa de diseño

3. Cronograma
4. Especificación del Sistema
5. Diagramas de estructura modular.
6. Reportes de avance

C. Documentación de la etapa de implementación.

4. Cronograma
5. Documentación de pruebas realizadas y resultados
6. Documentación de la evolución de los prototipos.
7. Reportes de problemas
8. Reportes de avances

Construya fácilmente una documentación completa: elabore un "diario" donde reporte los avances y problemas con el proyecto.

Al finalizar el sistema le resultará más fácil recopilar y estructurar los datos.

No olvide guardar todo, incluyendo el diario, los archivos fuentes (comentados) y diskettes usados.

5.2 Cómo nombrar variables

Los lenguajes de alto nivel proveen la facilidad de usar identificadores (símbolos) para representar valores variables, valores constantes, procedimientos, etcétera. Para obtener mayor ventaja de los identificadores conviene considerarlos como parte de la documentación.

El uso de variables y constantes, evitando el uso de números es una característica que facilita la transportabilidad (al evitar valores concretos) y permite ententer para qué sirve ese valor.

1. Escoger los nombres de los identificadores con cuidado, de tal modo que su significado se comprenda sin ambigüedades y explicarlos de manera detallada.

   Evitar nombres cuyo significado no se relacione con el problema, como suele pasar cuando se usan abreviaturas extrañas, una sola letra extraña, o nombres de amigos(as).

   Ejemplos:

```
Var
        lechuga : Integer;
        q : Real;

Procedure Julia;
```

2. Las variables locales deben tener nombres sencillos.

   Ejemplos:

```
Var
    i : Integer;
    cont : Integer;
    opción : Char;
```

3. Evitar los errores de ortografía deliberados, o asignar sufijos carentes de significado que suelen utilizarse para hacer identificadores diferentes.

   Ejemplos:

```
Var
    Cont1, Cont2 : Integer;
    Aux, Aux1 : Integer;
    Aux2 : Real;
    Aux3 : Char;
    Bandera : Boolean;
    Vandera : Boolean;
```

6. No escoger nombres que puedan confundirse fácilmente.

   Ejemplos:
   Abre y Abrir
   Dibuja, Dibujo y Dibujar
   Inicializa e Inicializar.

7. Tener cuidado con las letras l, O y 0 que pueden resultar confusos, especialmente al imprimir.

cuadro 5-2

NOTACION HUNGARA[7]

Es una convención para nombrar identificadores. Especialmente ayuda a reducir los errores de incompatibilidad de tipos, a diferenciar claramente entre variables, constantes y tipos y nombrar rápidamente los identificadores.

Variables

Se construyen mediante un prefijo, un tipo base y un sufijo calificador.

PREFIJO	TIPO BASE	CALIFICADOR
p = pointer	i = integer	First
arr = array	ch = char	
d = diferencia	r = real	Src
h = handle (manejador)	f = float	Dst
c = contador	w= word	Min
	by = byte	Max
	f = flag (bandera)	Lim
	b = boolean	Tmp
	st = string (pascal)	
	sz = string (c)	
	fh = file handler	
	l = long	
	u = unsigned	

Tipos Definidos por el Usuario

Agregar el prefijo "t_" o el sufijo "Type". Por ejemplo:

```
Type
    t_SetChar = Set Of Char;
    {forma alternativa:      }
    {SetCharType = Set Of Char;}
Var
```

[7]Soto Maldonado, Luis Daniel. "Poderosa Herramienta de Programación: La Notación Húngara." Memorias del IV Congreso Nacional sobre Informática y Computación, ANIEI. Aguascalientes, México, 1991.

```
    Set1 : t_SetChar;
```

Constantes

Utilizar mayúsculas en todas las letras; si son varias palabras separarlas con "_". Por ejemplo:

```
Const
    MIN = 0;
    MAX = 100;

Var
    ci : Integer; { contador de enteros }

For ci := MIN To MAX Do
```

Procedimientos y Funciones

Identificar con nombres significativos, usando minúsculas y separando las palabras con mayúsculas (sin preposiciones). Utilizar los verbos en infinitivo. Por ejemplo:

```
Procedure GenerarArchivo (stFileName : String);
```

Si hay funciones similares que provienen de módulos diferentes, se agrega como prefijo la abreviatura del módulo del que proceden. Por ejemplo:

```
Procedure TXTGotoXY (byX,byY : Byte);
Procedure GRGotoXY  (wX,XY   : Word);
```

En funciones el prefijo indica el tipo que regresa. Por ejemplo:

```
Function ICalculaAño (ci : Integer) : Integer;
```

5.3 Documentación interna

La documentación interna son los comentarios que se encuentran junto con el programa fuente y son ignorados por el compilador.

Se usan para describir cosas significativas para un programador: qué hace cada sección, quién lo programó, etcétera.

Comentar a medida que se escribar el programa. Preparar una parte de la documentación antes de empezar. Modificar la documentación a medida que se modifica un programa.

El primer elemento de documentación es el código. Un buen código ya es una guía, mientras una gran cantidad de comentarios no rescata un mal código. Los cuadros 5-4, 5-5, 5-6 y 5-8 muestran varios ejemplos de programas comentados.

cuadro 5-3

Los comentarios generales de un programa son:

a) Encabezado del programa o módulo.
b) Convención utilizada para abreviaturas.
c) Límites de cada sección importante.
d) Variables, constantes, procedimientos y funciones.
e) Funcionamiento y parámetros requeridos por cada módulo.
f) Funcionamiento detallado de los trucos.

¿Qué comentar?

1. Encabezado general del programa (o librería):

a) nombre del programa o líbrería,
b) nombre del programador,
c) fecha de realización,
d) versión actual,
e) propósito de todo el programa,
f) breve descripción del programa o líbrería,
g) historial del desarrollo del programa y,
h) bibliográfía o referencia a documentación externa.

Cuadro 5-4

Ejemplo de un encabezado con historial

```
UNIT PlotData; { versión 2.01 }

{ Propósito: Rutinas para graficar funciones, dados los datos }
{ Autor: Jorge Vasconcelos Santillán                          }

{ Versión 1.00 - Rutina original ViewGraph, para datos reales
          1.01 - Se agrega el visualizador en coordenadas polares
          1.02 - Se hace rutina paralela para datos enteros
          1.03 - Se modifica la 1.01 para anexar datos a un mismo despliegue
          2.00 - Se generaliza la rutina para graficar indistintamente enteros
                 o reales. Se elimina la opcion en coordenadas polares.
                 Se optimiza el calculo de la retícula, eliminandose la
                 posibilidad de dimensionar directamente la ventana, y
                 calculándolo en base a una cantidad de Pixels por Division.
                 Se agrego la opcion para limpiar un área.
          2.01 - Aumento de las posibilidades de casting para graficar
                 enteros cortos sin signo. Se agrega la función overview
                 para graficar datos sobre otros. Se añaden las rutinas
                 SetGridColor y SetDataColor. Se mantiene SetColors por, y
                 se colocó nuevamente SetDimVent, por  compatibilidad.
                 Se mejora la rutina SetAutoRange. Se agregan rutinas para
```

```
                etiquetar. }
```

2. Encabezado particular de cada rutina:

   a) nombre de la rutina,
   b) descripción de parámetros (entradas y salidas),
   c) propósito de la rutina,
   d) breve descripción del método aplicado,
   e) bibliográfía o referencia a documentación externa.

3. Explicación del nombre y uso de los identificadores principales (variables, constantes, procedimientos y funciones). De utilizar abreviaturas, indique la convención utilizada.

4. Descripción breve de las funciones de cada sección y del funcionamiento de cada módulo, así como los parámetros requeridos. Indicar si alguno de los parámetros puede tomar sólo algunos valores específicos, o tienen un significado especial.

   Describa el algortimo empleado en el módulo utilizando un lenguaje común y corriente (como ejemplo véase los algoritmos del cuadro 5-7).

cuadro 5-5

Ejemplos de subrutinas y parámetros comentados

```
Procedure CursorMouse(X,Y:Integer; var CURSOR:ARRCUR);
    {OBJETIVO: Asigna la forma del cursor al cursor del ratón y
               fija el punto de seleccion dentro de la forma del cursor.
     PARAMETROS:
                    X,Y son los pixels dentro del cursor que contienen el
                      punto de selección (hot pixel).
                  CURSOR es un arreglo que describe la forma de cursor.
                  ARRCUR es un tipo definido como ARRAY[0..31] of INTEGER
    }

                        {.....................................}

Procedure PosMouse(var mbt:t_Boton; var mx,my:integer);
    {OBJETIVO: Regresa la posición del ratón y el botón que fue
               presionado
     PARAMETROS:
                  mbt = 1, si el botón izquierdo se presionó
                      = 2, si el botón derecho se presionó
                      = 3, si ambos botones se presionaron
                      = 4, botón medio presionado
                      = 5, botónes izquierdo y medio
                      = 6, botón derecho y medio
                      = 7, los tres botones presionados
                  mx = coordenadas en el eje x
                  my = coordenadas en el eje y}
```

Si en un momento dado el preparar la documentación puede provocar que se pierda alguna idea fundamental para el funcionamiento del programa, entonces efectuar pequeños comentarios e ir marcando el programa, para posteriormente ampliar esos comentarios.

5. Usar un estilo de programación claro y elegante para facilitar la lectura. Destacar las partes relacionadas y los bloques.

    a) Usar líneas en blanco para diferentes partes del programa.
    b) Sangrías consistentes (entre dos y cuatro espacios).
    c) Marcar con un comentario el inicio y el final de cada sección importante .

6. No comentar lo obvio:

    a) No repetir lo que realiza el código.
    b) No usar expresiones coloquiales propias de un área de trabajo en particular.

7. Evitar los trucos o explicarlos claramente.

    Como ejemplo, considere este programa en lenguaje C[8]. No le diremos qué es, ni que hace, pero le aseguramos que funciona.

```
main() {int _o_oo_,_ooo;for(_o_oo_=2;;_o_oo_++){for(_ooo_=2;
_o_oo_%_ooo_!=0;_ooo_++);if(_ooo_==_o_oo_)printf("\n%d",_o_oo_);}}
```

8. Si el lenguaje es árido (por ejemplo, ensamblador, lisp u otro) entonces describa el algoritmo aplicado mediante comentarios.

    Por ejemplo:
    m = i++; /* m toma el valor de i y luego i se incrementa en 1 */

Cuadro 5-6 (a)

Un programa árido sin comentar

```
(defun DFS (origen destino arbol)
  (setq q (list origen 0))
  (loop ((null q))
    (setq expl (car q) )
    (setq q (cdr q) )
    ((equal  expl destino))
    (setq l (expande expl arbol))
    (setq q (append l q) )
  )
)
```

[8]Rutina que aparece en Embree, Paul M. and Kimble, Bruce. *C Languaje for Digital Signal Processing.* p.111. Prentice Hall, 1991.

Cuadro 5-6 (b)

Un programa árido con comentarios

```
(defun DFS (origen destino arbol)
  (setq q (list origen 0))                ; inicializar lista de nodos
  (loop ((null q))                        ; while q (no vacio)
    (setq expl (car q) )                  ; expl = first (q)
    (setq q (cdr q) )                     ; q    = tail  (q)
    ((equal  expl destino))               ; meta encontrada => exito
    (setq l (expande expl arbol))         ; l = expande (expl)
    (setq q (append l q) )                ; q = l + q
  )
)
```

cuadro 5-7

DESCRIPCION DE ALGORITMOS.

En los siguientes ejemplos se describe, con lenguaje cotidiano, el funcionamiento de varios algoritmos

1. Suma de dos números

```
{
Solicitar al usuario los números Ay B,
sumarlos y enviarle el resultado     }

    Inicio
    Leer A,B
    C = A + B
    Escribir C
    Fin
```

2. Sumar los cien primeros naturales

```
{
Se efectuan cien repeticiones en las que se
van acumulando las sumas de cada número entero
}

    Inicio
    SUMA     = 0
    CONTADOR = 1
    Repetir
        SUMA     = SUMA     + CONTADOR
        CONTADOR = CONTADOR + 1
    Hasta que CONTADOR = 100
    Escribir SUMA
    Fin
```

3.Tres rutinas en Pascal

```
    Function Xp (Xr : Real) : Integer;
    {
    Traslación del eje X, al centro de la pantalla.
    Xr es el punto original y Xp está trasladado
    }
    Begin
        Xp := Trunc(Xr + GetMaxX/2);
```

```
End;

Function Yp (Yr : Real) : Integer;
{
Traslación del eje Y, al centro de la pantalla.
Yr es el punto original y Yp está trasladado
}

Begin
    Yp := GetMaxY - Trunc(Yr + GetMaxY/2);
End;

Function ObtenMascara (X : Word) : Byte;
{
Dada la coordenada X encuentra el número de
byte que le corresponde y luego en Máscara
enciende el bit preciso dentro del byte
}
Var
    Numero,
    Mascara : Byte;
Begin
    Numero      := 7 - X mod 8;
    Mascara     := 1 shl Numero;
    ObtenMascara := mascara
End;
```

cuadro 5-8

Un ejemplo de código comentado

```
{ ____________________________________________________________________ }
{ _                UNIDAD MANEJADORA DE ARREGLOS DINAMICOS           _ }
{ _                     (arreglos de dos dimensiones)                _ }
{ _                      versión 1.0  Octubre, 1993                  _ }
{ _                      Jorge Vasconcelos Santillán                 _ }
{ ____________________________________________________________________ }

{ _--------------------------------------------------------------------_ }
{ _ > EXPLICACION: La unidad ArrBytes permite definir (dinámicamente)  _ }
{ _   un arreglo de dos dimensiones en tiempo de ejecución. Se proveen _ }
{ _   las rutinas necesarias para inicializar y finalizar el arreglo,  _ }
{ _   (de cualquier tamaño, para cualquier tipo de datos); y para leer _ }
{ _   e introducir valores en las casillas del arreglo.                _ }
{ _                                                                    _ }
{ _   La unidad se basa en reservar un espacio de memoria (necesario   _ }
{ _   para almacenar NxM casillas de un mismo tipo) y en calcular la   _ }
{ _   posición interna de ese espacio equivalente a la casilla (i,j)   _ }
{ _   mediante apuntadores a bytes.                                    _ }
{ _                                                                    _ }
{ _   Se define una estructura para manejar los arreglos dinámicos,    _ }
{ _   ArrByte, que soporta un apuntador a la memoria reservada, las    _ }
{ _   dimensiones del arreglo, y el tamaño de los datos que se almace- _ }
{ _   nen.                                                             _ }
{ _                                                                    _ }
{ _   Para poder utilizar los arreglos antes deben inicializarse indi- _ }
{ _   cando las columnas y renglones necesarios y el tipo de dato que  _ }
{ _   contendrán las casillas. Cuando se haya finalizado el uso del    _ }
{ _   arreglo debe liberarse la memoria reservada para el arreglo,     _ }
{ _   mediante la rutina de cierre del arreglo.                        _ }
```

```
{ _------------------------------------------------------------------------_ }

UNIT ArrBytes;

INTERFACE
             { ____________________________________________ }
             { _          Tipo  Disponibles               _ }
             { ____________________________________________ }


Type
    ArrByte = Record
       byTamDato : Byte;         { Tamaño del tipo de datos de cada casilla }
         wNumBytes,              { Número de bytes de memoria reservada     }
         wMaxRen,                { Total de renglones en en arreglo         }
         wMaxCol   : Word;       { Total de columnas en en arreglo          }
         pBuff     : Pointer;    { Apuntador a la memoria reservada         }
     End;
             { ____________________________________________ }
             { _         Librería Disponible              _ }
             { ____________________________________________ }


Procedure InitArray    (var Arreglo : ArrByte; wRen,wCol : Word;
                        DataSize : Byte);
Procedure CloseArray   (var Arreglo : ArrByte);
Procedure InsertarDato (var Arreglo : ArrByte; wRen,wCol : Word; var Dato);
Procedure LeerDato     (var Arreglo : ArrByte; wRen,wCol : Word; var Dato);


{ _------------------------------------------------------------------------_ }
{ _                   DESCRIPCION DE LAS RUTINAS                         _ }
{ _                                                                      _ }
{ _ InitArray: Inicializa Arreglo[wRen,wCol], teniendo cada celda datos _ }
{ _            de tamaño DataSize.                                       _ }
{ _                                                                      _ }
{ _ CloseArray: Termina el uso del Arreglo indicado.                     _ }
{ _                                                                      _ }
{ _ InsertarDato: Asigna un valor a una casilla. Es equivalente a        _ }
{ _               Arreglo[wRen,wCol] := Dato                             _ }
{ _                                                                      _ }
{ _ LeerDato: Regresa el valor contenido en una casilla. Equivale a      _ }
{ _               Dato := Arreglo[wRen,wCol]                             _ }
{ _------------------------------------------------------------------------_ }


IMPLEMENTATION

{ ------------------------------------------------------------------------ }

Procedure InitArray (var Arreglo : ArrByte; wRen,wCol : Word; DataSize  Byte);
   { Reserva el espacio de memoria necesario para almacenar }
   { wRen X wCol X DataSize, inicializa en ceros Arreglo y  }
   { define todas las características de Arreglo.          }

    Begin
         With Arreglo Do
              Begin
           byTamDato := DataSize;
              wMaxRen  := wRen;
              wMaxCol  := wCol;
              wNumBytes := wMaxRen*wMaxCol*byTamDato;
           (* Tratar de validar mejor la reserva de memoria *)
           if MaxAvail < wNumBytes Then
              Begin
              WriteLn('Memoria insuficiente para crear arreglo');
               Halt(1);
               End;
              GetMem (pBuff,wNumBytes);
              FillChar (pBuff^,wNumBytes+1,0);
              End;
    End;
```

```
{ ------------------------------------------------------------------------ }


Procedure CloseArray (var Arreglo : ArrByte);
    { Libera la memoria reservada para Arreglo }
    { y limpia las características del Arreglo }
    Begin
         FreeMem (Arreglo.pBuff,Arreglo.wNumBytes);
         Fillchar (Arreglo,SizeOf(Arreglo),0);
    End;

{ ------------------------------------------------------------------------ }

Procedure InsertarDato (var Arreglo : ArrByte; wRen,wCol : Word; var Dato);
    { Calcula el byte correspondiente a wRen y wCol, }
    { (dentro de la memoria reservada) y efectúa una }
    { transferencia de bytes desde Dato hacia el byte}
    { localizado.                                    }

    Var
         wPosByte,
         wPosDeRen,
         wPosDeCol  : Word;
         pby        : ^Byte;
    Begin
         With Arreglo Do
              Begin
           {Calcular posición del byte}
              wPosDeRen := (wRen-1)*(wMaxCol*byTamDato);
              wPosDeCol := (wCol-1)*byTamDato+1;
              wPosByte  :=  wPosDeRen+wPosDeCol;

           {Apuntar al principio de la memoria reservada}
              pby       := pBuff;

           {Moverse hasta la posición calculada}
              Inc (Longint(pby),wPosByte);

           {Transferir datos}
           Move (Dato,pby^,byTamDato); {pby^ := Dato^}
              End;
    End;

{ ------------------------------------------------------------------------ }

Procedure LeerDato (var Arreglo : ArrByte; wRen,wCol : Word; var Dato);
    Var
         wPosByte,
         wPosDeRen,
         wPosDeCol  : Word;
         pby        : ^Byte;
    Begin
         With Arreglo Do
              Begin
           {Calcular posición del byte}
              wPosDeRen := (wRen-1)*(wMaxCol*byTamDato);
              wPosDeCol := (wCol-1)*byTamDato+1;
              wPosByte  :=  wPosDeRen+wPosDeCol;

           {Apuntar al principio de la memoria reservada}
              pby       := pBuff;

           {Moverse hasta la posición calculada}
              Inc (Longint(pby),wPosByte);

           {Transferir datos}
           Move (pby^,Dato,byTamDato); { Dato^ := pby^ }
              End;
    End;

{ ------------------------------------------------------------------------ }

BEGIN
END.
```

5.4 Documentación técnica

El **manual técnico** es documentación externa que describe con profundidad las características técnicas* y funcionamiento del programa (o sistema). Está destinado a los programadores y sirve de referencia para darle mantenimiento al sistema durante su vida útil.

Cuadro 5-9

Los elementos generales de un manual técnico son:

a) Propósitos y funcionamiento del programa.
b) Esquema lógico del funcionamiento del programa
c) Nuevos tipos definidos, principales variables globales, archivos utilizados, etc.
d) Explicación de fórmulas y procesos complejos.
e) Especificación de datos de entrada y datos de salida.
f) Formato de los archivos y de las pantallas.
g) Datos utilizados en las pruebas.
h) Puntos débiles y futuras expansiones.
i) Guía de Referencia de Rutinas.
j) Diagramas.
k) Listado del programa fuente.
l) Otras características técnicas

Este manual suele quedarse en el lugar de desarrollo, tanto para futuros cambios, como para verificar que fue correctamente instalado; por ejemplo si una vez que el programa ha sido puesto en funcionamiento ocurre un error, los datos de prueba servirán como parámetro para tratar de detectar cuál fue el percance. Otra información útil que puede tener este manual es el tamaño (en bytes) de los archivos ejecutables, pues un cambio inesperado podría sugerir la presencia de un virus.

Otro de los puntos interesantes son los puntos débiles, mencionarlos puede servir para saber por dónde podría fallar el programa en algún momento, o por donde se le puede mejorar.

* Si usted ha trabajado con componentes electrónicos, seguramente conocerá la importancia del manual que detalla todas las características y funcionamiento de los mismos. Otro ejemplo de documentación técnica son los manuales que describen elementos importantes de una computadora, como su mapa de memoria, asignación de puertos o interrupciones, y los libros que indican los parámetros de esas interrupciones y su propósito.

El cuadro 5-10 muestra una plantilla que cuenta con elementos mínimos y ejemplos para elaborar el manual técnico de su propio proyecto. Adecuelo a sus necesidades y redáctelo con un lenguaje conciso y formal.

Para elaborar el **manual técnico**, el programador debe explicarse el programa a sí mismo (como si fuese otro programador). No debe dejar detalles importantes a su memoria, pues al cabo de un tiempo los olvidará.

cuadro 5-10

Plantilla general para un manual técnico

☞ *Este es un modelo de los temas que puede incluir en el manual del usuario, algunas de las secciones incluyen una redacción propuesta para abordar ese punto en particular, y otras sólo mencionan qué deberá incluirse.*

Indice

☞ *Este es el índice general, pueden agregársele otros capítulos.*

1. Propósitos y funcionamiento del programa

Los própositos del programa son:

☞ *Incluya los propósitos propios de sus proyecto.*

1) Obtener una _____, programada bajo _____, y que efectúe correctamente las funciones básicas de _____ .

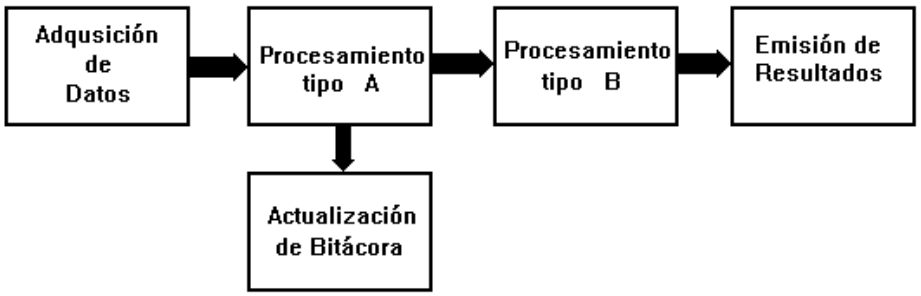
2) Funcionamiento bajo el sistema operativo _____.
3) Aplicación de una interfaz con el usuario amigable y sencilla.

☞ *Incluya los elementos que fundamentan su programa o las necesidades que motivaron el desarrollo de una solución particular, así como el camino seguido.*

El funcionamiento de _____ se basa en el (los) algoritmo(s) de _____ y en la teoría _____.

2. Esquema lógico del funcionamiento del programa

☞ *Describa el funcionamiento lóqico de su programa, puede redactarlo o usar diagramas de bloques.*



3. Fórmulas y procesos complejos

☞ *Describa las principales fórmulas utilizadas y los procesos cuya construcción resultó difícil. Aquí se muestra sólo un ejemplo.*

Las coordenadas de los puntos se calculan mediante las fórmulas trigonométricas:

$$\boldsymbol{X} = \boldsymbol{r}\ \text{Cos}\ (\boldsymbol{q_1})$$
$$\boldsymbol{Y} = \boldsymbol{r}\ \text{Cos}\ (\boldsymbol{q_2})$$
$$\boldsymbol{Z} = \boldsymbol{r}\ \text{Cos}\ (\boldsymbol{q_3})$$

donde $\boldsymbol{r}$ es la distancia entre el punto y el origen del sistema,
$\boldsymbol{q}$ es el ángulo entre el vector y los ejes del sistema.

4. Librerías requeridas

☞ *Indique los nombres y descripción de las librerias que tuvo que utilizar para construir su proyecto.*

5. Datos utilizados en las pruebas

☞ *Haga una lista de los datos (con su respectivo tipo) que se utilizaron para probar el programa y de los resultados que se obtuvieron.*

Entrada	**Resultado**
x:5.12	
y:0.12	
z:-1.2	m:16

6. Puntos débiles y futuras expansiones

Puntos débiles:

☞ *Indique los errores que ha detectado y que no han sidor corregidos. Incluya también las sugerencias que mejorarían al programa.*

1) El manejo de los menús necesita optimización.
2) No se pueden editar los datos mientras se les captura.
3) Ocurre un error bajo circusntancias desconocidas con las entradas reales.
4) La pantalla no se autorrefresca después de que ocurre un error.

Futuras expansiones:

☞ *Enliste los planes de mejoramiento de su programa. Sólo incluya lo que tenga pensado hacer.*

1) Trasladar la interfaz a un ambiente gráfico.
2) Activar la opción de usar mouse.
3) Corregir las fallas hasta ahora detectadas.

7. Referencia de nuevos tipos definidos

☞ *Describa definió los nuevos tipos. Aquí mostramos algunos ejemplos.*

Tipo: archCampos
Descripción: Archivo con estructura campos.
Estructura: file of regCampos

Tipo: cad12
Descripción: Cadena de longitud 12.
Estructura: string [12]

Tipo: puntRegCad

Descripción: Puntero a registros de tipo cadena.
Estructura: ^regCad
Tipo: regCad
Descripción: Registro para manejo de listas ligadas con datos de tipo cad.
Estructura: cadena : string {Cadena de 1 a 255 caracteres}
posReg : longint {Posición del registro en el archivo de datos}
ant : puntRegCad { Nodo anterior.}
sig : puntRegCad { Nodo siguiente.}

8. Referencia de constantes y variables globales

Constantes

☞ *Indique las constantes globales en su programa, nombre, valor y uso.*

IDENTIFICADOR	VALOR	COMENTARIO
ENTER	#13	Tecla de aceptación.
ESC	#27	Tecla de escape.
NULO	#0	Valor nulo.
MAX_WORD	2*MaxInt	Máximo valor de un word
DEFAULT	15	Número por default

Variables globales

☞ *Indique las variables globales en su programa, nombre, tipo y uso.*

IDENTIFICADOR	TIPO	COMENTARIO
maxX	integer	Máxima coordenada gráfica en X.
maxY	integer	Máxima coordenada gráfica en Y.
nomArchivo	cad12	Nombre del archivo donde se guardan los datos
opcion	char	Permite el control de los menús.
borrado	boolean	Controla el borrado físico de registros en disco
raíz	Nodo	Raíz de una lista ligada.

9. Referencia de procedimientos y funciones

☞ *Indique las rutinas usadas en su programa, nombre, parámetros y uso.*

Rutina: procedure AbreVentana (Num : Byte);
Función: Borra la zona de la pantalla donde aparecerá la ventana. Posiciona el cursor en el origen de la ventana.
Parámetro: Num - indica el número asignado a la ventana que se abrirá.

Rutina: procedure Marco (x1,y1,x2,y2 : Word);
Función: Dibuja un marco en la pantalla, dados los límites.
Parámetros: x1, y1 - esquina superior izquierda del marco
x2,y2 - esquina inferior derecha del marco.
Rutina: function Existe (nomArchivo : String) : Boolean;
Función: regresa un resultado TRUE si el archivo especificado existe, y FALSE si el archivo no existe.
Parámetro: nomArchivo - Nombre del archivo, puede incluir la ruta de acceso.

Apéndices

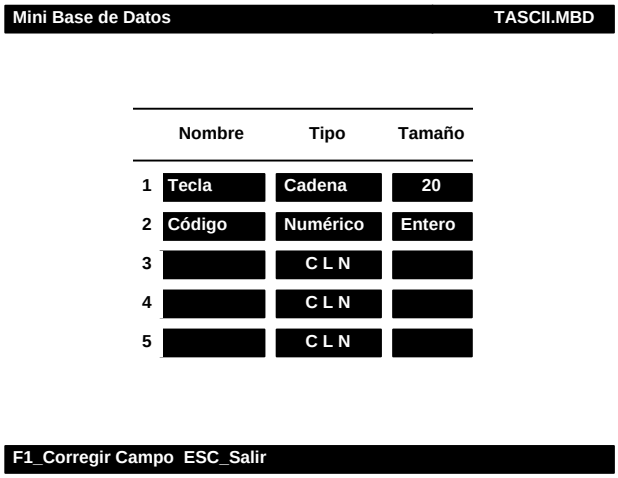
A-1. Formato de los archivos y pantallas.

☞ *Indique qué forma tienen los registros de archivos y la estructura de las pantallas.*

Formato de los registros del archivo: AGENDA.DAT

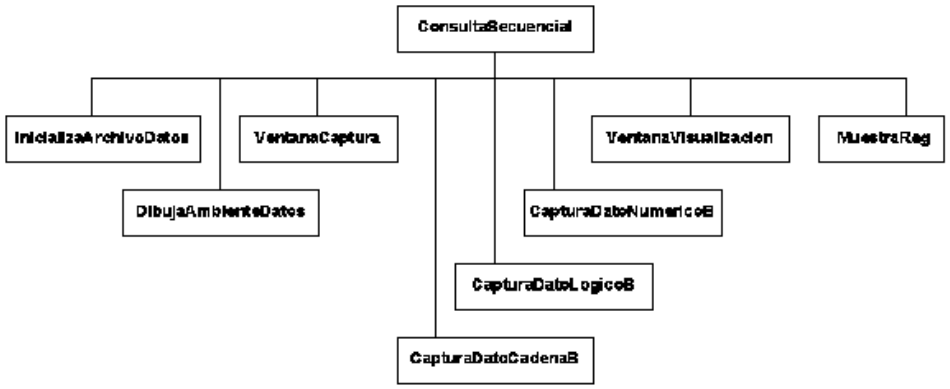
Campo	Tipo	Tamaño
Nombre	String	30
Direccion	String	50
Telefono	String	7
Edad	Byte	1

Formato de la pantalla de captura:



A-2. Diagramas.

☞ *Incluya aquí todos los diagramas que se hayan elaborado para construir el programa.*



A-3. Listado del programa fuente.

☞ *Anexe una copia impresa del listado del programa fuente, sin retocarlo.*

A-4 Características Técnicas

☞ *Enliste las principales características técnicas.*

- Hardware y software requerido
- Sistema operativo y versión.
- Memoria requerida.
- Tamaño de los archivos ejecutables.
- Arbol de directorios recomendado.
- Método de instalación.

5.5 Documentación para el usuario

El **manual para el usuario** es un documento que describe los objetivos, opciones y características externas del programa (o sistema), necesarios para su operación.

En general, los programas y sistemas no suelen producirse para otros programadores, por lo que este manual debe ser lo suficientemente claro y sencillo para que el usuario u operador sepa qué esperar del producto, cómo operarlo correctamente y cómo solucionar algunos problemas.

El manual del usuario, junto con la interfaz y las "*ayudas"*, forman la *carta de presentación* del programa, por lo que resultan primordiales. Todas las ventajas que puedan tenerse, resultan inútiles cuando el usuario no entiende cómo usarlas.

Para elaborar el **manual del usuario**, el programador debe ponerse en el lugar de una persona que desconoce totalmente el programa, e incluso, con conocimientos mínimos de computación.

cuadro 5-11

Los elementos generales de un manual del usuario son:

a) Descripción clara y amplia del objetivo del programa.
b) Requerimientos mínimos de hardware.
c) Sistema operativo requerido.
d) Pasos necesarios para instalarlo por primera vez y para des-instalarlo.
e) Pasos necesarios para ejecutarlo.
f) Descripción de opciones, comandos y menúes.
g) Descripción y muestra de salidas que se pueden obtener.
h) Ejemplos de operación.
i) Posibles errores y cómo solucionarlos.
j) Nombres de los archivos que se incluyen en los diskettes y breve descripción de los mismos.
k) Glosario

Desafortunadamente, a pesar de su importancia, el programador suele tener dificultades para desarrollar este documento, debido a que está tan familiarizado con el proyecto, que las características y detalles le parecen obvios, y le resulta difícil ponerlos de manifiesto.

Para desarrollar un manual del usuario, el cuadro 5-11 propone una plantilla que cuenta con los elementos mínimos que el usuario necesitará conocer. Bastará que inserte los datos, y redacte con sencillez, considerando que va dirigida a una persona que desconoce totalmente el programa, e incluso puede carecer de conocimientos de computación. (No tome estas plantilla como un formato absoluto, sólo es una guía que deberá adaptarse a cada necesidad.)

Una vez desarrollado este manual, los textos pueden incorporarse (con las adecuaciones necesarias) a la ayuda en línea del programa* .

Finalmente, como se mencionó en párrafos anteriores, la interfaz puede considerarse también como parte de la documentación para el usuario. Mientras más intuitiva sea, más apoyo será para el usuario (considere que, contra las opiniones de moda, el simple hecho de implementar una interfaz gráfica no significa que resultará intuitiva).

Las utilerías (librerías, compactadores, herramientas, etcétera) suelen producirse para aprovechamiento de programadores y desarrolladores. Aun cuando también requieren de un manual del usuario, este puede desarrollarse bajo la consideración que el destinatario es un especialista.

cuadro 5-11

* Las "*ayudas en línea"* son parte de la interfaz del programa principal, y se encargan de enlazar un grupo de archivos (con texto que describe el uso de algún elemento del programa) con el programa. Básicamente, la ayuda en línea permite mostrar en pantalla las explicaciones, sin alterar los contenidos anteriores de la pantalla, en cualquier momento de la operación del programa. La cosntrucción del manejador de ayuda está bastante relacionada con la construcción de la interfaz con el usuario, y rebasa los alcances de este libro.

Plantilla general para un manual del usuario

☞ *Este es un modelo de los temas que puede incluir en el manual del usuario, algunas de las secciones incluyen una redacción propuesta para abordar ese punto en particular, y otras sólo mencionan qué deberá incluirse.*

Presentación

☞ *Haga una breve presentación de su programa, resaltando sus ventajas*

Bienvenido al sistema XXXXXX, la herramienta de _____ más agradable que se ha desarrollado.

Al adquirir este producto usted da un paso adelante en tecnología, pues este sistema convertirá su computadora en una poderosa herramienta para _____. Con XXXXXX usted podrá [*mencionar características principales]*.

Antes de empezar, dedique algunos minutos a leer este manual, donde encontrará toda la información que necesita para aprender a usar XXXXXX y obtener los máximos beneficios.

Organización del Manual

☞ *Describa la organización física del manual, indicando el objetivo de cada sección; puede ser equivalente a una tabla de contenido.*

Este manual está dividido en las siguientes secciones:

- 1. Antes de empezar: preparativos para la instalación.
- 2. Instalación: transferencia de los archivos a su computadora.
- 3. Conociendo su Computadora: fundamentos de operación de su equipo de cómputo.
- 4. Características del Programa: Características y ventajas del programa.
- 5. Ejecución del Programa: Como empezar a usar el programa.
- 6. Uso del Programa: Elementos para la operación correcta del programa.
- 7. Apéndices: Solución de problemas, datos técnicos, archivos del sistema, glosario.

1. Antes de Empezar

☞ *Describa las recomendaciones que el usuario deberá seguir antes de instalar por primera vez el programa*

El producto que que usted adquirió consta de [*número*] diskettes de [*tamaño*] pulgadas, [*número*] manuales, y [*mencionar otras cosas*].

Requerimientos del Sistema

☞ *Indicar los requerimientos mínimos de hardware y software que se necesitan para ejecutar correctamente el programa*

- Computadora [*marca*], [*modelo*]; o compatible.
- [*cantidad*] KBytes de memoria RAM.
- Monitor [*tipo*].
- [*cantidad*] MB de espacio en disco duro.
- [*sistema operativo*] versión [*número*] o posterior.
- [*otros periféricos*]

Respalde sus diskettes

Le recomendamos que haga una copia de respaldo de los diskettes que acompañan este producto, antes de iniciar el proceso de instalación. Guarde los originales en un lugar seguro. Si no sabe cómo copiar discos consulte la sección [*numero*] de este manual.

Información de último momento

Constantemente modificamos nuestros productos para ofrecerle mayores ventajas, por ello hay información que no alcanzó a imprimirse en este manual. Por favor, lea el archivo LEEME.TXT (puede utilizar cualquier editor de texto) que se encuentra en el primer diskette de instalación, y conozca los últimos datos disponibles sobre el sistema XXXXXX.

Convenciones usadas

Para facilitarle la interpretación de la información, este manual utiliza los siguientes formatos de texto:

convención	indicación
Negritas	nombres de comandos, modificadores y texto que debe escribirse exactamente como aparece
Cursiva	parámetros, términos nuevos y texto que debe escribirse de manera variable
MAYUSCULAS	nombres: directorios, archivos y abreviaturas
< >	teclas que deberán oprimirse

2. Instalación

- Indique el procedimiento para trasladar los programas en diskette al disco duro (o diskettes de trabajo) del usuario

Instalación en Disco Duro

1. Encienda su computadora.
2. Inserte el diskette de instalación en la unidad de discos A o B.

3. Cambie la unidad por default hacia A o B, según sea el caso.
4. En la línea de comandos escriba: **Instalar** y oprima <ENTER>.
5. Siga cuidadosamente las instrucciones que aparecerán en la pantalla. Si debe cancelar la instalación, oprima <ESC> en cualquier momento.
6. Al terminar, retire el diskette de instalación.

☞ *En caso de que existan otras posibilidades de instalación, diskettes, CD-ROM, o deba instalarse hardware, también deberá indicarse*

☞ *Las siguientes secciones deberán redactarse según las características propias de cada sistema*

---

3. Conociendo su computadora

☞ *Describa aquí los elementos previos que considera necesarios para que el usuario pueda usar su programa.*

Antes de desarrollar esta sección, piense a quién va dirigido el sistema. Por ejemplo, si el producto es un tutorial para niños, conviene que proporcione mucha información previa, tanto para el niño como para el padre.

Algunos temas que puede incluir aqui son:

Qué es y para qué sirve una computadora.
Concepto de hardware y software.
Partes de la computadora.
Cuidados de la computadora.
Necesidad del disco duro y los discos flexibles.
Cuidados de los discos flexibles.
Cómo encender la computadora.
Usos y uso del software.
Comandos básicos del sistema operativo.

☞ *Omita esta sección si el usuario será un programador o especialista en cómputo*

4. Características del Programa

☞ *Describa las características y ventajas propias del producto*

Descripción
Aplicaciones principales
Tipo de interface
Facilidades y ayudas
Soporte técnico

5. Ejecución del Programa

1. Encienda su computadora.
2. Cambiese al subdirectorio [*nombre*].
3. Teclee [nombre] y oprima <ENTER>.

Al iniciar el programa aparecerá en pantalla el menú principal y se encontrará listo para trabajar.

6. Uso del Programa

☞ *Describa con detalle cada uno de los siguientes puntos*

Area de trabajo y uso del teclado. *Zona de la pantalla (menús, barras, íconos, etcétera), y funciones de las teclas principales y/o el mouse.*

Menú pricipal: *mencionar cada una de las opciones que sean mostradas. Es muy importante mencionar cómo salvar el trabajo, cómo recuperar un trabajo salvado y cómo salir del programa.*

Funciones de las opciones del menú principal: *describir funcionamiento de una de las opciones disponibles.*

Submenús: *mencionar cada una de las opciones que sean mostradas.*

Funciones de las opciones de los submenús: *describir funcionamiento de una de las opciones disponibles.*

Presentación de resultados: *Cómo se obtien los resultados, qué formato tienen, etcétera.*

Obtención de ayuda: *Qué teclas oprimir para obtener ayuda, qué tipo de ayuda aparece, cómo quitar la ayuda, etcétera.*

7. Apéndices

☞ *Incluya información para los siguientes temas*

Solución de Problemas: *Cómo resolver los problemas más comunes a que se enfrentará el usuario: Discos llenos, Sistema operativo incompatible, Errores al introducir datos,Insuficiente memoria RAM, Demasiados archivos abiertos y todos aquellos errores propios del trabajo normal del sistema.*

Características Técnicas: *Algunos datos técnicos que pudieran ser de utilidad al usuario.*

Archivos que contienen los diskettes: *Esta lista permite verificar*

una eventual pérdida por discos originales dañados y solicitar una reposición.

Soporte técnico: *A dónde acudir en caso de problemas.*

Glosario: *incluya las palabras que no son del uso común del usuario y se incluyen en el manual.*

Indice

cuadro 5-12

Desarrollando el programa de instalación

El programa de instalación se encarga de trasladar los archivos de un sistema de los diskettes originales al disco duro (o diskettes de trabajo) del usuario.

Este programa puede ser tan sencillo, que sólo copie archivos de un lugar a otro, o tan complejo que permita personalizar la instalación. Puede estar desarrollado en modo texto o en un atractivo ambiente gráfico.

Esta es una lista de sugerencias para construir u programa de instalación básico.

1. Crear los directorios necesarios
2. Copiar únicamente los archivos necesarios (ejecutables, datos, imágenes) para ejecutar el programa.
3. Si hay una versión anterior borre los archivos antiguos.
4. Si se cancela la instalación borre los archivos copiados y directorios creados.
5. Si tiene que descomprimir, copiar los archivos empacados y el programa desempacador al disco duro para acelerar el proceso, pero deberán borrarse una vez completada la instalación.

Capítulo 6

❖*Depuración de programas*

6.1 ¿Es inevitable depurar?

El proceso de **prueba** es la fase en la cual se ejecuta el programa con algunos datos, especialmente seleccionados, para encontrar los errores que pueda haber

El proceso de **depuración** es la fase en la cual se hacen modificaciones al programa para eliminar los errores.

Hay dos tipos de depuración: difícil y profunda. Una **depuración difícil** significa que hay problemas con la lógica del programa, mientras que una **depuración profunda** sólo mejora el programa, lo libera de errores *sencillos* o busca código optimizable.

Además de buscar errores, la depuración permite detectar algunos detalles importantes para el mejoramiento del programa; por ejemplo:

1. Expresiones booleanas que requieren simplificación.
2. Segmentos de código que nunca se ejecutan,
3. Instrucciones en las que se ocupa el 90% del tiempo, que pueden ser optimizables, e inclusive
4. Necesidad de volver a empezar desde el principio.

Desafortunadamente, no existe ningun método que permita probar por completo, en un tiempo razonable, un programa de gran complejidad. Esto se debe a que mientras mayor cantidad de errores se detectan (y corrigen), más difícil resulta detectar los siguientes.

Ante la imposibilidad de hacer pruebas exhaustivas se recurre a una selección cuidadosa de **datos de prueba**. Si el programa funciona correctamente con esos datos, se puede tener mucha confianza sobre su futuro comportamiento.

El momento de dejar de buscar errores es cuando el esfuerzo de hacerlo es mayor que el costo de los fallos aún no detectados.

Para reducir las posibilidades de error se debe tener en cuenta que la mayoría de los errores ocurre en las etapas de especificación y diseño (por prestar poca atención a la naturaleza del problema), y se reducen siguiendo algún método estructurado.

También se generan errores en la traducción del diseño al código; y finalmente suele haber errores en la comprobación y depuración, la misma corrección de errores puede dar pauta a otros.

Cuadro 6-1

CUATRO IDEAS VALIOSAS SOBRE DEPURACION

1o. Si se apresura a programar, tendrá que depurar para siempre.

2o. Comenzar de nuevo suele ser más fácil que parchar un programa viejo.

3o. Si es preciso modificar más del 10% de un programa es mejor reescribirlo.

4o. Los parches tienden a introducir tantos errores como los que corrigen.

6.2 Algunas técnicas para comprobación de rutinas

I. Módulos vacíos

En la actualidad, gracias a las ventajas de los compiladores, no resulta prudente capturar la totalidad del código antes de probar si compila bien.

Para compilar correctamente un programa (que se integrará de varios módulos aún no hechos) puede usar programas vacíos (formados sólo por **begin** y **end**). Esto permite verificar que las declaraciones de tipos y variables son correctas sintácticamente. Posteriormente, conforme se vaya necesitando, agregue los detalles, pero sobre todo, no olvide *"llenarlos"*.

Depure y pruebe cada subprograma cuando esté terminado y no espere hasta que el proyecto haya sido codificado por completo.

II. Módulos manejadores

Para probar y depurar un subprograma (aislado) se puede escribir un pequeño programa auxiliar (manejador del subprograma) que proporcione la entrada necesaria, efectúe la llamada y evalúe el resultado.

El manejador permite aislar cada subprograma y estudiarlo por separado para detectar rápidamente errores.

El tiempo y esfuerzo requerido para depurar está en proporción inversa al cuidado con que fue diseñado el algoritmo y escrito el código.

III. Impresión de listados

No siempre conviene depurar sobre el monitor (los monitores de video cansan la vista, permiten una visión muy limitada y no se pueden transportar fácilmente).

Algunas veces conviene imprimir (cuando la depuración es profunda), lo que permite una visión diferente, marcar el código y facilita la transportación (tampoco conviene imprimir cuanto código escriba).

Cuando la depuración es difícil hay que regresar (o hacer) el algoritmo, hacer pruebas de escritorio e incluso utilizar diagramas para analizar la lógica del programa.

Ponga más cuidado en no equivocarse que en depurar.

IV. Marcar el camino

Cuando el ambiente de desarrollo carece de facilidades de depuración conviene colocar en algunas partes del código banderas o señales que le permitan saber cuando se ha pasado por esa sección, o que faciliten el monitoreo de las variables.

El siguiente ejemplo muestra un código en lenguaje Lisp. Para monitorear el comportamiento del código aparecen unas funciones **print1** (aparecen con el comentario **{depuración}**) que despliegan el contenido de la variable a su derecha. También se aplicó una *"salida de emergencia"* para lograr que el programa terminase. Observe que cada marca fue comentada.

```
(defun Hill (origen destino arbol)
  (setq q (list origen 0))        ; inicializar lista de nodos (cd millas)
  (print q)                        ; {depuracion}
  (loop ((null q))                 ; while q (no vacio)
    (setq expl (car q) )          ; expl <- first (q)
    (setq q (cdr q) )              ; q    <- tail  (q)
    (prin1 expl) (bl)              ; {depuracion}
    (prin1 q) (bl)                 ; {depuracion}

    ; esta es una salida de emergencia
    ; que deberá ser quitada en un
    ; próximo refinamiento
      ((equal expl 0))
    ; termina salida
```

```
    ((equal  expl destino))          ; meta encontrada => exito
    (setq l (expande expl arbol))    ; l <- expande (expl)
    (setq q (append
            (mayor (cdr l) (car l) destino)
             q))                     ; q <- mayor(l) + q
    (print q)                        ; {depuracion}
  )
)
```

V. Datos de prueba

Para probar un módulo conviene desarrollar un conjunto de datos de prueba, que sea representativo de una clase de datos que se comportarán de la misma manera. Para elegirlos se debe atender a las especificaciones del problema.

Como norma, determine los valores críticos y pruebe el módulo con el valor crítico, con un valor mayor y con uno menor.

La calidad de los datos de prueba es más importante que la cantidad.

Los resultados emitidos por módulos que efectúen cálculos se deben calcular a mano antes de la ejecución de prueba.

Es válido suponer que si una rutina opera correctamente con un dato de un tipo determinado, operará correctamente con los datos del mismo tipo dentro del rango válido (comprobación por clases de equivalencia).

Para evitar perder archivos, establezca una bandera que indique si ha sido actualizado, y que permita enviar un mensaje antes de salir del programa, en caso de que no se le haya salvado.

Los valores que se introducen en un rutina deben siempre comprobarse, especialmente aquellos que acepten datos de los usuarios, que manipulen archivos (los archivos pueden corromperse o leerse inadecuadamente), o que reciban datos desde los puertos de comunicaciones.

cuadro 6-2

♦ METODO DE CAJA NEGRA PARA VERIFICACION DE ERRORES:

Considere al módulo como una caja negra que recibe un dato y arroja un resultado. Usted debe evaluar si los resultados están acordes con las entradas proporcionadas. Este método no evalúa la lógica del programa.

Para utilizar este método elija valores:

a) verificables manualmente,
b) típicos y realistas,
c) valores extremos y
d) valores ilegales.

♦ METODO DE CAJA DE CRISTAL PARA VERIFICACION DE ERRORES:

Este método permite analizar la estructura lógica del programa, para ello utiliza los datos que activen cada alternativa del programa.

Para utilizar este método debe probar cada una de las opciones que ofrece su programa (y algunas que no ofrece).

Aplique este método a cada pequeño módulo conforme lo escriba y luego use esos mismos datos en secciones más amplias.

VI. Códigos de error

Puede utilizar el valor retorno de funciones como código de error y manejar la comunicación de datos mediante parámetros.

El lenguaje C maneja mucho este conpcepto, por ejemplo la instrucción

```
if  ((in=fopen("ARCH.DAT","r")) == NULL)
```

abre un archivo y lo asocia con el manejador **in**, pero en caso de que el archivo no exista, se obtiene un valor de **NULL**.

Se debe asegurar que todos los valores que pasen por interfaces (entre rutinas y entre usuario-máquina) sean del tipo correcto (validadción) y con los rangos adecuados.

VII. Pila de depuración

Para facilitar el rastreo de errores se puede utilizar una pila. El procedimiento es el siguiente: al iniciar, cada rutina inserta un identificador único en la pila y al salir lo extrae; así, de ocurrir un error, puede reconstruirse la ruta analizando la pila.

Conviene que la pila se encuentre en disco, usando un archivo sin buffer (para evitar pérdida de datos); de este modo si el sistema *se cae*, la pila no se pierde.

Ejemplo:

```
void fun1 (void)
    {
    PUSH_DEP (ID_FUN1);
    .....
    .....
    POP_DEP
    }

int main ()
    {
    PUSH_DEP (ID_MAIN);
    ......
    ......
    POP_DEP;
}
```

VIII. Validación

Siempre valide la asignación de memoria dinámica, las transacciones con archivos, y la coincidencia de tipos en los datos que pasan por interfaces.

cuadro 6-3

VERIFICACIONES ANTES DE ENTREGAR PROGRAMAS

Salvar antes de ejecutar.

Salvar antes de conectar periféricos.

Asegurarse que el programa ejecutable corresponde a la última versión del programa fuente.

Copiar el programa en dos diskettes (original y respaldo), incluyendo ejecutable, fuentes, documentación, ejemplos, rutinas de instalación y nombre del programador.

Efectuar una serie de pruebas finales, tanto en el programa fuente, como en el ejecutable, y especialmente sobre los diskettes.

Revisar la presencia de virus.

Durante el desarrollo hacer respaldos constantemente, pero en diskettes diferentes.

6.3 Errores comunes

Hay tres grandes grupos de errores que pueden ocurrir en un programa:

1. **Errores de sintaxis**: Se escriben mal palabras del lenguaje y por lo tanto no permiten que el programa se ejecute. Suelen ser fáciles de corregir, ya que el compilador suele marcar el error.

   Por ejemplo:

   ```
   If (A Mod 2) = 0 Then
         impar :=  FALSE;
   Else
         impar := TRUE;
   ```

2. **Errores de lógica**: Las instrucciones se escriben correctamente, pero utilizando un algoritmo erróneo, por lo que permiten que el programa se ejecute, pero entregan resultados erróneos o impredecibles. Su facilidad de corrección depende de cómo haya sido diseñado el programa .

   Por ejemplo:

   ```
   If  Edad < 18 Then Read (Voto);
   ```

   Un error de lógica, muy común y también muy fácil de detectar, es que los contadores o índices se pasen por uno o les falte uno.

3. **Errores de ejecución**: Se producen una vez que el programa está en funcionamiento y provocan que se interrumpa la ejecución. Esto suele deberse a que se rompe alguna regla del lenguaje o del sistema operativo (errores de tipos, demasiados archivos, etc).

   Por ejemplo:

   ```
   y = Sqrt (x);
   ```

   Esta instrucción (en Pascal) causará la *caída* del programa si en algún momento **x** toma un valor negativo.

cuadro 6-4

Lista de comprobación de errores[9]

VARIABLES

1. ¿Cada variable tiene un nombre diferente por módulo?, ¿El tamaño de los identificadores es importante para el compilador?
2. ¿Se ha modificado el valor de una variable cuando ésta aún era útil, por ejemplo, en ciclos o parámteros?
3. ¿Los subíndices de los arreglos son enteros y están dentro de los límites?
4. ¿Los arreglos empiezan en 1 o en 0?

CALCULOS

5. ¿Qué tipo de datos producen los cálculos y a qué variables se asignan?
6. ¿Los cálculos con reales se efectúan usando variables reales?
7. ¿El resultado de un cálculo es un número demasiado grande o demasiado pequeño para la computadora?
8. ¿Un divisor es tan pequeño que provoca una división por cero?
9. ¿Son significativos los errores por redondeo?
10. ¿Se utilizó correctamente la jerarquía de operadores? La jerarquía llega a variar según el lenguaje y la notación.
11. ¿Los resultados generados coinciden con las calculadoras?

COMPARACIONES

12. ¿Se comparan tipos simples con tipos simples y no con estructuras?
13. ¿Se hace distinción entre mayúsculas y minúsculas?
14. ¿Se comparan cadenas (*strings*) de igual longitud?
15. ¿Se utilizaron adecuadamente los operadores booleanos de los relacionales? (recuerde A>B OR C es diferente de A>B OR A>C?
16. Al comparar reales tener en cuenta los errores de redondeo.

[9]Basada en una idea de G.J. Myers (The Art of Software Testing), publicada en *Enciclopedia Práctica de Inforática*, Vol. 3. Ediciones Algar. España, 1986.

CONTROL

17. ¿Los ciclos y los algoritmos terminan sin importar el caso?

18. ¿Los ciclos y los algoritmos tienen un solo punto de entrada y un solo punto de salida?

19. Si falla un **IF-THEN** el control pasa a la sentencia **ELSE** o a la siguiente instrucción.

20. ¿Qué pasa si no se satisface ninguna condición de un grupo de **IF**'s anidados?

21. ¿Hay errores de stack debidos a un exceso de llamadas recursivas o de variables locales?

LIBRERIAS

22o. ¿Están actualizadas las librerías?

23o. ¿El compilador está usando las librerías correctas?

24o. ¿Hay identificadores (variables globales o procedimientos) iguales entre librerías diferentes?

ARCHIVOS

25. ¿Los archivos de configuracióm son correctos?

26. ¿Hay suficiente espacio en disco?

27. ¿El disco no está dañado físicamente?

28. ¿El archivo existe al abrirse?

29. ¿El archivo es exclusivo del sistema?

30. ¿Se vació el buffer de transferencia de archivos, antes de salir del programa?

APUNTADORES

31. ¿Hay memoria suficiente para cada nuevo apuntador?

32. ¿Se reservó la memoria necesaria antes de utilizar el apuntador para guardar datos?

33. ¿Están aterrizados todos los apuntadores no usados?

34. ¿Está efectuando transferencias con apuntadores aterrizados?

35. ¿Está apuntando a áreas de memoria seguras, sin corromper al sistema?

36. ¿Se está liberando la memoria reservada?

37. ¿La lista ligada tiene un nodo raiz?

6.4 Manejo de errores

Los errores durante la ejecución del programa son inevitables, por lo que un manejo *elegante* de ellos es importante en un buen programa. Aquí hay cuatro sugerencias para manejar los errores cuando ocurran.

1. Evitar que un error "*truene*" un programa.
2. Los errores deben generar una clave.
3. Un módulo debe tomar la clave y enviar un mensaje de error
4. Si el error debe finalizar la ejecución del programa se deben cerrar los archivos, regresar al modo de video original y devolver el control al Sistema Operativo.

En este ejemplo mostramos un módulo manejador de errores y un par de módulos que hacen uso de él. Observe que el manejador de errores es el que toma la decisión de abortar el programa.

```
{  Módulo Manejador de Errores  }

Procedure Error (code : Byte);
    {Manejador de errores}
    Begin
        Case code Of
        1: WriteLn ('ERROR: SE HAN SOBREPASADO LOS REGISTROS');
        2: WriteLn ('ERROR: SE HA SOBREPASADO LA MEMORIA DE PROGRAMA');
        3: WriteLn ('ERROR: Instrucción Inválida');
        4: WriteLn ('ERROR: Parámetros insuficientes');
        5: WriteLn ('ERROR: Registros fuera de rango');
        End;

        If Code  < 3 Then Halt(code+300);  {Evitar colisión con los errores estándar de DOS}
    End;


{ Otros módulos del programa }

Procedure Sumar (NumReg : Byte);
    Begin
        If Registros[NumReg] = 255 Then Error(5);
        Inc (Registros[NumReg]);
    End;
```

```
Procedure Ejecutar (Codigo : Byte);
    { Determina qué operación deberá efectuar la URM }
    { pasa los parámetros necesarios.                 }
    { En caso de error termina el programa.           }

    Begin
        Case Codigo Of
{Error}     0 : Error(3);
          255 : Error(4);

{funcionamiento normal}
           1 : Cero   (Parametros[1]);
           2 : Sumar  (Parametros[1]);
           3 : Copiar (Parametros[1],Parametros[2]);
           4 : Saltar (Parametros[1],Parametros[2],Parametros[3]);
         End;
```

cuadro 6-5

SUGERENCIAS DE PROGRAMACION CON LISTAS LIGADAS[10]

1. Dibuje los diagramas de "antes" y "después", del comportamiento de la lista, mostrando los principales apuntadores y la manera como deberían comportarse.

2. Para utilizar nodos nuevos (registros que incluyen un apuntador):
   1) Cree el nuevo nodo.
   2) Introduzca los valores en los campos de datos respectivos.
   3) Aterrice el (los) campo(s) liga del (los) nodo (s).
   4) Enlaec el nuevo nodo con el resto de la lista.

3. No deje apuntadores sin definir (aterrizar o apuntar hacia un lugar seguro).

4. Verifique que su algoritmo funciona para un lista vacía y para una lista con sólo un nodo.

5. Trate de referenciar sólo un nodo a la vez (es decir, evite las construcciones del tipo **p^.liga^.liga^.dato**). Las referencias repetidas indican que el algoritmo puede mejorarse.

6. Evite que dos apuntadores señalen a un mismo nodo, salvo que uno de los apuntadores sea para recorrer una lista.

[10]Adaptada de Kruse, Robert L. *Estrutura de Datos y Diseño de Programas*. Prentice Hall, México, 1988.

cuadro 6-6

Un catálogo de errores inesperados

A continuación presentamos una colección de programas y segmentos de código que algorítmicamente parecen estar bien, y sin embargo, al momento de compilarse o ejecutarse, presentan algún error. Estúdielos con cuidado.

Estos son errores con los que se ha topado el autor, puede ocurrir que bajo circunstancias diferentes, no se presenten. Hay que tener en mente que algunos de estos errores se deben a las directivas de compilación, y algunos se corrigen con un coprocesador matemático o la librería de emulación correspondiente.

```
Program Errores_Deliberados;

Var
    i1,i2,i3 : Integer;
    r1,r2,r3: Real;
            a : Array [1..10] of Char;
             f :  File;

Begin
    i1 := 32767;              límite de los enteros de dos bytes
    i2 := 1;
    r1 := sqrt(i1+i2);        la evaluación se hace entera, dando un número negativo y
                              "tronando" el programa.
    i1 := 6;
    i2 := 6;
    a[i1+1] := '5';
    a[i1+i2] := '7';           el subíndice queda fuera de rango


    Assign (f,'AUX.DAT');
    Rewrite (f,1);
    BlockWrite (f,a,10);      el sistema reportará un error de dispositivo no preparado
                              (o similar). Esto se debe a que AUX es un archivo predefinido
                              por el DOS e ignora la asignación anterior.
    Close (f);


    r1 := 0.7853981634;         r1» 45° (en radianes)
    r2 := Sin (r1);             r2 = 0.70710678119
    r3 := Cos(r1);              r3 = 0.70710678118
    r1 := r2-r3;                r1 debería ser cero, ya que Sen(45°)-Cos(45°) = 0
    If r1 = 0 Then
        WriteLn ('Hola 4);      la condición nunca se cumple ya que en lugar de cero
                                 r1 = 4.5474735089E-12
End.
```

```
/* SERIE DE FIBONACCI */
#include <stdio.h>
#include <conio.h>

main()
{
unsigned int a,b,c,num;

clrscr();
a = b = 1;
printf ("\n\n%u %u ",a,b);
for (num=1;num<=100;num++){
    c = a + b;
    printf ("%u ",c);
    a = b;
    b = c;
    }
}
```

☞ *Al ejecutar este programa se observa que los primeros resultados son correctos, pero tras algunas iteraciones empiezan a salir mal. Esto se debe a que se sobrepasa el límite superior de los números enteros , y en lugar de producirse un error, la variable empieza a tomar valores a partir del límite inferior de los enteros (esta es una característica de los números representados mediante complemento a dos).*

```
Const
  MAX = 15;

Var
  i : Integer;
Input : Array [1..MAX] Of Real;
  dx,x : Real;

Begin
  dx := 2*PI / MAX;
  x  := 0.0;
  i  := 1;

  While x <= 2*PI Do
     Begin
     Input[i] := Sin (x);
     x := x + dx;
     Inc (i);
     End;
End.
```

☞ *Tal como está este programa se cicla infinitamente o provoca un error de **fuera de rango**. Esto se debe a que para completarse el rango (0 <= x <=2*Pi) se debe iterar 16 veces (no 15). Si no se detecta que el índice se salió de rango, entonces ese último ciclo provoca que se corrompan las variables **dx** y **x** perdiéndose la condición de parada.*

☞ *Estos dos programas son algebráicamente idénticos, sin embargo no lo son computacionalmente. Las operaciones con números reales pueden conducir a un error de fuera de rango (se detecta un error), mientras que las operaciones con enteros pueden emitir resultados erróneos al sobrepasarse sus límites (no se detecta el error).*

```
Var
    Sx, Sy,
    X,Y,Z : Integer;

Begin
   Sx := D * X / Z;
   Sy := D * Y / Z;
End;
```

*La operación **D** * **Z** se efectúa como multiplicación de enteros y con el resultado se hace la división real.*

```
Var
    Sx, Sy,
    X,Y,Z : Integer;

Begin
   Sx := D * (X / Z);
   Sy := D * (Y / Z);
End;
```

*La operación **X** / **Z** se efectúa como división de reales y con el resultado se hace la multiplicación real.*

☞ *Este programa debería fallar, porque **sin(0)+Cos(0) -1** = 0 y produce una indeterminación. Sin embargo, los errores de redondeo provocan que no sea cero exacto y el programa no falla (el error sí se produce con coprocesador matemático).*

```
Var
    i,r : real;
Begin
  i := 0.0;
  While i <= 2*Pi Do
     Begin
     writeln (sin (i));
     writeln (cos(i));
     r := 1 / (sin (i) + (Cos (i)-1));
     WriteLn (r);
     i := i + 0.5;
     End;
End.
```

☞ *Este programa debería fallar, porque se y produce una indeterminación. Sin embargo, los errores de redondeo provocan que el programa no falle.*

```
Program PuntoFlotante;
Var
h,x,y,e,f,q : Real;
Begin
  H := 1/2;                    1/2 = 0.5
  X := 2/3 - H;                2/3 - 1/2 = 1/6 = 0.166666...
  y := 3/5 - H;                3/5 -1/2  = 1/10 = 0.1
  E := (x+x+x) - H;            3/6 - 1/2 = 0
  F := (y+y+y+y+y) - H;        5/10 - 1/2 = 0
  Q := E/F ;                   aquí debería producirse un error.
End.
```

Capítulo 7

❖ *Optimización de programas*

Optimizar programas consiste en minimizar el tiempo de ejecución o la cantidad de memoria utilizada.

No se puede optimizar en todo.

Un **programa eficiente** es el que minimiza el tiempo de ejecución. Estos son especialmente importantes en procesos que requieren gran cantidad de repeticiones. Sin embargo, si un algorimo requiere demasiada optimización, puede ser indicador de que se está usando la máquina equivocada.

Desafortunadamente, optimizar algoritmos aumenta su dificultad y dificulta su comprensión.

7.1 Sugerencias de optimización

1. Los accesos a memoria secundaria son lentos. Conviene reducir la cantidad de accesos leyendo grandes bloques de datos cada vez y almacenándolos en RAM.

El cuadro 7-1 muestra un ejemplo de cómo un programa de acceso a disco puede ser mejorado.

cuadro 7-1

```
Program Copy1;  { Versión ineficiente }
Var
fOld,
fNew  : File;
   by : Byte;

Begin
{ abrir archivo original }
Assign (fOld,'ARCHIVO1.DAT');
Reset (fOld,1);

{ crear archivo destino }
Assign (fNew,'ARCHIVO1.DAT');
Reset (fNew,1);

   { copiar byte a byte }
   While Not Eof (fOld) Do
      Begin
      BlockRead (fOld,by,1);
      BlockWrite (fNew,by,1);
      End

   { cerrar archivos }
   Close (fOld);
   Close (fNew);
End.

{------------------------------------------------------------------}
```

```
Program Copy2;  { Versión mejorada }
Const
MAX_BUFF = 10000;

Var
fOld,
fNew   : File;
  buffer : Array[1..MAX_BUFF] of byte;

Begin
{ abrir archivo original }
Assign (fOld,'ARCHIVO1.DAT');
Reset (fOld,1);

{ crear archivo destino }
Assign (fNew,'ARCHIVO1.DAT');
Reset (fNew,1);

  { copiar bloques de datos }
  While Not Eof (fOld) Do
     Begin
     BlockRead (fOld,buffer,MAX_BUFF);
     BlockWrite (fNew,buffer,MAX_BUFF);
     End

  { cerrar archivos }
  Close (fOld);
  Close (fNew);
End.
```

Los programas que manejan muchos datos mediante archivos, como los manejadores de bases de datos, deben efectuar las actualizaciones físicas al final de la sesión; por ejemplo, no conviene borrar cada vez que se ordena eliminar un registro, sino marcarlo y al terminar de usar el programa entonces borrar físicamente todos los registros borrados.

Para evitar que el usuario se aburra, conviene que los archivos se lean y escriban mientras el usuario está ocupado haciendo alguna otra cosa (por ejemplo, leer instrucciones).

---

2. La aritmética de enteros es mucho más rápida que la de punto flotante.

    1) Reemplace lo más posible reales por enteros (ahorrará memoria y tiempo).
    2) En algunos casos es posible multiplicar por un factor, convertir a entero, efectuar operaciones enteras y luego dividir entre el factor.
    3) Evite colocar contadores reales en los ciclos. Si necesita incrementos fraccionarios, mejor calcule el número (entero) de iteraciones antes de entrar al ciclo.

3. Para las computadoras, hacer multiplicaciones y divisiones ocupa muchísimo más tiempo que hacer sumas y restas. Minimice el número de multiplicaciones reagrupando términos.

Por ejemplo la expresión

$$A = ac + ad + bc + bd$$

requiere de tres sumas y cuatro multiplicaciones. La misma igualdad puede reexpresarse factorizándola:

$$A = (a+b)\ (c+d)$$

Esta última expresión sólo ocupa dos sumas y una multiplicación. La mejora no resulta importante si el programa ejecutará una sola vez esta línea; pero si por el contrario ha de repetirla miles (incluso millones) de veces, la mejora es realmente apreciable.

Otro ejemplo es la siguiente igualdad:

$$m = \frac{1 + \frac{x}{y}}{1 - \frac{x}{y}}$$

Esta expresión requiere de tres divisiones y dos sumas (la resta es una suma con signo negativo). Observe cómo podemos reacomodar los términos para optimizar las operaciones:

$$\frac{1+\frac{x}{y}}{1-\frac{x}{y}} = \frac{\frac{y+x}{y}}{\frac{y-x}{y}} = \frac{y+x}{y-x}$$

Por lo tanto, la última expresión sólo contiene dos sumas y una división:

$$m = \frac{y + x}{y - x}$$

Un último ejemplo, calcular la *n-ésima* potencia de un número x tiene la siguiente expresión:

$$x^n = x \cdot x \cdot x \ldots x \cdot x \text{ (n veces)}$$

Esta forma directa requiere de ***n-1*** multiplicaciones. Para disminuir el número de multiplicaciones podemos calcular el cuadrado del número (una sola vez) y entonces efectuar sólo la mitad de las multiplicaciones.

Si ***n*** es par

$$P = x^n = \begin{cases} y = x^2 \\ P = y \cdot y \ldots y \cdot y \quad (\frac{n}{2} \text{ veces}) \end{cases}$$

Si ***n*** es impar

$$P = x^n = \begin{cases} y = x^2 \\ Q = y \cdot y \ldots y \cdot y \quad (\frac{n-1}{2} \text{ veces}) \\ P = x \cdot Q \end{cases}$$

---

4. Si domina el lenguaje ensamblador, le convendrá escribir rutinas en código máquina, y enlazarlas con los programas en lenguajes de alto nivel.

   Las rutinas escritas en ensamblador se ejecutarán mucho más rápido que las escritas en lenguajes de alto nivel.

---

5. Evite calcular un mismo valor una y otra vez en los ciclos, calcule afuera e incorpórelo como una variable. Evite también llamadar repetidamente funciones que regresen el mismo valor en cada iteración (véase el ejemplo del cuadro 7-2).

cuadro 7-2

```
Procedure PintarEstrellas;
{ Colocar aleatoriamente puntos en la pantalla,
 para simular estrellas }

Const
   TOTAL_ESTRELLAS = 10000;
Var
   i,X,Y,col : Integer;

Begin
  Randomize;

  { Limpiar pantalla }
  ClearDevice;
  For i := 1 to TOTAL_ESTRELLAS Do
    Begin
     { Calcular coordenadas }
```

```
        X := Random (GetMaxX);      Durante todo el ciclo se hace una llamada a las función
    GetMaxX
        Y := Random (GetMaxY);      Durante todo el ciclo se hace una llamada a las función
    GetMaxY


        { Calcular color }
        col := Random (GetMaxColor);      Durante todo el ciclo se hace una llamada a las función
                                          GetMaxColor

        { Poner punto según coordenadas y color }
        PutPixel (X,Y,col);
        End;
    End.

{-------------------------------------------------------------------------}
    Procedure PintarEstrellas;
    { Colocar aleatoriamente puntos en la pantalla,
     para simular estrellas }

    Const
       TOTAL_ESTRELLAS = 10000;

    Var
      i,X,Y,col : Integer;
      MaxX,MaxY,
      MaxCol    : Integer;

    Begin
       Randomize;

      { Calcular valores máximos }
      MaxX := GetMaxX;            Sólo se hace una llamada a cada función
      MaxY := GetMaxY;
      MaxCol := GetMaxColor;

      { Limpiar pantalla }
      ClearDevice;
      For i := 1 to TOTAL_ESTRELLAS Do
         Begin
         { Calcular coordenadas }
         X := Random (MaxX);
         Y := Random (MaxY);
         { Calcular color }
         col := Random (MaxCol);

         { Poner punto según coordenadas y color }
         PutPixel (X,Y,col);
         End;
    End.
```

☞ *La rutina putpixel consume mucho tiempo, por ello es mejor calcular y guardar los valores y despues sólo graficarlos. Esto se muestra en el siguiente programa.*

```
{-------------------------------------------------------------------------------------------------------------}

   Procedure PintarEstrellas;
    { Colocar aleatoriamente puntos en la pantalla,
     para simular estrellas }
```

```
Const
   TOTAL_ESTRELLAS = 10000;



Type
   Punto = Record
        X,Y,Col : Integer;
   End;

Var
   i,X,Y,col : Integer;
  MaxX,MaxY,
  MaxCol    : Integer;

  Estrellas : Array [1..TOTAL_ESTRELLAS] of Punto;

Begin
   Randomize;

  { Calcular valores máximos }
  MaxX := GetMaxX;
  MaxY := GetMaxY;
  MaxCol := GetMaxColor;

  { Limpiar pantalla }
  ClearDevice;
   For i := 1 to TOTAL_ESTRELLAS Do
      Begin
      { Calcular coordenadas }
      X := Random (MaxX);
      Y := Random (MaxY);
      { Calcular color }
      col := Random (MaxCol);

      { Almacenar en el arreglo las coordenadas y color }
      Estrellas[i].X   := X;
      Estrellas[i].Y   := Y;
      Estrellas[i].Col := Col;
      End;

  { Colocar estrellas }
   For i := 1 to TOTAL_ESTRELLAS Do
   With Estrellas[i] Do
      PutPixel (X,Y,col);
End.
```

6. Si es posible evite el uso de listas ligadas porque desperdician espacio. Si tiene experiencia con memoria dinámica, puede desarrollar un grupo de rutinas para dimensionar dinámicamente grandes bloques de bytes. El cuadro 5-8 muestra una librería para manejar arreglos dinámicos, sin recurrir a listas ligadas.

7. Algunos sistemas para desarrollo de programas poseen directivas de compilación que facilitan la detección de errores, sin embargo pueden reducir la cantidad de memoria disponible o la velocidad de ejecución. Usted puede deshabilitar esas facilidades pero aumenta su responsabilidad sobre el código.

---

8. Use constantes en lugar de variables, la constante se sustituye al momento de compilarse. Esto evita una acceso a memoria en busca del contenido de una variable.

---

9. Si tiene problemas de memoria (y el programa no es de procesamiento numérico) utilice la mínima precisión posible.

---

10. No permita que nadie le diga que un código claro o bien hecho no es eficiente, solamente que le sea probado y le sea medido.

Cuando la elección en un programa se debe hacer entre la claridad y eficiencia, generalmente se elegirá a la clardidad o la legibilidad del programa.

7.2 Uso de memoria dinámica

Conceptos

El empleo de memoria dinámica no hace que una aplicación sea más rápida que su implementación con variables estáticas.

Hay una tendencia a creer que el uso de apuntadores revoluciona y mejora sus programas. Así mismo, suele confundirse memoria dinámica con apuntadores.

El apuntador en sí es una variable estática (es una localidad de memoria de cuatro bytes, que guarda el número que le corresponde a otra localidad de memoria), que es indispensable para el manejo de memoria dinámica, pero que puede usarse en memoria estática.

Algunos programadores llegan a creer que nunca deberían usar el indexamiento de arreglos porque los punteros son más eficaces.

Si se requiere acceder el arreglo en estricto orden ascendente o descendente, entonces los apuntadores son más fáciles de usar y más rápidos.

Si se requiere acceder al arreglo aleatoriamente entonces es mejor la indexación (será tan rápido como evaluar una compleja expresión de puntero) y es más fácil de entender.

El uso de apuntadores sólo es benéfico cuando se utilizan para recorrer una estructutra de datos de principio a fin, en orden (secuencialmente).

Los apuntadores también son útiles como manejadores (handles) de grupos de bytes o para pasar parámetros sin tipo o de diferentes tipos.

Desventajas:

1. La liberación de bloques de memoria entre bloques aún no liberados origina fragmentación.Esto provocará que aunque exista memoria suficiente (dispersa) no se puede otorgar, porque el bloque más largo (contiguo) no satisface la demanda.

2. Es común olvidar liberar bloques, o liberarlos varias veces.

3. Es común emplear memoria no obtenida.

4. Es común rebasar los límites de la memoria asignada.

La asignación dinámica de memoria es útil cuando no se sabe por adelantado con cuántos datos se van a procesar.

Apéndice

❖ *Casos prácticos de planeación de programas*

A.1 Presentación de los casos

Un programa es una herramienta que permite desarrollar o simplificar cierto trabajo; los máximos beneficios del programa se obtienen cuando éste ha sido especialmente diseñado para el problema específico.

Muchos trabajos requieren programas complejos o grandes sistemas (como es el caso de un vuelo espacial o de las transacciones de una red bancaria). Sin embargo, también hay una gran cantidad de trabajos y problemas que no requieren de sistemas tan complejos y basta con un programa, bien planeado y diseñado, hecho con el lenguaje que domine el programador.

En esta situación suelen estar los negocios pequeños, los proyectos escolares, las necesidades familiares y las propias, en que la inversión requerida (en tiempo, esfuerzo y dinero) para elaborar un sistema sofisticado rebasa las posibilidades reales. Para estos casos, un programa que satisfaga las necesidades del usuario, cumpliendo un mínimo de criterios de calidad, será una herramienta muy valiosa, aunque sea modesta. Para lograr tales programas es necesario una cuidadosa planeación antes de su desarrollo.

Para ilustrar mejor estas ideas, en esta sección presentamos dos casos:

- El diseño de un programa para controlar un inventario bibliotecario.
- El análisis para elaborar un programa que administre un registro civil.

En el primer caso vamos directo al diseño puesto que los procedimientos involucrados son comunes en la programación y basta con pensar cómo acomodarlos adecuadamente. Por otra parte, el segundo caso es más complejo en cuanto al movimiento de la información y de los procesos requeridos, por ello resulta necesario hacer un cuidadoso análisis de la situación que luego pueda traducirse fácilmente en el diseño de solución.

En ningún caso se muestra el código fuente, así no se resta generalidad al ejemplo. Tampoco se ha planeado hasta el último detalle ya que eso tiende a complicar la implementación; por el contrario, una planeación general orienta la programación y le deja suficiente holgura al programador para elaborar los detalles.

A.2 Programa para inventario bibliotecario

Definición del problema

En cierta escuela secundaria se requiere de un programa para controlar el inventario de su biblioteca.

Básicamente, el programa debe sustituir al tradicional fichero de la biblioteca y tener facilidades para registrar nuevas adquisiciones, cambiar datos cuando se requiera (en caso de haber errores de captura o por pérdida de libros) y buscar fichas bibliográficas.

Para aumentar su utilidad el programa también debe generar la clave de clasificación por cada libro nuevo y permitir la consulta de fichas ya sea por autor, por título o por clave.

Análisis del problema

Este programa es conceptualmente muy sencillo y; para su elaboración se requiere a realizar las operaciones básicas del manejo de archivos: altas, bajas, cambios y consultas.

Se requiere un módulo para cada tipo de operación cuya activación será hecha a través de un menú que presentará al usuario las operaciones posibles.

La información a manejar proviene de las fichas bibliográficas, las cuales, una vez capturadas, se guardarán en un mismo archivo. El tipo de consultas requeridas (por autor, por título o por clave) conduce naturalmente a la necesidad de utilizar índices que faciliten la búsqueda de datos.

La clave de clasificación es única, por tanto servirá para elaborar el índice primario, mientras que los autores y los títulos se utilizarán como índices secundarios.

Diseño general del programa

Dependiendo de las facilidades provistas por el ambiente de desarrollo (manejadores de bases de datos, lenguajes para la administración o de propósito general), habrá que desarrollar cierto trabajo previo: definir la estructura de los archivos, construir las rutinas de entrada o salida de datos, y preparar la rutina para generar la clave. Si se conoce poco del ambiente de desarrollo entonces es muy conveniente considerar que sus facilidades son mínimas y empezar el programa construyendo las rutinas de más bajo nivel (generamente las de entrada y salida de datos).

- **Estructura de los archivos**

Como el problema es manejar archivos empezamos por definir (manualmente) la estructura que tendrán los registros. En este proyecto utilizamos cuatro archivos: el principal (que guardará todas las fichas bibliográficas), el de claves, el de autores y el de títulos.

La estructura del archivo principal es similar al formato de una ficha bibliográfica; por comodidad llamaremos LIBRO a esta estructura:

Campo	Tipo	Longitud
Clasificación	Cadena	10
Título	Cadena	50
Autor	Cadena	50
Editorial	Cadena	25
Colección	Cadena	15
Lugar y País	Cadena	15
Año	Número	4
Páginas	Número	4
Tipo de encuadernación	Cadena	10
Ejemplares	Número	2
Bandera de borrado	Booleano	no visible

El índice principal consta de dos campos, uno para la clasificación y otro para el número del registro (en el archivo principal) que corresponde a esa clasificación; por comodidad llamaremos CLAVE a esta estructura:.

Campo	Tipo	Longitud
Clasificación	Cadena	10
Número de registro	Número	5

Los índices secundarios también constan de dos campos, uno para el autor (o el título) y otro para la clasificación . De este modo al buscar un título se obtendrá la clasificación del libro y luego deberá buscarse en el índice primario hasta encontrar el número de registro de la ficha bibliográfica correspondiente (fig. A-1). Por comodidad llamaremos AUTOR y TITULO a estas estructuras:

Campo	Tipo	Longitud
Autor	Cadena	50
Clasificación	Cadena	10

Campo	Tipo	Longitud
Título	Cadena	50
Clasificación	Cadena	10

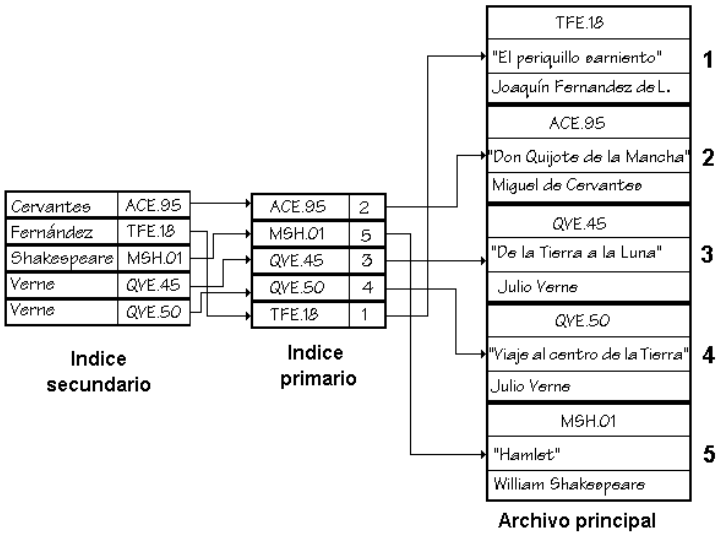


Figura A-1. Ejemplo de un archivo de libros con un índice primario (clave) y un índice secundario (autor).

Al construir el programa debe recordarse que todas las rutinas que manejan archivos siguen este patrón de funcionamiento: (*i*) abrir, (*ii*) procesar y (*iii*) cerrar. También es muy importante verificar la existencia del archivo antes de abrirlo.

• Rutinas de entrada y salida

Estas rutinas son la interfaz con el usuario y deben estar hechas de tal modo que los datos mostrados resulten fáciles de entender y manipular por el usuario.

Conviene que la primera versión del programa utilice rutinas muy sencillas (para no desviar la atención del problema principal) pero suficientemente robustas para tratar los datos adecuadamente. De ser necesario después se emigrará hacia una interfaz más compleja.

Para la entrada o salida de información sobre libros se hicieron las rutinas *Captura/Edición* y *Mostrar*.

La rutina *Captura/Edición* recibe como parámetro una variable con estructura LIBRO, muestra su contenido en pantalla, permite su edición y devuelve la misma variable ya modificada. Cuando la acción se limita a la captura de datos, el parámetro de entrada es previamente inicializado con blancos. Los datos recibidos son convertidos a mayúsculas (capitalizados) para evitar errores durante las búsquedas.

Por otra parte, la rutina *Mostrar* recibe cómo parámetro una variable de tipo LIBRO y envia su contenido a la pantalla.

El último preliminar, la rutina *Generar Clave*, recibe cómo parámetro una variable de tipo LIBRO y produce una clave de clasificación, acorde a esos datos, que se agrega al parámetro de entrada. La clave se capitaliza y también se valida su unicidad.

Sobre estas rutinas se construye el resto del programa. La figura A-2 muestra el diagrama jerárquico de los principales módulos que componen el programa.

Diagrama de los módulos

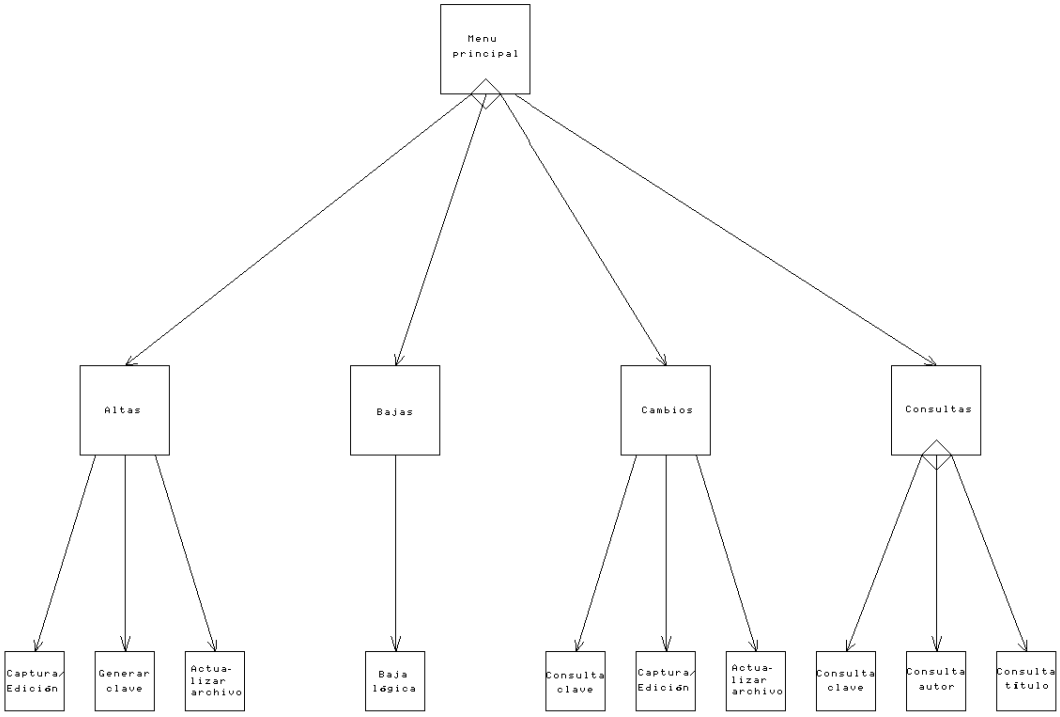


Figura A-2. Diagrama jerárquico de los módulos que componen el programa de inventario bibliotecario.

- **Menú principal**

El menú principal sirve para que el usuario elija alguna de las principales funciones del programa. En este caso las opciones son: (1) altas, (2) bajas, (3) cambios, (4) consultas, y (5) terminar.

Por cada opción muestrada en un menú debe construirse un módulo; en este caso se elaboraron cuatro grandes módulos: *altas*, que registra un nuevo título en los archivos; *consultas*, que recupera la ficha bibliográfica de un libro a partir de algún dato; *cambios*, que sirve para modificar alguno de los datos previamente guardados y el módulo *bajas*, que se encarga de eliminar un registro no deseado.

• Módulo de altas

Algoritmo de cada módulo

Este módulo registra las fichas bibliográficas nuevas. Para ello solicita al operador los datos de indentificación del libro y les agrega automáticamente la clave de clasificación. Enseguida guarda los datos en el archivo principal y actualiza los índices.

El algoritmo seguido por este módulo es:

1º Leer ficha bibliográfica y guardar sus datos en LIBRO (utilizando la rutina *Captura/Edición*).
2º Agregar CLAVE de clasificación a LIBRO (utilizando la rutina *Generar Clave*).
3º Actualizar archivos con la información contenida en LIBRO.

• Módulo de consultas

Este módulo recupera una ficha bibliográfica a partir de alguno de sus datos: la clave, el autor o el título. Como se tienen varias posibilidades su funcionamiento se controla con un menú cuyas opciones son: (1) clave, (2) autor, y (3) título.

Este módulo utiliza dos diferentes estrategias de búsqueda (fig. A-3), una para el índice primario y otra para los índices secundarios (autor y título).

La consulta primaria consiste en recibir la clave de clasificación y buscarla en el índice primario para encontrar el número de registro correspondiente. Enseguida se recupera el registro del archivo principal colocado en la posición recién indicada y se muestran los datos al usuario.

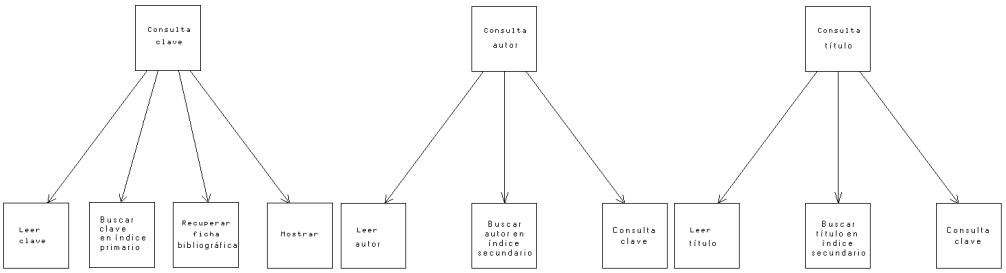


Figura A-3. Detalle de la composición de los módulos de consulta.

El algoritmo seguido por la consulta primaria es:

1º Leer CLAVE de clasificación.
2º Buscar CLAVE en el índice primario y devolver el NÚMERO DE REGISTRO asociado.
3º Leer el registro (del archivo principal) ubicado en la posición indicada por NÚMERO DE REGISTRO y colocar sus datos en LIBRO .
4º Desplegar los datos guardados en LIBRO (utilizando la rutina *Mostrar*).

El algoritmo seguido por las dos consultas secundarias (autor o título) es:

1º Leer LLAVE (autor o título).
2º Buscar LLAVE en el índice secundario correspondiente y devolver la CLAVE de clasificación asociada.
3º Utilizar CLAVE para recuperar el registro haciendo una consulta primaria.

Estos dos últimos pasos deben efectuarse varias veces por que una misma llave secundaria pueden tener varias claves asociadas (por ejemplo, cuando un mismo autor ha publicado varios libros).

- **Módulo de cambios**

Este módulo permite modificar alguno de los datos guardados. Para ello solicita la clave del libro y localiza la ficha correspondiente, mostrándola y permitiendo su edición.

El algoritmo seguido por este módulo es:

1º Leer CLAVE del registro a modificar.
2º Efectuar consulta primaria, colocar los datos en LIBRO y guardar la posición del registro.
3º Mostrar al usuario los datos de LIBRO y permitirle su edición (utilizando la rutina *Captura/Edición*).
4º Guardar LIBRO en el archivo principal en la misma posición que tenía antes de la edición.

- **Módulo de bajas**

Este módulo se encarga de eliminar un registro no deseado. El operador sólo tiene acceso al borrado lógico con lo cual podría recuperarlo en cualquier momento durante la sesión de trabajo. El programa automáticamente deberá hacer las bajas físicas al terminar la sesión.

El borrado lógico consiste en recibir la clave de clasificación y hacer una consulta primaria para recuperar el LIBRO correspondiente. Enseguida se activa la bandera de borrado de LIBRO y se le vuelve a guardar en el archivo principal.

El algoritmo seguido por el borrado lógico es:

1º Leer CLAVE del registro a borrar.
2º Efectuar consulta primaria, colocar los datos en LIBRO y guardar la posición del registro.
3º Activar bandera de borrado de LIBRO.
4º Guardar LIBRO en el archivo principal en la misma posición que tenía.
5º Borrar el NÚMERO DE REGISTRO asociado a CLAVE en el indice primario.

Para recuperar el registro basta con desactivar la bandera de borrado y restituir el índice afectado.

El borrado físico consiste eliminar del archivo principal todos los registros que hayan sido borrados lógicamente.

El algoritmo seguido por el borrado físico es:

1º Copiar cada registro del archivo principal hacia un archivo temporal, excepto si tiene activada la bandera de borrado.
2º Borrar archivo principal.
3º Renombrar el archivo temporal con el nombre que tenía el principal.
4º Reconstruir los índices.

Para reconstruir los índices basta borrar los archivos de éstos y leer cada uno de los registros del archivo principar para obtener los datos de clave, autor y título y construir los índices de manera similar a la captura desde teclado.

- **Módulo de actualización**

Este módulo se encarga introducir una nueva ficha bibliográfica en el archivo principal y en los índices.

Su algoritmo es el siguiente (fig. A-4):

1º Obtener LIBRO.
2º Agregar LIBRO al final del archivo principal.
3º Obtener CLAVE del registro LIBRO, agregar al índice primario y reordenarlo.
4º Obtener TÍTULO del registro LIBRO, agregar al índice de títulos y reordenarlo.
5º Obtener AUTOR del registro LIBRO, agregar al índice de autores y reordenarlo.

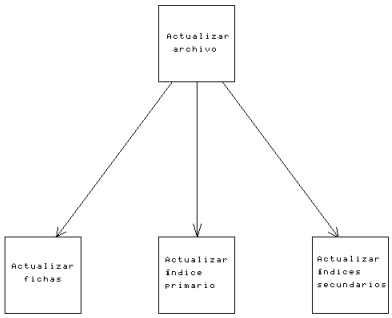


Figura A-4. Detalle de la composición del módulo actualización de archivos.

A.3 Programa para registro civil

Definición del problema

El registro civil es la institución que acredita los actos relacionados con el estado y capacidad de las personas, como su nacimiento o su enlace matrimonial. En sus oficinas se efectúan los procedimientos necesarios para dejar constancia de tales hechos.

Con objeto de organizar y sistematizar las actividades desarrolladas en una (hipotética) oficina del registro civil se desarrollará un programa de cómputo que automatice el registro de los actos, la emisión de constancias (previas a las actas oficiales) y el mantenimiento de los archivos de la oficina. Además, el programa facilitará el trabajo de manipulación de datos y permitirá un mejor cuidado de los libros de actas al reducir su uso directo.

Los actos de estado civil considerados para elaborar el programa son:
1) Nacimientos.
2) Matrimonio y divorcios.
3) Defunciones

Este problema está basado en el manejo de datos y tiene cierta complejidad por lo que conviene analizarlo con diagramas de flujo de datos.

• Panorama general

Análisis del problema

El diagrama de contexto nos permite identificar todos los datos que estarán en movimiento, así como las personas involucradas en su manejo (fig A-5).

Los principales datos que fluirán a través del programa son el nombre de los participantes, las actas civiles, las referencias de localización, la identificación de la oficina, el tipo de acto efectuado y la autorización de actos.

Las personas involucradas actúan como fuentes o beneficiarios de la información del programa; éstas son:

1. El juez del registro civil. Es el titular de la oficina y debe autorizar validez los movimientos.
2. Los participantes en el acto. Son los afectados (recién nacidos, o contrayentes de matrimonio) y los testigos.
3. El interesado es la persona que necesita de la información asentada en los libros de actas. En esta categoría caen los receptores del acto, el juez, un historiador, etcétera.

Figura A-5. Diagrama de contexto del programa para registro civil.

- **Componentes principales**

El diagrama de nivel "0" sirve para identificar las principales actividades realizadas y agruparlas en rubros cualitativamente diferentes. En el problema del registro civil encontramos cuatro grandes rubros: (*i*) registro de los actos, (*ii*) expedición de las constancias, (*iii*) emisión de reportes, y (*iv*) actualización de los datos (fig. A-6). En el mismo diagrama podemos identificar claramente cuáles datos son entradas y cuáles son salidas, según indique la flecha del flujo.

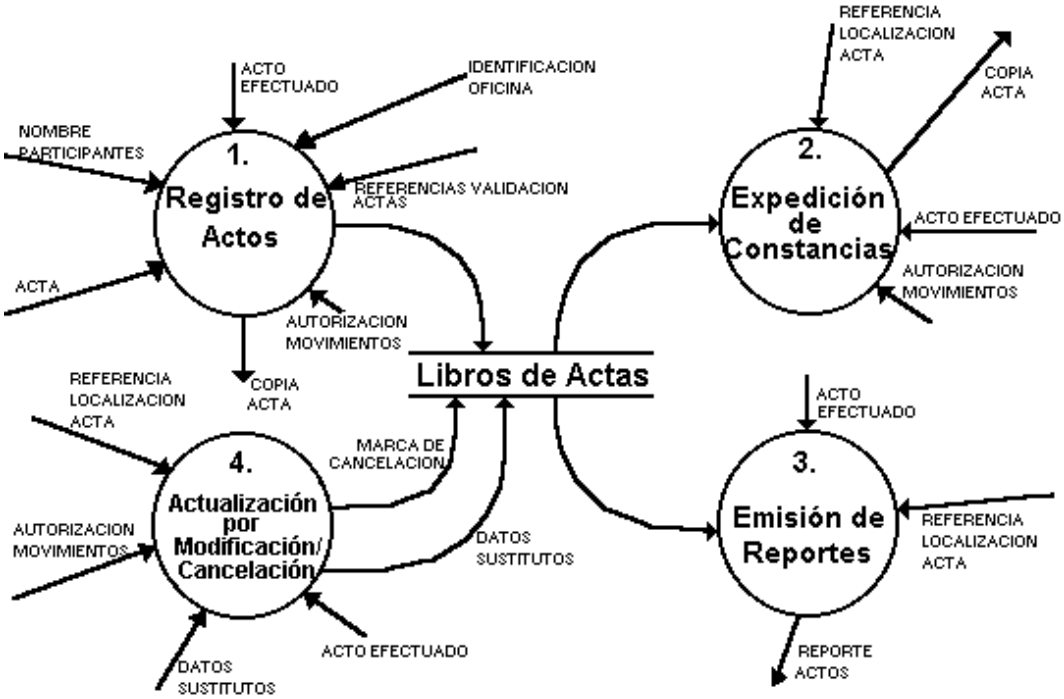


Figura A-6. Diagrama de nivel 0; principales componentes del programa para registro civil.

- **Registro de actos**

Este diagrama nos muestra los procesos involucrados en el asentamiendo de datos de un acto civil. La figura A-7 nos muestra que básicamente hay tres procesos:

- **La captura de datos**. Toda la información relacionada con el acto será concentrada en un formato especial que pueda ser revisado antes de elaborar el acta oficial y así minimizar los errores.
- **El registro en el libro de actas**. Dependiendo del acto celebrado (nacimiento, matrimonio o defunción) se efectuará un vaciado de los datos del formato en el libro correspondiente.
- **Emisión de constancias**. Este documento servirá para comprobar que se verificó el acto en tanto se puede emitir una copia certificada del acta.

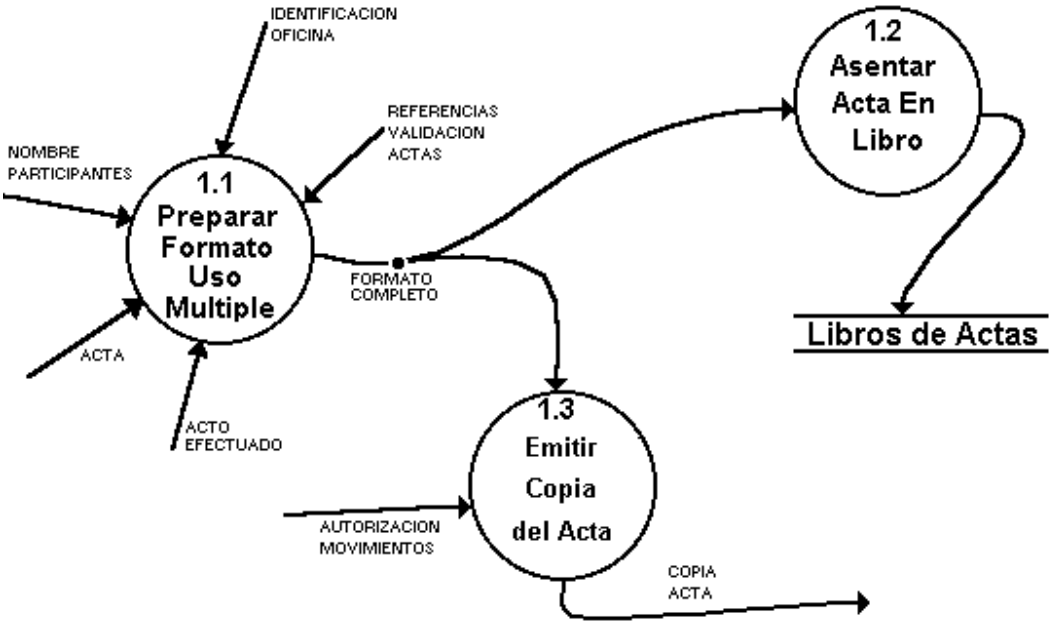


Figura A-7. Diagrama de nivel 1 del subprograma para registro de actos.

- **Expedición de constancias**

Este diagrama nos muestra los procesos para comprobar documentalmente la realización de un acto civil. La figura A-8 nos muestra que básicamente hay tres procesos:

- **Preparar un patrón de búsqueda**. Consiste en recabar los datos necesarios para el acta localizar en el libro correspondiente. En este caso se requieren: el tipo, la fecha del acto y el número de referencia del acta.
- **Búsqueda del patrón**. Consiste en revisar el libro de actas hasta encontrar aquélla que corresponde a los datos solicitados en el patrón.
- **Emisión de constancias**. Este documento servirá para comprobar que se localizó el acta en tanto se puede emitir una copia certificada.

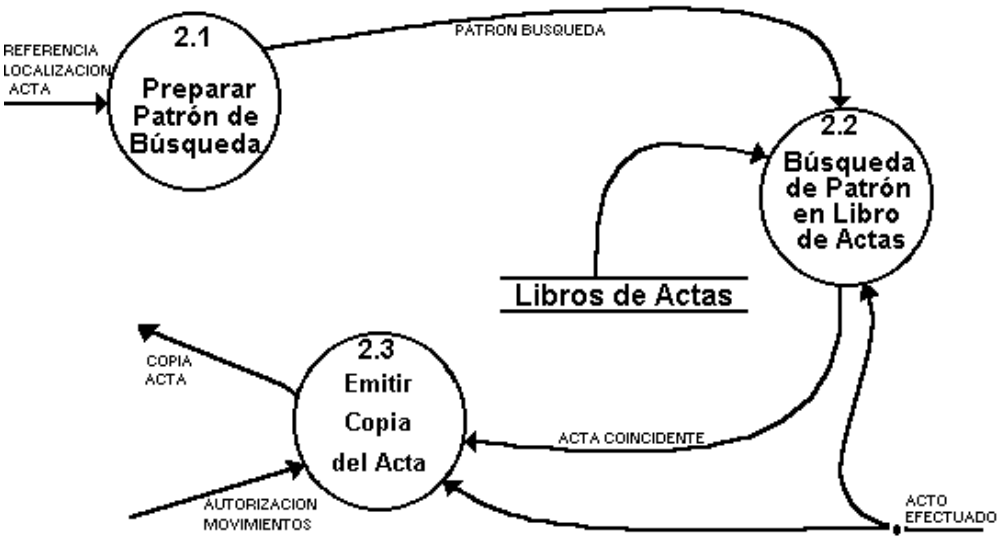


Figura A-8. Diagrama de nivel 1 del subprograma para emisión de constancias.

- ## Emisión de reportes

Este diagrama muestra los procesos para elaborar listas de los actos realizados. En la lista pueden abarcarse todos los actos o sólo aquéllos que cumplen con alguna caracteristica en común (el tipo de acto, el período de realización, la coincidencia de apellidos, etc.). La figura A-9 nos muestra que básicamente hay tres procesos:

- **Preparar un patrón de búsqueda**. Consiste en recabar ciertos datos que limiten la cantidad de actas a ser incluídas en el reporte.
- **Búsqueda del patrón**. Consiste en revisar los libros para encontrar cada acta que cumpla con los datos solicitados en el patrón.
- **Emisión del reporte**. Este proceso imprimirá la lista de actos que cumplieron con las especificaciones del patrón de búsqueda.

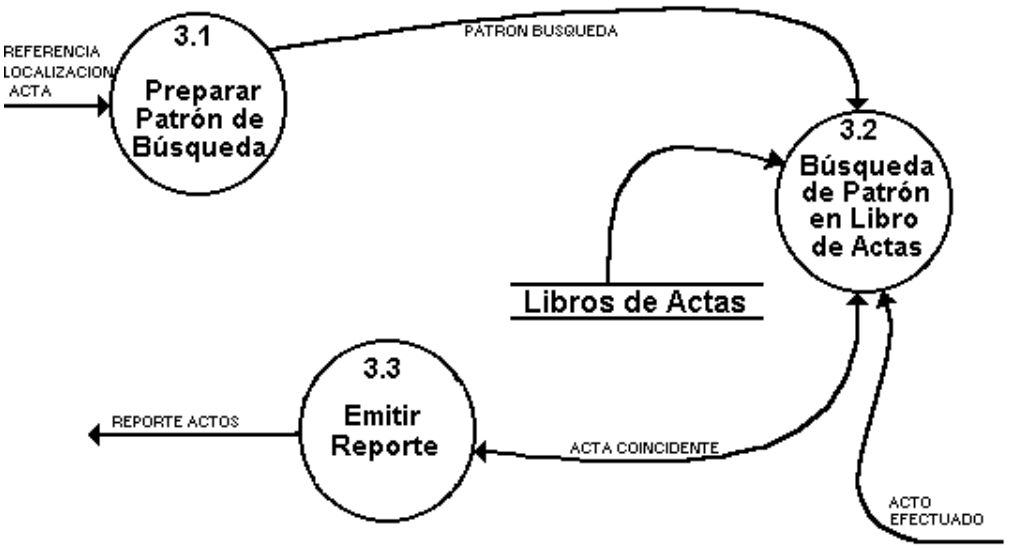


Figura A-9. Diagrama de nivel 1 del subprograma para elaboración de reportes.

- **Modificación o cancelación de actas**

Aun cuando todos los procedimientos se verifiquen con el mayor esmero pueden ocurrir errores durante la captura de los datos; por ello resulta conveniente añadir un procedimiento para su corrección y en su caso eliminación. (Sin embargo, la actualización de datos se realizará únicamente en los archivos de la computadora y no en los libros de actas, por lo que estos últimos deberán seguirse manejando com mucho cuidado.)

En este diagrama se muestran los procesos para efectuar la modificación o cancelación de actas. La figura A-9 nos muestra que básicamente hay cinco procesos:

- **Preparar un patrón de búsqueda**. Consiste en recabar el tipo de acto y el número de referencia del acta a modificar o borrar.
- **Búsqueda del patrón**. Consiste en revisar los libros para encontrar el acta a modificar.
- **Confirmación del movimiento**. Consiste en solicitar autorización para modificar los datos asentados o eliminar un acta.
- **Emitir marca de cancelación**. Este proceso coloca una "marca" en el acta para que sea considerada como "inválida" (aunque aún esté en el libro de actas).
- **Modificación del acta**. Este proceso permite modificar alguno de los datos asentados previamente en un acta.

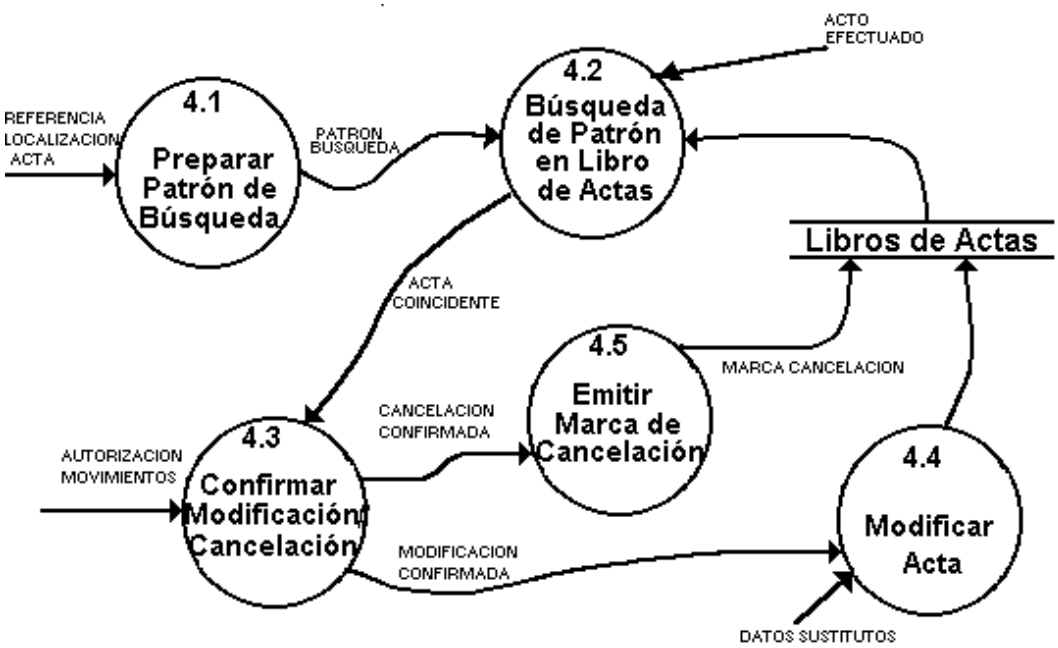


Figura A-10. Diagrama de nivel 1 del subprograma para modificación o cancelación de actas.

Terminado el análisis se puede pasar al diseño del programa. En el caso que nos ocupa basta traducir cada una de las burbujas que aparecen en los diagramas de nivel "1" a módulos (procedimientos) del programa mientras que las flechas se traducen en variables y parámetros a pasar a los módulos.

De este modo los diagramas de flujos de datos se convierten en un *mapa* que guía la construcción del programa.

Indice