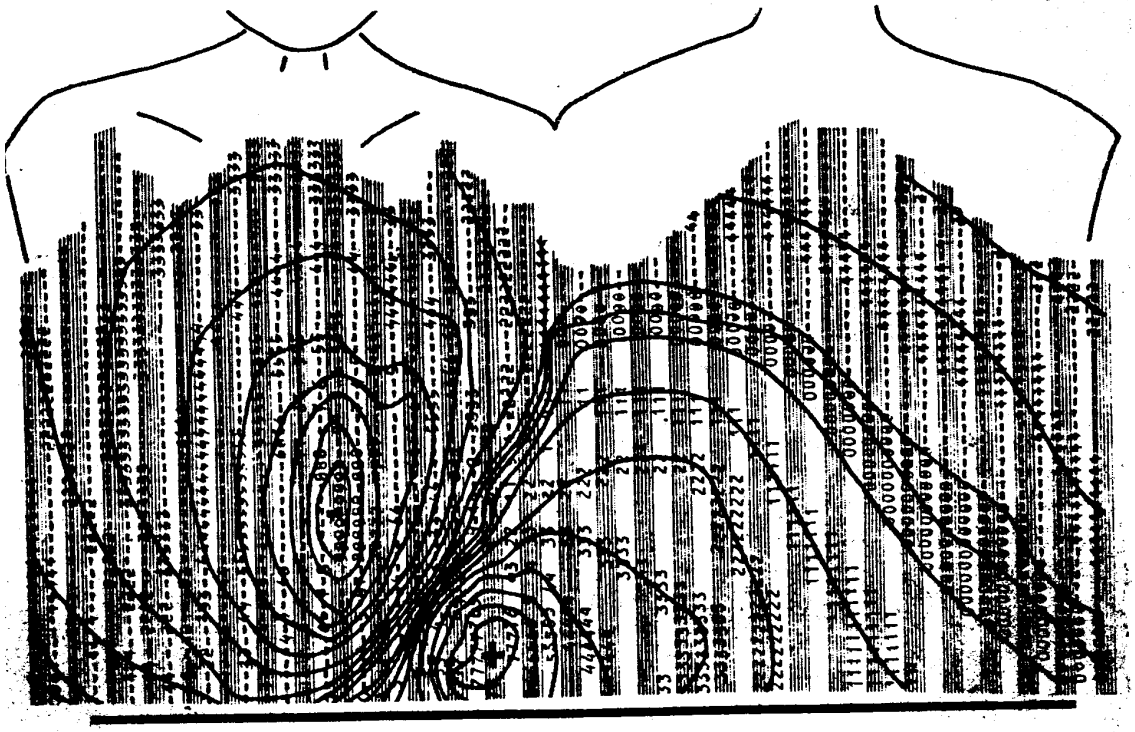
REVISTA DEL SEMINARIO de ENSEÑANZA Y TITULACION

Vol. VII num. 52



1991

PANORAMA GENERAL DE LA PROGRAMACION ESTRUCTURADA

Amparo López Gaona
Depto. Matemáticas, Fac. Ciencias, UNAM

Con frecuencia se cree que programar de manera estructurada consiste en hacerlo usando cierto conjunto de instrucciones y evitando otro, en particular evitando la famosa instrucción de salto incondicional o *GO TO*. En este documento se describe en qué consiste la programación estructurada y se proporcionan algunas reglas para lograr programar de esa manera.

La *programación estructurada* es una técnica que sistematiza y organiza el ciclo de diseño, codificación, pruebas y mantenimiento de programas, con el propósito de desarrollar programas correctos y confiables previniendo errores y facilitando su depuración; minimizando así los costos al incrementar la productividad.

Para lograr programar estructuradamente, es necesario hacer un diseño con el cual se llegue a tener la solución en forma de módulos independientes y que éstos se puedan codificar de manera estructurada, con el objetivo de simplificar la etapa de pruebas y la de mantenimiento.

TECNICAS DE DISEÑO

Para facilitar el desarrollo de programas es necesario empezar haciendo un buen diseño de la solución al problema que se desea resolver. El no hacerlo es como si se deseara construir una casa sin planos y se empezara por pegar ladrillos; de esta manera es probable que se termine la casa, pero no resulte segura, y puede emplearse más tiempo y material que si se hubiera hecho primero un diseño y se trabajara de acuerdo con él.

El diseño puede hacerse usando una metodología propia o alguna de las conocidas, entre éstas se encuentran la de arriba a abajo o la de abajo a arriba.

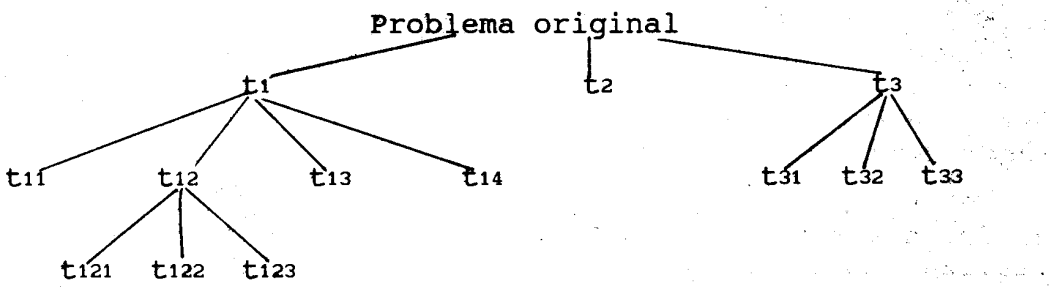
El que cada quién emplee su propia metodología de diseño puede ser malo porque sólo el que la usa sabe en qué consiste y al llegar otra persona puede no entender de que se trata.

La técnica de diseño llamada *de abajo-arriba* implica pensar inmediatamente en la codificación de algunas partes elementales. Con este enfoque se tienen que tomar las decisiones acerca de detalles de nivel inferior, como la definición de formatos de entrada, antes de conocer las implicaciones de tales decisiones. Se puede emplear más tiempo en la definición de esos detalles.

La técnica de *diseño de arriba a abajo* ha sido usada en otras áreas por ejemplo en la elaboración de ensayos, tesis, reportes de libros, revisiones críticas, tareas de investigación sobre algún tema.

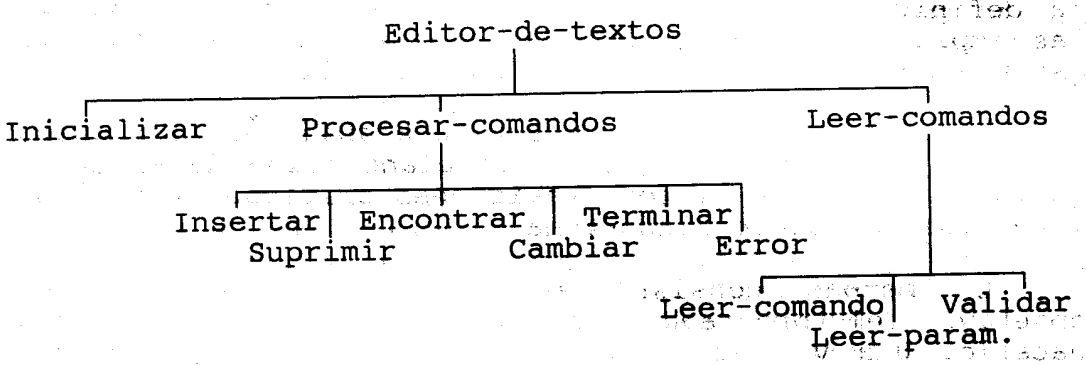
El método consiste en empezar por definir el objetivo del programa -*qué* se desea hacer, no *cómo* hacerlo. Una vez establecido el objetivo se define una serie de tareas a realizar para lograrlo. El diseño continúa con la división de cada tarea en sub-tareas más detalladas. Este proceso de *refinamiento* continúa hasta que la descripción de cada sub-tarea, sea tan simple como para poder llevarse a cabo sin mayor detalle.

El proceso de desarrollo se puede ver como la estructura jerárquica mostrada en la figura



En la figura las subtareas t_1, t_2 y t_3 se necesitan para resolver el problema original. Las subtareas t_{11}, t_{12}, t_{13} y t_{14} se necesitan para realizar la subtarea t_1, y así sucesivamente.

Como ejemplo práctico se presenta un diagrama de diseño para elaborar un editor de textos. Para un editor es necesario tener ciertas condiciones iniciales, leer cada comando y procesarlo. Leer un comando significa hacerlo junto con sus parámetros y validarlos. Los comandos con se puede trabajar son: insertar, suprimir, encontrar, cambiar, terminar y enviar mensajes de error:



Ventajas del diseño de arriba a abajo

Probablemente la principal ventaja de este diseño es su contribución a hacer manejable un programa. Esto se logra mediante el proceso llamado *abstracción*. Es decir, inicialmente se trata con una operación sólo desde un punto de vista general, totalmente separada de los detalles de la subestructura. Esta abstracción permite, por ejemplo, entender un mapa complejo y llegar a un lugar dado sin importar las condiciones de la carretera.

Otro punto a su favor es el hecho de poder *retrasar decisiones* hasta que la codificación de los módulos se realiza. Para las decisiones de detalles de bajo nivel esto ocurrirá al final del desarrollo del proyecto, cuando se tiene una mejor idea de qué hacer.

Las estructuras de datos, se desarrollan también de arriba-abajo. Las decisiones iniciales se toman en base a la necesidad de tales estructuras de datos y a la organización de nivel superior. Las decisiones de nivel inferior tales como el formato exacto de los campos individuales y los valores iniciales dentro de los campos pueden posponerse para refinamientos posteriores. Este es un punto importante: diseño estructurado, abstracción y retraso de decisiones se aplican tanto a algoritmos como a estructuras de datos.

Otra ventaja de este método es que se puede validar cada unidad del programa antes de que el código esté totalmente terminado. El número de proposiciones individuales no es el único factor que afecta el tiempo de depuración. Los efectos de sus interacciones también contribuyen al problema.

Desde luego, es una ventaja depurar un programa como una serie de unidades pequeñas en lugar de como un todo. El diseño de arriba-abajo, impone una jerarquía de tareas, definiendo un conjunto natural de sub-unidades que pueden probarse y verificarse individualmente para su posterior integración en la solución global.

Otra ventaja, es la facilidad para delegar y manejar toda la carga de trabajo. Al tener la secuencia de los módulos del programa en forma jerárquica resulta natural que una persona sea la responsable de supervisar el proyecto general y sus sub-tareas principales.

MODULOS INDEPENDIENTES

Las tareas y sub-tareas del diseño se transforman en módulos al momento de programarse. Las características de los módulos que comprenden la solución de un problema son:

- Independencia
- Coherencia Lógica
- Tamaño

Independencia

Los módulos deben ser auto-contenidos e independientes de otros módulos en el sistema. Específicamente deben ser independientes de:

1. La fuente de las entradas
2. El destino de la salida
3. La historia de activaciones de este módulo

El objetivo de hacer unidades independientes es poder eliminar un módulo y colocar otro sin afectar el resto del programa, siempre y cuando las especificaciones de entrada/salida sean idénticas, para ambos módulos. Por ejemplo al usar la rutina para cálculo del seno en la siguiente instrucción

y := sin(theta) - 1.0

No interesa si se usó una serie de Taylor o algún otro método, siempre y cuando tenga las siguientes características:

a. Acepte un parámetro real, *x*, cuyo valor esté en radianes
b. Regrese un valor real, *v*, tal que v= Sen(x)
c. Lo haga en una cantidad razonable de tiempo.

La mejor manera de lograr indendencia de módulos es:

1. Evitando la modificación innecesaria de variables globales. Esto no quiere decir que no existan variables globales pero cuando se requiera modificarlas se debe hacer con mucho cuidado.
2. Declarando variables locales
3. Evitando cambios a los parámetros de entrada, tratando de pasarlos por valor.y los de salida por referencia.
4. No haciendo algo que no esté definido en la especificación del programa.

Estas reglas indican que un procedimiento sólo debe afectar a sus parámetros por referencia. Como ejemplo, suponga que se tienen números enteros entre 1 y 10, además de -1 para indicar datos erróneos, y se desean ordenar de manera ascendente, con los -1 al final. El siguiente procedimiento puede resolver el problema:

```
procedure ordena(var lista : arreglo; n : integer);
{Procedimiento que ordena una lista de n valores en el
rango de 1 a 10 en orden ascendente usando el algoritmo
de la burbuja.}

var
   i        : integer;      { Indice para el ciclo }
   ordenado : boolean;      { Para verificar si ya está
                              ordenado}
   temp     : integer;      { Temporal para intercambios }

begin
  for i := 1 to n do
    if lista[i] < 0 then
      lista[i] := +11; { muy malo!!!!}

  repeat
    ordenado := true;
    for i := 1 to n-1 do
      if lista[i] > lista[i+1] then
      begin
        ordenado := false;
        { Intercambia dos valores}
        temp := lista[i];
        lista[i] := lista[i+1];
        lista[i+1] := temp;
      end;
  until ordenado;
end;
```

Lo malo es cuando se desee usar este procedimiento en algún otro programa suponiendo que sólo ordena datos porque cada vez que se tengan números negativos los transforma.

Lo que debe hacerse es eliminar los -1 antes de llamar al procedimiento de ordenamiento para que éste se limite a ordenar la lista como debe ser.

En resumen, *independencia* requiere un conjunto de módulos construídos de tal manera que un cambio a alguno no cause un cambio inesperado a los otros módulos del programa.

Coherencia Lógica:

Se debe resistir la tentación de incluir más de una tarea en un módulo, debido a que el resultado sería una unidad grande, compleja y confusa.

Es fácil incluir operaciones de tareas relacionadas pero distintas en un solo módulo, Por ejemplo las tareas t_{31} y t_{32} del diagrama jerárquico original, se requieren para ejecutar la tarea t_3, por tanto probablemente estén relacionadas. Sin embargo, es incorrecto manejarlas en un solo módulo.

Ejemplos de tareas relacionadas

1. Leer datos
   Validar

2. Leer una cadena
   Buscar un patrón específico

3. Calcular la media
   Calcular el coeficiente de variación

Existen razones importantes para conservar los módulos lógicamente coherentes y la más importante está relacionada con la *modificación*. Es normal que la especificación del problema necesite cambios; para minimizar los errores durante el proceso de modificación se debería modificar el código lo menos posible. Las secciones de código que no resulten afectadas por las nuevas especificaciones no deben modificarse. Una forma de evitar esto es aislar cada operación indivisible.

Por ejemplo si A y B son tareas separadas, ordenar e intercalar respectivamente, pero se codifican como un solo módulo. Al modificar A, desarrollando un algoritmo más eficiente existe el peligro de propiciar un error en B, por ejemplo modificando algún índice.

Además de las ventajas ganadas durante la modificación se pueden lograr beneficios significativos en la fase de prueba. Por ejemplo, si A tiene *m* diversas ramificaciones y B tiene *n*. Al escribir ambas tareas en un solo módulo, se pueden requerir al menos m*n conjuntos de datos para probar exahustivamente este módulo. Por otro lado, si se codifican como dos módulos separados se

requieren sólo *m* casos de prueba para A y *n* para B, haciendo un total de *m+n* casos en total.

Igual de importante que mantener coherencia lógica entre tareas del mismo nivel lo es entre tareas de diferente nivel. Excluyendo detalles innecesarios e inapropiados de nivel inferior que obscurecen el próposito y la función de un procedimiento.

Tamaño de los módulos

El tamaño adecuado para un módulo no es una característica aislada, si se diseña cada módulo para realizar una tarea, lógicamente será pequeño.

No existe un tamaño estándar lo que se propone es que no exceda una página de código, es decir como de 60 líneas aunque hay quien dice que no debe exceder 2 páginas.

Pero el tamaño no está en relación al papel sino en la coherencia lógica. Si nos encontramos escribiendo un módulo de 200 o 300 líneas es muy probable que estemos incluyendo varios detalles de otro nivel.

CODIFICACION ESTRUCTURADA

La codificación estructurada es lo que generalmente se conoce como programación estructurada. Su objetivo principal es crear programas legibles y entendibles, usando principalmente tres formas básicas de codificación: secuencia, iteraciones y condicionales. Para apreciar la legibilidad de un código estructurado ver los siguientes fragmentos para encontrar el mayor de tres valores.

```
repeat
  read(x, y, z);
  if (x >=y) and (x >= z) then
    mayor := x
  else if (y >= x) and (y >= z) then
         mayor := y
       else mayor := z;
until eof
```

```
8:    read (x, y, z);
      if x >= y then goto 1;
      if y >= z then goto 5;
      goto 4;
1:    if x >= z then goto 2;
      goto 4;
5:    mayor := y;
      goto 3;
2:    mayor := x;
      goto 3;
4:    mayor := z;
3:    if not eof then goto 8
```

Una *proposición secuencial* ejecuta una operación y luego continúa con la siguiente. Ejemplos de éstas, en Pascal, son la asignación, lectura, escritura, proposiciones compuestas y llamadas a procedimientos.

Una *proposición condicional* ejecuta una prueba para decidir cuál proposición debe realizarse a continuación.

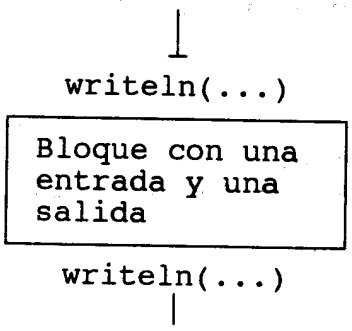
Una *proposición iterativa* ejecuta repetidas veces la proposición llamada *cuerpo* hasta que se satisface alguna condición específica.

Un programa bien estructurado contiene principalmente proposiciones de estos tres tipos. La principal característica del código estructurado es que éste se compone de segmentos de código anidados de manera adecuada a los cuales se entra solo por la parte superior y se sale por la inferior. Todos los programas pueden codificarse de esta manera.

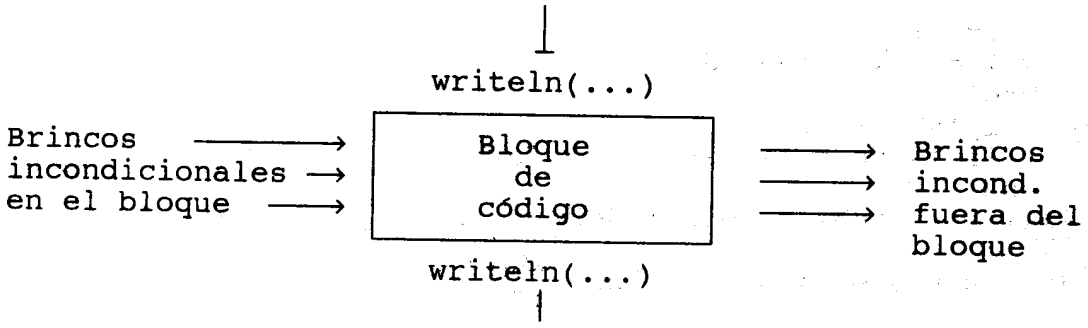
Sin embargo, la mayoría de los lenguajes tienen la instrucción goto. El uso indiscriminado de goto's puede dar como resultado un programa díficil de entender y por consecuencia díficil de trabajar con él y darle mantenimiento.

El problema al intentar entender un programa así, es que se necesita seguir dos o más trayectorias distintas que pueden ser partes totalmente separadas del programa.

Codificar estructuradamente es muy importante durante las etapas de depuración y prueba de los programas. Si un bloque de programa tiene una sola entrada y una sola salida, resulta relativamente fácil probar la correctez del código. Basta con escribir los valores de las variables importantes a la entrada y a la salida del bloque.



Si los valores fueron correctos al entrar e incorrectos al salir, el error se encuentra en ese bloque. Sin embargo al permitir brincos arbitrarios no se puede determinar con tanta facilidad el error ya que puede estar en ese bloque o en otros lo cual incrementa significativamente la complejidad de la tarea de depuración.



Existe una técnica importante en programación llamada *verificacion de programas* que intenta verificar la correctez de un programa usando matemáticas formales en lugar de hacerlo empíricamente.

La meta de la codificación estructurada no es eliminar los goto sino incrementar la claridad y legibilidad. Los goto se deben usar siempre que esto mejore la organización de un programa. Los gotos no deben usarse:

1) Para brincar hacia atrás, pues esto implica repetición y para eso existen proposiciones. Por ejemplo

```
      suma := 0.0;
      i := 0;
100:  i := i + 1;
      suma := suma + a[i]
      if (suma <= 100.0) and (i < n) goto 100
```

Puede escribirse como:

```
suma := 0.0;
i := 0;
repeat
    i := i + 1;
    suma := suma + a[i]
until (i >= n) or (suma > 100.0)
```

2) Para salir de un ciclo antes de que se satisfaga la condición normal, pues esto oscurece la condición de salida del ciclo. Por ejemplo:

```
  i := 0;
  while i < tamano do
  begin
    i := i + 1;
    if llave = lista[i] then goto 1
  end;
1:if i <= tamano then { determina porque termino }
```

Es más claro si se escribe

```
  i := 0;
  encontro := false;
  while (i < tamano) and (not encontro) do
  begin
    i := i + 1;
    if llave = lista[i] then encontro := true;
  end;
  if encontro then      {determina porque termino }
```

Puede ser útil usar los gotos cuando ocurre un error o una terminación anormal que cause romper el flujo normal de la lógica. En tal caso el goto debeser para brincar hasta la salida de ese módulo y debe estar comentado. En algunos lenguajes de programación modernos y alguno antiguo se tienen instrucciones para hacer esto y eliminar completamente los gotos, y estas son las llamadas para manejo de errores.
Por ejemplo:

```
for i := 1 to max do
begin
  suma_plano := 0;
  for j := 1 to renglon do
  begin
    suma_renglon := 0;
    for k := 1 to columna do
    begin
      if matriz[i,j,k] < 0 then
        goto 99;      {transfiere al final de la rutina}
      suma_renglon := suma_renglon + matriz[i,j,k]
    end;
    writeln('La suma del renglon ...');
    suma_plano := suma_plano + suma_renglon;
  end;
  writeln('La suma del plnao ',i,' es ',suma_plano);
end;
  ....
99: end;
```

Aquí se uso de una manera específica y limitada: para terminar el módulo debido a un error fatal, además se agregó un comentario para aclarar el lugar en donde será el salto. De no usar el goto el código quedaría:

```
for i := 1 to max do
begin
  suma_plano := 0;
  error := false;
  for j := 1 to renglon do
  begin
    suma_renglon := 0;
    for k := 1 to columna do
    begin
```

```
      if matriz[i,j,k] < 0 then error := true;
      if not error then
        suma_renglon := suma_renglon + matriz[i,j,k]
    end;
      if not error then
      begin
         writeln('La suma del renglon '...);
         suma_plano := suma_plano + suma_renglon;
      end;
  end;
  if not error then
     writeln('La suma del plano ',i,' es ',suma_plano);
end;
  if not error then
  begin
  ....
  end;
end;
```

Con respecto a la codificación estructurada se mencionaron los mecanismos que proporciona el lenguaje, pero otros puntos que deben tomarse en cuenta son los que se mencionan a continuación:

Muchas veces se cree que si escribe de manera obscura e intrincada es mejor. Se debe recordar que el código escrito será leído y estudiado por otras personas, por tanto debe ser claro y reflejar la operación que está haciendo. La motivación en todos los programas debe ser la *simplicidad y claridad*, por tanto se debe decir exactamente lo que se desea.

Nada contribuye tanto a la claridad y legibilidad de los programas como los nombres descriptivos para todos los identificadores, no se deben usar abreviaturas inventadas por uno ni nombres que no tengan que ver con el propósito del identificador. Debe ser suficiente ver el programa principal para determinar qué hará el programa. La única razón para revisar los procedimientos es determinar cómo se está haciendo cada tarea. Por ejemplo la siguiente rutina

```
function x (a1 : y ; a2 : integer) : integer;
var  i, w : integer;
begin
  w := a1[1];
  for i:= 1 to a2 do
  if a1[i] > w then
    w := a1[i];
    x:= w;
end;
```

resulta más fácil de entender si se presenta como sigue:

```
{  Función que obtiene el mayor de n elementos enteros
almacenados en el arreglo A }
function mayor (a : enteros ; n : integer) : integer;
var
     i, grande  : integer;
begin
  grande := a[1];
  for i := 2 to n do
    if a[i] > grande then
      grande := a[i];
  mayor := grande;
end;
```

Es conveniente incluir al principio de cada módulo un comentario que indique el contenido del mismo y su objetivo,

1. Un resumen de qué hace el programa y qué método usa
2. El nombre del programador
3. La fecha en que se escribió
4. Una referencia a documentación de este módulo
5. Una breve historia de todas las modificaciones del programa y el porqué de las mismas.

Para cada rutina tener

6. Las condiciones de entrada. El estado inicial del subprograma y los valores iniciales pasados a él.
7. Las condiciones de salida. Qué valores regresa al termino.

Debido a que las declaraciones pueden ser un prólogo del programa es conveniente poner un comentario indicando el uso de cada variable o constante.

Otros comentarios que resultan útiles es explicar cada bloque o separarlos por espacios en blanco, indicar en cada *end* a que *begin* pertenecen.

Comentarios que no son útiles son aquellos que escriben el significado obvio de las proposiciones o que son cripticos. El código debe indicar *cómo* se realiza alguna operación y el comentario el *porqué*. Por tanto se deben evitar los comentarios poco claros, técnicos.

Para no caer en el error de hacer comentarios obvios o poco útiles se deben escribir éstos a la par del programa no al concluir éste.

Por último como los módulos sufren cambios es necesario actualizar los comentarios porque no hacerlo es peor que no ponerlos.

Mejora la legibilidad un módulo bien identado. Todas las proposiciones de un mismo nivel deben estar alineadas

```
while not eof do
begin
readln(n);
for i:= 1 to n do
begin
read(ch);
write(ch)
end
end
write(n)
```

```
while not eof do
begin
  readln(n);
  for i:= 1 to n do
  begin
    read(ch);
    write(ch)
  end
end
write(n)
```

Programas robustos

Hasta ahora se ha hablado de claridad y legibilidad pero existe otro punto muy importante que se debe tener en cuenta y este es la *robustez.*

Un programa es robusto si produce resultados coherentes, no necesariamente útiles o correctos, para cualquier conjunto de datos, sin importar si éstos son ilegales, inapropiados o patológicos. Esto incluye enviar mensajes de error explicando porqué las operaciones no pueden ejecutarse, si es posible indicando cómo corregir los datos.

Las guías de robustez requieren que el programa no termine de manera inesperada bajo ninguna causa y éstas incluyen:

1. Validar los datos de entrada. Que estén en los rangos permitidos, que sean del tipo esperado, en caso contrario enviar algún mensaje en el que se indique claramente el tipo de error ocurrido y de ser posible cómo corregirlo, o bien terminar. La regla es "Si entra basura, basura se obtendrá".

2.otección contra errores en tiempo de ejecución. Un error de corrida ocasionará que el programa termine de manera anormal. Por ejemplo aunque se haya validado cierta variable *x* de grados que esté entre 0 y 360, al calcular tan(x) estará indefinido si *x* es 90 o 270. En Pascal los errores de ejecución más comunes son:

1. Arreglos fuera de rango
2. Subrango fuera de límites
3. Conversión real a entera con el valor absoluto del real mayor que máximo entero.
5. Argumentos inadecuados pra alguna función
   sqrt(x) x<0
6. División por cero
7. Hacer referencia a datos antes de inicializarlos

Para evitar que aborte el programa es necesario incluir validaciones antes de ejecutar tales operaciones.

3. Protección contra errores de representación. Entre éstos se encuentran:

a. Errores de exactitud. Nunca se debe preguntar por el valor exacto de una variable real, ya que se pueden tener errores debido a la falta de exactitud. Por ejemplo:

$(1/5)_{10} = (0.001100110011...)_2 = (0.1992)_{10}$

Por tanto

$1/5 + 1/5 + 1/5 + 1/5 + 1/5 = 0.996 \quad 1$

Lo aconsejable es preguntar por <= o >=, pero nunca por igualdad o desigualdad estricta. Por ejemplo se desea deten algún ciclo cuando x = 2.2 no debe hacerse como lo sigue:

```
x := 0.0
while x <> 2.2 do
begin
  ...          { Procesa el valor actual de x }
  x := x + 0.2
end;
```

Debe hacerse como se muestra a continuación

```
x := 0.0
while x <= 2.0 do
begin
  ...          { Procesa el valor actual de x }
  x := x + 0.2
end;
```

Otro de los errores con números reales es el llamado *error de truncación*, estos son ocasionados por la aproximación de un proceso matemático infinito con un número finito de operaciones de programación.

Generalidad

La generalidad libera a un programa de dependencia de cualquier conjunto específico de datos, esta propiedad es muy deseable porque reduce la necesidad de modificar programas conforme cambie la necesidad del usuario final. Ejemplo de un programa poco flexible y sin generalidad

```
cont := 0;
suma := 0;
for i := 1 to 25 do
  if (calif[i] >= 0) and (calif[i] <= 100) then
  begin
    cont := cont + 1;
    suma := suma + calificacion[i]
  end;
promedio := suma / cont;
```

Además no es robusto porque la división puede ser entre cero. Para hacerlo un poco más general usar constantes

```
const
  datos  = 25;      { Número de datos }
  maxima = 100;     { Calificación máxima }
  minima = 0;       { Calificación mínima }
      ...
cont := 0;
suma := 0;
```

```
error := false;
for i := 1 to datos do
  if (calif[i] >= minima) and (calif[i] <= maxima) then
  begin
    cont := cont + 1;
    suma := suma + calificacion[i]
  end;
if cont > 0 then
  promedio := suma / cont
else
  error := true;
```

Sin embargo esto no es suficiente, si los datos son susceptibles de cambios lo mejor es usar variables y la forma de darles valor inicial es leyendolos.

Para escribir procedimientos generales, cualquier valor que pueda cambiar de una invocación a otra se debe usar como parámetro.

La generalidad no es solo cuestión de codificación, sino más bien de diseño.

Portabilidad

En tanto la generalidad hace un programa independiente de cualquier conjunto de datos, la portabilidad implica hacerlo independiente del hardware o sistema operativo específico.

Para ello es necesario usar la parte estándar de los lenguajes y lo particular documentarlo bien y como rutinas para facilitar las modificaciones.

Comportamiento de la Entrada/Salida

Hasta ahora todo a tenido relación con el mantenimiento. Al usuario final no le importa si está estructurado o no el programa; él lo ve como una caja negra. No importa que tan estructurado esté el progama, si su interfaz es poco clara nadie lo usará. Algunos puntos que se deben tener en cuenta con respecto a la entrada/salida son:

Hacer una presentación limpia y clara con ayudas que el usuario pueda omitir si ya conoce el sistema.

Siempre indicar qué dato se espera enviando un prompt o una pantalla de captura.

Evitar programas que caigan en ciclos infinitos pidiendo los datos correctos.

Siempre terminar los datos con alguna señal, nunca contando el número de datos.

Pedir los datos de manera que resulte natural para el usuario no para el programa.

Usar valores por omisión para que la cantidad de datos requeridos se reduzca.

Principalmente, emplear suficiente tiempo en producir salidas que sean elegantes, legibles y directamente usables por el usuario final.

CONCLUSION

Las tres fases de la programación estructurada cubren todos los aspectos del proceso de programación. El diseño arriba-abajo dirige el programa con un enfoque de una tarea grande y compleja desarrollando una jerarquía de sub-tareas pequeñas. La *modularidad* involucra la implementación de estas sub-tareas como módulos distintos, lógicamanete coherentes e independientes con ciertas características de programación. Por último las técnicas de *codificación estructurada* tienen que ver propiamente con la codificación de los módulos definidos. Estos tres principios definen una técnica de manejo de programación para la implementación de programas correctos y confiables.

AL ENVIAR SUS ARTICULOS TENGAN EN CUENTA LAS SIGUIENTES NORMAS:

ENVIAR SUS ARTÍCULOS MECANOGRAFIADOS EN MÁQUINA ELÉCTRICA CON LETRA TIPO "LETHER GOTHIC" A RENGLÓN Y MEDIO Y HOJAS TAMAÑO "CARTA" O BIEN EN CUALQUIER PROCESADOR DE PALABRAS QUE TENGA "DRIVER" PARA IMPRESORA "LASER", CON LETRAS TIPO "GÓTICO" EN FORMATO MEDIO OFICIO A RENGLÓN SEGUIDO. EN CASO NECESARIO DIFERENCIAR LOS NÚMEROS "0" Y "1" DE LAS LETRAS "O" Y "L".

EL NOMBRE DEL AUTOR JUNTO CON SUS DATOS, INCLUYENDO ALGÚN TELÉFONO, DEBERÁN APARECER EN UNA HOJA POR SEPARADO DEL TEXTO.

ENVIAR ORIGINALES Y NO COPIAS.

EN CADA ARTÍCULO DEBERÁ INCLUIRSE RELACIÓN DE LIBROS Y ARTÍCULOS CON SOLAMENTE BAJO EL TÍTULO DE "REFERENCIAS" O BIEN "BIBLIOGRAFÍA" SOBRE EL TEMA.

CADA AUTOR RECIBIRÁ CONSTANCIA DE RECEPCIÓN DE SU ARTÍCULO Y UNA VEZ APROBADA SU PUBLICACIÓN CONSTANCIA DE EN QUE NÚMERO APARECERÁ.

CADA AUTOR RECIBIRÁ CINCO EJEMPLARES DE LA REVISTA.

LOS TRABAJOS Y CUALQUIER CORRESPONDENCIA DEBERÁN REMITIRSE A:
"REVISTA DEL SEMINARIO DE ENSEÑANZA Y TITULACIÓN"
FACULTAD DE CIENCIAS, DEPARTAMENTO DE MATEMÁTICAS
CUBÍCULO 240, MÉXICO 04510, D.F.

LA REVISTA DEL SEMINARIO DE ENSEÑANZA Y TITULACIÓN CONSIDERA PUBLICABLES ARTÍCULOS DE CIENCIAS BÁSICAS, EN PARTICULAR DE MATEMÁTICAS, DE LA MANERA MÁS AMPLIA, DESDE LA MÁS EXTREMA DIVULGACIÓN HASTA ARTÍCULOS TAN TÉCNICA O CONCEPTUALMENTE SOFISTICADOS COMO LO REQUIERA SU CONTENIDO Y EL ESTILO DEL AUTOR, ABARCANDO ASPECTOS DE SU ENSEÑANZA, APLICACIONES E INVESTIGACIÓN. EN SU CASO EL ARBITRAJE DE LOS ARTÍCULOS SERÁ ANÓNIMO, LOS ÁRBITROS NO RECIBEN LOS NOMBRES DE LOS AUTORES Y VICEVERSA.