

26. PROGRAMACIÓN MODULAR. DISEÑO DE FUNCIONES. RECURSIVIDAD. LIBRERÍAS.

ESQUEMA DE CONTENIDOS

1. PROGRAMACIÓN MODULAR.....	2
1.1. CONCEPTOS INICIALES.....	2
1.1.1. Módulo.....	2
1.1.2. Interfaz e implementación.....	3
1.1.2.1. Interfaz.....	3
1.1.2.2. Implementación.....	4
1.2. REFINAMIENTO Y MODULARIDAD.....	5
1.2.1. Refinamiento sucesivo.....	5
1.2.2. La división en módulos y la inteligibilidad de los programas.....	5
1.2.3. El diseño de arriba abajo (TOP DOWN) concepto y utilidad.....	6
1.2.4. Tipos de módulos.....	6
1.2.4.1. Según los mecanismos de activación.....	6
1.2.4.2. Según el camino de control.....	6
1.3. OCULTACIÓN DE LA INFORMACIÓN.....	7
1.4. INDEPENDENCIA FUNCIONAL Y CALIDAD DEL SOFTWARE.....	7
1.4.1. Concepto.....	7
1.4.2. Cohesión.....	8
1.4.3. Acoplamiento.....	8
1.5. ESTRUCTURA DEL PROGRAMA Y JERARQUÍA DE CONTROL.....	9
1.5.1. Organización de módulos en niveles y transferencias de control.....	9
1.5.2. Principios de un buen método de diseño según Meyer.....	10
1.5.2.1. Principio de descomponibilidad modular.....	10
1.5.2.2. Principio de componibilidad modular.....	10
1.5.2.3. Principio de comprensibilidad modular.....	10
1.5.2.4. Principio de continuidad modular.....	11
1.5.2.5. Principio de protección modular.....	11
2. DISEÑO DE FUNCIONES.....	11
2.1. PROCEDIMIENTOS.....	11
2.1.1. Llamada a procedimientos.....	11
2.1.2. Parámetros.....	12
2.1.2.1. Concepto.....	12
2.1.2.2. Clases.....	12
2.1.3. Definición de procedimientos.....	12
2.1.4. Declaración de procedimientos.....	14
2.2. FUNCIONES.....	14
2.2.1. Concepto.....	14
2.2.2. Llamada a funciones.....	14
2.2.3. Definición de funciones.....	15
2.2.4. Declaración de funciones.....	15
3. LA RECURSIVIDAD.....	16
3.1. CONCEPTO.....	16
3.1.1. Recursividad en algoritmos.....	16
3.1.2. Recursividad en datos.....	17
3.2. RAZONAMIENTO RECURSIVO.....	17
3.3. RECURSIÓN E ITERACIÓN.....	18
3.4. COSTE DE LAS FUNCIONES RECURSIVAS.....	18
3.5. APLICACIONES DE LA RECURSIVIDAD.....	19
4. BIBLIOTECAS (LIBRERÍAS).....	20
4.1. CONCEPTO.....	20
4.2. REQUISITOS.....	21

4.3. UN CASO ESPECÍFICO, LAS A.P.I.S.	22
4.3.1. <i>Concepto</i>	22
4.3.2. <i>La API de Windows</i>	22
4.3.2.1. <i>Introducción y breve referencia</i>	22
4.3.2.2. <i>Enlace estático y dinámico</i>	23
4.3.3. <i>La API de Windows y las DLL</i>	23
5. BIBLIOGRAFÍA	24
6. UNIDAD DIDÁCTICA	25
7. DESTINATARIOS	25
8. UBICACIÓN, DISEÑO CURRICULAR BASE	25
9. RELACIÓN CON EL PROYECTO CURRICULAR DE ETAPA	26
10. OBJETIVOS-CAPACIDADES	26
10.1. CAPACIDADES TERMINALES	26
10.2. ESPECÍFICOS	27
10.2.1. <i>Conceptuales</i>	27
10.2.2. <i>Procedimentales (eje estructurador)</i>	27
10.2.3. <i>Actitudinales</i>	27
11. CONTENIDOS	27
12. ACTIVIDADES PROPUESTAS (ADICIONALES A LA EXPOSICIÓN)	28
13. MATERIAL DIDÁCTICO	28
14. INTERDISCIPLINARIEDAD	28
15. DISTRIBUCIÓN TEMPORAL	29
16. METODOLOGÍA Y ESTRATEGIAS	29
17. EVALUACIÓN, RECUPERACIÓN Y PROMOCIÓN	29
17.1. DEL ALUMNO	29
17.2. DE LA ACTIVIDAD DOCENTE	30
18. BIBLIOGRAFÍA	30
18.1. INICIACIÓN	30
18.2. PROFUNDIZACIÓN	30
18.3. OTRAS FUENTES DOCUMENTALES	31

1. PROGRAMACIÓN MODULAR

1.1. CONCEPTOS INICIALES

1.1.1. MÓDULO

Un módulo es una colección estática de declaraciones definidas en un ámbito de visibilidad particular y oculto al resto del programa con el que se comunica por una sección de interfaz donde se incluyen la lista de exportaciones. Usando módulos se construyen las unidades en que se ha de descomponer cualquier programa mínimamente importante. Los módulos se conectan entre sí dando lugar a una estructura modular que permite resolver el problema de programación planteado.

La programación modular es un primer paso en la evolución hacia la programación orientada a objetos y por componentes (aunque en un futuro no muy lejano es seguro que surgirán nuevos desarrollos). No obstante, en el presente tema no vamos a hacer más que breves alusiones a la O.O.P. ya que existe un tema específico del programa que estudia esta potente herramienta.

Para efectuar un buen diseño modular los algoritmos que se van a desarrollar se han de concebir como una jerarquía de módulos intercomunicados donde cada uno de ellos presenta una función clara y diferenciada y en la que ningún módulo accede directamente al interior de otros módulos sino que siempre utiliza los mecanismos de interfaz.

Las técnicas de diseño modular se estudian también en otros diversos apartados del presente temario.

1.1.2. INTERFAZ E IMPLEMENTACIÓN

Un módulo actúa como una caja negra con la cual el resto del programa interactúa a través de una sección de interfaz. La interfaz es pues una colección de declaraciones de constantes, tipos, variables, procedimientos, funciones, etc. La otra sección principal de un módulo es la de implementación que incluye el código de los procedimientos y demás elementos constitutivos de la parte ejecutable del módulo. Ambos elementos, interfaz e implementación, también se suelen denominar como vista pública y vista privada de un módulo.

1.1.2.1. Interfaz

Un módulo puede ofrecer a la "comunidad" de módulos sus propios recursos, tipos y procedimientos. Con el fin de controlar qué elementos son exportables al resto se utiliza la versión soportada por el lenguaje de desarrollo del concepto genérico de lista de exportaciones, dentro de lo que se conoce, también genéricamente, como sección de interfaz del módulo.

Normalmente, en lenguajes anteriores a la O.O. es posible exportar cinco tipos de elementos: constantes, tipos, variables, procedimientos y funciones.

La importación y la exportación constituyen acciones simétricas. Para que, tal y como vemos en la Figura 1, un módulo pueda importar un tipo, procedimiento u otro elemento es preciso que otro lo exporte. Por supuesto, es posible que el mismo módulo se comporte como exportador e importador.

El lenguaje C no diferencia explícitamente entre elementos que se importan y elementos que se exportan. El concepto de módulo en C se corresponde con el de fichero. Así, la implementación de un programa en módulos se efectúa distribuyendo el código entre diversos ficheros. Todos los procedimientos y funciones de un fichero son exportables excepto aquellos que contienen la palabra reservada *static* en su declaración. Para crear un módulo en C, lo primero que se ha de hacer es generar un archivo de cabeceras (header), usualmente con la extensión h, por ejemplo *miModulo.h*, que incluirá las declaraciones de todos los elementos que se desean hacer exportables. Seguidamente, se editará el archivo de implementación, al que normalmente se le da la extensión c. En este segundo archivo es preciso incluir las declaraciones del primero y para ello se utilizará la sintaxis:

```
# include "miModulo.h"
```

Por otro lado, C no permite exportar tipos ocultando su implementación, es decir, si se desea exportar un tipo usando un archivo de cabeceras será necesario definir completamente el tipo con lo que la implementación será pública. Esta es una de las limitaciones de C que ha resuelto C++

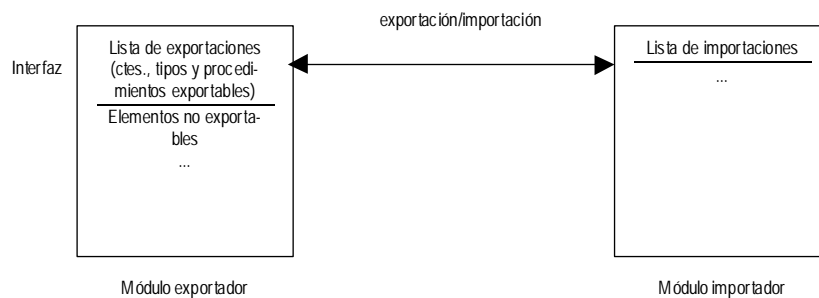


Figura 1 Módulos exportadores e importadores.

Un lenguaje no O.O. que permite exportar tipos (punteros y enumerados) ocultando su implementación es MODULA 2 (usando lo que se denomina exportación opaca). Este lenguaje utiliza los denominados *módulos de definición* para hacer explícito todo lo que los futuros usuarios deben conocer si desean usar los elementos que los módulos exportadores ofrecen al exterior. Veamos un ejemplo en MODULA-2:

```
DEFINITION MODULE MVarios;
(*  lista de exportaciones
=====*)
EXPORT QUALIFIED
EsMayus, ConvAMayus, Visualiza, DMV;
(* declaración de los objetos
===== *)
VAR
DMV : BOOLEAN;
PROCEDURE EsMayus (C : CHAR) : BOOLEAN;
PROCEDURE ConvAMayus (C : CHAR) : CHAR;
PROCEDURE Visualiza (C : CHAR);
END MVarios.
```

en este ejemplo, el módulo *MVarios* hace exportables los procedimientos que devuelven valores (funciones) *EsMayus* y *ConvAMayus*, el procedimiento *Visualiza* y la variable *DMV*.

1.1.2.2. Implementación

En la parte de implementación se detallan las definiciones y el diseño de todos los elementos que el módulo contiene. La implementación puede incluir, asimismo, la lista de importaciones correspondiente a los elementos exteriores utilizados por el módulo.

Como hemos visto, cuando se lleva a cabo un diseño por descomposición modular, suele ocurrir que algunos módulos hacen referencia a la interfaz de otros. Ello puede suceder cuando un módulo usa un tipo declarado en otro sitio o llama a un procedimiento o función que se han definido fuera del propio módulo.

Para servirse de un tipo o procedimiento ajeno a un módulo es obligatorio especificar el lugar donde se han definido originalmente dichos elementos. Además, es conveniente que esa declaración de objetos ajenos al módulo pero usados en él, se lleve a cabo en una sección bien diferenciada. Para tal fin se puede utilizar alguna versión de la denominada lista de importaciones.

Cada lenguaje presenta sus peculiaridades, así, en C, las importaciones se pueden incluir explícitamente declarando los procedimientos y funciones en el mismo módulo o haciendo uso de un archivo de cabeceras con la sintaxis:

```
# include "archivoDeExportaciones.h"
```

donde *archivoDeExportaciones* corresponde al nombre del archivo concreto que se utilice. En cuanto a la implementación se lleva a cabo usando la técnica indicada en 1.1.2.1.

MODULA-2 incorpora un tipo de módulo específico para propósitos de implementación con la sintaxis *IMPLEMENTATION MODULE*. Este tipo de módulos se utiliza, junto con los *DEFINITION MODULE*, cuando se están codificando módulos exportadores, ya que de no ser así, es decir si se va a codificar un módulo que no exporte nada al exterior, se puede usar un módulo del tipo principal o local.

MODULA-2 es un lenguaje que tiene perfectamente definido el concepto de lista de importación. Así, en este lenguaje, para importar variables, constantes, procedimientos (que incluyen también el concepto clásico de funciones) o tipos de otros módulos se utiliza la sintaxis que se expone en el siguiente ejemplo:

```
FROM InOut IMPORT ReadInt, WriteInt, WriteLn ...;
```

donde *InOut* sería el módulo externo y *ReadInt*, *WriteInt* ... los elementos importados. Las listas de importaciones se pueden usar tanto en los módulos de definición como en los de implementación.

Veamos un ejemplo correspondiente a la implementación del visto en 1.1.2.1.

```
IMPLEMENTATION MODULE MVarios;
(*  lista de importaciones
=====*)
FROM InOut IMPORT WriteString, WriteLn;

(* implementación de los objetos
===== *)

PROCEDURE EsMayus (C : CHAR) : BOOLEAN;
BEGIN
RETURN ( C >= 'A' AND C <= 'Z' );
```

```

END EsMayus;

PROCEDURE ConvAMayus (C : CHAR) : CHAR;
BEGIN
    RETURN CAP(C);
END ConvAMayus;

PROCEDURE Visualiza (C : CHAR);
BEGIN
    WriteString (C); WriteLn;
END ConvAMayus;

BEGIN
    DMV := FALSE; (* se ejecuta antes que el módulo principal *)

```

END MVarios.

este módulo utiliza dos procedimientos definidos en el módulo *InOut*.

1.2. REFINAMIENTO Y MODULARIDAD

1.2.1. REFINAMIENTO SUCESIVO

El *refinamiento sucesivo*, propuesto por Niklaus Wirth en 1971, fue una de las primeras estrategias de diseño descendente. En ella, la arquitectura de un programa se desarrolla en niveles sucesivos de refinamiento de los detalles procedimentales. Se desarrolla una jerarquía descomponiendo una declaración macroscópica de una función de forma sucesiva, hasta que se llega a las sentencias del lenguaje de programación. Wirth en su artículo "*Program Development by Stepwise Refinement*" da una visión general del concepto:

"En cada paso (del refinamiento), una o varias instrucciones del programa dado, se descomponen en instrucciones más detalladas. Esta descomposición sucesiva o refinamiento de especificaciones termina cuando todas las instrucciones están expresadas en términos de la computadora usada o del lenguaje de programación... Conforme se refinan las tareas, también los datos pueden ser refinados, descompuestos o estructurados, siendo lo natural refinar las especificaciones del programa y las de los datos en paralelo.

Cada paso de refinamiento implica algunas decisiones de diseño. Es importante que... el programador sea consciente de los criterios subyacentes (en las decisiones de diseño) y de la existencia de soluciones alternativas..."

El refinamiento es, realmente, un proceso de elaboración. Se comienza con una declaración de la función (o una descripción de la información). Es decir, la declaración describe la función o la información conceptualmente, pero no proporciona información sobre el funcionamiento interno de la función o sobre la estructura interna de la información. El refinamiento hace que el diseñador amplíe la declaración original, dando cada vez más detalles conforme se producen los sucesivos refinamientos (elaboraciones).

1.2.2. LA DIVISIÓN EN MÓDULOS Y LA INTELIGIBILIDAD DE LOS PROGRAMAS

El concepto de modularidad se refiere al hecho de que el software se divida en componentes con nombres y ubicaciones determinados, que se denominan "módulos" y que se integran para satisfacer los requisitos del problema. Bertrand Meyer, uno de los grandes teóricos de las técnicas de OO, dijo en una ocasión que la "modularidad es el atributo individual del software que permite a un programa ser intelectualmente manejable" porque facilita la comprensión del programa.

Para ilustrar este punto, consideremos la siguiente disquisición, basada en observaciones sobre la resolución humana de problemas.

Sea $C(x)$ una función que define la complejidad de un problema x y $E(x)$ una función que define el esfuerzo (en tiempo) requerido para resolver un problema x . Para dos problemas, p_1 y p_2 , si

$$C(p_1) > C(p_2)$$

se deduce que

$$E(p1) > E(p2)$$

Para un caso general, este resultado es intuitivamente obvio. Se tarda más tiempo en resolver un problema difícil.

Se ha encontrado otra propiedad interesante, a partir de la experimentación sobre la resolución humana de problemas, es la siguiente

$$C(p1+p2) > C(p1) + C(p2)$$

que indica que la complejidad de un problema compuesto por $p1$ y $p2$ es mayor que la complejidad total cuando se considera cada problema por separado. Se puede deducir que

$$E(p1+p2) > E(p1) + E(p2)$$

indica que es más fácil resolver un problema complejo cuando se divide en trozos más manejables.

De la desigualdad anterior se podría concluir que, si partiéramos el software indefinidamente, el esfuerzo requerido para desarrollarlo sería insignificamente pequeño. Sin embargo conforme crece el número de módulos, el esfuerzo (coste) asociado a los interfaces entre los módulos también crece. Por lo tanto, debe evitarse tanto la modularización excesiva como que ésta quede pobre. Pero, ¿cómo de modular debe hacerse el software? El tamaño de un módulo depende de su función y de su aplicación.

1.2.3. EL DISEÑO DE ARRIBA ABAJO (TOP DOWN) CONCEPTO Y UTILIDAD

El diseño descendente o diseño de arriba abajo (Top Down) utiliza los conceptos de refinamiento y modularidad descritos anteriormente. Consiste en una serie de descomposiciones sucesivas del problema inicial, que describen el refinamiento progresivo del conjunto de instrucciones que van a formar parte del diseño.

La utilización de esta técnica de diseño tiene los siguientes objetivos básicos

- Simplificación del problema y de los bloques resultantes de cada descomposición.
- Las diferentes partes del problema pueden ser diseñadas/desarrolladas de modo independiente e incluso por diferentes personas.
- El diseño final queda estructurado en forma de bloques o módulos, lo que hace más sencilla su implementación y posterior mantenimiento.

La principal ventaja del diseño Top Down es que aminora la dificultad de resolución y posterior mantenimiento de los problemas de diseño. Como desventaja asociada tenemos que, a medida que se divide el problema en sub-problemas y el número de módulos crece, se produce un incremento de los interfaces entre éstos con la consiguiente complejidad asociada.

1.2.4. TIPOS DE MÓDULOS

Los módulos en los que se puede dividir un programa se pueden clasificar atendiendo a dos criterios:

- ◇ Según los mecanismos de activación
- ◇ Según el camino de control

1.2.4.1. Según los mecanismos de activación

Existen dos mecanismos de activación. Convencionalmente, un módulo es invocado mediante *referencia* por ejemplo: una sentencia "de llamada". Sin embargo, en las aplicaciones de tiempo real o en entornos GUI y ejecución dirigida por eventos, un módulo puede ser invocado mediante una interrupción o por una llamada desde el bucle de mensajes del sistema operativo; esto es, un suceso exterior produce una discontinuidad en el procesamiento, que da como resultado el paso de control a otro módulo. Los mecanismos de control son importantes porque pueden afectar a la estructura del programa resultante.

1.2.4.2. Según el camino de control

El camino de control de un módulo describe la forma en la que se ejecuta internamente. Los módulos *convencionales* tienen una única entrada y una única salida y ejecutan secuencialmente una tarea en cada momento. Al-

gunas veces se necesitan caminos de control más sofisticados. Por ejemplo, un módulo puede ser *reentrante*. Esto es, un módulo se diseña de forma que de ninguna manera pueda modificarse a sí mismo o a las direcciones que referencia localmente. Así, el módulo puede ser usado para más de una tarea concurrentemente.

Dentro de una estructura de programa, un módulo puede ser clasificado como

- Un módulo *secuencial* que se referencia y se ejecuta sin interrupción aparente por parte del software de la aplicación.
- Un módulo *incremental* que puede ser interrumpido, antes de que termine, por el software de la aplicación y, posteriormente, restablecida su ejecución en el punto en que se interrumpió. Este tipo de módulo se suele denominar *corrutina*.
- Un módulo *paralelo* que se ejecuta a la vez que otro módulo, en entornos de multiprocesadores concurrentes. Una denominación utilizada para este tipo es *corrutina*.

1.3. OCULTACIÓN DE LA INFORMACIÓN

El concepto de modularidad lleva a todo diseñador de software a plantearse una pregunta fundamental: ¿cómo descomponer una solución de software obteniendo el mejor conjunto de módulos?. El principio de ocultamiento de información propuesto por Parnas sugiere que los módulos se han de "caracterizar por decisiones de diseño que los oculten unos a otros". En otras palabras, los módulos deben especificarse y diseñarse de forma que la información (procedimientos y datos) contenida dentro de un módulo sea inaccesible a otros módulos que no necesiten tal información.

El ocultamiento implica que para conseguir una modularidad efectiva hay que definir un conjunto de módulos independientes, que se comuniquen con los otros sólo mediante la información que sea necesaria para realizar la función del software. La abstracción ayuda a definir las entidades procedimentales (o de información) que componen el software. El ocultamiento establece y refuerza las restricciones de acceso a los detalles procedimentales internos de un módulo y a cualquier estructura de datos localmente utilizada en el módulo.

El uso del ocultamiento de información como criterio de diseño para los sistemas modulares, revela sus mayores beneficios cuando se hace necesario realizar modificaciones, durante la prueba y, más adelante, el mantenimiento del software. Debido a que la mayoría de los datos y de los procedimientos estarán ocultos a otras partes del software, será menos probable que los errores introducidos inadvertidamente durante la modificación se propaguen a otros lugares del software.

1.4. INDEPENDENCIA FUNCIONAL Y CALIDAD DEL SOFTWARE

1.4.1. CONCEPTO

El concepto de independencia funcional es una derivación directa del de modularidad y de los conceptos de abstracción y ocultamiento de información. En sendos artículos sobre diseño de software, Parnas y Wirth aludieron a las técnicas de refinamiento que aumentan la independencia modular. Posteriores artículos de Stevens, Myers y Constantine asentaron el concepto.

La independencia funcional se adquiere desarrollando módulos con "una clara" función y una "aversión" a una excesiva interacción con otros módulos. Dicho de otra forma, se trata de diseñar software de forma que cada módulo se centre en una subfunción específica de los requisitos y tenga una interfaz sencilla, cuando se ve desde otras partes de la estructura del software.

La pregunta que inmediatamente surge es por qué es tan importante la independencia funcional en el desarrollo de aplicaciones informáticas. El software con modularidad efectiva, es decir, con módulos independientes, es fácil de desarrollar porque su función puede ser partida y se simplifican los interfaces (considérense las implicaciones cuando el desarrollo es realizado por un equipo). Los módulos independientes son más fáciles de mantener (y de probar) debido a que se limitan los efectos secundarios producidos por las modificaciones en el diseño/código, se reduce la propagación de errores y se fomenta la reutilización de los módulos. Resumiendo, la independencia funcional es la clave de un buen diseño y el diseño es la clave de la calidad del software.

La independencia se mide con dos criterios cualitativos: la **cohesión** y el **acoplamiento**. La cohesión es una medida de la fortaleza funcional relativa de un módulo. El acoplamiento es una medida de la interdependencia relativa entre los módulos.

1.4.2. COHESIÓN

La cohesión es una extensión del concepto de ocultamiento de información, descrito en 1.3. Un módulo cohesivo ejecuta una tarea sencilla de un procedimiento de software y requiere poca interacción con procedimientos que ejecutan otras partes de un programa. Dicho de forma sencilla, un módulo cohesivo sólo hace (idealmente) una cosa.

La cohesión, al ser una magnitud más bien difusa, puede presentarse como un "espectro". Lo deseable siempre es conseguir una gran cohesión, aunque un punto medio del espectro normalmente es aceptable. La escala de cohesión no es lineal. Es decir, una cohesión baja es mucho "peor" que una de la mitad del espectro, que, a su vez, es casi tan "buena" como una gran cohesión. En la práctica, un diseñador no tiene que preocuparse por el grado preciso de cohesión de un módulo específico. En vez de ello, debe comprender el concepto general y evitar los niveles bajos de cohesión en el diseño de los módulos.

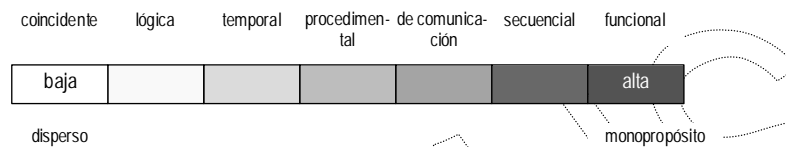


Figura 2 Espectro de la cohesión modular.

Un módulo que realice un conjunto de tareas que están débilmente relacionadas entre sí, si es que lo están, es *coincidentalmente* cohesivo. Un módulo que realiza tareas que están relacionadas de forma lógica (p. ej.: un módulo que produzca toda la salida, independientemente de su tipo) es *lógicamente* cohesivo. Cuando un módulo contiene tareas que están relacionadas por el hecho de que se ejecutan en el mismo momento, se dice que tiene cohesión *temporal*.

Los niveles moderados de cohesión están relativamente cercanos unos a otros en su grado de independencia modular. Cuando los elementos de procesamiento de un módulo están relacionados y deben ejecutarse en un orden específico, existe cohesión *procedimental*. Cuando todos los elementos de procesamiento se concentran sobre un área de una estructura de datos, se presenta una cohesión de *comunicación*.

Un módulo con cohesión *secuencial* es aquel cuyos elementos desempeñan tareas tales que los datos de salida de una sirven como datos de entrada para la siguiente. Por último, un módulo tiene cohesión *funcional* si en su seno se ejecuta una única tarea.

1.4.3. ACOPLAMIENTO

El acoplamiento es una medida de la interconexión entre los módulos de una estructura de programa y está totalmente vinculado al concepto de ocultación de la información. Al igual que la cohesión, el acoplamiento puede asimilarse a un "espectro". El acoplamiento depende de la complejidad de los interfaces entre los módulos, del punto en el que se hace una entrada o referencia a un módulo y de los datos que pasan a través de la interfaz.

En el diseño de software buscamos el más bajo acoplamiento posible. La conectividad sencilla entre módulos da como resultado un software que es más fácil de comprender y menos propenso al "efecto onda o colateral" producido cuando los errores aparecen en una posición y se propagan a lo largo del sistema.

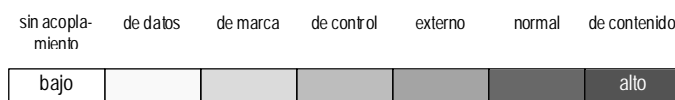


Figura 3 Espectro del acoplamiento entre módulos.

Mientras un módulo presente una lista de parámetros simples (no estructurados), se tiene un bajo acoplamiento (acoplamiento de *datos* en el espectro). Una variación del acoplamiento de datos, denominado acopla-

miento por *estampado*, se presenta cuando se pasa una porción de una estructura de datos (en vez de argumentos simples) a través de una interfaz de módulo.

En niveles moderados, el acoplamiento se caracteriza por el paso de control entre módulos. El acoplamiento de *control* es muy frecuente en la mayoría de los diseños de software. En su forma más sencilla, existe acoplamiento de control si un módulo pasa a otro un elemento de información con el objeto de controlar su lógica interna.

En el denominado acoplamiento *externo* los módulos están ligados a un entorno externo al software. Un ejemplo es la gestión de E/S donde a cada dispositivo se le asocian módulos específicos. Este tipo de acoplamiento es esencial pero debe reducirse al mínimo siendo altamente recomendable dejar en manos de los sistemas operativos toda la relación de bajo nivel con equipos físicos. Otro tipo de alto acoplamiento, denominado *común*, es el que se produce cuando dos módulos se refieren a un área de datos común a ambos. Por ejemplo, en el entorno operativo Windows 3.x, existe acoplamiento común entre todos los programas Windows que se ejecutan en un momento dado ya que todos comparten la misma cola de mensajes del sistema. Así, si un proceso corrompe dicha cola o interrumpe su funcionamiento, el resto de los procesos en ejecución también sufrirá las consecuencias.

El mayor grado de acoplamiento es el acoplamiento por *contenido* que ocurre cuando un módulo se refiere, de algún modo, al interior de otro. Se trata de una violación total del principio de ocultación de la información y produce, generalmente, resultados indeseables sobre las características funcionales del software diseñado.

1.5. ESTRUCTURA DEL PROGRAMA Y JERARQUÍA DE CONTROL

1.5.1. ORGANIZACIÓN DE MÓDULOS EN NIVELES Y TRANSFERENCIAS DE CONTROL

La *jerarquía de control*, también denominada *estructura del programa*, representa la organización (frecuentemente jerárquica) de los componentes del programa (módulos) e implica una jerarquía de control. No representa pues, aspectos procedimentales del software, tales como la secuencia de procesos, la ocurrencia u orden de decisiones o la repetición de operaciones.

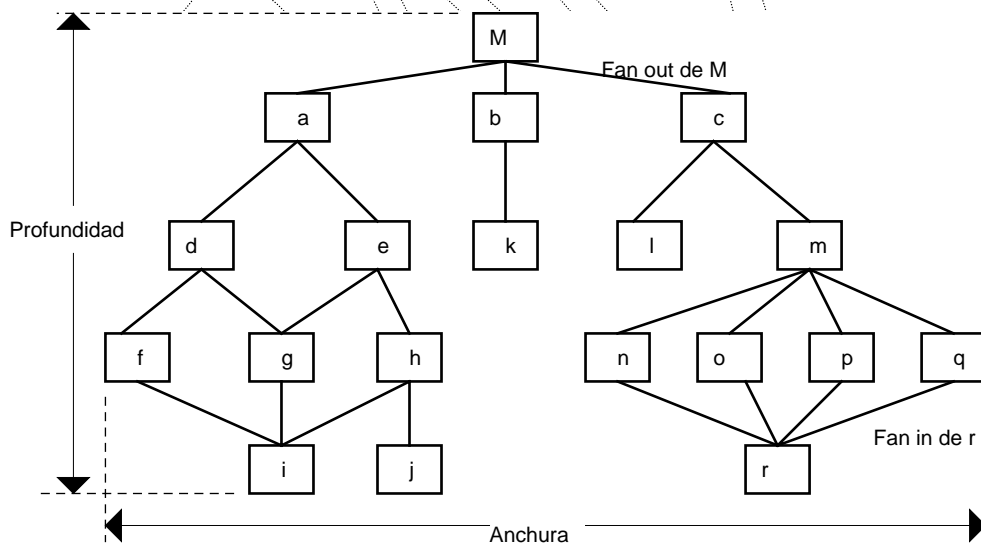


Figura 4. Árbol de estructura con términos usados en dicha técnica.

Para representar la jerarquía de control se utilizan muchas notaciones diferentes. La más común es un diagrama en forma de árbol denominado *diagrama de estructura*, como el que muestra la Figura 4 que también presenta alguno de los términos usados. Sin embargo, también se pueden utilizar otras notaciones igualmente efectivas, tales como los diagramas de Warnier-Orr y de Jackson. Para facilitar posteriores tratamientos de la estructura, vamos a definir algunas medidas simples y algunos términos sencillos. Refiriéndonos a la figura, la profundidad y la anchura son una indicación del número de niveles de control y de la amplitud global del control, respectivamente.

El *grado de salida (fan out)* es una medida del número de módulos que están directamente controlados por otros módulos. El *grado de entrada (fan in)* indica cuántos módulos controlan directamente a un módulo dado.

Las relaciones de control entre los módulos se expresan de la siguiente forma: un módulo que controla a otro módulo se dice que es superior a él, e inversamente, un módulo controlado por otro se dice que es un subordinado del controlador. Por ejemplo, refiriéndonos a la figura, el módulo *M* es superior a los módulos *a*, *b* y *c*. El módulo *h* es subordinado del módulo *e* y en última instancia es subordinado del módulo *M*. Las relaciones en función de la anchura (p. ej.: entre los módulos *d* y *e*), aunque se pueden expresar en la práctica, no es necesario definir las con una terminología explícita.

La jerarquía de control también representa dos características, sutilmente diferentes, de la arquitectura del software: la *visibilidad* y la *conectividad*.

La visibilidad (*scope*) indica el conjunto de componentes del programa a los que un módulo dado puede invocar o utilizar sus datos, incluso cuando lo haga indirectamente. Se trata pues de un concepto relacionado en parte con la ocultación de la información. Así, un dato es visible por un módulo dado cuando desde éste se puede usar o variar su valor y, un módulo es visible desde otro, cuando este último puede invocar la ejecución de aquél.

La conectividad indica el conjunto de componentes a los que directamente se invoca o de los que se utilizan sus datos en un determinado módulo. Por ejemplo, un módulo que en un momento dado provoca la ejecución de otro módulo, está conectado a ese último.

Podemos concluir diciendo que la visibilidad es pues una medida de la conectividad potencial de un programa y que cuanto mayores sean estas magnitudes menor es el nivel de ocultamiento de la información y por tanto más elevadas las posibilidades de que aparezcan los molestos efectos laterales.

1.5.2. PRINCIPIOS DE UN BUEN MÉTODO DE DISEÑO SEGÚN MEYER

Bertrand Meyer, uno de los principales teóricos de las técnicas orientadas a objetos, indica en [Meyer 99] cuáles son, a su juicio, los criterios que es necesario aplicar para obtener un buen diseño modular.

1.5.2.1. Principio de descomponibilidad modular

Un diseño modular cumplirá este criterio si facilita la fragmentación de un problema en varios subproblemas o, en otro nivel conceptual, favorece que los módulos de que se compone sean particionables en submódulos, y éstos, a su vez, en otros, hasta llegar a un nivel que permita aprehender y abordar su tratamiento individual.

1.5.2.2. Principio de componibilidad modular

Se trata aquí del grado de facilidad con que pueden combinarse de forma flexible (aunque no siempre con total independencia) los módulos de que se compone un diseño modular. Este criterio está directamente relacionado con la posibilidad de reutilización de componentes para la creación de nuevos sistemas software en diferentes entornos y escenarios. De hecho, en justa correspondencia, la reutilización ha sido definida como "la capacidad efectiva de incorporar objetos creados para un sistema software dentro de un sistema software diferente". Hay que tener en cuenta, empero, que el criterio de descomponibilidad no conduce necesariamente a éste, pues la mera descomposición suele abocar -como ocurre en el enfoque top-down- a módulos no reutilizables (cuestionémoslo, si no, cómo podríamos "recomponer" a un ser vivo anteriormente descuartizado). La idea que, en definitiva, subyace en las ideas expuestas es la del uso de distintos repositorios de módulos (quizás modelados como OODBMS's) que permitan la construcción independiente de sistemas software.

1.5.2.3. Principio de comprensibilidad modular

De poco sirve una estructura o configuración modular si ésta no puede ser parcialmente asignada o comprendida por un observador externo. Si atendemos, por otro lado, a la consideración de que el límite humano medio para el procesamiento de información, en el marco de la memoria inmediata, se sitúa en tres distintos conjuntos conteniendo un máximo de tres ítems cada uno, la necesidad de poder extraer de un diseño modular un pequeño subsistema comprensible para el lector se torna aún más perentoria. El presente criterio ilustra, pues, la necesidad de que los módulos en un diseño modular sean autoexplicativos o, a lo sumo, extiendan su comprensibilidad a alguno de los módulos adyacentes.

1.5.2.4. Principio de continuidad modular

De forma parecida a como ocurre con la continuidad de funciones $y=f(x)$, donde, informalmente, puede decirse que un pequeño cambio en x produce un necesariamente pequeño cambio en y , subyace bajo este criterio una idea que se ha totemizado, y con razón, en los manuales de estilo de diseño modular: "pequeños cambios han de originar pequeñas repercusiones".

1.5.2.5. Principio de protección modular

Se pretende que la propagación en tiempo de ejecución de un error en un módulo resulte local a éste o, como máximo, se extienda a algunos de los módulos adyacentes.

2. DISEÑO DE FUNCIONES

Uno de los componentes que los lenguajes estructurados incorporan son un tipo de secuencias algorítmicas individualizadas que pueden recibir o no valores de entrada y que también pueden devolver o no valores de salida. Se trataría de sustituir todo un conjunto de instrucciones que puede incluir cualquier combinación de las denominadas estructuras básicas (secuencial, condicional, iterativa) por un identificador que puede incorporar la declaración de valores. Posteriormente, en cualquier lugar del programa se podrá invocar al conjunto de instrucciones nominado utilizando el identificador. Tras la ejecución, y según el tipo de estructura utilizaza, se pueden obtener resultados devueltos.

El título dado al tema en el programa oficial distingue entre procedimientos y funciones. Esta diferencia tiene su origen en las especificaciones de algunos lenguajes estructurados como PASCAL. Sin embargo, procedimientos y funciones se diferencian en muy poca cosa. Tan es así, que en realidad, para conseguir los objetivos de modularidad y reutilización de código, un lenguaje sólo necesitaría contar con alguno de los dos tipos. No obstante, contar con los dos es bastante más cómodo. Es frecuente utilizar el nombre de subprogramas para hacer referencia tanto a procedimientos como a funciones.

2.1. PROCEDIMIENTOS

Como ya hemos indicado, un procedimiento sirve para definir partes de un programa mediante la asociación de un identificador. Posteriormente dichas partes se pueden activar utilizando sentencias de llamada. Un procedimiento es pues un algoritmo diseñado de tal modo que es susceptible de ser llamado por otros algoritmos que, a su vez, pueden ser procedimientos. Al algoritmo que llama se le suele denominar algoritmo principal o, en su caso, procedimiento principal.

2.1.1. LLAMADA A PROCEDIMIENTOS

Para que un algoritmo A llame a otro B , dentro del código de A tendrá que incluirse una instrucción de llamada a B . La llamada provocará la ejecución de B que trabajará sobre los valores que le indiquemos y nos devolverá los resultados que se haya previsto en su diseño. En tiempo de ejecución, las instrucciones que siguen a la llamada al procedimiento contarán ya con los resultados devueltos.

La llamada a un procedimiento se realiza escribiendo su nombre seguido de las expresiones sobre las que deseemos que trabaje. Estas van encerradas entre paréntesis y en el orden en que han sido especificadas en el procedimiento llamado. Para ilustrar lo dicho supongamos que tenemos el procedimiento `visuSeg` que recibe una cantidad de horas, minutos y segundos y efectúa la conversión a segundos para su visualización.

Si en el procedimiento la declaración de variables ha sido:

`h, m, s : entero`

entonces la llamada al procedimiento, se hará

`vi suSeg(hora, minuto, segundo)`

lo que producirá una asignación de las variables usadas en la llamada con las usadas en la definición según la posición que ocupan:

`h ↔ hora`

m ↔ minuto
s ↔ segundo

2.1.2. PARÁMETROS

2.1.2.1. Concepto

Tanto las variables usadas en la definición del procedimiento como los valores utilizados en la llamada se denominan parámetros. A los primeros se los denomina parámetros formales (en el ejemplo anterior *h, m, s*) y a los segundos parámetros reales (en el ejemplo *hora, minuto, segundo*). Los parámetros reales, según los casos, pueden ser constantes, variables definidas en el ámbito del procedimiento llamante o expresiones (mezcla de constantes, variables y operadores).

La lista de parámetros formales de un procedimiento $\{pf_1, pf_2, \dots, pf_n\}$ y la de parámetros reales de cualquier llamada al mismo $\{pr_1, pr_2, \dots, pr_m\}$ deben cumplir las siguientes condiciones:

- $m = n$
- Dadas una pareja de parámetros que ocupan la misma posición en sus respectivas listas pf_i y pr_i su tipo debe ser igual o, al menos, compatible. Sin embargo, su nombre no tiene por qué ser igual.

2.1.2.2. Clases

En función de su papel en el procedimiento, los parámetros, formales o reales, pueden ser de tres clases:

- Parámetros de entrada: son aquellos que se utilizan para aportar datos al procedimiento. Si dentro de éste se produce un cambio en el valor del parámetro formal el parámetro real no se verá afectado. Los parámetros reales en este caso pueden ser constantes, variables o expresiones.
- Parámetros de salida: son aquellos que se utilizan para exportar datos desde el procedimiento. No aportan valor inicial por lo que son directamente inicializados por el procedimiento que les asigna valores. Así, los cambios producidos en el valor del parámetro formal afectarán al parámetro real que deberá ser una variable.
- Parámetros de entrada/salida: son aquellos cuya función incluye a las dos anteriores. Por un lado aportan valores y por otro son modificados por el procedimiento para exportar valores. Los parámetros reales tienen también que ser variables.

La clase a la que pertenece cada parámetro la establece el programador a partir de las características del procedimiento.

2.1.3. DEFINICIÓN DE PROCEDIMIENTOS

Para definir un procedimiento se utilizará la sintaxis propia del lenguaje de programación que se esté utilizando. En general, todos los lenguajes suelen dividir esta definición en dos partes:

- Cabecera o interfaz: incluye el identificador del procedimiento usualmente precedido de una palabra reservada tal que *proc*, y la lista de parámetros formales con cero o más parámetros. En esta lista se indica el tipo de los parámetros y su clase. Para la clase se debe emplear alguna notación específica. Nosotros, a efectos de explicación, utilizaremos las siguientes palabras reservadas que precederán a los parámetros:
 - ◊ De entrada, van precedidos por la palabra reservada *ent*.
 - ◊ De salida, por la palabra *sal*.
 - ◊ De entrada/salida, por la palabra *entSal*.
- Cuerpo: el cuerpo del procedimiento lo componen las declaraciones e instrucciones en las que se ejecuta el algoritmo propio del procedimiento.

¹ Es frecuente encontrar textos que llaman declaración a lo que aquí se denomina definición y viceversa. Nuestra opción está fundada en la denominación empleada por reputados expertos y en cuestiones de índole semántica. Así, en castellano y en el contexto en que nos encontramos, una definición es una "proposición que expone con claridad y exactitud los caracteres genéricos y diferenciales de una cosa material o inmaterial" y no un breve esquema de sus principales características.

Veamos un ejemplo: supongamos un procedimiento que devuelva la posición, p , ocupada dentro de los n primeros elementos de un vector $a[1..1000]$ por el primer elemento igual a un valor, v , que se suministra. Si el valor no se encuentra devolverá un cero:

Especificación:

```
a : vector [1..1000] de entero
v, n : entero
{P} ≡ { a = A ∧ v = V ∧ n = N ∧ (1 ≤ n ≤ 1000) }
operaciones
{Q} ≡ { (pos = 0) ∨ ((1 ≤ pos ≤ n) ∧ (∀ α ∈ {1..pos-1}. a[α] ≠ v) ∧ (a[pos] = v)) }
```

Procedimiento:

```
proc posPrimerValor (ent a : vector [1..1000] de entero; ent v, n : entero; sal pos
: entero)
{Pre: a = A ∧ v = V ∧ n = N ∧ (1 ≤ n ≤ 1000)} /* Ver nota2 */
var
j : entero
fvar
j := 0; pos := 0;
mientras (pos = 0) ∧ (j ≠ n) hacer
j := j + 1;
si
a[j] = v → pos := j
fsi
fmientras;
{Post: (pos = 0) ∨ ((1 ≤ pos ≤ n) ∧ (∀ α ∈ {1..pos-1}. a[α] ≠ v) ∧ (a[pos] = v))}
fproc
```

Como vemos, comenzamos y terminamos con las palabras reservadas *proc* y *fproc* entre las que se incluye:

1. Cabecera
2. Precondición
3. Cuerpo
4. Postcondición

Por ejemplo, si el procedimiento principal (procedimiento en el que buscamos un valor específico en una tabla de equivalencias) es

```
var
...
tablaEquiv : vector [1..1000]
valorABuscar, posicion, iteraciones: entero
...
fvar
...
{suponemos cargada tablaEquiv con valores válidos ∧ valorABuscar = 146514 ∧ n =
100}
posPrimerValor (tablaEquiv, valorABuscar, iteraciones, posicion);
...
```

entonces los valores iniciales de a , v y n , indicados en la precondición por A , V y N , serán $A = \text{tablaEquiv}$, $V = 146514$ y $N = 100$.

La llamada a un procedimiento afecta exclusivamente a sus parámetros de entrada/salida o de salida. Si los datos de la llamada cumplen la precondición, entonces el estado posterior a la llamada cumplirá su postcondición, aplicada sobre los parámetros reales, y sobre aquellas propiedades que no se hayan visto alteradas por no involucrar parámetros modificados por el procedimiento.

En el procedimiento anterior, dentro de la sección de declaración de variables se declara a j . Esta es una variable local, es decir su visibilidad cae únicamente dentro del ámbito del procedimiento y toma valores únicamente durante la ejecución de éste. Al producirse la llamada al procedimiento las variables locales del mismo toman valores. Cuando el procedimiento termina su ejecución, las variables locales son eliminadas y el espacio de memoria que ocupaban es restituido al heap de la aplicación. El que el ámbito de estas variables sea local significa que aunque en otro punto del programa haya variables homónimas, éstas deben considerarse diferentes a las locales al procedimiento las cuales, si no se especifica otra cosa, prevalecerán sobre las exteriores.

² Algunos autores incluyen en la precondición la sustitución de los parámetros formales por los reales. Otros consideran esta sustitución obvia y no la incluyen en la precondición.

2.1.4. DECLARACIÓN DE PROCEDIMIENTOS

En programación real, es frecuente distinguir entre definición, donde se incluye no sólo el interfaz sino también el cuerpo del procedimiento, de declaración. Ésta incluye sólo la cabecera y su utilización no exige de efectuar la definición, que se puede llevar a cabo en el propio programa (en este caso se hará después de la declaración) o en otro programa diferente que será vinculado al actual cuando se produzca la compilación o el montaje final de la aplicación.

Un caso en el que siempre es necesario declarar primero y luego definir es cuando dos procedimientos se llaman mutuamente y se está usando un compilador de una única pasada. Si no declaramos previamente al menos uno de ellos, siempre se producirá un error al compilar el código del primer procedimiento ya que el compilador intentará acceder al código del segundo procedimiento, invocado desde el primero, código que todavía no se ha definido.

Como ejemplo veamos la declaración del anterior procedimiento:

```
proc posPrimerValor (ent a :vector [1..1000] de entero; ent v,n :entero; sal pos
: entero);
```

2.2. FUNCIONES

2.2.1. CONCEPTO

Una función es un procedimiento con unos parámetros con características peculiares: a excepción de un parámetro de salida, todos los demás son de entrada. El parámetro de salida sirve para albergar el valor devuelto por la función.

2.2.2. LLAMADA A FUNCIONES

No es casual que las funciones tengan un único parámetro de salida. De hecho, las funciones de salida tienen sentido como clase de subprograma diferente a los procedimientos en que calculan un valor, valor que se asigna al parámetro de salida y al que desde fuera de la función se accede usando el identificador de la propia función empleado en la llamada. Es por ello que en la llamada a una función el parámetro de salida no debe aparecer en la lista de parámetros. Esta estructura, además de facilitar la escritura de las aplicaciones, evita efectos laterales indeseados que se pueden producir cuando se usan parámetros de e/s típicos de los procedimientos.

Por ejemplo, si necesitamos una función, similar al procedimiento de 2.1.1 que devuelva el número de segundos equivalente a *hor* horas, *min* minutos y *seg* segundos, y lo asigna a la variable *totalSegundos* ejecutaremos la siguiente llamada

```
total Segundos := equivSeg(hor, mi n, seg)
```

donde *equivSeg* es el identificador de la función. Si en la función la declaración de variables ha sido:

```
h, m, s : entero
```

entonces se producirá una asignación de las variables usadas en la llamada con las usadas en la definición según la posición que ocupan:

```
h ↔ hor
m ↔ mi n
s ↔ seg
```

La instrucción de llamada evalúa la función y realiza una asignación interna al identificador que luego puede ser asignado, en la misma llamada, a una variable. Las llamadas a funciones pueden también aparecer en expresiones complejas diferentes a la asignación. El requisito que se debe siempre cumplir es que el tipo del valor que la función retorna sea compatible con el requerido en la expresión. A este tipo del valor devuelto se le conoce como tipo de la función.

2.2.3. DEFINICIÓN DE FUNCIONES

La definición de funciones es muy parecida a la de los procedimientos. No obstante, existen una serie de diferencias que se recogen a continuación:

- Las palabras reservadas de inicio y fin son, respectivamente, *func* y *ffunc*.
- Como hemos dicho, la función tiene un tipo igual al del valor que devuelve. Para especificar este tipo, en la definición de la función, lo añadiremos al final de la cabecera, tras la lista de parámetros y utilizando la sintaxis normal de especificación de tipos, es decir, dos puntos, ':', seguidos del tipo devuelto.
- Al ser todos los parámetros de entrada, en la lista de parámetros formales no se utiliza la etiqueta *ent*.
- El valor devuelto por la función se indica usando la palabra reservada *retorna* seguida por la expresión correspondiente a dicho valor. Es usual adoptar el convenio de que esta instrucción aparezca únicamente al final de la función y que sea la única forma de terminar la función.

Cuando el resultado devuelto por una función se asigna a una variable, la postcondición equivale a la postcondición de la función substituyendo los parámetros formales por los reales, y la expresión devuelta por la variable a la que se asigna.

Veamos el ejemplo de 2.1.3 resuelto con funciones.

```
func posPrimerValor (a : vector [1..1000] de entero; v, n : entero) : entero
  {Pre: a = A ∧ v = V ∧ n = N ∧ (1 ≤ n ≤ 1000)}
  var
    j : entero
  fvar
    j := 0; pos := 0;
  mi entras (pos = 0) ∧ (j ≠ n) hacer
    j := j + 1;
  si
    a[j] = v → pos := j
  fsi
  fmi entras;
  retorna pos
  {Post: (pos = 0) ∨ ((1 ≤ pos ≤ n) ∧ (∀ α ∈ {1..pos-1}. a[α] ≠ v) ∧ (a[pos] = v))}
ffunc
```

Como vemos el resultado es muy parecido al del procedimiento. Ahora, el programa principal quedará del siguiente modo:

```
var
  ...
  tablaEquiv : vector [1..1000]
  valorABuscar, posicion, iteraciones : entero
  fvar
  ...
  posicion := posPrimerValor (tablaEquiv, valorABuscar, iteraciones);
  ...
```

En cuanto a la clase de parámetros formales, hay lenguajes, como PASCAL y C que permiten que en una definición de función se usen no solamente parámetros de entrada sino también de salida y de entrada/salida. Sin embargo, es propio de un estilo de programación correcto utilizar en las funciones sólo parámetros de entrada.

2.2.4. DECLARACIÓN DE FUNCIONES

Al igual que los procedimientos y por los mismos motivos las funciones también se pueden declarar. Así, la declaración de la función ejemplo del anterior apartado sería:

```
func posPrimerValor (a : vector [1..1000] de entero; v, n : entero) : entero;
```

3. LA RECURSIVIDAD

3.1. CONCEPTO

3.1.1. RECURSIVIDAD EN ALGORITMOS

Existe recursividad en algoritmos cuando un algoritmo se invoca a sí mismo o es invocado en otro algoritmo previamente llamado. La recursividad requiere dos condiciones para su correcto funcionamiento:

- Las sucesivas invocaciones deben ser efectuadas con versiones cada vez más reducidas del problema inicial.
- Debe existir una condición de fin de las llamadas o fin de la recursividad, sin esta condición de terminación, el algoritmo no podría construirse siguiendo esta técnica y su ejecución produciría un ciclo infinito.

La recursividad y la iteración son los dos mecanismos suministrados por los lenguajes de programación para describir cálculos que han de repetirse un cierto número de veces.

Pese a la limitada atención que la literatura científica informática ha prestado hasta hace poco a la recursividad³, no hay duda de que las definiciones recursivas y las demostraciones por inducción son, en muchos casos, el método más natural de describir funciones y tipos de datos. Véase si no el caso de la definición de factorial o tipos de datos como los árboles y las listas. Por otro lado, cualquier algoritmo iterativo puede ser expresado en forma recursiva. De hecho, en la mayoría de los casos, las versiones recursivas serán más compactas y de comprensión más sencilla.

Ejemplos de este tipo de algoritmos son funciones como el factorial de un número o generar la serie de números de Fibonacci (Leonardo de Pisa). Veamos este caso que, paradójicamente, es también una demostración de un uso inadecuado de la recursividad.

EJEMPLO: números de Fibonacci

$$(F(i) = F(i - 1) + F(i - 2))$$

Un algoritmo recursivo sería:

```
función fib (n: entero) : entero;
inici o
  si (n ≤ 1) then
    fib := 1
  si no
    fib := fib(n-1) + fib(n-2)
  fsi
fin;
```

Como se ha dicho, el ejemplo puesto no es el mejor caso de aplicación de la recursividad (en el formato usado ya que existen variaciones que utilizan cálculo matricial y recursividad que sí son mucho más rápidas). La razón de que este algoritmo recursivo sea tan lento radica en que se repiten las llamadas a algunas funciones. Así para calcular $F(N)$ se realiza la llamada a $F(N-1)$ y $F(N-2)$, ahora como $F(N-1)$ hace una llamada a $F(N-2)$ y $F(N-3)$, hay ya dos llamadas distintas para calcular $F(N-2)$. De este modo por cada llamada se generan otras dos una de las cuales puede estar repetida. Por todo ello la velocidad de crecimiento es casi 2^n es decir nos encontramos con un algoritmo $O(2^n)$ (en realidad se trata de $O(\phi^n)$ donde ϕ es 1,6180339... es decir 1 + la proporción áurea)

Otro ejemplo es la búsqueda binaria en un vector ordenado. Se trata de ir buscando en la mitad superior o inferior del vector según que el valor buscado sea mayor o menor que el elemento central de dicho vector. Veamos el código en PASCAL:

```
CONST tamMax = 1000;
TYPE
  tipoElemento = ...;
  vector = ARRAY [1..tamMax] of tipoElemento;
...
FUNCIÓN buscaElem (v : vector; inicio, final : integer; x : tipoElemento) : integer;
```

³ En los textos [Peña 98] y [Brassard 97] hay estudios bastante interesantes sobre la recursión, el paso de algoritmo recursivo a iterativo, el cálculo del coste computacional de los algoritmos recursivos, etc.

```

{ EFECTO: devuelve la posición (valor de índice) del elemento x dentro del rango
del vector ordenado v determinado por los índices inicio y final. Devuelve un 0
si no lo encuentra. Se supone que el vector no tiene valores repetidos}
VAR
tamVent, medio : integer;
BEGIN
IF (x < v[inicio]) OR (x > v[final]) OR (inicio > final) THEN
  BuscaElem := 0 { no se encuentra }
ELSE BEGIN
  medio := (inicio + final) DIV 2;
  IF x = v[medio] THEN
    buscaElem := medio { se encuentra el valor }
  ELSE BEGIN
    {el siguiente IF ELSE selecciona la mitad inferior o superior de v}
    IF x < v[medio] THEN final := medio - 1
    ELSE inicio := medio + 1;
    buscaElem := buscaElem(v, inicio, final, x) {llamada recursiva}
  END
END
END; {buscaElem}

```

3.1.2. RECURSIVIDAD EN DATOS

La recursividad en datos consiste en definir tipos de datos que incluyen en su definición el tipo definido. Esta construcción es muy útil y se utiliza profusamente en las estructuras denominadas "dinámicas". Así, por ejemplo, la definición en pseudo PASCAL de una lista doblemente encadenada sería algo así como:

```

TYPE
  tipoElemento = ....;

  lista = ^nodo;

  nodo = REGISTRO
    elemento : tipoElemento;
    siguiente, anterior : lista
END;

```

Es en el tratamiento de estos datos donde la recursividad en algoritmos resulta el mejor método, con unos planteamientos mucho más claros y concisos que las correspondientes versiones iterativas.

3.2. RAZONAMIENTO RECURSIVO

Aunque con ambos se puede obtener el mismo resultado, los procedimientos de razonamiento recursivo difieren notablemente de los propios del método iterativo. Así, si se plantea un problema de cálculo *PC* consistente en efectuar cierta operación sobre unos datos de partida *D*, la solución iterativa consistirá en plantear un problema alternativo, *pc*, más sencillo que el inicial, sobre unos datos también más sencillos, *d*, problema que, una vez resuelto, se aplicará un cierto número de veces *n*, sobre los datos *d*, hasta conseguir resolver el problema original.

El razonamiento recursivo es bastante distinto. Se trata de preguntarnos si sería posible resolver *PC* sobre *D* suponiendo que ya se ha resuelto para otros datos *D'* del mismo tipo que los originales *D* pero, de algún modo, más sencillos. Así, se obtiene una relación de recurrencia entre los datos que permite resolver el problema recurriendo reiteradas veces a versiones más sencillas del mismo tipo de datos. Como vemos, se trata de aplicar el mismo cálculo sobre datos del mismo tipo. Con ello, el número de elementos envueltos en la solución del problema es menor que en el caso de la iteración.

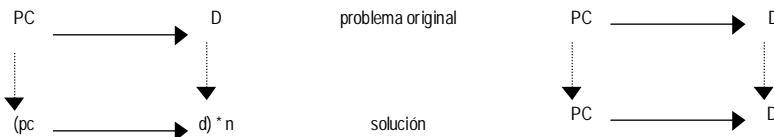


Figura 5 Soluciones iterativa y recursiva al mismo problema.

Un ejemplo al caso anterior puede ser elevar un número *a* a una potencia *p*. El problema *PC* es pues calcular a^p y los datos de partida son *a* y *p*. En el caso del planteamiento iterativo el problema se traduce en tomar *p* veces el producto $a \cdot x$ donde *x* es un acumulador cuyo valor inicial es $a^0 = 1$. Así, $a^0 = 1$, $a^1 = a \cdot 1 = a$, $a^2 = a \cdot (a \cdot 1) = a \cdot a = a^2$.

En efecto, el problema de multiplicar un número por otro es de naturaleza diferente al de la exponenciación.

El razonamiento recursivo para el mismo ejemplo es diferente. Suponiendo que sabemos cómo calcular a^{P-1} será fácil calcular a^P . Para ello es suficiente con obtener el producto $a^P = a * a^{P-1}$. El nuevo problema, a^{P-1} , es exactamente del mismo tipo que a^P pero más sencillo. El mismo método empleado en resolver D a partir de D' se puede aplicar para resolver D' en base a unos datos D'' más sencillos. Así, se forma una sucesión $D > D' > D'' \dots$ donde los datos son cada vez más simples. Como ya hemos indicado, esta sucesión ha de ser finita. Para ello es preciso llegar a unos datos D_n tan sencillos como para ser resueltos directamente sin aplicar PC. Esta solución, conocida como caso trivial, nos permitirá ir conociendo los resultados, en sentido inverso al de la sucesión formada, es decir desde D_n , ... hasta D'' , D' y D con lo que obtendremos el resultado final. En nuestro ejemplo tenemos un caso trivial:

- $a^0 = 1$

Así, $a^1 = a * a^0 = a * 1 = a$, $a^2 = a * a^1 = a * a * a^0 = a * a$, etc.

De todas formas, al final, las expresiones algebraicas desplegadas de la versión iterativa y de la versión recursiva son, como hemos visto, exactamente iguales.

3.3. RECURSIÓN E ITERACIÓN

Todo algoritmo recursivo puede convertirse en su equivalente iterativo. Así, por ejemplo, en el caso de los números de Fibonacci como para calcular $f(N)$ todo lo que se necesita es $f(N-1)$ y $f(N-2)$, solamente es necesario registrar los dos números de Fibonacci calculados más recientemente. Un algoritmo más eficiente que calculase, usando programación iterativa, hasta los n primeros números de Fibonacci, sería el siguiente:

```

procedimiento fib(n: entero; fi bo ent/sal: vector[1.. n] de entero);
var i, ult, penult, rep: entero;
inicio
  si n>1 entonces
    fi bo[2] := 1; fi bo[1] := 1
    si n>2 entonces
      para i := 3 hasta n hacer
        fi bo[i] := fi bo[i - 1] + fi bo[i - 2]
    fsi
  si no
    si n>0 entonces
      fi bo[1] := 1
    si no
      error
    fsi
  fsi
fin;

```

Donde el parámetro `fi bo` lo hemos definido como de entrada/salida (lo que en Pascal se indica con `var` dentro de la lista de parámetros). El coste computacional de este algoritmo está definido por $O(n)$.

En el segundo ejemplo de 3.1.1, y en general en todos aquellos en que la única llamada recursiva está al final del procedimiento, la transformación a algoritmo no recursivo es bastante directa y, como es usual, incrementa la eficiencia del proceso. Dejamos como ejercicio dicha transformación en el caso del ejercicio de la búsqueda binaria.

3.4. COSTE DE LAS FUNCIONES RECURSIVAS

El tiempo de cómputo no es el único factor de coste con el que tenemos que llevar cuidado. Por supuesto, los tiempos de desarrollo y la longitud y claridad del código son importantes, pero son temas más relacionados con la ingeniería del software que con la algoritmia. Sin embargo, un tema que sí es interesante analizar es el consumo de memoria del programa. Si un programa tarda mucho en ejecutarse, la solución más a mano es simple, tómesese un descanso y espere a que termine. Ahora bien, si el programa consume demasiada memoria es posible que ni tan siquiera pueda ejecutarse. De ahí la importancia de controlar también este factor.

De nuevo, los programas recursivos e iterativos difieren. En un programa iterativo es fácil conocer la memoria consumida. Basta con mirar las declaraciones de variables y el posible uso de asignaciones dinámicas de memoria. Así, en la versión iterativa empleada en 3.3 se declara una matriz de n elementos y algunas otras sencillas variables.

Sin embargo, el análisis del espacio de memoria ocupado por las invocaciones recursivas es mucho más complejo. De este modo, cada llamada recursiva del algoritmo en 3.1 toma una cantidad de espacio como sigue:

- Espacio para variables locales y parámetros de función.
- Espacio adicional para el contexto de la función cada vez que se llama a sí misma.

Las invocaciones activas en un momento dado forman un camino concreto en un árbol de llamadas. Así, en el caso del cálculo de fib(5) el árbol completo sería el de la Figura 6, donde vemos que para la raíz y los nodos internos cada llamada genera, a su vez, dos llamadas mientras que para las hojas se genera una sola llamada. Así una breve reflexión nos permitirá concluir que, si hemos de calcular fib(n), al final del desarrollo el número de hojas del árbol es precisamente fib(n) y el de nodos internos (incluida la raíz) fib(n)-1 con lo que el total de invocaciones será de fib(n) + 2*(fib(n) - 1) = 3*fib(n) - 2. De este modo, para calcular fib(5) necesitaremos espacio para 13 invocaciones, pero, para calcular fib(10) ya serán 163 las llamadas y si queremos llegar a los mil millones tan sólo es necesario calcular fib(45).

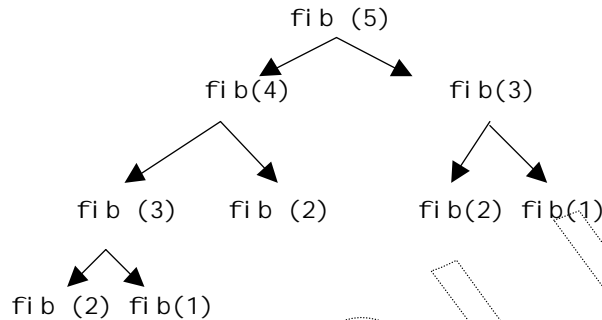


Figura 6 Árbol de llamadas del cálculo recursivo de números de Fibonacci.

Así, hemos comprobado que, en el caso del cálculo recursivo de los números de Fibonacci, el espacio que necesita el algoritmo recursivo puede hacer inviable su ejecución.

El resumen de lo expuesto sería algo así como, la recursividad permite definir ciertos problemas de modo mucho más natural que aplicando iteración. Sin embargo, en algunos casos es posible que tengamos que pagar esta sencillez con una menor eficiencia en el consumo de recursos.

3.5. APLICACIONES DE LA RECURSIVIDAD

Como ya comentamos en 3.1.2, existen casos en los que el uso de la recursividad viene prácticamente impuesto por la organización de los datos que se van a tratar. Un ejemplo paradigmático es el recorrido de un árbol. Se trata de una estructura típicamente recursiva y la mejor solución para su tratamiento es emplear algoritmos recursivos. No obstante, aún en estas ocasiones es posible, aunque a nuestro juicio no recomendable, eliminar la recursividad de tratamientos usando la misma técnica que usan los compiladores para ello, es decir, mediante el uso de pilas para ir guardando los sucesivos contextos del procedimiento en cada una de las llamadas a la rutina recursiva. Con los modernos lenguajes de programación esta práctica ha dejado de ser necesaria puesto que la recursividad es una característica nativa de dichos lenguajes.

Ejemplo: recorrido en orden de un árbol binario en C (suponemos un elemento de tipo char y un procedimiento visitar() también predefinido).

```

struct nodo
{
    char elemento;
    struct nodo *izq, *der;
};

typedef struct nodo tNodo;

enOrden (tNodo *t)
{
    if (t != NULL)
    {
        enOrden (t->izq);
        visitar (t);
        enOrden (t->der);
    }
}
  
```

```

    }
}

void main (void)
{
    ...
    tNodo z, *pz;
    pz = &z;
    pz -> izq = NULL; pz -> der = NULL;
    ...
    enOrden (pz);
    ...
}

```

Por otro lado, existen tipos de problemas en cuya resolución la recursividad puede prestar una ayuda inestimable. Así, es el caso de los algoritmos de *divide y vencerás*⁴. Esta técnica consiste, en esencia, en dividir el problema en dos o más subproblemas del mismo tipo que el original pero de menor complejidad. Si se da solución a estos subproblemas se tendrá fácilmente una solución para el problema principal. A su vez, para resolver los subproblemas se puede optar por dos posibilidades:

- Si el subproblema es ya lo suficientemente reducido para solucionarse directamente estamos ante un ejemplo de solución "trivial".
- Si el subproblema es aún de gran complejidad se aplica recursivamente la técnica.

Un ejemplo (bastante deficiente) de *divide y vencerás* es la función recursiva *fib* que calcula los números de Fibonacci en 3.1.1. El fallo está en que los subproblemas no son completamente independientes sino que están traslapados por lo que el algoritmo tiene una grave falta de eficiencia ya comentada. Otro ejemplo, bastante más adecuado, es el que acabamos de ver en este mismo apartado sobre el recorrido de un árbol binario. Evidentemente, aquí el razonamiento *divide y vencerás* es inmediato: para recorrer un árbol binario en orden, primero recorreremos el subárbol izquierdo, luego visitaremos la raíz y finalmente recorreremos el subárbol derecho; como cada subárbol es, a su vez, un árbol, la recursividad *divide y vencerás* está servida.

Es generalmente admitido que cualquier algoritmo recursivo que se codifique con una única auto-llamada no entra dentro de la categoría de *divide y vencerás* (aunque todo es cuestión de proponérselo y liar convenientemente la cosa).

Otro uso de la recursividad es en la codificación de los algoritmos conocidos como de retroceso (*backtracking*) que suponen una evolución del conocido sistema de *fuerza bruta*, esto es, atacar un problema mediante la prueba sistemática de todos los casos posibles hasta dar con la solución o llegar al final de los casos. La fuerza bruta no suele producir resultados demasiado alentadores ya que genera algoritmos con coste computacional de tipo exponencial. Por ello, y en los casos en que resulte posible, la utilización del retroceso que se aplica cuando se llega a una situación no deseada y se vuelve hacia atrás (es lo que se conoce como *poda* es decir la eliminación de un camino), puede suponer una sustancial mejora.

Un ejemplo tomado de la vida real sería todas las formas admisibles de colocar el mobiliario de una vivienda. Evidentemente, con unos cuantos muebles y varias habitaciones el número de posibilidades es muy alto. Sin embargo, en cuanto se coloque, por ejemplo, la cama en la cocina, el dueño de la vivienda nos dirá que la cosa no va demasiado bien y que es necesario volver sobre nuestros pasos y tomar por otro camino. Así, podremos seguir probando, avanzando, podando y retrocediendo, hasta que todo el mobiliario esté colocado.

4. BIBLIOTECAS (LIBRERÍAS)

4.1. CONCEPTO

Una biblioteca (librería si traducimos *library* "por libre") es un conjunto documentado, probado y, en su caso, previamente compilado, de procedimientos y funciones que es posible invocar desde otro programa. Actualmente, el concepto de biblioteca ha sido superado por el de biblioteca de clases y, dando un paso más, por el de biblioteca

⁴ Como en otros casos también se puede codificar *divide y vencerás* por el procedimiento iterativo.

de componentes. No obstante, como en el temario actual existen capítulos dedicados a la programación orientada a objetos nos centraremos ahora en las bibliotecas no OO, que por otro lado siguen de plena actualidad.

Prácticamente desde sus inicios, la mayoría de los lenguajes de programación de alto nivel se vienen suministrando con bibliotecas básicas, generales, pero éstas son normalmente insuficientes para el desarrollo eficaz de aplicaciones. Por este motivo, y para lenguajes de uso común⁵, es posible encontrar en el mercado diversas bibliotecas comerciales que incluyen innumerables procedimientos capaces de solucionar multitud de los problemas de programación a los que un desarrollador se puede enfrentar con los lenguajes en cuestión.

Las bibliotecas son un claro ejemplo de reutilización del software. La reutilización de líneas de código individuales es la forma más simple y la que se utilizó antes. Para ello lo único que hay que hacer es "cortar" y "pegar" con un editor de textos. No obstante, este método amén de primitivo es bastante ineficaz ya que hay que repetir todo el código y no suministra mecanismos de mantenimiento. Se puede conseguir un aprovechamiento bastante superior usando bibliotecas organizadas en un marco de referencia (Booch 94). Un marco de referencia⁶ es una colección de tipos de datos, procedimientos y funciones que proporciona un conjunto de servicios para un dominio particular: exporta así una serie de mecanismos y herramientas que los clientes pueden usar o adaptar.

Los marcos de referencia pueden ser transversales al tipo de aplicación, lo que significa que pueden aplicarse a una amplia gama de aplicaciones. Ejemplos de esta categoría son las bibliotecas básicas generales, las bibliotecas matemáticas o las bibliotecas para interfaces gráficas de usuario. También existen marcos de referencia verticales que se aplican en dominios concretos. Posibles ejemplos pueden ser bibliotecas para control de comunicaciones telefónicas, para cálculo de estructuras en ingeniería, etc.

4.2. REQUISITOS

Una biblioteca que proporcionase soluciones para todos los posibles problemas de programación con un lenguaje dado sería enorme. Los analistas, por tanto, deben seleccionar a la hora de crear las bibliotecas sean éstas para consumo propio o para su venta en el mercado. Adaptando el trabajo de Booch [Booch 94] para las bibliotecas de clases establecemos a continuación los requisitos de las bibliotecas básicas.

Una biblioteca básica debe proporcionar una colección de estructuras de datos, funciones y procedimientos independientes del tipo de aplicación donde se vayan a usar. Esta colección debe ser suficiente para cubrir las necesidades de la mayoría de las aplicaciones en los lenguajes que permitan su uso. Además, una biblioteca ideal debe ser:

- **Completa:** la biblioteca debe proporcionar una familia de subprogramas, unidas por un interfaz compartido pero empleando cada representación diferente, de manera que los desarrolladores puedan seleccionar las que sean más apropiadas para la aplicación de que se trate.
- **Adaptable:** todos los aspectos específicos de la plataforma deben estar claramente identificados y aislados, de manera que puedan realizarse sustituciones y adaptaciones locales (por ejemplo, mediante el uso de funciones propias que intermedien entre el código de la aplicación y las invocaciones a los procedimientos de biblioteca).
- **Eficiente:** los componentes deben ser de fácil incorporación al código propio (eficiencia en términos de recursos de compilación) utilizar cantidades razonables de memoria y tiempo de ejecución (eficiencia en ejecución) y de uso comprensible y seguro (eficiencia en términos de recursos de desarrollo).
- **Segura:** es un requisito fundamental que la biblioteca esté completamente probada en todos los entornos previsibles. Uno de los indicadores de esa robustez es el uso de excepciones para identificar condiciones para las cuales se violan las precondiciones de un algoritmo. Cuando estas excepciones se generen el sistema debe ser capaz de mantener la estabilidad sin que se produzcan reacciones anómalas, rupturas bruscas de la secuencia de ejecución o corrupciones en el espacio de direcciones del programa.

⁵ El lenguaje de codificación original de una biblioteca y el usado en el programa que invoca a los procedimientos de dicha biblioteca no tienen por qué coincidir.

⁶ Grady Booch aplica este concepto a las bibliotecas de clases pero su adaptación a las bibliotecas clásicas es inmediata.

- Simple: característica que es cada vez más difícil de cumplir (en este sentido ayudan mucho las técnicas O.O.). Se trata de dotar a la biblioteca de una organización clara y consistente que facilite la identificación y selección de las estructuras y procedimientos adecuados para el fin requerido.
- Extensible: los desarrolladores propios deben ser capaces de añadir funcionalidad a la biblioteca sin alterar su integridad arquitectónica original.
- Independiente de la plataforma final de ejecución: característica que está adquiriendo cada vez una mayor importancia. Se trata de hacer la biblioteca lo más independiente que sea posible del hardware y sistema operativo donde finalmente se ejecute la aplicación que se está desarrollando. Para ello se crean bibliotecas abstractas que actúan como interfaz. Estas bibliotecas se conectan de forma transparente para el desarrollador con otras que sí dependen de los servicios de la plataforma. Dicha conexión se puede producir, bien al compilar el código, con lo que habrá que recompilar para cada tipo de plataforma, bien en tiempo de ejecución de forma dinámica, como ocurre, por ejemplo, con las bibliotecas de clases de Java, lo que obliga al uso de las conocidas como "máquinas virtuales".

4.3. UN CASO ESPECÍFICO, LAS A.P.I.S.

4.3.1. CONCEPTO

Una API (Application Programming Interface), en castellano, interfaz para la programación de aplicaciones es un conjunto de bibliotecas de programación, que elaboran y publican los fabricantes de elementos tales como sistemas operativos o dispositivos hardware, para permitir a los programadores de aplicaciones utilizar los servicios y posibilidades de dichos elementos. Por extensión, se denomina API a cualquier grupo de funciones que son parte de ciertas aplicaciones pero que son utilizables desde otras aplicaciones. Sin la creación y publicación de APIs la construcción, por ejemplo, de compiladores para Windows 3.x, hubiera supuesto un trabajo improbable que muy poca gente habría estado dispuesta a afrontar.

4.3.2. LA API DE WINDOWS

4.3.2.1. Introducción y breve referencia

Cuando comenzó la era Windows, cualquier programador ávido de desarrollar para este entorno/sistema operativo debía hacer uso directo de las funciones de la API. Ello provocó fuertes dolores de cabeza e incluso males peores a más de un intrépido, pues se trata de cientos de funciones y estructuras escritas en C con nombres increíblemente complicados, multitud de parámetros, etc. En resumen, que no se caracterizan precisamente por su ergonomía. No obstante, poco a poco fueron apareciendo los marcos de referencia (marcos de trabajo o *frameworks*). Se trata de bibliotecas de funciones intermedias que hacen de colchón entre las llamadas y necesidades del desarrollador de aplicaciones y los servicios ofrecidos por la API del sistema operativo. Famosos ejemplos son las MFC (Microsoft Foundation Classes) o las distintas familias de bibliotecas de clases que han venido acompañando a los productos de Inprise (antes Borland) entre las que destacan por méritos propios las denominadas VCL (Visual Component Library) biblioteca orientada a objetos de componentes visuales y otras clases que despliegan sus innumerables posibilidades en un alarde de funcionalidad, buena estructuración y eficacia. Otro ejemplo, las bibliotecas que acompañan a la utilizada herramienta PowerBuilder, encapsulan, en su versión para Windows 32, más de 700 llamadas a los servicios de la API de Windows. Por supuesto, en cualquiera de las herramientas citadas también es posible efectuar llamadas directas a los servicios de la API Windows.

Seguidamente se relacionan las principales versiones, aparecidas hasta la fecha de la API de Windows:

- Win16: API de Windows de 16 bits. Es la API original de Windows y Windows para Trabajo en Grupo 3.x. También es soportada por Windows 9x y Windows NT en modo 16 bit.
- Win32: API de 32 bits completa. Es la API original de Windows NT. Cada uno de los sistemas operativos Windows NT, Windows 95, Windows 98 y Windows CE (Windows para PalmTops) incorpora un subconjunto distinto de Win32. Windows 2000 (la última versión de NT) es el único que soporta plenamente todas las llamadas de Win32.

- Win32c: subconjunto de Win32 que es soportado por Windows 95. Con esta API es posible invocar a cualquier función de Win32 pero muchas de las funciones no han sido desarrolladas (aquellas relacionadas con características específicas de Windows NT) por lo que simplemente devuelven el control sin hacer nada.
- Win32s: reducido subconjunto de Win32 soportado por Windows y Windows 3.x para Trabajo en Grupo.

4.3.2.2. Enlace estático y dinámico

Como ya hemos indicado, la mayoría de los lenguajes permiten al programador la creación de bibliotecas de funciones que pueden ser invocadas desde su programa y aparecen como si estuviesen elaboradas dentro del propio lenguaje. Usualmente, en entornos MS-DOS/Windows, los módulos de programas que contienen las funciones están precompilados en archivos de programas objeto (.obj) que pueden agruparse en archivos de bibliotecas (.lib) utilizando un bibliotecario (que puede ser un programa auxiliar o parte de un entorno integrado de desarrollo IDE).

Cuando se debe crear una versión final ejecutable de una aplicación, un enlazador analiza los archivos objeto de la aplicación buscando referencias a funciones que no están definidas en el propio programa, luego recorre todos los archivos de bibliotecas cuyo uso se ha solicitado, buscando las funciones que faltan. El enlazador extrae los módulos que contienen las funciones invocadas, los incluye en el archivo ejecutable y los enlaza con las llamadas del programa de aplicación. A este proceso se lo conoce como *enlace estático*, ya que toda la información de direccionamiento que necesita el programa para el acceso a las funciones de biblioteca queda fijada definitivamente cuando se crea el ejecutable y permanece invariable en tiempo de ejecución. Tradicionalmente, los enlazadores incluyen los módulos enteros cuando se los enlaza en los ejecutables finales aunque las últimas versiones de IDE ya son capaces de extraer únicamente el código correspondiente a la función referida.

No obstante, en entornos gráficos y multitarea como Windows, dado el alto número de funciones de biblioteca que es preciso incorporar a cada programa, es usual que cualquier pequeña aplicación, del tipo de las de "Hola Mundo" ocupe cantidades considerables de espacio. El problema se agudiza cuando dichas aplicaciones se van cargando en memoria. Cada programa que se esté ejecutando incluirá sus propias copias, por ejemplo, de rutinas de gestión de ventanas. Si suponemos que dichas rutinas ocupan 250 KB de RAM y tenemos dos aplicaciones ejecutándose, existirán dos copias de las citadas rutinas con lo que se desaprovecharán 250 KB de memoria.

Microsoft resolvió el problema del desperdicio de memoria provocado por el enlace estático introduciendo la posibilidad de usar bibliotecas de enlace dinámico.

Con el enlace dinámico, los módulos de programas conteniendo las funciones también son precompilados en archivos de programas objeto (.obj), pero, en lugar de agruparlos en archivos de bibliotecas, son enlazados en un formato especial de archivo ejecutable de Windows conocido como DLL⁷, biblioteca de enlace dinámico. Cuando se construye una DLL, el constructor especifica qué funciones van a ser accesibles desde otras aplicaciones en ejecución mediante la técnica, ya estudiada, denominada exportación.

Al crear un archivo ejecutable para Windows, el enlazador analiza los archivos objeto de la aplicación y elabora una lista de todas aquellas funciones que no están ya incluidas en el código del programa junto con la indicación de las bibliotecas de enlace dinámico donde se encuentran.

Cuando se ejecuta una aplicación con acceso a DLLs, cada vez que se invoca una función de las ubicadas en la biblioteca de enlace dinámico, la dirección real de enlace es calculada y la función se enlaza dinámicamente con la aplicación. De este modo, aunque existe una única copia en memoria por cada DLL, los programas pueden compartir las funciones incluidas en dicha biblioteca.

4.3.3. LA API DE WINDOWS Y LAS DLL

La mayoría de las funciones de la API Win32 se encuentran divididas entre las tres bibliotecas de enlace dinámico principales, Kernell, User y GDI, y algunas otras secundarias. Seguidamente resumimos su funcionalidad:

- Kernell32.dll: contiene las funciones de operaciones de bajo nivel. Incluye las funciones de administración de memoria y otros recursos, gestión de procesos y otras operaciones relacionadas.

⁷ En Windows, los archivos que incluyen las bibliotecas de enlace dinámico tienen, principalmente, la extensión .dll pero esto no es obligatorio. Así, existen DLL con extensión .ocx, .drv, .exe, etc.

- User32.dll: funciones relacionadas con la administración y uso de Windows. Incluye la operatoria de mensajes, menús, cursores, gestión de relojes, comunicaciones y la mayor parte de las funciones que no tienen efectos visuales.
- GDI32.dll: biblioteca de la interfaz de dispositivo gráfico. Contiene las funciones relacionadas con los dispositivos de salida, principalmente pantallas e impresoras, tales como funciones de dibujo, metaarchivos, coordenadas, fuentes tipográficas, etc.
- ComDlg32.dll, LZ32.dll, Version.dll: suministran recursos para diálogos comunes, compresión de archivos y control de versiones.

Gracias a las especiales características de las DLL, cuando Microsoft quiere añadir alguna funcionalidad a sus sistemas operativos Windows, lo único que tiene que hacer es agregar una nueva biblioteca de enlace dinámico. Cada vez que una DLL aparece se publica la correspondiente extensión a las API. A continuación se recogen algunas de ellas:

- ComCTL32.dll: introdujo controles nuevos como la lista en árbol (ListTree) y el control de edición de texto en formato "texto enriquecido" RTF (rich text format).
- MAPI32.dll: proporciona funciones que permiten a las aplicaciones trabajar con correo electrónico.
- NetAPI32.dll: suministra funciones para acceso y control de redes.
- ODBC32.dll: una de las DLL usadas en ODBC (Open Database Connectivity) sistema que permite a cualquier aplicación utilizar cualquier base de datos que soporte ODBC.
- WinMM.dll: acceso a las capacidades multimedia del sistema.

La API de Windows está en permanente proceso de revisión y evolución. La complejidad y el número de funciones crece tan rápidamente que hace prácticamente imposible a cualquier desarrollador mantenerse actualizado. No obstante, no es tan importante conocer todos los detalles de esta inmensa maraña de herramientas como su estructura principal que es bastante parecida a la de otros sistemas operativos con soporte nativo GUI.

5. BIBLIOGRAFÍA

- **Aho, Alfred V.; Hopcroft, John E.; Ullman, Jeffrey D.:** Estructuras de datos y algoritmos. Addison Wesley Ib. 1988
- **Appleman, Daniel:** Visual Basic 5. Guía del programador para el uso de la API de Win32. InforBook's. 1997
- **Allen Weiss, Mark:** Estructuras de datos y algoritmos. Addison Wesley Ib. 1995
- **Balcázar:** Programación metódica. McGraw-Hill. 1993
- [Booch 94] **Booch, Grady:** Object-oriented analysis and design with applications. Benjamin/Cummings. 1994 (existe una versión en castellano)
- **Brassard G.; Bratley, B.:** Fundamentos de Algoritmia. Prentice Hall. 1997
- **Castro, Jorge; Cucker, Felipe et al:** Curso de programación. McGraw-Hill. 1993
- **Collado Machuca y O.:** Estructuras de Datos, Realización en PASCAL. Díaz de Santos. 1987
- **Fontaine, A.B.:** MODULA -2. Masson 1987.
- **Heileman, Gregory L.:** Estructuras de datos, algoritmos y programación orientada a objetos. McGraw-Hill. 1997
- **Joyanes Aguilar, Luis; Zahonero Martínez, Ignacio:** Estructuras de datos. Algoritmos, abstracción y objetos. McGraw Hill. 1998
- [Meyer 99] **Meyer, Bertrand:** Construcción de software orientado a objetos. 2ª ed. Prentice Hall. 1999
- **Peña Marí, Ricardo:** Diseño de programas. Formalismo y abstracción. 2ª ed. Prentice Hall. 1998
- **Pressman, Roger S.:** Ingeniería del software. Un enfoque práctico. 4ª ed. McGraw-Hill. 1998
- **Sanchís Lorca, F.J.; Morales Lozano, A.:** Programación con el lenguaje PASCAL. Paraninfo 1985.
- **Scholl, P.C.; Peyrin, J.P.:** Esquemas algorítmicos fundamentales. Secuencias e iteración. Masson 1991
- **Sedgewick, Robert:** Algoritmos en C++. Addison Wesley-Díaz de Santos. 1995
- **Wirth, Niklaus:** Algoritmos + Estructuras de datos = Programas. Del Castillo. 1980

Los textos señalados con → componen la fuente documental principal en relación a los contenidos del tema. Los que aparecen con el autor subrayado son, también en lo que se refiere al tema actual y a nuestro juicio, textos que deberían figurar en la biblioteca de cualquier profesional "serio" de la docencia informática. Es posible que

ambos criterios no coincidan. También es posible que un texto que aquí no se recomienda aparezca recomendado en otro tema.

26. PROGRAMACIÓN MODULAR. UNIDAD DIDÁCTICA

6. UNIDAD DIDÁCTICA

La unidad didáctica va a hacer referencia a todo el tema que equivale a parte de sendos bloques de conocimientos en los currículos del Ministerio de Educación y Ciencia de los ciclos superiores de informática ASI y DAI. Asimismo encontramos referencias al tema de programación estructurada y modular en el módulo *Sistemas Operativos y Lenguajes de Programación* del ciclo superior *Sistemas de Telecomunicación e Informáticos* en el cual también tienen atribución docente los profesores de Informática.

En realidad, la programación modular es un tema complementario al de la programación estructurada. Aunque es perfectamente posible utilizar programación estructurada y no modular y viceversa ambas técnicas son perfectamente complementarias y compatibles y resulta bastante adecuado hacer un enfoque sucesivo de las dos comenzando por las técnicas de estructura y terminando por las de modularidad del código. Evidentemente, las técnicas de diseño de funciones y procedimientos se estudiarían únicamente en una de las dos unidades didácticas (en los conceptos que se proponen en ambas unidades están repetidas).

Por otro lado, comentar que la modularidad alcanza su máxima expresión en la programación orientada a objetos y que hoy por hoy no creemos adecuada hacer una exposición sobre programación modular que no incluya, de algún modo, las técnicas de objetos. No obstante, respetamos aquí la voluntad de los diseñadores del programa actual que han considerado la OOP como algo especial y han reservado su estudio a un tema específico.

7. DESTINATARIOS

Existen dos perfiles de destinatarios:

1. El alumno que accede por lo que es la vía normal tras cursar ESO y Bachillerato. Su edad de partida son los 19 años.
2. El alumno de procedencia y nivel académicos diversos, que accede directamente por prueba de acceso o convalidación de estudios anteriores.

En ambos casos la metodología a aplicar no suele diferir ya que se trata de personas lo suficientemente maduras para ser tratadas como adultas y, la experiencia nos lo dice, bastante motivadas.

8. UBICACIÓN, DISEÑO CURRICULAR BASE

En los ciclos formativos de grado superior, se puede ubicar en el módulo de *Fundamentos de Programación* para el ciclo *Administración de Sistemas Informáticos* y en *Programación en Lenguajes Estructurados* para el ciclo *Desarrollo de Aplicaciones Informáticas*, ambos módulos son de 1er curso del currículo en el territorio MEC. También se ubicará en el módulo *Sistemas Operativos y Lenguajes de Programación* del ciclo superior *Sistemas de Telecomunicación e Informáticos*. La docencia de todos estos módulos corresponde a profesores del Cuerpo de Enseñanza Secundaria.

DESARROLLO DE APLICACIONES INFORMÁTICAS	Curr.	MEC		Andal.		Galicia		Valenc.		Catal.	
		Sem.	Tot.	Sem.	Tot.	Sem.	Tot.	Sem.	Tot.	Sem.	Tot.
Módulos 1º											
Programación en lenguajes estructurados	210	12	380	11	352	9	295			10	330

ADMINISTRACIÓN DE SISTEMAS INFORMÁTICOS	Curr.	MEC		Andal.		Galicia		Valenc.		Catal.	
		Sem.	Tot.	Sem.	Tot.	Sem.	Tot.	Sem.	Tot.	Sem.	Tot.
Fundamentos de programación	160	9	285	8	256	8	260			8	240

Las características horarias de los módulos de los ciclos de Informática son las que figuran en la tabla anterior.

Por otro lado y formando parte también del marco de referencia que atañe al desenvolvimiento del hecho docente hemos de considerar el Proyecto Educativo de Centro como fuente de información sobre la situación del entorno en el que se inserta el centro educativo.

9. RELACIÓN CON EL PROYECTO CURRICULAR DE ETAPA.

A la hora de desarrollar la programación de Departamento y la programación diaria en el aula correspondiente a ésta y a otras disciplinas, se debe tener en cuenta el sistema referencial que suponen los proyectos curriculares de la etapa existentes en cada centro y que definen los criterios básicos relacionados con objetivos generales de la etapa, la secuenciación de estos objetivos y de los contenidos correspondientes a cada curso, metodología aplicable, los criterios de evaluación, promoción y recuperación, así como las medidas de atención a la diversidad. En este sentido la programación de una unidad didáctica perteneciente a una asignatura o módulo, se relaciona con la del resto del Departamento y suele coincidir plenamente con ésta en los aspectos de metodología aplicable, criterios de evaluación y promoción, etc.

Otro elemento a tener en cuenta es el Proyecto Curricular del Ciclo de que se trate. Es éste un referente mucho más cercano a la programación de aula que hay que considerar a la hora de su desarrollo. Cada PCC, debe cumplir, entre otros, los siguientes requisitos:

1. Ser realizado por todos los profesores que imparten docencia en los Ciclos Formativos incluidos en cada Familia Profesional, coordinados por el Jefe de Departamento.
2. Incluir el Plan de tutoría y orientación profesional.
3. Contemplar las orientaciones acerca del uso de los espacios específicos y de los medios y equipamientos asignados al Departamento.

10. OBJETIVOS-CAPACIDADES

La unidades de competencia a las que se asocian los módulos son las siguientes:

- En DAI es la número 3: elaborar, adaptar y probar programas en lenguajes de programación estructurados y de cuarta generación. Como contenido organizador tomaremos la misma unidad de competencia.
- En ASI es la número 4: Proponer y coordinar cambios para mejorar la explotación del sistema y las aplicaciones. En este caso, como contenido organizador tomaremos el siguiente que es más cercano a las características del módulo *Fundamentos de programación*: Elaborar, adaptar y probar programas para mejorar la explotación del sistema y las aplicaciones.

10.1. CAPACIDADES TERMINALES

Expresan lo que se espera del alumno al terminar sus estudios profesionales.

- En DAI:
 - ◊ Elaborar programas utilizando programas estructurados cumpliendo con las especificaciones establecidas en el diseño.
 - ◊ Integrar y enlazar programas y rutinas siguiendo las especificaciones establecidas en el diseño.
- En ASI: realizar, a su nivel, los cambios propuestos en el sistema y/o aplicaciones de acuerdo con las prestaciones requeridas.

10.2. ESPECÍFICOS

10.2.1. CONCEPTUALES

- Describir el concepto de algoritmo y programa.
- Explicar el concepto de estructura y el de módulo.
- Describir la utilidad de las librerías y de los enlazadores de los sistemas operativos así como su forma de empleo (también procedimental).
- Aplicar estrategias de programación modular (también procedimental).

10.2.2. PROCEDIMENTALES (EJE ESTRUCTURADOR)

- Ante problemas de procesamiento de datos con reglas de procesamiento expresadas en lenguaje natural:
 - ◊ Interpretar el problema.
 - ◊ Elegir las estructuras básicas de programación necesarias para la resolución del problema.
 - ◊ Construir el algoritmo.
 - ◊ Edición, compilación y prueba.
 - ◊ Análisis de la idoneidad del algoritmo.
- Desarrollar la capacidad de abstracción necesaria para resolver problemas de tratamiento de la información.
- Integrar y enlazar módulos de programación, rutinas y utilidades, siguiendo las especificaciones del diseño.
- Elaborar módulos, funciones o procedimientos utilizando las técnicas de programación modular.

10.2.3. ACTITUDINALES

- Fomentar la actitud positiva en el alumno hacia la actitud de mantener relaciones fluidas con los miembros del grupo funcional en el que está integrado, responsabilizándose de la consecución de los objetivos asignados al grupo, respetando el trabajo de los demás, organizando y dirigiendo tareas colectivas y cooperando en la superación de dificultades que se presenten, con una actitud tolerante hacia las ideas de los compañeros y subordinados.
- Promover el gusto por el uso adecuado de las técnicas de descomposición modular.
- Promover en el alumno una actitud positiva hacia el hábito de trabajo.
- Fomentar entre los alumnos el aprecio por el rigor intelectual (exactitud en los datos y en la terminología, precisión en la documentación).
- Desarrollar en el alumno la toma de decisiones.

11. CONTENIDOS

PROGRAMACIÓN MODULAR

1.1. CONCEPTOS INICIALES

1.1.1. Módulo

1.1.2. Interfaz e implementación

1.2. REFINAMIENTO Y MODULARIDAD

1.2.1. Refinamiento sucesivo

1.2.2. La división en módulos y la inteligibilidad de los programas

1.2.3. El diseño de arriba abajo (TOP DOWN) concepto y utilidad

1.2.4. Tipos de módulos

1.3. OCULTACIÓN DE LA INFORMACIÓN

1.4. INDEPENDENCIA FUNCIONAL Y CALIDAD DEL SOFTWARE

1.4.1. Concepto

1.4.2. Cohesión

1.4.3. Acoplamiento

1.5. ESTRUCTURA DEL PROGRAMA Y JERARQUÍA DE CONTROL

1.5.1. Organización de módulos en niveles y transferencias de control

1.5.2. Principios de un buen método de diseño según Meyer

2. DISEÑO DE FUNCIONES

2.1. PROCEDIMIENTOS

- 2.1.1. *Llamada a procedimientos*
- 2.1.2. *Parámetros*
- 2.1.3. *Definición de procedimientos*
- 2.1.4. *Declaración de procedimientos*

2.2. FUNCIONES

- 2.2.1. *Concepto*
- 2.2.2. *Llamada a funciones*
- 2.2.3. *Definición de funciones*
- 2.2.4. *Declaración de funciones*

3. LA RECURSIVIDAD

3.1. CONCEPTO

- 3.1.1. *Recursividad en algoritmos*
- 3.1.2. *Recursividad en datos*

3.2. RAZONAMIENTO RECURSIVO

3.3. RECURSIÓN E ITERACIÓN

3.4. COSTE DE LAS FUNCIONES RECURSIVAS

3.5. APLICACIONES DE LA RECURSIVIDAD

4. BIBLIOTECAS (LIBRERÍAS)

4.1. CONCEPTO

4.2. REQUISITOS

4.3. UN CASO ESPECÍFICO, LAS A.P.I.S.

- 4.3.1. *Concepto*
- 4.3.2. *Presentación de la API de Windows*
- 4.3.3. *La API de Windows y las DLL*

12. ACTIVIDADES PROPUESTAS (ADICIONALES A LA EXPOSICIÓN)

5. Realización de supuestos con algoritmos básicos o de mayor nivel utilizando las técnicas de programación estructurada y modular.
6. Implementación de los supuestos anteriores en un determinado lenguaje. El lenguaje escogido depende de factores adicionales pero podría ser PASCAL.
7. Si es posible, compilación, depuración, montaje y ejecución de los programas resultantes para comprobar la funcionalidad de las propuestas.
8. Reparto de ejercicios prácticos complementarios para su resolución en clase y en el domicilio particular.

13. MATERIAL DIDÁCTICO

Se utilizarán:

- Fotocopias de apuntes elaborados por el profesor.
- Transparencias con ejemplos gráficos de herramientas de programación estructurada. Posteriormente dichos ejemplos se formularán en un lenguaje de programación.
- Bibliografía básica y complementaria.

Como principales herramientas serán usados:

- Equipos informáticos.
- Herramientas necesarias para desarrollar software ejecutable en el lenguaje elegido.

14. INTERDISCIPLINARIEDAD

El estudio de los algoritmos se vincula estrechamente a las *Estructuras de la Información* y a cualquier módulo que utilice la programación como herramienta. Por ejemplo, al estudiar las Bases de Datos se puede hacer referencia

a los algoritmos de indización que éstas utilizan. Al analizar los protocolos de acceso al medio en redes locales observar los algoritmos implicados, etc. En todo caso, el utilizar y comprender las técnicas de programación estructurada y modular ayudarán a mejorar la comunicación entre los profesionales de la programación.

15. DISTRIBUCIÓN TEMPORAL

Para el módulo de *Programación en Lenguajes Estructurados* la ubicación en la primera mitad del primer trimestre una vez vista la unidad dedicada a la programación estructurada. El número de horas destinado al tema sería de 18 lo que llevaría a una ocupación temporal aproximada de 1,5 semanas. En cuanto al módulo *Fundamentos de Programación* la distribución en bloques sería parecida pero disminuyendo el total de horas ocupadas por los contenidos a 14. Por otro lado, siendo tan fundamentales los contenidos de esta unidad didáctica para la formación de cualquier programador informático, es preciso que los conceptos en ella tratados se vayan retomando a lo largo de toda la formación del profesional.

Para el módulo de *Sistemas Operativos y Lenguajes de Programación* del ciclo superior *Sistemas de Telecomunicación e Informáticos* se establece una duración mínima anual de 155 horas de las que no más de 2 ó 3 se dedicarían al estudio del tema. Es evidente el bajo nivel que se alcanzaría en este caso pero, por otro lado, es lógico, dado que para los especialistas en Sistemas de Telecomunicación e Informáticos esta formación tiene un carácter menos fundamental que en las de informática.

16. METODOLOGÍA Y ESTRATEGIAS

Al principio de la unidad temática se entregará al alumno la suficiente documentación junto con orientaciones para el completo aprendizaje del tema.

El profesor realizará la exposición verbal con abundante soporte gráfico de los puntos fundamentales que componen la unidad temática acompañada de numerosos ejemplos prácticos de aplicación.

Durante el trabajo en el aula, que incluirá necesariamente la realización de abundantes prácticas con y sin soporte informático, el profesor actuará como asesor intentando orientar las tareas de autoaprendizaje en lugar de facilitar directamente la solución a los problemas planteados. Se trata de conseguir que el alumno participe en la elaboración de los procesos conducentes a su propia instrucción creando así el marco de referencia adecuado para la generación de situaciones de aprendizaje significativo.

La distribución de los espacios en el aula será flexible pero dando tratamiento de preferencia a las agrupaciones de trabajo de tres o cuatro miembros sobre todo para las fases de resolución de tareas propuestas.

17. EVALUACIÓN, RECUPERACIÓN Y PROMOCIÓN

17.1. DEL ALUMNO

La evaluación será **continua e integradora**. Se respetarán en todo caso las funciones formativa y sumativa de la misma. En cuanto a la evaluación inicial, al tratarse de la programación de una unidad didáctica y no del curso completo, consideramos que ya se efectuó anteriormente iniciándose desde ese momento el funcionamiento de la evaluación continua.

En los ciclos, la evaluación se realizará tomando como referencia las capacidades y los criterios de evaluación de cada uno de los módulos.

Como principales instrumentos para llevar a cabo la evaluación de los alumnos se van a utilizar:

- Observación de la actitud y trabajo diario del alumno durante la clase: desenvolvimiento normal, intervenciones (expresión oral), ejercicios de clase. (Evidentemente este tipo de instrumento es imprescindible para conseguir una evaluación **continua** que es la indicada por la Administración (RD 676/1993 de 7/05 de directrices generales sobre los títulos y enseñanzas mínimas de la F.P.)
- Resultado de trabajos y otras actividades de ejecución individual o grupal.
- Exámenes con preguntas cortas (mejor tipo test).

- Exámenes con supuestos (problemas complejos para cuya resolución es necesaria la utilización combinada de conocimientos y aptitudes específicos).
- Además serán tenidas en cuenta otras fuentes de información sobre los alumnos tales y como:
- Entrevistas realizadas con los mismos alumnos y coevaluación.
 - Información procedente de los padres o tutores.
 - Información procedente de otros docentes (de reuniones de departamento, reuniones de evaluación, del Departamento de Orientación ...).
 - Proceso de autoevaluación de los alumnos.

En cuanto a los **mínimos** cuya superación se considera necesaria por parte del alumno los enunciamos a continuación:

- Características de la programación modular.
- Recursividad.
- Funciones.
- Estructura modular de programas.
- Librerías.

En todo caso para los ciclos superiores se considerarán mínimas las capacidades profesiones establecidas en cada uno de los R.D. que definen el currículo base.

Para los alumnos que no alcancen los objetivos mínimos fijados, durante el período de explicación de la siguiente unidad didáctica, se llevarán a cabo **tareas de refuerzo y recuperación** consistentes en la realización de actividades adicionales a las indicadas.

En su caso, y de ser necesario, se llevarán a cabo, con el asesoramiento del departamento de orientación, las **adaptaciones curriculares** que alumnos con determinadas carencias/necesidades requieran.

17.2. DE LA ACTIVIDAD DOCENTE

La evaluación de la práctica docente es un componente fundamental dentro del proceso general de evaluación académica. Se deben evaluar:

- Los procedimientos de enseñanza.
- La labor docente del profesorado en relación con el logro de los objetivos generales del currículo.
- La programación docente.
- El desarrollo curricular en relación a la racionalidad de espacios y horarios y al funcionamiento de la orientación académica y profesional.

18. BIBLIOGRAFÍA

18.1. INICIACIÓN

- **Alcalde/García:** Metodología de la Programación. McGraw-Hill 1987
- **Gallego:** Técnicas de programación. Grado superior. McGraw-Hill 1998 (ciclo Sistemas de Telecom.)
- **Joyanes Aguilar, Luis:** Fundamentos de Programación, 2ª ed. McGraw-Hill 1996
- **Joyanes; Rodríguez; Fernández:** Fundamentos de Programación. Libro de problemas. McGraw-Hill 1996
- **Sánchez, M.A.; Chamorro et al:** Programación estructurada y fundamentos de programación. Grado superior. McGraw-Hill 1996

18.2. PROFUNDIZACIÓN

Ver el tema.

18.3. OTRAS FUENTES DOCUMENTALES

- **Secretaría de Estado de Educación; ANELE:** Administración de Sistemas Informáticos: Desarrollo Curricular del Ciclo Formativo de Grado Superior de F.P. 1995
- **Secretaría de Estado de Educación; ANELE:** Desarrollo de Aplicaciones Informáticas: Desarrollo Curricular del Ciclo Formativo de Grado Superior de F.P. 1995

©ABACUS

Ap. Correos 72
Beniján - 30570 Murcia
e-mail: info@abacusnt.com

ABACUS