Tema 3: Programación modular

El proceso de diseño, realización y verificación de un programa exige, como cualquier proceso de resolución de un problema, seguir un método. Este consiste en establecer una serie de pasos o etapas de resolución, adecuarse a una conjunto de principios o directivas y utilizar unas herramientas determinadas, generalmente gráficas, para formular la solución de cada etapa.

En este capítulo vamos a tratar primero los criterios metodológicos esenciales de programación, presentando simultáneamente las herramientas más usuales.

Hay demasiados argumentos para justificar un metodología de la programación y para hacerlo desde el momento que se inicia el aprendizaje de la programación. A continuación se enumeran unos cuantos con el propósito de que desde el principio el lector valore la importancia de adecuarse a un método lo más estricto posible en el diseño de programas.

El uso de un metodología trae como consecuencia los siguientes puntos:

- Hace más fácil la verificación de la corrección del programa por el autor o por otras personas. Dicha verificación se puede realizar antes de la codificación, lo que es deseable.

- Facilita igualmente la depuración del programa para eliminar posibles errores tanto lógicos como sintácticos.

- Permite probar el programa de una manera más rigurosa y completa.

Además debe pensarse que un programa requerirá siempre mantenimiento, por tanto habrá que corregir errores no detectados en el proceso de prueba, se deberá adecuarse una vez realizado a situaciones particulares distintas a las previstas en el momento de su realización (por ejemplo un entorno operativo diferente) y habrá que ampliarlo, incorporando tareas o funciones que no habían sido formuladas inicialmente. Normalmente estas tareas de mantenimiento posiblemente las llevará a cabo otra persona o personas distintas de las que realizaron el programa y ¿cómo hacerlo si es difícil o imposible al leer el código del programa y entender qué hace o cómo lo hace?

De lo que se ha expuesto se pueden derivar ya una reglas o directivas metodológicas:

- El programa debe ser legible.

- Debe estar documentado por lo menos en aquellas partes donde interpretar su lógica sea difícil.

- El problema que el programa pretende resolver debe dividirse o descomponerse en problemas más sencillos de manera que puede plantearse su solución, entenderse y verificarse cada parte que resulta de la descomposición sin atender al resto.

Este último criterio de descomposición ya desde el principio fue utilizado en la programación y llevó al planteamiento de la primeras estrategias metodológicas: la descomposición funcional y la programación modular.

3.1. Descomposición funcional y diseño descendente.-

La descomposición funcional como método para el diseño de programas se centra en las tareas o funciones que debe realizar el programa o algoritmo.La primera, será enunciar el problema a resolver con una frase que exprese lo que tiene que hacer el programa. Si, por ejemplo, se trata de "resolver" el algoritmo para la tarea:

hacer un huevo frito

Esta misma frase serviría como enunciado de la función que el algoritmo resuelve.

Seguidamente se expresa la solución en forma de una secuencia de "funciones" más simples:

```
Calentar_aceite
Echar_huevo
Freir_huevo
Sacar_huevo
```

En una tercera fase haciendo una abstracción del resto del problema se elige una de las funciones anteriores, por ejemplo *Calentar_aceite* y se vuelve a descomponer funcionalmente en:

```
Encender_fuego
Poner_sartén
Echar_aceite
MIENTRAS  Aceite NO Caliente
Hacer
      Esperar
FIN_MIENTRAS
```

De igual forma se puede prodecer con *Freir_Huevo*, quedando:

```
MIENTRAS NO cuajado
      Esperar
FINMIENTRAS
SI Con_sal
      Entonces
            Echar_sal
FINSI
```

Como se observa, la descomposición puede seguir adelante, ¿Cuando se para? La respuesta no puede ser general. Si se piensa en un programa de procesamiento de datos hay un primer criterio: cuando la función a la que se llegue se corresponde con una sentencia o secuencia de sentencias válidas del lenguaje en el que se vaya a codificar.

En cualquier caso el objeto de la descomposición funcional es el que debe servir de guía para tratar de descomponer un problema en otros más sencillos o más fáciles de resolver. Se debe descomponer de forma que el grado de complejidad no impida que la lógica de la función se entienda por el programador de un solo vistazo.

De esta manera son dos los principios que rigen la descomposición funcional:

- La abstracción funcional
- El diseño descendente

La abstracción funcional consiste en plantearse un problema descomponiéndolo en otros más elementales sin preocuparse por el momento, a ese nivel, cómo se resolverán los problemas que resultan de la descomposición, haciendo **abstracción** u olvidándose de ellos.

La descomposición funcional se adecua a lo que se denomina diseño descendente de un programa, que consiste en abordar el problema empezando por lo más general y, descendiendo, se van resolviendo tareas o funciones más particulares. Este mismo concepto se puede aplicar en el proceso de codificación y prueba, codificando e incluso probando las funciones más generales en primer lugar, antes de realizar las que resultan de su descomposición.

3.2. Programación modular.-

El principio de abstracción planteado anteriormente conduce de una manera natural a la programación modular. Esta consiste básicamente en dividir un problema en otros más simples para poder resolverlo. De forma que el programa resulta ser un conjunto de pequeñas partes interrelacionadas que dan solución a problemas muy específicos. De esta forma se puede y debe entenderse un programa como un conjunto de secciones (módulos) que resuelven una tarea o realizan una función, todas ellas interrelacionadas.

Antes de seguir conviene precisar mejor que se entiende por módulo. En principio se podría pensar que es una parte del programa que realiza una función específica y bien determinada. En la práctica es deseable que sea así, pero el concepto de módulo no puede ser tan restrictivo.

A continuación se enumeran las características propias de un módulo que derivan de lo dicho hasta ahora:

- Un módulo es un segmento o parte de un programa.
- Tiene un identificador, por ejemplo "*Calentar_aceite*".
- Puede ser invocado desde otro u otros puntos del programa.
- Después de llevarse a cabo, devuelve el control al punto donde fue invocado.
- Consecuentemente hay un punto de retorno en el segmento de código que constituye el módulo. Aunque puede haber más de uno, pero no es deseable.

En el siguiente ejemplo se presenta en pseudocódigo un segmento de programa que incluye un módulo:

```
------
------
PRESENTA_LINEA ()
------
------

FUNCION PRESENTA_LINEA
I = 0
MIENTRAS (I < 20)
HACER
      ESCRIBIR '-'
      I = I + 1
FINMIENTRAS
VOLVER
FIN PRESENTA_LINEA
```

Todos los lenguajes de programación de alto nivel, en mayor o menor grado, permiten la modularización. Los subprogramas en Cobol, procedimientos y funciones en Pascal, funciones en C, Java, etc , son ejemplos de estructuras modulares.

Como se ha dicho, desde el punto de vista de la programación modular, un programa es un conjunto de módulos **intercornectados**. La interconexión entre módulos viene determinada por la invocación que en un punto del programa o desde un módulo se hace a otro módulo diferente. Entre ellos existirá, habitualmente una transferencia de datos, ya que cada módulo recibe una serie de datos requeridos para la función o tarea que tiene que llevar a cabo y modifica otros, que devuelve al módulo que le invocó. El mecanismo por el cual un módulo intercambia información con otros, dependerá no solo de las estructuras modulares que provea el lenguaje en el que se codifica, sino de la solución particular que se de al problema.

3.3.Concepto de función.-

Las funciones son los bloques esenciales, subprogramas que se combinan para construir programas. Una función toma una serie de entradas (o ninguna), realiza alguna tarea concreta y por último produce una salida (o ninguna) llamada valor de retorno.

Se podría pensar en principio que una función que no tenga entradas o que no devuelva salidas no sería muy útil. Pero sin embargo esto no es cierto. Por ejemplo en una función que imprima las cabeceras de un listado, carece de entrada de datos y no produce ninguna salida de los mismos, pero sí realiza una tarea concreta y útil. Existen también funciones que carecen de entradas o salidas. Por ejemplo una función que produzca un número aleatorio no tiene por qué tener ningún valor de entrada. O una función que sitúe el cursor en una posición determinada de la pantalla toma como entradas la fila y la columna, pero no devuelve ningún valor.

La utilidad práctica de las funciones se resume básicamente en tres aspectos importantes dentro del diseño de programas: modularización, ahorro de memoria y ocultación de la información.

1. Modularización. Las funciones se parecen a los subsistemas que los ingenieros emplean para construir dispositivos electrónicos. Cada subsistema realiza una tarea o conjunto de tareas. Algunos los suficientemente versátiles para ser usados en distintos dispositivos, desde satélites hasta equipos estereofónicos.

2. El propósito primordial de las funciones es ayudar a organizar los programas en unidades manejables, permitiendo dividirlos en pequeñas unidades para alcanzar el objetivo final. Por tanto aunque se pueda escribir un programa muy extenso como una única función main(), no es necesario ni recomendable. En su lugar, se debe dividir en una serie de funciones separadas, cada una diseñada para realizar una única tarea.

3. Ahorro de memoria. Si en un programa hay dos o más ocasiones que se utilizan la misma secuencia de instrucciones, con los mismos o diferentes datos, se puede ahorrar memoria construyendo una función que incluya dicha sentencias. Ahorrar espacio es una consecuencia del hecho de que las instrucciones necesitan estar sólo una vez en el programa ejecutable.

4. Ocultar información. Para minimizar el riesgo de que los datos empiecen a corromperse (tomar valores no previstos) sólo deben estar disponibles para aquellas partes del programa que los requieran, usando variables locales, ya que están son creadas y eliminadas dentro de la propia función.

La definición de un función consiste de un encabezamiento y un cuerpo, que sólo debe aparecer una sola vez en el programa.

El encabezamiento de una función consiste en el nombre de la función y una lista de parámetros opcionales separados entre comas y encerrados entre paréntesis.

Por tanto en términos generales está línea de encabezamiento se podría escribir:

```
<identificador_función> (<parámetro_1>, ..., <parámetro_n>)
```

El nombre de la función y los parámetros opcionales serán la forma de invocar a la función y transferir la información necesaria desde el punto de su llamada para su correcta ejecución. Si la función no requiere parámetros, también se debe escribir los paréntesis (ésta es una característica de las funciones).

Los parámetros que están en la definición de una función se llaman parámetros formales. Son unos lugares reservados para los argumentos reales. Estos son la expresiones y valores pasados por el programa que realiza la llamada en el momento de la ejecución y se introducen en los lugares reservados por los parámetros formales.

El encabezamiento de una función no muestra cómo ésta realiza su tarea. Este es el propósito del cuerpo de la función. Un cuerpo de una función se compone de un bloque de sentencias encerradas inicio y el FIN de la función. Dicho bloque debe de contener las declaraciones de datos y definiciones, sentencias ejecutables y una sentencia de retorno. Cada uno de estos elementos son opcionales.

La declaración de las variables y las definiciones deben aparecer al principio el bloque. A menos que se indique lo contrario, todas las variables que se definan dentro del cuerpo de la función son automáticas, y por tanto locales a la función. Esto es, comienzan a existir cuando se pasa el control a la función y son destruidas cuando ésta devuelve el control al punto de su llamada. Y por tanto no son conocidas fuera del ámbito o bloque de la función.

Las sentencias ejecutables son el corazón y el alma de una función. Estas pueden incluir llamadas a otras funciones, las cuales pueden llamar a su ver otras y así sucesivamente. De hecho, una función puede tener una llamada a ella misma, un comportamiento conocido como recursividad. La sentencia DEVOLVER se utiliza para terminar el proceso de una función, transfiriendo el control a la función que realizó la llamada, devolviendo en determinadas ocasiones un valor. La sintaxis de la sentencia DEVOLVER para una función que devuelva un valor es :

```
DEVOLVER [(expresion)]
```

El elemento opcional *expresion* debe proporcionar un resultado del tipo especificado en la declaración. Las funciones del tipo *void* también pueden usar la sentencia *return*. Sin el componente *expresion*, la sentencia *return* devuelve el control a la función que efectuó la llamada. Si no hubiera sentencia *return* y se llega al final de una función, su llave de cierre devuelve el control a la sentencia que la llamó.

Existen dos modelos de pasos de parámetros:

1. **Paso por valor:** El paso por valor quiere decir que se pasan copias de los argumentos proporcionados en la llamada de la función. Ningún cambio de los datos que suceda dentro de la función afecta a las variables originales de las que fueron copiados, asegurándose así la integridad de los datos.

2. **Paso por referencia**: Esta alternativa para pasar datos a las funciones permite a la función llamada acceder a las variables directamente, obviando el mecanismo de protección ofrecido por el método de paso por valor. El paso por referencia permite a una función devolver más de un valor a la función que realizó la llamada; ya que en circunstancia normales sólo se puede devolver un valor único utilizando la sentencia *DEVOLVER*.

A continuación se muestran ejemplos de uso y definiciones de funciones.

Ejemplo 1: Realizar una función que indique si un número es par o no. Utilizar dicha función en un programa.

```
PROGRAMA mostrar_pares
     numero = 0
     continuar = “s”

     MIENTRAS continuar == “s”
     HACER
          leer numero
          SI es_par (numero)
               ENTONCES
                    escribir numero + “ es par”
               SINO
                    escribir numero + “no es par”
          FINSI
          escribir “Desea continuar? (s/n)”
          leer continuar
     FINMIENTRAS
FIN mostrar_pares

FUNCION es_par (VAR n)
     resultado = FALSO
     SI n %2 == 0
          ENTONCES
               resultado = CIERTO
     FINSI

     DEVOLVER resultado
FIN es_par
```

Ejemplo 2: Realizar una función que intercambie el valor de dos números. Utilizar dicha función en un programa.

```
PROGRAMA muestra_mayor
     mayor = 0
     menor = 0

     leer mayor
     leer menor

     SI mayor < menor
          ENTONCES
               intercambia (menor, mayor)
     FINSI

     escribr “El mayor es “ + mayor

FIN muestra_mayor

FUNCION intercambia (VAR REF m, VAR REF n)
     auxiliar = 0

     auxiliar = m
     m = n
     n = auxiliar

FIN intercambia
```

3.4. Objetivos de la programación modular.-

Los objetivos de la programación modular se ajustan parcialmente a los objetivos de diseño propuestos al inicio del capítulo. Por una parte no es cierto, cómo sí lo es en una metodología dirigida por la descomposición funcional, que el diseño tenga que ser descendente.

La programación modular no presupone un diseño descendente, sino que en muchos casos puede propiciar el diseño ascendente: identificar aquellas funciones o tareas elementales que no están disponibles pero que permitirían un grado de abstracción superior en la resolución del problema planteado, implementarlas o realizarlas antes de iniciar el diseño de la arquitectura modular en su conjunto. En este caso no se empieza por lo más general sino por lo más particular, aunque esto no es incompatible con que al abordar el problema en su conjunto si se sigue una técnica de diseño descendente.

Para aclarar lo que se esta explicando, piénsese en tratar de resolver un problema matemático que exija el tratamiento de números complejos. El lenguaje no

dispone de herramientas de tratamiento para éstos. Será razonable que antes de plantearse sistemáticamente la resolución del programa, se empiece por construir, verificar y probar funciones que lleven a cabo las operaciones sobre complejos. El objeto es disponer de las herramientas básicas, antes de empezar, es decir funciones implementadas por el programador, que permitan tratar el problema con un nivel de abstracción superior.

El diseño ascendente es muy común en la realización de funciones genéricas para la gestión de interfaces de usuario (presentación de mensajes, presentación de ventana de ayuda, gestión de menús, etc.) o de tratamiento de dispositivos de salida como la impresora.

No es por tanto el diseño descendente, como tampoco es la abstracción funcional, lo que caracteriza a la programación modular.

Si se quiere decir concisamente que es lo propio de la programación modular, sería, como ya se ha repetido varias veces, el concebir el programa como una organización de componentes interrelacionados, de forma que cada componente pueda ser, en un determinado grado, pensado, verificado y probado independientemente. Y, en consecuencia, sea viable utilizar componentes o módulos, preconstruidos, reutilizar módulos ya diseñados y construidos por el mismo programador o por otros. En resumen, se tratar de aplicar a la Ingeniería del Software el mismo tipo de técnicas de diseño que se utiliza en la Ingeniería del Hardware, donde al construir un sistema no se diseñan a propósito cada uno de los circuitos integrados que van a ser componentes del sistema, sino que se recurre al catálogo de módulos ya construidos, que se adapten a las especificaciones necesarias.

3.5. Características de un módulo.-

En los apartados anteriores se ha referido al hablar de la descomposición funcional que se debería seguir fraccionando una función hasta conseguir que su grado de complejidad fuera tan pequeño, que el mismo programador, o mejor un observador neutral o revisor de diseño, pudiera comprender lo que hace sin dificultad. Pero esto resulta muy impreciso, ¿cómo se puede evaluar o medir el grado de complejidad de una función o de un módulo? A continuación se describen tres características de un módulo que permiten hacerlo: el tamaño, la complejidad lógica y el número de elementos de la interfaz.

Es evidente que cuanto más pequeño sea un módulo más sencillo será de entender, verificar y probar. En este sentido hay autores que recomiendan que no supere el tamaño de una pantalla o de una página impresa. Aunque esto resulta arbitrario, se debe aceptar como regla que un módulo no supere las 40 o 50 líneas sino excepcionalmente. La longitud o número de líneas de un módulo no dice sin embargo gran cosa sobre su dificultad de comprensión lógica. En este sentido son más significativas las otras dos características.

La complejidad lógica se puede determinar por el número de sentencias de

control (tanto de selección como iterativas) que tiene el módulo. Si es muy elevado, una persona al leer el código no podrá entenderlo en su totalidad. Tendrá necesidad de centrar su atención en una parte o segmento de él y hacer abstracción del resto. Es deseable que este proceso de abstracción se hubiera realizado por el programador fraccionando el módulo en otros más simples.

McCabe propuso cuantificar la complejidad lógica midiendo lo que denominó complejidad ciclomática (el número de caminos alternativos que puede seguir el flujo del programa en el módulo).

En el ejemplo:

```
SI cond1
      sentencia1;
SINO

      sentencia2;
      MIENTRAS cond2
            sentencia3;
      FIN_MIENTRAS
      sentencia4;
FINSI
```

Como puede observarse, el número de caminos que puede seguir el flujo del programa son 3, por tanto el número que cuantifica la complejidad ciclomática o número de McCabe es de 3.

Como se puede comprobar utilizando el ejemplo, la complejidad ciclomática se puede evaluar utilizando las siguientes reglas:

- Se cuentan el número de sentencias de selección o de iteración y se le añade 1.

- Por lo tanto, cada sentencia de control cuenta por 1. Pero esto será si la condición es un expresión lógica elemental. Si la condición es una expresión lógica compuesta como:

  SI **(a>b) Y (a<c)**

  entonces la sentencia cuenta doble. Es decir cuenta un valor igual al número de términos elementales de la expresión lógica de la condición de control.
- Si se trata de una sentencia CASO cuenta por un valor igual al numero de alternativas disminuido en 1.

Se propuso que la complejidad ciclomática, así determinada, no debería ser superior a 10.

Es cierto, sin embargo que esto sólo no puede medir la complejidad lógica del módulo porque no atiende a las referencias a los datos, que pueden suponer una

dificultad adicional para entender qué hace el mismo.

En los dos segmentos siguientes:

```
SI seguir
      total=total+nuevovalor;
FIN_SI

SI valido[contador]
      total[cliente][contador]= total[cliente][contador]+nuevovalor;
FIN_SI
```

aunque la complejidad ciclomática es la misma, la dificultad de comprensión del segundo es mayor por el hecho de utilizar estructuras de datos, que implican atender al significado de un número mayor de elementos de datos.

Finalmente, el tercer factor a considerar para estimar la calidad del diseño de un módulo es la complejidad de la interfaz de comunicación con los otros módulos. En el caso de las funciones se trata del número de parámetros que recibe y el valor de retorno. Es deseable que la interfaz sea mínima, que el número de elementos de datos que se transfieren sea lo más reducido posible y que éstos no sean estructurados. Este será un factor determinante no sólo para que se más simple el módulo, sino su invocación desde otro módulo. Cuanto mayor sea el número de parámetros de una función más difícil será de entender, verificar y depurar la función, pero también será más difícil de entender la llamada que se haga a la función y por lo tanto el comportamiento de la función en la que se hace la llamada.

El criterio de diseño expuesto, como en los casos anteriores, no se puede considerar como absoluto. Es interesante señalar que no se debe minimizar el número de parámetros de una función a costa de la claridad o carácter significativo de su llamada. Es mejor construir una función que requiera una llamada como:

```
pinta_recuadro(x0,y0,x1,y1)
```

que otra como:

```
pinta_recuadro(p0,p1)
```

aunque el número de parámetros sea menor. (Además téngase en cuenta que, en el ejemplo, el número de datos elementales en ambos casos es de 4, puesto que en la segunda función *p0* y *p1* son puntos, es decir, o estructuras de datos de dos elementos.

3.7. Cohesión y acoplamiento.-

A parte de las tres características explicadas en el apartado anterior, existen otras características estructurales que deben guiar el diseño: la cohesión de un módulo y el acoplamiento entre los diferentes módulos que forman la arquitectura

modular.

La cohesión evalúa el grado de interrelación entre los elementos que constituyen un módulo, mientras que el acoplamiento evalúa el grado de interconexión entre los módulos.

Un buen diseño debe minimizar el acoplamiento, aumentando la independencia de cada módulo respecto a los demás, y hacer máxima la cohesión en cada módulo, haciendo que todos sus elementos (de datos, de proceso y de control) estén fuertemente relacionados entre sí.

El acoplamiento es mínimo cuando la conexión de un módulo con otro solo se da en un punto: el de invocación o disparo de un módulo desde el superior y el de entrada o inicio del módulo en el caso del módulo subordinado. Esto será cierto si los módulos son funciones: existe conexión únicamente en el punto donde la primera función hace la llamada a la segunda. En el siguiente ejemplo se muestra el pseudocódigo de invocación a una función y el segmento del mapa de estructura que representa la conexión de las funciones como módulos:

```
FUNCION PRINCIPAL
-----
-----
N_TOTAL = EVALUAR_TOTAL (N_FACT)
-----
-----
FIN PRINCIPAL
-----
-----

FUNCION EVALUAR_TOTAL (var N_FACT)
----
----
DEVOLVER TOTAL
FIN EVALUAR_TOTAL
```

Existen otros tipo de conexiones entre módulos. La más típica y frecuente es la conexión **por entorno común**, que consiste en que las funciones o módulos comparten variables de ámbito global (accesibles desde ambas). Si una función utiliza, accede, a una variable que es asignada en otra función, existe una conexión entre las dos funciones, no porque la primera derive el control a la segunda sino porque accede a una variable que es asignada en la segunda y, en consecuencia, el comportamiento de la primera función está ligado en ese punto al de la segunda. Para entender mejor esto, piénsese que si la función en la que se hace la asignación de la variable de entorno común está mal programada y falla, la otra función también podrá fallar, pero no porque esté mal programada sino porque está conectada con otra que sí lo está.

Otro ejemplo habitual de conexión no deseable entre funciones es el caso en el que se pasan parámetros de control a una función, como en el ejemplo:

```
presenta_cliente(identificador_cliente, editando)
```

Aunque parezca que el parámetro *editando* no es sino uno más como lo es *identificador_cliente*, no es así, se trata, y para ello se transfiere, de un parámetro de control que va a determinar la derivación del flujo en la función que lo recibe. Existe por lo tanto una conexión de la función *presenta_cliente* respecto de la función que la invoca y en la que se asigna el valor del parámetro *editando*. En el ejemplo sería mejor construir dos funciones:

```
visualiza_cliente(identificador_cliente)
edita_cliente(identificador_cliente)
```

que no serían dependientes del valor de ningún parámetro de control.

Resumiendo, para que el acoplamiento sea mínimo es deseable que la conexión de una función se límite a los parámetros de llamada y el valor de retorno y que dichos parámetros sean datos y no parámetros de control. Se debe tener en cuenta que no podrá evitarse en muchos casos un cierto grado de acoplamiento, sobre todo el acoplamiento por entorno común, en el diseño de programas, y, en este caso lo importante es ser conscientes de ello y especificar o documentar el acoplamiento para que en el momento en que se haga cualquier cambio en uno de los módulos se tenga en cuenta si puede afectar al comportamiento de otros.

Mientras que el acoplamiento evalúa la interconexión entre módulos y debe minimizarse, la interrelación entre los elementos que constituyen un módulo, cohesión, debe ser máxima. La cohesión de un módulo viene determinada por el grado de afinidad entre la tareas o procesos que lleva a cabo el módulo y entre los datos que trata. Así podría puede pensarse que una función siempre será cohesiva, pues no realiza sino **una** tarea especifica y bien determinada. Pero no es, ni puede ser, siempre así. Son muy frecuentes módulos como éstos cuya descripción se escribe a continuación:

inicializar

leer y presentar cliente

que realizan dos o más tareas. En el primero puede llevar a cabo las funciones o tareas relacionadas con la preparación de un proceso determinado, que consistirá en verificar la existencia de un archivo, abrirlo, tal vez leer la estructura del archivo, etc. Se dice que en el módulo inicializar existe cohesión temporal, es decir se incluyen tareas o funciones que se deben realizar al mismo tiempo o consecutivamente en el tiempo. En el segundo ejemplo, la conjunción "y" indica claramente que se realizan dos tareas. Ambas están relacionadas por el hecho de que actúan sobre datos comunes. En ocasiones será razonable dividirlo en dos funciones, pero no necesariamente será así. A este tipo de cohesión o interrelación entre las tareas que lleva a cabo el módulo se le denomina cohesión por comunicaciones.

El tipo de cohesión óptima o máxima es la denominada cohesión funcional, en la

que todas las operaciones de un módulo colaboran a la realización de una tarea bien definida. Un método para determinar si existe cohesión funcional y no se da otro tipo de cohesión es describir la tarea que realiza un módulo mediante un enunciado: "verbo + objeto directo", como por ejemplo

imprimir cliente
calcular total_gastos

Si es posible hacerlo y no aparece más de un verbo o más de un objeto directo, y, en consecuencia no se utiliza en el enunciado conjunciones como "y" o separadores como "," que indique las presencia de una lista de acciones o de objetos o estructuras de datos sobre las que se realizan las tareas, existirá exclusivamente cohesión funcional.

Este método no es en ocasiones muy riguroso porque, aunque el enunciado se ajuste a lo prescrito, puede ser que el verbo enmascare varias operaciones o tareas, como en el caso del *inicializar* del ejemplo de antes, o el objeto directo o de tratamiento sea en realidad una estructura de datos o varias, como sería el caso de imprimir cliente, si cliente representa el *NOMBRE* y *APELLIDOS* del mismo, más la *CUENTA*, mas la relación de los *MOVIMIENTOS* en la *CUENTA*, más el *ESTADO* de la *CUENTA*, etc.

3.8. Ejercicios propuestos.-

1º Realizar un programa que calcule e área y perímetro de un rectángulo.

2º Diseñar un programa que obtenga el salario neto de n trabajadores de acuerdo a las siguientes premisas:

- Las 38 primeras horas semanales se cobran a la tarifas ordinaria.
- Cualquier hora extra realizada se cobra a 1.5 veces la tarifa ordinaria.
- Las primeras 50000 pesetas están libres de impuestos. Las siguientes 40000 están sometidas a unas retenciones del 25% y las restantes al 45%.

Pedir por teclado el número de trabajadores y la tarifa ordinaria.

3º El número de combinaciones de m elementos tomados de n en n es:

m sobre n = (m!) / (m! (m-n)!)

donde m! = m * (m-1) * ...* 1

Realizar un programa que lea desde el teclado los valores de m y n. Compruebe que m es mayor que n y calcule el número de combinaciones.

4º El máximo común divisor (m.c.d.) entre dos números enteros mayores que cero viene dado por la siguiente forma:

$$m.c.d.(m,n) = \begin{cases} n, & \text{si } m \,\%\, n = 0 \\ m.c.d.(n,r), & \text{si } m \,\%\, n = r \text{ con } r > 0 \end{cases}$$

Escribir un programa que calcule el máximo común divisor entre dos números.

5º Para obtener el número del Tarot de una persona, hay que sumar los números de su fecha de nacimiento y reducirlos a un sólo dígito. Realizar un programa que lea una fecha de teclado y escriba el número del Tarot en base a la fecha leída. La fecha estará formada por tres números enteros, el día, el mes y el años (4 dígitos).

Ejemplo:

Supongase que una persona nace el día 1 de Julio de 1966. La suma de los tres números da como resultado 1 + 7 + 1966 = 1974. El resultado obtenido no está formado por un sólo dígito, por lo que habrá que sumar las cuatro cifras que componen el número: 1 + 9 + 7 + 4 = 21. Al igual que antes, el resultado no está formado por un dígito. por lo que hay que repetir el proceso, 2 + 1 = 3. El resultado obtenido es el número del Tarot 3.

6º Crear una función que calcule la temperatura media de un día a partir de la temperatura máxima y mínima. Crear un programa principal, que utilizando la función anterior, vaya pidiendo la temperatura máxima y mínima de cada día y vaya mostrando la media. El programa pedirá el número de días que se van a introducir.
Nota: Si se te ocurre alguna otra función puedes utilizarla.

7º Diseñar una función que calcule el área y el perímetro de una circunferencia. Utiliza dicha función en un programa principal que lea el radio de una circunferencia y muestre su área y perímetro.

8º Crear el programa, utilizando la técnica de programación estructurada, para un determinado comercio en el que se realiza un descuento dependiendo del precio de cada producto. Si el precio es inferior a 6 euros, no se hace descuento; si es mayor o igual a 6 euros y menor que 60 euros, se hace un 5 por 100 de descuento, y si es mayor o igual a 60 euros, se hace un 10 por 100 de descuento. Finalizar visualizando el precio inicial, el valor del descuento y el precio final.

9º Crear una programa que pida el límite inferior y el límite superior de un intervalo. Si el límite inferior es mayor que superior, intercambiar las variables. A continuación se irán pidiendo números enteros positivos hasta que se introduzca un número negativo. Al terminar se mostrará la siguiente información:

- Cantidad de números dentro del intervalo abierto.
- ¿Se ha introducido algún número igual a alguno de los límites del intervalo?

Algunas de las funciones que puedes utilizar son las siguientes:

- Leer_limites_del_intervalo: Función que lee los dos límites del intervalo.
- Intercambiar_variables: Función que intercambia el valor de dos variables.
- Numero_fuera_del_intervalo: Función que indica si número está fuera del intervalo.
- Comprobar_limites: Función que te dice si el número introducido es igual a algunos de los límites.

Si piensas en alguna otra función no dudes en utilizarla.

10º Escribir un programa, haciendo uso de funciones, que visualice un calendario de la forma:

L	M	M	J	V	S	D
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

El usuario indica únicamente el mes y el año.

La fórmula que permite conocer el día de la semana correspondiente a una fecha es:

• Meses de enero o febrero:

n= a + 31 *(m-1) + d (a-1) div 4 – 3 * ((a+99) div 100) div 4;

• Restantes meses:

n= a + 31 * (m-1) + d - (4*m + 23) div 10 + a div 4 - (3*(a div 100 + 1)) div 4;

Donde a=año; m=mes; d=día;

Nota: n mod 7 indica el día de la semana (1=lunes, 2= martes, etc.)
div es \, división entera

11º Queremos crear un programa que trabaje con fracciones a/b. Para representar una fracción vamos a utilizar dos enteros:

Vamos a crear las siguientes funciones para trabajar con funciones:

• Leer_fracción: La tarea de esta función es leer por teclado el numerador y el denominador. Cuando leas una fracción debes simplificarla. La función de simplificar se verá a continuación.

• Escribir_fracción: Esta función escribe en pantalla la fracción. Si el dominador es 1, se muestra sólo el numerador.

• Calcular_mcd: Es una función auxiliar que nos va ayudar a realizar las otras funciones. Esta función recibe dos número y devuelve el máximo común divisor.

• Simplificar_fracción: Esta función simplifica la fracción, para ello hay que dividir numerador y dominador por el mcd del numerador y denominador.

• Sumar_fracciones: Función que recibe dos funciones n1/d1 y n2/d2, y calcula la suma de las dos fracciones.

• Restar_fracciones: Función que resta dos fracciones, igual que la anterior pero al calcular el numerador hay que restar.

• Multiplicar_fracciones: Función que recibe dos fracciones y calcula el producto.

• Dividir_fracciones: Función que recibe dos fracciones y calcula el cociente.

Crear un programa que utilizando las funciones anteriores muestre el siguiente menú:

1. Sumar dos fracciones: En esta opción se piden dos fracciones y se muestra el resultado.
2. Restar dos fracciones: En esta opción se piden dos fracciones y se muestra la resta.
3. Multiplicar dos fracciones: En esta opción se piden dos fracciones y se muestra el producto.
4. Dividir dos fracciones: En esta opción se piden dos fracciones y se muestra el cociente.
5. Salir