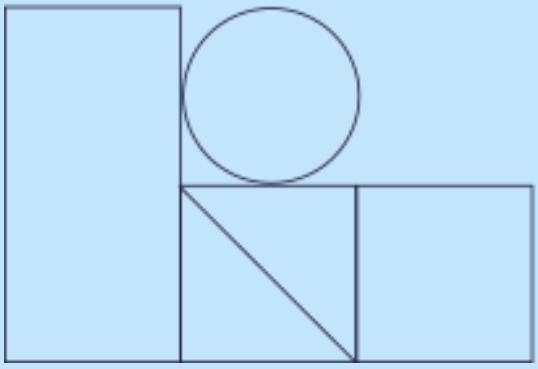
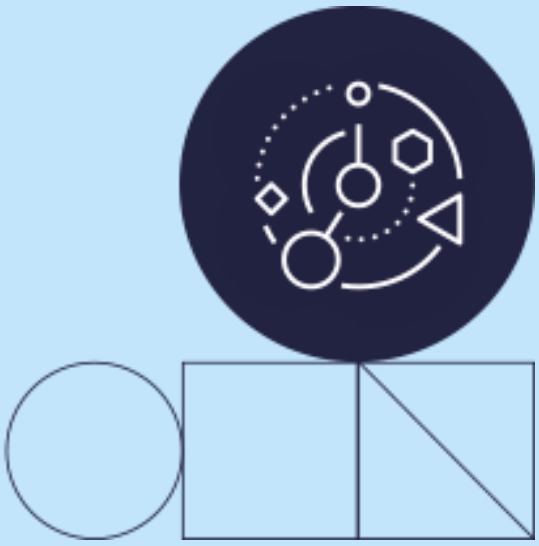
IBM SkillsBuild

IBM

Conceptos básicos de programación

Segmentación de programación modular, recursividad, global





Índice

Introducción

A la hora de desarrollar un programa pueden planteársenos preguntas del tipo: ¿sigue nuestro programa una programación en módulos? ¿Tiene una estructura básica o es un caos? ¿Están los procedimientos y funciones bien desarrollados? Nuestro objeto de estudio en este tema será conocer cómo es una programación modular, su estructura básica y cómo deben ser las funciones y procedimientos de los programas. Se presenta históricamente como una evolución de la programación estructurada para solucionar problemas de programación más grandes y complejos de lo que ésta puede resolver.

La razón principal para utilizar un ordenador es para resolver problemas (en el sentido más general de la palabra), ó en otras palabras, procesar información para obtener un resultado a partir de unos datos de entrada.

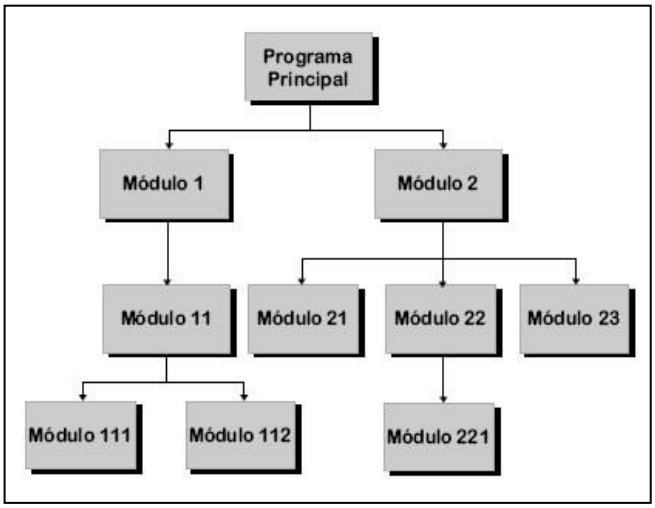
Durante la corta historia de los computadores, el modo de programar ha sufrido grandes cambios. La programación era en sus comienzos todo un arte (esencialmente cuestión de inspiración); posteriormente diversas investigaciones teóricas han dado lugar a una serie de principios generales que permiten conformar el núcleo de conocimientos de una metodología de la programación. Ésta consiste en obtener “programas de calidad". Esto se puede valorar a través de diferentes características que se exponen a continuación, no necesariamente en orden de importancia:

- **La corrección** del programa que, obviamente, es el criterio indispensable, en el sentido de que se desean obtener programas correctos que resuelvan el(los) problema(s) para los que están diseñados.
- **La comprensibilidad**, que incluye la legibilidad y la buena documentación, características que permiten una mayor facilidad y comodidad en el mantenimiento de los programas.
- **La eficiencia**, que expresa los requerimientos de memoria y el tiempo de ejecución del programa.
- **La flexibilidad** o capacidad de adaptación del programa a variaciones del problema inicial, lo cual permite la utilización del programa durante mayor tiempo.
- **La “transportabilidad "**, que es la posibilidad de usar el mismo programa sobre distintos sistemas sin realizar cambios notables en su estructura.

Teniendo en cuenta que un programa, a lo largo de su vida, es escrito solo una vez, pero leído, analizado y modificado muchas más, cobra una gran importancia adquirir técnicas de diseño y desarrollo adecuadas, como la programación modular que se presenta a continuación, para obtener programas con las características mencionadas.

Programación modular

La programación modular es la técnica de programación basada en la filosofía del diseño descendente, que consiste en dividir el problema original en diversos subproblemas (y estos a su vez en otros más pequeños, obteniendo una estructura jerárquica o en árbol) que se pueden resolver por separado, para después recomponer los resultados y obtener la solución al problema. Un subproblema se denomina módulo y es una parte del problema que se puede resolver de manera independiente.



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 en árbol que permite resolver el problema de programación planteado.

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

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.

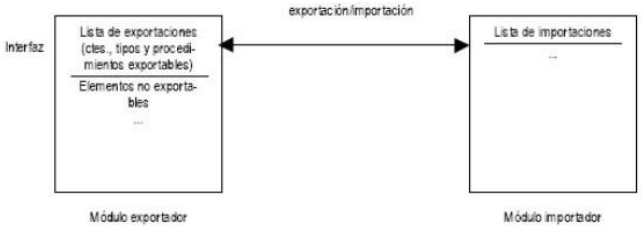
Ventajas

- Facilita el diseño descendente
- Disminuye la complejidad del algoritmo
- Disminuye el tamaño total del programa
- Reusabilidad: ahorro de tiempo de programación
- División de la programación entre un equipo de programadores reducción del tiempo de desarrollo
- Facilidad en la *depuración:* comprobación individual de los módulos
- Programas más fáciles de modificar
- Estructuración en *librerías* específicas (biblioteca de módulos)

Interfaz

Un módulo puede ofrecer a la "comunidad" de módulos sus propios recursos, tipos y procedimientos dentro de lo que se conoce como sección de interfaz del módulo. Normalmente es posible exportar cinco tipos de elementos: constantes, tipos, variables, procedimientos y funciones.

La importación y la exportación constituyen acciones simétricas. Para que 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.



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.

Tipos de módulos

Según su función dentro del programa:

- Programa o módulo principal
- Módulos o módulos secunda

Según su **uso:**

- Funciones: devuelven un valor (evaluación de la función)
- Procedimientos: realizan tareas, pero no devuelven ningún valor directamente.

Según los mecanismos de activación:

- Invocados por referencia
- Invocados mediante interrupción (en entornos de tiempo real)

Según el camino de control (describe la forma en la que se ejecuta internamente):

- Módulos convencionales: tienen una única entrada y una única salida y ejecutan secuencialmente una tarea en cada momento.
- Módulos reentrantes: diseña de forma que de ninguna manera pueda modificarse a si 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 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 conrrutina.

Refinamiento y modularidad

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.

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).

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.

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, p1 y p2, si C(p1) > C(p2) 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(pl+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(pl+p2) > E(p1) + E(p2)

Esto 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 seria insignificantemente 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.

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 mas 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 subproblemas y el número de módulos crece, se produce un incremento de los interfaces entre éstos con la consiguiente complejidad asociada.

Ocultación de la información

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.

Independencia funcional y calidad del software

El concepto de independencia funcional es una derivación directa del de modularidad y de los conceptos de abstracción y ocultamiento de información.

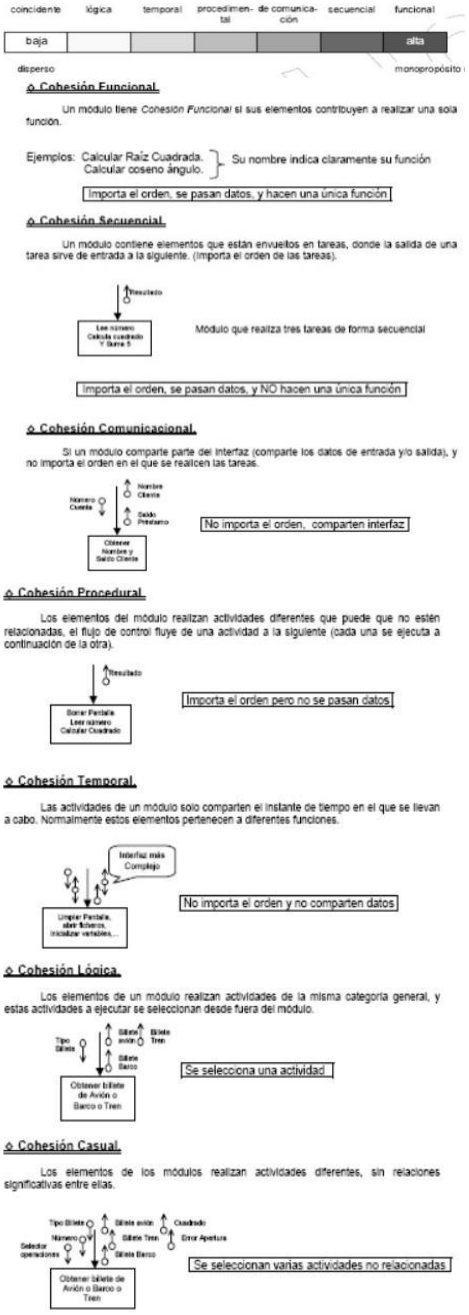
La independencia funcional se adquiere desarrollando módulos con "una clara" función y una "aversión" a una excesiva interacción con otros módulos. Dicho de otra forma, se trata de diseñar software de forma que cada módulo se centre en una subfunción específica de los requisitos y tenga una interfaz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

Es importante la independencia funcional en el desarrollo de aplicaciones informáticas porque 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.

Cohesión

Mide el grado de conexión funcional entre los elementos (instrucciones, definición de datos, llamadas a módulos) de un mismo módulo. Cuanto más fuerte sea la cohesión mejor será el mantenimiento del módulo.



Acoplamiento

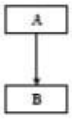
Es el grado de interdependencia entre los módulos de un sistema. Este criterio debe minimizarse, con ello se logra atenuar el ruido del sistema (los errores de un módulo no se propagan a otros) y se realiza el mantenimiento sin mirar en el interior de otros módulos.

Acoplamiento Normal.

La diferencia entre los tres tipos de acoplamiento normal radica en el tipo de información que se intercambian.

Dos módulos A y B están **acoplados normalmente** si se cumplen las condiciones:

1.- A invoca a B.
2.- B realiza su función retomando el control a A.
3.- Toda información que comparten o se pasan, es por medio de los parámetros presentes en la llamada.

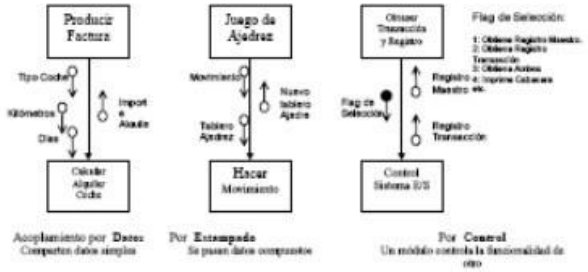


Acoplamiento (Normal) por Datos.
Dos módulos A y B están acoplados por datos si están acoplados normalmente y todos los datos que se intercambian son elementales.

Acoplamiento (Normal) por Estampado.
Dos módulos A y B acoplados normalmente están acoplados por estampado si uno le pasa a otro datos compuestos como vectores y registros. (Los datos compuestos provocan indirección, ya que se debe consultar su estructura en algún sitio).

Acoplamiento (Normal) por Control.
Dos módulos A y B acoplados normalmente están acoplados por control si uno le pasa a otro datos con la intención de controlar su lógica interna.

Ejemplos Acoplamiento Normal:



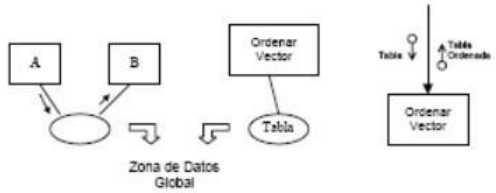
A un módulo se le debe enviar únicamente la información que necesite. Síntoma de una mala organización de los módulos es la presencia de **datos vagabundos**: datos que viajan por gran parte del D.E. sin ser utilizados por la mayoría de los módulos por los que pasan.

Acoplamiento Común (o global).

Dos módulos A y B están acoplados globalmente si se refieren a una misma zona de datos o variable global

Las variables globales no son aconsejables porque:

- Un error en un módulo puede aparecer en otro que comparte la variable.
- Es difícil saber que módulo modifica los datos
- Poca reutilización de módulos y difícil mantenimiento.



Acoplamiento por contenido.

Dos módulos están acoplados por contenido si uno se refiere al interior de otro de una de las siguientes maneras: modificando o leyendo sus datos internos o saltando al interior de su código (GOTO).

• Comparación de los distintos tipos de Acoplamiento.

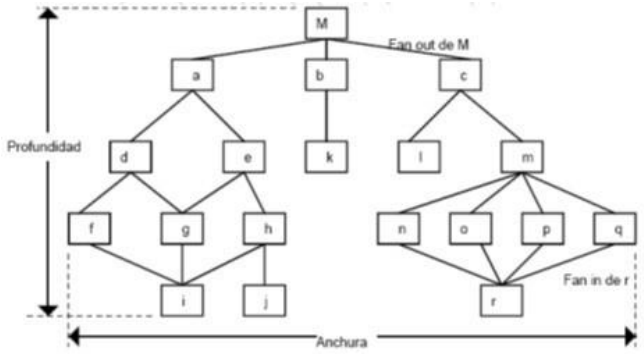
Tipo de Acoplamiento	Modificabilidad	Legibilidad	Reusabilidad Módulos
Por Datos	Buena	Buena	Buena
Por Estampado	Buena	Media	Media
Por Control	Pobre	Pobre	Pobre
Global	Media	Mala	Pobre
Contenido	Mala	Mala	Mala

Visión pesimista: Dos módulos pueden presentar varios tipos de acoplamiento, en este caso se considera que tienen el peor de los acoplamientos que presentan.

Estructura del programa y jerarquía 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.

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.



- 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.
- 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.

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.

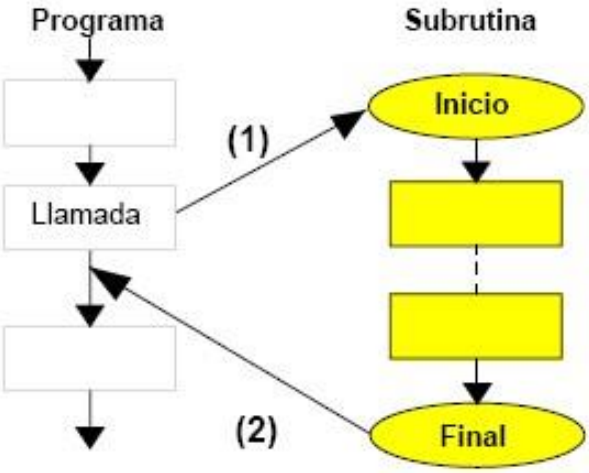
Se trataría de sustituir todo un conjunto de instrucciones que puede incluir cualquier combinación de las denominadas estructuras básicas (secuencial, condicional, iterativa) por un identificador que puede incorporar la declaración de valores. Posteriormente, en cualquier lugar del programa se podrá invocar al conjunto de instrucciones nominado utilizando el identificador.

Tras la ejecución, y según el tipo de estructura utilizada, se pueden obtener resultados devueltos.

Procedimientos

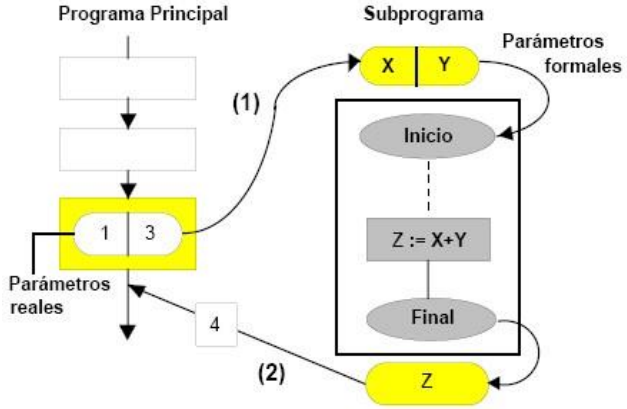
Sirven para definir partes de un programa mediante la asociación de un identificador. Posteriormente dichas partes se pueden activar utilizando sentencias de llamada. Un procedimiento es pues un algoritmo diseñado de tal modo que es susceptible de ser llamado por otros algoritmos que, a su vez, pueden ser procedimientos

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 llamada.



Parámetros

Los parámetros *son tanto las variables usadas en la definición del procedimiento como los valores utilizados en la llamada*. A los primeros se los denomina parámetros formales y a los segundos parámetros reales. 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 {pf1, pf2,..., pfn} y la de parámetros reales de cualquier llamada al mismo {pr1, pr2,..., prm} deben cumplir las siguientes condiciones:

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

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.

Paso de Parámetros

- ***Por valor:*** únicamente nos interesa el valor, no las modificaciones que pueda tener dentro del subalgoritmo. Se trabaja con una copia del valor pasado. Son parámetros unidireccionales, que pasan información desde al algoritmo al subalgoritmo. Puede ser cualquier expresión evaluable en ese momento.
- ***Por referencia:*** se pasa una referencia a la posición de memoria donde se encuentra dicho. Se utilizan tanto para recibir como para transmitir información entre el algoritmo y el subalgoritmo. Debe ser obligatoriamente una variable.

Definición de procedimientos

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 procedure, 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 los componen las declaraciones e instrucciones en las que se ejecuta el algoritmo propio del procedimiento.

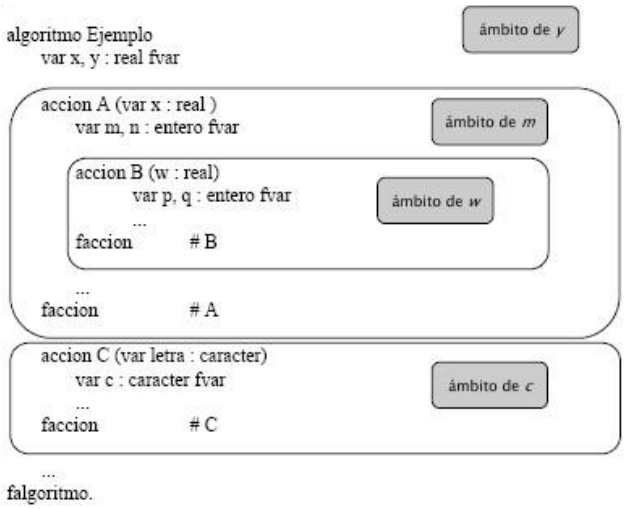
Funciones

Son procedimientos 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.

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.

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.

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



- 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.

Ámbito de identificador

Conjunto de sentencias donde puede utilizarse ese identificador.

- **Variables locales:** variable declarada dentro de un subprograma y, por tanto, sólo disponible durante el funcionamiento del mismo.
- **Variables globales:** variables declaradas en el programa principal y, por ello, pueden ser utilizadas por el programa principal y por todos sus subprogramas.
- ***Efecto Lateral:*** efecto de un módulo sobre otro módulo que no es parte de la interfaz definida explícitamente entre ellos.

Reglas para el cálculo del ámbito de un identificador:

- Un identificador declarado en un bloque es accesible únicamente desde ese bloque y todos los bloques incluidos en él (se considera *local* a ese bloque). Un parámetro formal se considera también una declaración local al bloque de la función.
- Los identificadores declarados fuera de cualquier bloque se consideran *globales* y pueden ser utilizados desde cualquier punto del fichero.
- Cuando tenemos un bloque dentro de otro bloque y en ambos se declaran identificadores con el mismo nombre, el del bloque interno "oculta" al del bloque externo.

Recursividad

- Un objeto es recursivo si su definición requiere la definición previa del objeto en un caso más sencillo.
- Una función es recursiva si su resolución requiere la solución previa de la función para casos más sencillos.
- **Un algoritmo** A que resuelve un problema P es recursivo si está basado **directa o indirectamente** en sí mismo.

Los algoritmos recursivos son especialmente apropiados si el propio problema o el propio cálculo a realizar o la propia estructura con la que trabaja el problema aceptan una definición recursiva. Sin embargo, la existencia de tales definiciones no garantiza que la mejor forma de resolver un problema pase por utilizar un algoritmo recursivo.

Siempre que diseñemos un algoritmo recursivo habrá que asegurarse no sólo de que el número de llamadas es finito (la recursión acaba) sino también de que es pequeño, ya que hace falta espacio de memoria (pila de recursión) para almacenar en cada llamada los objetos locales, los parámetros de la llamada y el estado del proceso en curso para recuperarlo cuando acabe la llamada actual y haya que reanudar la antigua.

Tipos de recursión

Recursión lineal:

Si cada llamada recursiva genera, como mucho otra llamada recursiva

- FINAL: si la llamada recursiva es la última operación que se efectúa, devolviéndose como resultado lo que se haya obtenido de la llamada recursiva sin modificación alguna.
- NO FINAL: El resultado obtenido de la llamada recursiva se combina para dar lugar al resultado de la función que realiza la llamada.

Ejem: Función recursiva que calcule el factorial de un número entero positivo.

```
funcion Fact(n:entero) retorna entero
inicio
  si n=0 entonces
    devolver(1)
  si_no
    devolver(n * Fact(n-1))
  fin_si
fin_funcion
```

Recursión múltiple:

Si alguna llamada puede generar más de una llamada adicional.

Ejem: Función recursiva que calcule el término n de la serie de Fibonacci. Siendo n entero positivo.

```
funcion Fibo(n:entero) retorna entero
inicio
  si n=0 o n=1 entonces
    devolver(1)
  si_no
    devolver(Fibo(n-1) + Fibo(n-2))
  fin_si
fin_funcion
```

Recursión anidada:

Hay recursión anidada cuando uno de los argumentos de la función recursiva es el resultado de la llamada recursiva.

Ejem: Calcular la raiz n de un número entero x

```
Funcion raices (n,x: entero) retorna real
inicio
    según sea
    n=1: devolver(raiz2(x))
    n>1: devolver raices(n – (n div 2), raices((n div 2),x))
  fin_según
fin_funcion
```

Etapas del diseño recursivo:

Un diseño recursivo constará de las siguientes etapas:

- Definición del problema.
- Análisis de casos. Identificación de la función limitadora.
- Trascripción algorítmica y verificación de cada caso.
- Validación de la inducción: la función limitadora decrece estrictamente en las llamadas.

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.

Librerías

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. Las bibliotecas son un claro ejemplo de reutilización del software*.

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.
- **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".

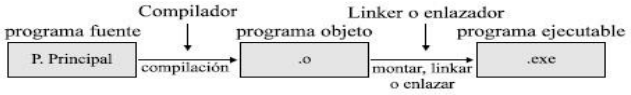
APIs

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.

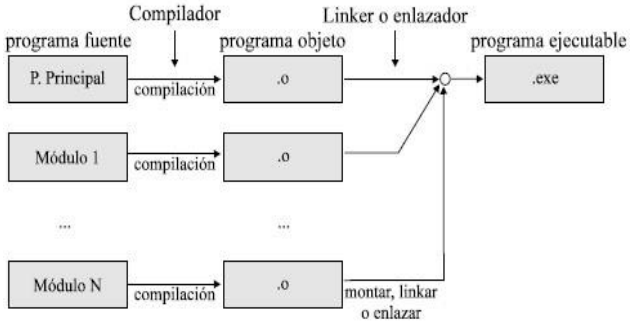
Cuando dividimos un problema en distintos módulos, no siempre ha de corresponder cada módulo a un subprograma. Puede corresponder a todo un conjunto de subprogramas que estarán agrupados en un fichero independiente. La principal utilidad de dividir el programa en ficheros distintos es que se pueden programar y compilar por separado. De esta manera no hace falta compilar cada vez todo el programa.

Esto se realiza normalmente para programas muy grandes o simplemente cuando queremos realizar una librería de subprogramas.

- Compilación de un programa en un sólo fichero:



- Compilación de un programa en varios ficheros:



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 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 (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.

El enlace estático produce un gran desperdicio de memoria dado que cada programa incluye una copia propia de cada librería utilizada (ej: imaginar el manejo de ventanas en Windows)

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.