Programación Modular

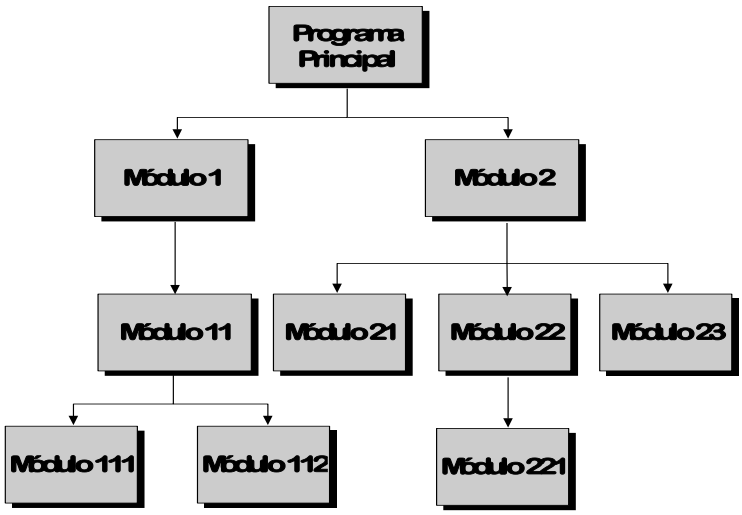
1. Introducción

La forma más razonable de encarar el desarrollo de un programa complicado es aplicar lo que se ha dado en llamar **Programación Top - Down** o Programación descendente.

El diseño descendente resuelve un problema efectuando descomposiciones en otros problemas más sencillos a través de distintos niveles de refinamiento. La programación modular consiste en resolver de forma independiente los subproblemas resultantes de una descomposición. El resultado de dividir reiteradas veces un problema en problemas más pequeños es una estructura jerárquica o en árbol.

El código de un programa, además de para realizar una cierta acción, debe escribirse pensando que posteriormente **puede ser necesario modificarlo o mantenerlo** y que la persona encargada de ello puede ser la misma que ha creado inicialmente el programa u otra distinta. Tomando esto como base, es necesario que el código de un programa sea lo más claro posible y uno de los elementos que más contribuye a esta claridad es precisamente la modularidad.

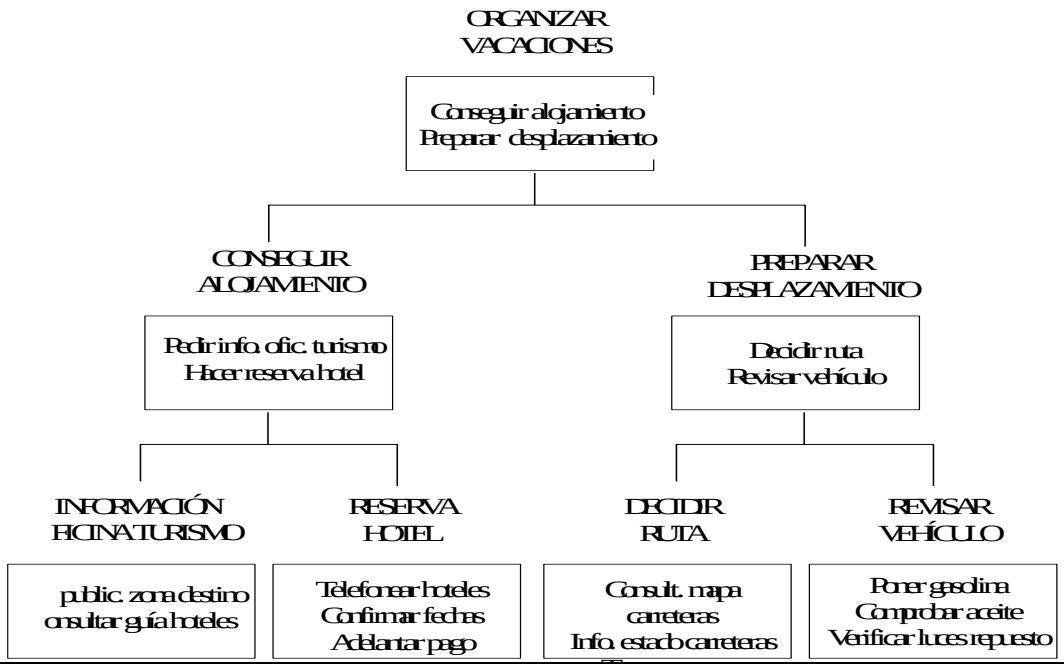
La modularidad consiste en la **división de un programa en distintos componentes o módulos**, cada uno de los cuales tendrá un nombre o identificativo propio y una ubicación determinada dentro del programa y que se podrán invocar y ejecutar.



La operación de estructurar un programa comienza desde su propio análisis, momento en el que se ha de determinar qué operaciones será necesario realizar en distintos puntos, qué bloques de código se repetirán a lo largo del programa, etc. A partir de estos datos procederemos a aislar esas porciones de código del código general del programa, creando los **subprogramas** que sean convenientes. Cada una de estas entidades actuará como si se tratase de un pequeño programa, al que se llama cuando es necesario, pasando y obteniendo los parámetros apropiados.

Mantener el código de un programa que está estructurado en múltiples subprogramas es mucho más simple, ya que el programador sabe que el código existente en uno de estos bloques es totalmente independiente del código del resto de programa y que, por lo tanto, lo puede analizar separadamente de forma más fácil.

Ejemplo. Realizar el análisis descendente de la tarea de ORGANIZAR VACACIONES



2. Subprogramas

Un **subprograma**, también denominado **método** o **función**, es un conjunto de operaciones o sentencias identificado mediante un nombre único. Toda referencia o llamada a este nombre implica la **ejecución de las operaciones contenidas** en dicha función o procedimiento.

Constituyen módulos o bloques que resuelven un problema parcial y que se pueden utilizar en cualquier parte del código. Se utilizan para ayudar a organizar y simplificar programas largos y complejos.

Se debe utilizar un subprograma cuando el conjunto de operaciones que define se va a utilizar en más de un lugar de nuestro código **evitando de esta forma la duplicidad del código**.

También se debe utilizar una función cuando hace **más fácil comprender la lógica del programa y su mantenimiento**.

Características de los subprogramas:

- La llamada a un subprograma en un programa da como resultado la ejecución del cuerpo de la función. Se ejecuta hasta que se alcanza la última sentencia (o se ejecuta la sentencia **return**), momento en el cual se transfiere de nuevo el control al programa principal o subprograma que lo hubiera llamado.

- La forma de comunicar la información a una función es mediante los argumentos o **parámetros**.

- El subprograma debe definirse antes de utilizarse.

Un subprograma **ha de definirse antes de su utilización,** estableciendo el identificador por el que se le conocerá, los parámetros que necesita y el bloque de código a ejecutar.

En dicho bloque el subprograma o método puede declarar sus propias variables, tipos, etc., de forma similar a como haríamos en el bloque principal, con la única diferencia de que los identificadores creados tendrían un ámbito local.

Como hemos dicho un subprograma o función consiste en una serie de sentencias en C# que realiza una acción determinada y devuelve un valor.

- **Sintaxis para crear una función**

  La forma general para crear una función es:

```
tipo nombreFuncion(parámetros)
{
     sentencias;

     return valor;
```

```
}
```

Ejemplo de función

El siguiente código crea una función denominada **cuadrado** que devuelve el cuadrado de un número entero (*int*):

```
// Función que devuelve el cuadrado de un valor.
int cuadrado(int valor)
{
    // res será una variable local.
    int res;

    res = valor * valor;

    // La sentencia return hace que la función devuelva
    // el valor contenido en res.
    return res;
}
```

Vamos a analizar línea a línea la función cuadrado.

```
int cuadrado(int valor)
```

es la cabecera de la función. Indica que la función se llama **cuadrado**, que devuelve un valor de tipo entero (**int**) y que se le pasa un parámetro de tipo **int** (más adelante veremos con más tranquilidad el concepto de parámetro).

```
    int res;
```

Estamos definiendo una variable local. Su ámbito de uso es nuestra función.

```
    res = valor * valor;
```

Sentencia de asignación.

```
    return res;
```

Hace que nuestra función devuelva el valor contenido en la variable res.

Si luego quisiéramos llamar a la función cuadrado lo podríamos hacer de la siguiente forma:

```
private void btnCalcularCuadrado_Click(object sender, EventArgs e)
{
    int num, resultado;

    num = int.Parse(txtNum.Text);

    resultado = cuadrado(num);

    MessageBox.Show("El cuadrado de " + num + " es " + resultado);
}
```

Ejemplo de función (suma de dos números):

```
// Función que devuelve la suma de dos números.
int suma(int num1, int num2)
{
    // res será una variable local.
    int res;

    res = num1 + num2;

    // Devolvemos el valor contenido en res
    return res;
}
```

```
private void btnSumar_Click(object sender, EventArgs e)
{
    int n1, n2, result;

    n1 = int.Parse(txtNum1.Text);
    n2 = int.Parse(txtNum2.Text);

    // Llamamos a la función pasando los parámetros
    // y recogiendo el valor que devuelve.
    result = suma(n1, n2);

    MessageBox.Show("El resultado de la suma es " + result);
}
```

Podría ocurrir que una función no necesite devolver ningún valor (en algunos lenguajes esto se denomina procedimiento). En ese caso el tipo que devuelve la función es void, lo cual indica que no devuelve nada y no es necesario utilizar return.

Ejemplo de función void:

```
// Función void. No devuelve ningún valor
void versionPrograma(int num)
{
    string texto;

    texto = "Esta es la versión "+ num + " de nuestro programa.";
    MessageBox.Show(texto);
}
```

```
private void btnVersion_Click(object sender, EventArgs e)
{
    versionPrograma(5);
}
```

3. Paso de parámetros

Como hemos dicho los parámetros son la forma de comunicación entre el programa principal y los subprogramas. Es la forma de **pasar datos** a las funciones o subprogramas. Estos parámetros pueden ser de entrada, de salida o de entrada salida.

Un aspecto importante en la programación modular es como se pasan o reciben las funciones y procedimientos. A este proceso se le conoce como paso de parámetros. Hay dos formas de paso de parámetros:

a. **Parámetros por valor:** Pueden ser variables o constantes que no cambian su valor en el cuerpo del subprograma. Son sólo **datos de entrada**.

b. **Parámetros por referencia:** Son variables que en el cuerpo del subprograma pueden cambiar su valor. Son **datos de entrada y salida**.

Es decir, cuando se pasa un parámetro por referencia se le indica al subprograma la posición en memoria donde está ubicada la variable que le pasas como parámetro; de esta forma los cambios de valor que se efectúen en dicha variable son definitivos incluso cuando dicha subrutina termine.

Cuando definimos una función en Visual C# .NET, describimos los datos y los tipos de datos para los que la función está diseñada para aceptar desde un subprograma que la llame. Los elementos definidos en la función se denominan ***parámetros***.

Cuando llamamos a la función, sustituimos un valor actual de cada parámetro. Los valores que pasamos en la posición de los parámetros se denominan ***argumentos***.

A pesar de esta sutil diferencia, los términos *argumento* y *parámetro* a menudo se utilizan indistintamente.

- La lista de argumentos debe tener tantos argumentos como parámetros requiera la función.
- El **orden de los argumentos influye**, es decir, al primer argumento se le asigna el primer parámetro, al segundo argumento se le asigna el segundo parámetro y así sucesivamente.
- El tipo de argumento debe coincidir con el tipo de parámetro al que esté asociado.

Parámetros por valor.

El paso de parámetros por valor **es la forma por defecto** en la que se pasan los parámetros a las funciones de C#.

En los ejemplos que hemos visto en los apartados anteriores se pasaban los parámetros por valor, es decir, si cambiara el valor del parámetro dentro de la función, ese cambio no se transmite al argumento de la llamada. Son parámetros de entrada, es decir, nos permiten pasar datos al subprograma.

Ejemplo de parámetro por valor:

```
void dobleValor(int num)
{
    num = num * 2;
}
```

```
private void btnValor_Click(object sender, EventArgs e)
{
    int num;

    num = 10;
    dobleValor(num);

    // El valor de la variable num sigue siendo 10
    MessageBox.Show("El valor de la variable num es " + num);
}
```

Como vemos en este ejemplo **num** en el botón no ha cambiado su valor a pesar de que en el subprograma si se cambia.

Parámetros por Referencia

Para definir un parámetro por referencia se pueden utilizar la palabra reservada **ref** o **out**.

Como hemos dicho antes, cuando definimos un parámetro por referencia, si éste cambia de valor el cambio se transmite al argumento.

La diferencia entre **ref** y **out** es que cuando utilizamos **ref** el argumento debe haber sido inicializado antes de llamar a la función (o tendremos un error de compilación) mientras que con **out** no es necesario.

Los parámetros **ref se podrían ver como parámetros de entrada/salida** y los **out como parámetros de salida**.

Ejemplo de parámetro por referencia:

```
void dobleReferencia(ref int num)
{
    num = num * 2;
}
```

```
private void btnReferencia_Click(object sender, EventArgs e)
{
    int num;

    num = 10;
    dobleReferencia(ref num);

    // El valor de la variable num pasa a ser 20
    MessageBox.Show("El valor de la variable num es " + num);
}
```

Notar que al llamar a una función con un parámetro por referencia hay que poner **ref** delante del argumento en la llamada.

Ejemplo:

Vamos a realizar un ejemplo en el que calculemos la nota media a partir de las notas de las 3 evaluaciones. Lo vamos a hacer con una función que devuelva la media y con una función void que devuelva la media en un parámetro de salida.

Ejemplo con función:

```
double mediaFuncion(double nota1, double nota2, double nota3)
{
    double res;

    res = (nota1 + nota2 + nota3) / 3;

    return res;
}
```

```
private void btnMediaFuncion_Click(object sender, EventArgs e)
{
    double n1, n2, n3, media;

    n1 = double.Parse(txtNota1.Text);
    n2 = double.Parse(txtNota2.Text);
    n3 = double.Parse(txtNota3.Text);

    media = mediaFuncion(n1, n2, n3);

    MessageBox.Show("La media es " + media);
}
```

Ejemplo con parámetro por referencia:

```
void mediaReferencia(double nota1, double nota2, double nota3,
                     out double res)
{
    res = (nota1 + nota2 + nota3) / 3;
}
```

```
private void btnReferencia_Click(object sender, EventArgs e)
{
    double n1, n2, n3, media;

    n1 = double.Parse(txtNota1.Text);
    n2 = double.Parse(txtNota2.Text);
    n3 = double.Parse(txtNota3.Text);

    mediaReferencia(n1, n2, n3, out media);

    MessageBox.Show("La media es " + media);
}
```

- **¿Cuándo devolver valores por return o por parámetros por referencia?**

En muchas ocasiones puede resultar confuso, a la hora de utilizar un subprograma, decidir si devolver valores mediante return o mediante parámetros por referencia.

Una posible regla a aplicar puede ser la siguiente:

- Si el subprograma va a devolver **un solo valor**, utilizar **return**.
- Si el subprograma **no va a devolver valor** o va a devolver **más de un valor,** entonces utilizar función void (no devuelve nada) y devolver los valores mediante varios parámetros por referencia.

A continuación, os planteo varios posibles ejemplos.

1. Imaginemos que queremos realizar un subprograma que devuelva la potencia de un número elevado a otro.

   En este caso lo mejor es utilizar una función con dos parámetros por valor (la base y el exponente) y que devuelva el resultado de la potencia.

2. Si queremos hacer un subprograma que simplemente imprima algo por pantalla se puede utilizar una función void.

3. Imaginemos que queremos hacer un subprograma que devuelva el valor de la división entera y del resto entre dos números.

   En este caso lo mejor sería hacer una función void con 4 parámetros, dos de entrada: el primer número y el segundo número, y otros dos parámetros de salida (out): el resultado de la división y el resto.

4. Si quisiéramos hacer un subprograma que intercambie el valor de dos números, lo mejor sería hacer una función void con dos parámetros de entrada/salida (ref).