Basado en Core C++ a Software Engineering Approach de Victor Shtern

Cohesion

- cuando una función tiene una **cohesión alta**, realiza **una tarea** concreta con cierto objeto o estructura de datos;

- cuando la **cohesión** es **baja** (ó débil), la función realiza **varias tareas** con un objeto **o** incluso **sobre distintos tipos** de objetos.

- Cuando la cohesión de una función es baja, ésta realiza varias tareas que no tienen mucho que ver unas con otras sobre varios objetos que no guardan tampoco ninguna relación.

- Es decir: **estos objetos** pertenenencen en realidad a alguna otra zona de código, *fueron desgajados de la parte del programa a la que deberían pertenecer* y reunidos, junto con otros, en esa función.

- Cuando una función tiene una cohesión alta es fácil darle nombre mediante un verbo, o un verbo junto con un nombre.

  El verbo indicaría la acción realizada por la función, y el nombre el objeto (o el sujeto) de la acción. Por ejemplo, ***insertaItem(), buscaCuenta()...***

- Cuando la cohesión es baja al intentar nombrar la función es necesario usar varios verbos o nombres, por ejemplo, **buscaOInsertaTtem().**

He aquí un ejemplo, no bueno precisamente (todo buen ejemplo de **cohesión baja** es tambi₫n un ejemplo de **módulo con un diseño pobre**)

```
void initializeGlobalObjects ()
{ numaccts = 0;                     // un objeto
  fstream inf("trans.dat",ios::in);
                                  // fichero de
transacciones
  numtrans = 0;                      // otro objeto
  if (inf==NULL) exit(1); }       // nuevamente el
fichero
```

En el ejemplo:

- numaccts está relacionado con el procesamiento de cuentas, debería inicializarse al comenzar a procesar las cuentas.
- numtrans debería inicializarse al comenzar a procesar transacciones, no está relacionado con el "procesamiento de cuentas".
- **hemos sacado una parte del procesamiento de cuentas y otra del de transacciones de su sitio natural para reunirlas en un módulo con una cohesión baja.**

- La cohesión no es tanto un criterio para orientar las decisiones de diseño
- como un criterio para evaluar la calidad del mismo.

- Una vez detectado un módulo con una cohesión baja, el dividirlo en varios módulos puede no ser una buena solución.

- En caso de duda es conveniente recurrir a algún otro criterio, además del de cohesión.

- Este criterio es muy útil, en cambio, para escoger entre alternativas de diseño, entre distintas formas de repartir el trabajo entre distintos módulos.

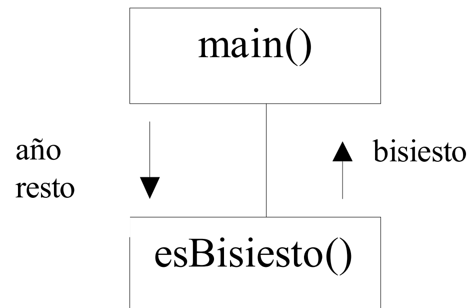
Acoplamiento implícito:

Cuando dos funciones se comunican a través de variables globales, que no aparecen en la interface, el acoplamiento es implícito. Consideremos un ejemplo

```
int main()
{int año,resto;  bool bisiesto;
cout << "Introduzca el año:  ";
cin >>año;
resto= año% 4;
if (resto!= 0)                    // no es divisible por 4
 bisiesto = false;             // luego no es bisiesto
else
   { if (año%100 == 0 && año%400 != 0)
bisiesto = false; //divisible por 100 mas no por 400
     else
  bisiesto = true; }   // en caso contrario: bisiesto
if (bisiesto)
  cout << año<< " es bisiesto\n";
else
  cout << año<< " no es bisiesto \n";
return 0;
}
```

Este programa es muy pequeño y , en realidad, no es necesario modularizarlo pero intentemos imaginar que se trata de un programa tan grande como para beneficiarse de la modularización, y estudiar la mejor forma de hacerlo. Lo descompondremos en dos funciones:

- main() se encargará de la interacción con el usuario
- esBisiesto() calculará el valor de bisiesto en función de año y resto.



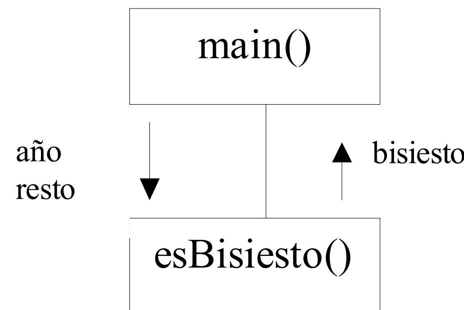
```
#include <iostream>
using namespace std;

int año,resto;  // variables globales
bool bisiesto;

void esBisiesto(); // entradas: año, resto;
         // salida: bisiesto
int main()
{ cout << "Introduzca el año:  ";
  cin >>año;
 resto= año% 4;     // acceso a variables globales
 esBisiesto();      // determina si es bisiesto el año
 if (bisiesto)
  cout << año<< " es bisiesto\n";
else
  cout << año<< " no es bisiesto \n";
  return 0;
  }
```

```
void esBisiesto() // entradas: año, resto;
         // salida: bisiesto
   { if (resto!= 0)      // no es divisible por 4
 bisiesto= false;       // luego no es bisiesto
     else if (año%100==0 && año%400!=0)
 bisiesto = false; //divisible por 100 mas no por 400
else
 bisiesto = true; } // en caso contrario: bisiesto
```

La carta de estructura muestra el flujo de datos entre funciones haciendo uso de las variables globales **año**, **resto** y **bisiesto**.



Al usar variables globales:

- sus valores pueden cambiarse accidental, o maliciosamente, y esto sería difícil de descubrir.

- no es fácil decir qué funciones acceden a estas variables.

- **lo peor**: el acoplamiento implícito.

debemos estudiar grandes zonas de código para entender el flujo de datos dentro de un programa

, mientras que si el acoplamiento fuera explícito bastaría con examinar las cabeceras de las funciones.

Acoplamiento explícito

Cuando el intercambio de información entre dos módulos se produce a través de la interface, es decir a través de la lista de parámetros, y no a través de variables globales, el acoplamiento entre los módulos es explícito.

Basado en ... Victor Shtern

```
#include <iostream>
using namespace std;

void esBisiesto(int año , int resto, bool
&bisiesto); // parametros
// entrada:año,resto; salida: bisiesto

int main()
{ int año,resto;
  bool bisiesto;
  cout << " Introduzca el año: ";
  cin >>año;
  resto= año% 4;
  esBisiesto(año,resto ,bisiesto);
  if (bisiesto)
    cout << año<< " es bisiesto \n";
  else
    cout << año<< " no es bisiesto \n";
  return 0;}
```

```
Void esBisiesto(int año, int resto, bool &bisiesto) {
// parametros
// entrada:año,resto; salida: bisiesto

if (resto!= 0)
 bisiesto = false;
     else if (año%100==0 && año%400!=0)
 bisiesto= false;
     else
 bisiesto= true; }
```

hemos sustituido la comunicación a través de variables globales por comunicación a través de parámetros

Consejo:

• Evitar los acoplamientos implícitos.

• Hacer uso de acoplamientos explícitos, a través de parámetros,

• así los programadores, puedan comprender la interface de la función con sólo mirar su cabecera, no todo el código de ésta y de la que la llama.

- Para hacer más inteligibles los programas no basta con usar acoplamientos explícitos.

- Es necesario también escoger correctamente los modos de los parámetros.

- En el ejemplo anterior:

  Los parámetros `año` y `resto` se pasan por valor. Por consiguiente no pueden ser parámetros de salida, sino de entrada.

  En cambio, el parámetro `bisiesto` se pasa por referencia. Esto indica que se trata de un parámetro de salida, o de entrada y salida

Consideremos por ejemplo la siguiente implementación de la función esBisiesto():

```
void esBisiesto(int &año, int &resto, bool &bisiesto)

  { if (resto!= 0)
 bisiesto= false;
    else if (año%100==0 && año%400!=0)
 bisiesto= false;
    else
 bisiesto= true; }
```

tiene una pega esta función:

- El paso por referencia de los tres parámetros, puede hacer pensar que los tres son modificados por esBisiesto(), para ser usados por la función que la llame.

- Para salir del error se deberá revisar todo el código de esBisiesto().

- Sería mucho mejor tener que mirar sólo la cabecera.

- El programador sabía que el único parámetro de salida era bisiesto, y debió plasmar este conocimiento en la interface haciendo que sólo este parámetro se pasara por referencia.

- Por el contrario, un paso de parámetro por referencia induce al mantenedor a pensar que el parámetro será modificado.

Un buen estilo de codificación podría ser el siguiente:

- Usar parámetros en vez de variables globales.

- Pasar los parámetros de entrada simples por valor y los de salida por referencia.

- Pasar los parámetros complejos y los class por referencia, usando el modificador const para los de entrada.

- Pasar los arrays que sean parámetros de entrada con el modificador const

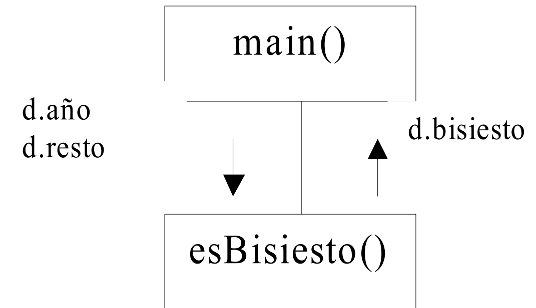
Cómo reducir la intensidad del acoplamiento

- Para disminuir el acoplamiento no basta con disminuir el número de parámetros;

- para conseguirlo se deberá **cambiar el diseño haciendo un reparto distinto de responsabilidades entre módulos**.

- Alguien podría pensar en agrupar los diversos parámetros en una estructura compleja. Vamos a ver cómo esto no necesariamente disminuye el acoplamiento. Veamos otra versión de `esBisiesto()`:

Basado en ... Victor Shtern




```
#include <iostream>
using namespace std;
struct DatoAño
{  int año,resto;
   bool bisiesto; } ;

void esBisiesto(DatoAño &datos)   // un solo parametro

int main()
{  DatoAño d;                     // variable local
   cout << "Introduzca el año:  ";
   cin >> d.año;              // damos valor a las entradas
  d.resto = d.año % 4;
   esBisiesto(d);          // las salidas son modificadas
   if (d. bisiesto)                     // usamos las salidas
          cout << d.año << " es bisiesto\n";
   else
          cout << d.año << " no es bisiesto\n";
   return 0;}
```

```
void esBisiesto(DatoAño &datos) // un solo parametro
  { if (datos.resto!= 0)
datos.bisiesto = false;
    else if (datos.año %100==0 && datos.año %400!=0)
 datos.bisiesto  = false;
    else
 datos.bisiesto = true; }
```

Efectivamente el número de parámetros es ahora menor, pero el flujo de datos es el mismo.

Ahora además, el programa es más difícil de entender y de reutilizar: esto no se puede hacer sin entender el tipo DatoAño.

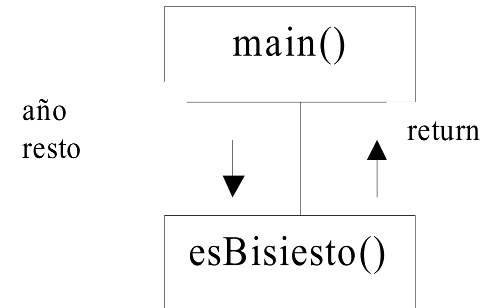
También se nos puede ocurrir disminuir el acoplamiento ocultando parámetros de salida.

```
#include <iostream>
 using namespace std;
bool esBisiesto(int año, int resto);

 int main()
{ int año,resto;  // sin variable bisiesto
  cout << " Introduzca el año:  ";
  cin >>año; resto= año% 4;
  if (esBisiesto (año,resto)==true cout << año<< " es 
bisiesto \n";
  else
    cout << año<< " no es bisiesto \n";
  return 0; }
```

Basado en ... Victor Shtern

```
bool esBisiesto(int año, int resto)  //menos
parametros
  { if (resto!= 0)
      return false;
    else if (año%100==0 &&
             año%400!=0)
      return false;
    else
      return true; }
```



- De todas formas el flujo de datos entre main() y esBisiesto() no ha disminuido.

- Nuevamente el acoplamiento sigue siendo el mismo porque el reparto de responsabilidades entre `main()` y `esBisiesto()` sigue siendo el mismo que en la primera versión del programa.

- Si queremos disminuir el acoplamiento debemos hacer un reparto de responsabilidades distinto, debemos cambiar el diseño.

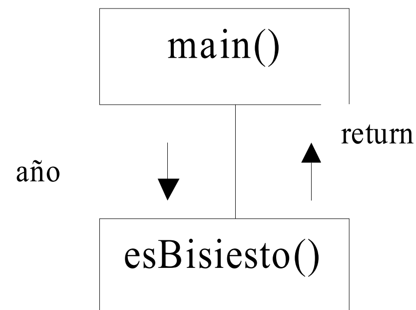
- Una de las formas de conseguirlo es identificar los componentes del flujo de datos que están ahí como consecuencia de haber separado lo que debería estar unido.

- Separar, en distintas funciones, cálculos que deberían figurar juntos obliga a éstas a comunicarse entre sí.

- Otra consecuencia de separar lo que debería estar unido es la necesidad de examinar la función cliente y la función servidora para entender el papel que desempeña un parámetro.

- Por ejemplo, el significado del parámetro resto no puede deducirse observando sólo la función esBisiesto(); sólo mirando a main() puede saberse que resto es el resto de dividir el año por 4....

Examinemos ahora el código y la carta de estructura de una nueva versión del programa en la que `resto` se calcula en `esBisiesto`(), sin necesidad de implicar a `main`() en este cálculo antes de llamar a `esBisiesto`().



```
#include <iostream>
using namespace std;

bool esBisiesto(int año); // menos parametros todavia

int main()
{ int año; // ya no esta resto aqui
  cout << "Introduzca el año: ";
  cin >>año;
  if (esBisiesto (año))
    cout << año<< " es bisiesto \n";
  else
    cout << año<< " no es bisiesto \n";
  return 0;
 }
```

```
bool esBisiesto(int año) // menos parametros todavia
   { int resto=año %4; // no separar lo que debe estar
unido
if (resto!= 0)
       return false;
     else if (año%100==0 && año%400!=0)
       return false;
     else
       return true; }
```

Mover el cálculo de resto de una función a otra es realmente un cambio en el diseño: es cambiar el reparto de responsabilidades entre funciones.

- Mover responsabilidades a los servidores no es siempre beneficioso, pero muchas veces si que lo es.

- Esta técnica es muy potente. Disminuir la comunicación entre funciones facilita el mantenimiento y la reutilización del código y disminuye también las necesidades de comunicación entre programadores cuando las funciones las escriben personas distintas (o la misma en diferentes épocas).

¿Hasta qué extremo habría que llegar, siguiendo esta línea?.

•¿Tendría sentido mover la petición de información y la variable año a esBisiesto()?.

•Esto disminuiría aún más el flujo de datos entre las funciones.

•Sin embargo, haría necesaria una mayor comunicación entre programadores en relación con la interface de usuario.

•Además estaríamos empeorando la cohesión de esBisiesto(): al combinar cálculo y entrada/salida.

- En la última versión del programa, main() era responsable de la interface de usuario, y esBisiesto() de los cálculos.

- *Diseminar la interface de usuario a través de varias funciones puede llegar a ser tan dañino como diseminar los cálculos.*

- Es mejor garantizar que las responsabilidades de las distintas funciones están bien delimitadas