Departamento de Informática | Universidad de Valladolid

Programación II (I.T.I. Gestión)

Programación Modular

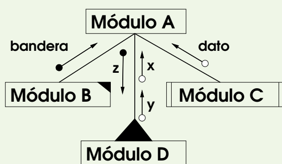
Félix Prieto
Esperanza Manso

Curso 2009/10

Definiciones

- Modularidad es el atributo individual del software que permite a un programa ser intelectualmente manejable. *Myers*
- Llamaremos módulo a un conjunto de sentencias de un programa que:
  - Están físicamente unidas
  - Están físicamente delimitadas con respecto al resto del programa
  - Tienen estructura o función bien delimitada
  - Son referenciables mediante un nombre
  - Respetan el principio de ocultación de información
- Principio de ocultación de información

Definiciones (y II)



Ventajas del uso de módulos

- Facilidad de construcción
- Facilidad de prueba y corrección
- Facilidad de comprensión
- Facilidad de modificación

Diagrama de estructura modular

- Sin orden de ejecución
- Sin código
- Con cierta especificación

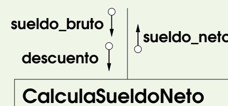
Bibliotecas

- Permiten:
  - Compilar por separado
  - Evitar repeticiones de código
  - Aumentar la fiabilidad
- Pero:
  - Las modificaciones pueden ser complejas
  - La fiabilidad puede bajar
  - Es difícil medir el ahorro conseguido
- Debemos establecer criterios de calidad:
  - Tamaño razonable
  - Máxima cohesión
  - Mínimo acoplamiento
  - Guías adicionales de diseño

Cohesión

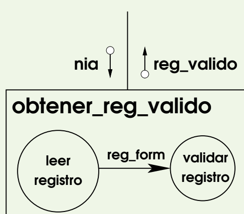
- Llamamos *cohesión* de un módulo al tipo de relación que se establece entre las actividades del mismo
- El nivel de cohesión del sistema es el del peor de sus módulos
- Debemos aumentarlo para mejorar...
  - comprensión
  - mantenimiento
  - reutilización
- Pero no es razonable aspirar a que un sistema tenga un grado de cohesión máxima.

Cohesión funcional



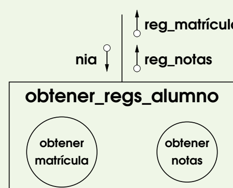
- Una única actividad simple y bien especificada
  - Simple no quiere decir fácil
  - La simplicidad tiene que ver con el punto de vista del programador
  - La función matemática es el paradigma de actividad bien especificada
  - Ojo con las especificaciones demasiado generales: Actividades iniciales del programa

Cohesión secuencial



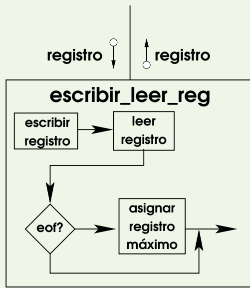
- Varias actividades, la salida de una es entrada de la siguiente
  - Menor probabilidad de reutilizar el módulo
  - Peligro de compartir código entre las tareas que lo forman
- Factorizar cuando las tareas sean «suficientemente grandes»

Cohesión comunicacional



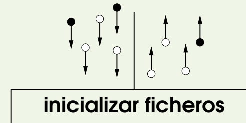
- Varias actividades que comparten la misma entrada y/o la misma salida
  - La probabilidad de reutilizar el módulo es ya mínima
  - Las actividades tienen muy poca relación entre si
  - Compartir código entre las tareas dificultará su posterior mantenimiento
- Empieza a ser muy recomendable factorizar el módulo

Cohesión procedural



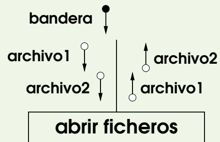
- Actividades diversas que ocurren en determinada secuencia
  - Las actividades tienen menos relación entre ellas
  - Pero es posible compartir código
  - El mantenimiento será más difícil
- Nivel de cohesión malo. El módulo debe ser factorizado

Cohesión temporal



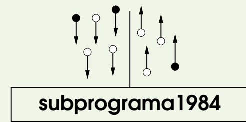
- Actividades diversas que ocurren al tiempo
  - Surge de la «necesidad» de modularizar
  - Dificulta el mantenimiento
  - No facilita la comprensión del sistema
- Nivel de cohesión muy malo. El módulo debe ser factorizado

Cohesión logica



- Actividades de la misma categoría
  - Puede parecer buena idea, pero. . .
  - Dificulta el mantenimiento y la comprensión
  - Aumenta el acoplamiento
- Nivel de cohesión pésimo. El módulo debe ser factorizado

Cohesión coincidente

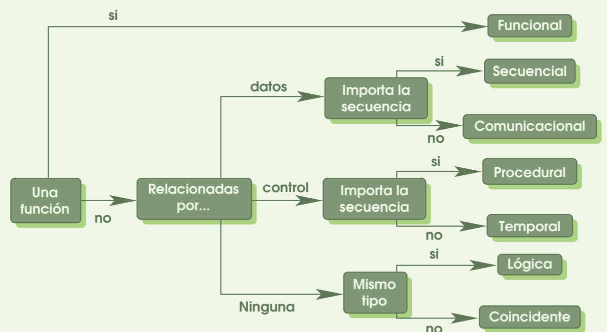


- No se nos ocurre porqué estas actividades están el el mismo módulo

¿Cómo clasificar?

- A partir de la información disponible sobre el módulo
  - Nombre, especificación,. . .
  - ¿Qué hará «razonablemente» el módulo?
  - ¿Tiene código?
- Cuando todas las actividades se relacionan por *los mismos* criterios *elegimos el mejor*
- Cuando las actividades se relacionan por *diferentes* criterios *elegimos el peor*

¿Cómo clasificar? (y II)



Acoplamiento

- Llamamos *acoplamiento* entre dos módulos al grado de dependencia que existe entre ellos
- El nivel de acoplamiento del sistema es el peor entre los existentes entre sus módulos
- Debemos reducirlo para facilitar...
  - comprensión
  - mantenimiento
  - reutilización
- Pero no podemos aspirar a que un sistema tenga grado de acoplamiento cero.

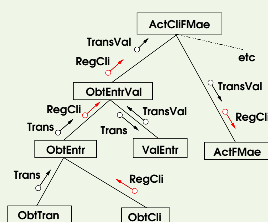
Bases para reducir el acoplamiento

- Podemos reducir el acoplamiento si:
  - Eliminamos relaciones innecesarias
  - Reducimos el número de relaciones necesarias
  - Reducimos la «firmeza» de las relaciones necesarias
- Preferimos conexiones. . .
  - estrechas (con pocos argumentos)
  - directas (comprensibles sin información adicional)
  - locales (información presente en la llamada)
  - obvias («forma» que esperan otros programadores)
  - flexibles (aplicando «indirección»)
- El acoplamiento se mide desde una *perspectiva humana*.

Tipos de acoplamiento

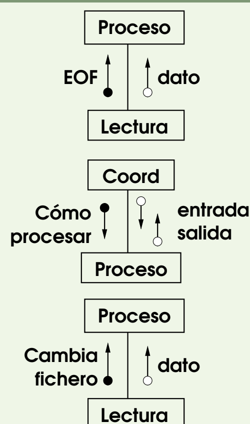
- *Minimal o normal* Llamada «normal» a una función o procedimiento.
  - *de datos* Todos los argumentos son datos
    - *simples* Todos los datos son simples
    - *por estructura* Alguno de los datos es estructurado
  - *de control* Alguna bandera de control entre los argumentos
  - *Híbrido* Algún dato actúa como bandera de control
- *Global* Utiliza zonas comunes de memoria
- *Patológico* Acceso directo entre bloques de código

Acoplamiento minimal de datos



- Evitar datos innecesarios
- Evitar estructuras innecesarias
- Evitar información innecesaria
- Evitar datos errantes

Acoplamiento minimal de control



- Mejor descripciones que órdenes
- Evitar órdenes ascencentes
- Ciertas banderas son imprescindibles
- No son banderas si procesamos datos binarios

Acoplamiento híbrido

- *Una «buena» idea:* Si el NIA es par para hombres e impar para mujeres me ahorro un campo del registro
  - ¿Para qué sirve ahorrar un campo en un registro?
- *Una idea «mejor»:* Los NIA del 1 al 10000 para alumnos de Ciencias, del 10001 al 20000 para Medicina,. . .
  - ¿Qué pasa con el alumno 10001 matriculado en Ciencias?
  - ¿Qué ocurrió el 1 de enero de 2000?
  - ¿Qué ocurrirá el 19 de enero de 2038?
- El acoplamiento híbrido sólo es justificable cuando supone un ahorro *esencial* de recursos escasos
- Aún en ese caso resulta peligroso y debe ser convenientemente documentado

Acoplamiento Global y Patológico

- Dos módulos se acoplan *globalmente* si utilizan zonas de datos comunes
  - El nivel de acoplamiento crece porque la dependencia entre módulos no es explícita
  - Los sistemas se vuelven más propensos a errores
  - Los sistemas se vuelven más difíciles de comprender, mantener y reutilizar
- Dos módulos se acoplan de modo *patológico* si uno de ellos accede de algún modo a los datos internos o el código del otro
  - Conduce a sistemas inmanejables

Criterios adicionales

- Abanico de entrada
- Abanico de salida
- Ámbito de aplicación
- División de las decisiones
- Tratamiento de errores
- Balanceado del sistema
- Tamaño de los módulos
- Duplicación de código
- . . .

Un problema frecuente

«Dado un elemento 'x' de algún tipo, y un conjunto 't' de elementos del mismo tipo, se trata de construir un programa `buscar` *que determine si 'x' está o no en 't'»*

¿Cuántas veces hemos resuelto este problema?

¿Porqué no reutilizamos software?

- No conocemos los elementos reutilizables
- Es demasiado caro
- Es inaccesible
- Reservas psicológicas
- Herramientas inadecuadas
- Noción inadecuada de módulo

Reutilización del Software

En 1968 McIlroy establecía los siguientes requisitos para la reutilización del software:

- Producción en masa de piezas software.
- Catálogos de piezas.
- Desarrollo a partir de piezas.

Formas de reutilización

Necesitamos determinar qué es lo que vamos a reutilizar

- Reutilización del código fuente.
- Reutilización del personal informático.
- Reutilización de diseños genéricos.

Esquema genérico del problema general

```
buscar(X:ELEMENTO, t: ESTRUCTURA_DE_ELEMENTO) : booleano
   pos: POSICION
inicio
   pos := POSICION_INICIAL (x,t);
   mientras no AGOTADA (pos, t)
       y entonces no ENCONTRADO(pos, x, t) hacer
          pos := SIGUIENTE(pos,x,t)
   fin_mientras
   devolver no AGOTADA(pos, t)
fin
```


Requisitos de reutilización

- Variación de tipos
  - Módulos con parámetros formales (genéricos)
- Variación de estructuras de datos y algoritmos
  - Polimorfismo
- Posibilidad de agrupar funciones
  - Clases
- Independencia de la representación
  - Herencia, polimorfismo y ligadura dinámica
- Pautas comunes a subgrupos
  - Herencia

Soluciones propuestas

- Creación de bibliotecas de funciones.
- Uso de módulos empaquetados.
- Sobrecarga de operadores.
- Generalización (módulos abstractos).

Creación de bibliotecas de funciones

- Funciona con problemas individuales:
  - de especificación sencilla
  - claramente diferenciados
  - sin estructuras de datos complejas
- Problemas:
  - Función enorme que hay que recompilar.
  - Colección de funciones muy parecidas.
  - Estructuras de datos dispersas.

Módulos empaquetados

- El módulo incluiría:
  - Definición de un tipo.
  - Operaciones relacionadas.
- Resuelve el agrupamiento de funciones.
- No resuelve las dificultades con problemas generales o muchos problemas relacionados.

Sobrecarga

- El mismo nombre de función para varios tipos de datos.
- Es una ventaja para los programadores de módulos cliente.
- Es un inconveniente para los programadores de los módulos servidores.
- Avance en el aspecto de variación de tipos.
- No resuelve la variación de algoritmos para el mismo tipo.
- Los algoritmos se asocian con la signatura sintáctica, no con el significado semántico.

Generalización

- Permite definir módulos genéricos.
- Substituyendo parámetros formales se obtienen instancias del módulo genérico.
- Proporciona mayor comodidad a los programadores de módulos servidores.
- Herramienta para trabajar con distintas estructuras de datos.

Generalización (y II)

- Pautas comunes con subgrupos.
  - Dos niveles de módulos.
  - Sólo un nivel utilizable y no modificable.
- Independencia de la representación.
  - Un módulo genérico no es utilizable en sí mismo sino a través de sus instancias.
  - Cada llamada a una función se refiere a una única versión de la operación, a pesar de la sobrecarga.

El siguiente paso

Plantear una función `buscar(x,t)` que signifique:

> *«Buscar el elemento x en t, usando el algoritmo apropiado, para cualquier estructura de datos t, sea la que sea, en tiempo de ejecución»*

Orientación a procesos

En resumen, la orientación a procesos presenta dificultades de cara a la reutilización:

- No tiene en cuenta la orientación evolutiva del software.
- La noción misma de un sistema como un proceso único es cuestionable.
- Se olvida con frecuencia la estructura de los datos.
- El diseño «top-down» no promueve la reutilización.

Orientación a objetos

La propuesta de la Orientación a Objetos es estructurar los sistemas informáticos en torno a los datos, representados mediante objetos. El problema es:

- Cómo encontrar los objetos.
- Cómo describir los objetos.
- Cómo describir las relaciones y aspectos comunes a objetos.
- Cómo utilizar los objetos para estructurar programas.

Tipos Abstractos de Datos

```
tipos
    PILA[X]
funciones
    vacia: PILA[X] -> BOOLEANO
    nueva: -> PILA[X]
    apilar: X x PILA[X] -> PILA[X]
    extraer: PILA[X] -> PILA[X]
    cima: PILA[X] -> X
precondiciones
    pre extraer(p: PILA[X]) = (no vacía(p))
    pre cima(p:PILA[X]) = (no vacía(p))
axiomas
    ∀ x: X, p: PILA[X]
    vacía(nueva())
    no vacía(apilar(x,p))
    cima(apilar(x,p))=x
    extraer(apilar(x,p))=p
```


Clases y TAD

- El diseño orientado a objetos construye sistemas informáticos como colecciones estructuradas de implementaciones de TAD.
- Las clases, los módulos en el paradigma OO, representan una implementación de un TAD, no el TAD en sí mismo.

Clases y TAD (y II)

```
Clase PILA[X]
    num_elementos : ENTERO;
    función vacía: BOOLEANO
      inicio
        ...
      fin;
    ...
    función cima : X
        (precondición: pila no vacía)
      inicio
        ...
      fin;
fin clase
```

Clases y herencia

- Las clases se diseñan como unidades interesantes y útiles en sí mismas.
- Relaciones importantes entre clases:
  - Cliente de (proveedora de)
  - Descendiente o heredera de (Ascendente o ancestro de)

```
Clase E_SECUENCIAL[T]
Clase TABLA[T] desciende de E_SECUENCIAL[T]
Clase LISTA[T] desciende de E_SECUENCIAL[T]
```


Polimorfismo

- Posibilidad de referirse en tiempo de ejecución a instancias de varias clases.
- Limitado por la herencia. Sólo a clases descendientes.
- Sistema de tipos más flexible.

Ligadura dinámica

Permite seleccionar la implementación concreta de una función según sea el tipo de la entidad en el momento de la ejecución.

```
Clase POLIGONO ...(incluye dibuja)
Clase RECTANGULO desciende de POLIGONO ...
    (incluye dibuja)
p: POLIGONO
r: RECTANGULO
...
p:=r;
p.dibuja
```


Requisitos de la OO según Meyer

- Estructura modular basada en datos
- Abstracción de datos
- Gestión de memoria automática
- Clases
- Herencia
- Polimorfismo
- Herencia múltiple