Algoritmos y estructuras de datos

www.u-tad.com

Grado en Ingeniería en Desarrollo de Contenidos Digitales

1. **Introducción a la construcción de software** ←
2. Algoritmos no recursivos
3. Algoritmos recursivos
4. Listas, pilas y colas
5. Ordenación avanzada
6. Tablas Hash
7. Árboles
8. Grafos

Introducción

- En esta asignatura consideraremos los siguientes elementos que también deberán cumplir nuestros programas, para considerar que están “bien” programados:
  - Bien comentado
  - Nomenclatura correcta
  - Máxima cohesión
  - Mínimo acoplamiento
  - Que siga el patrón MVC
  - Bien dividido en ficheros, clases y métodos/funciones
  - Buena presentación
  - Precondiciones
  - Indicada la complejidad temporal y espacial

Razones I

- La Ingeniería del Software es la disciplina que nos permite no sólo programar, sino programar “bien”
- Programar “bien” significa principalmente que se minimicen los gastos de mantenimiento de un software
- ¿Qué es el mantenimiento?
  - Adaptar un software ya hecho a ligeros cambios en los requisitos (lo que tiene que hacer el programa)
  - Usar el software como una pieza para crear un software distinto (reutilización)
    - Sin la reutilización, sería extremadamente caro crear software altamente complejo como un videojuego
  - Corregir el software, arreglar fallos
    - Principio importante: cualquier software real siempre tiene fallos
    - Principio importante: la probabilidad de fallos aumenta con el tamaño y complejidad del software
  - Facilitar el trabajo en equipo
    - El software actual es tan complejo que es necesario desarrollarlo en equipo
    - Los programadores solitarios sólo existen en las películas fantasiosas sobre hackers
    - Por lo tanto, mal asunto si un código es tan enrevesado que sólo lo entiende el programador que inicialmente lo hizo
      - Después de años, ni siquiera lo entenderá el programador que inicialmente lo programó

¡El mantenimiento en un software “mal programado” supone más del 80% de los gastos!

Razones II

- No sólo es importante la buena programación por la reutilización y el mantenimiento, sino también por:
  - Nuestra sociedad hace un uso intensivo del software, y por lo tanto demanda una altísima calidad en ello
    - Si el software no es bueno, se pueden perder millones de dólares
      - Ej: desastre del cohete Ariane 5 en 1996
      - Ej: el 6 de Mayo de 2010 la bolsa americana cayó un 10% en segundos por errores de software
      - Ej: ¡el dinero de nuestras cuentas bancarias es software!
    - Si el software no es bueno, se pueden perder vidas humanas
      - Ej: software de control de aviación aérea
    - Mas ejemplos: https://www.wired.com/2005/11/historys-worst-software-bugs/
    - Si el software no es bueno, ¿qué ventaja tenemos los ingenieros de software frente a los tecnicos que son mucho más baratos?
  - La profesión de Ingeniero de Software requiere ser una persona perfeccionista, cuidadosa, con visión matemática y detallista
    - Nos acostumbraremos a ello en este curso

Comentarios I

- Buena redacción y sin faltas ortográficas
- Claros y sin ambigüedades.
- Concisos. Sin paja.
- Comentario inicial en cada clase, antes de su cabecera. Expresa en pocas palabras para qué sirve y qué representa
- Comentario en cada declaración (no definición) de método, justo antes de su cabecera
  - Descripción general del método
  - Descripción de cada parámetro, indicando su significado, rango de valores y todo lo que se considere necesario
  - Descripción del resultado, indicando su significado, rango de valores y todo lo que se considere necesario
  - Precondiciones del método, expresadas matemáticamente siempre que sea posible, usando en caso necesario el resultado de otros métodos
  - Complejidad computacional O(), tanto de tiempo como de memoria, en el peor caso. El profesor podrá exigir también, en algunos ejercicios, el caso medio y el mejor caso.
- Comentario en cada declaración de variable de un método o atributo de una clase, explicando lo que es y para qué se utiliza
- Comentario en cada parte difícil del código (ej: bucles anidados)

Comentarios II

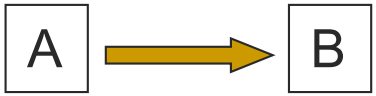
- Todos los comentarios van en los ficheros .h excepto los de las variables internas de un método y los de las partes difíciles del código
- Acostumbrarse a comentar todo es una buena práctica de programación por varias razones:
  - A lo que nosotros nos parece un «nombre obvio» puede que no lo sea para otra persona
  - Lo que hoy nos parece obvio, quizás lo veamos dentro de algunos meses y ya no lo sea tanto
  - Los programas que generan ayuda automáticamente requieren que todo esté comentado como estamos diciendo

Nomenclatura

- En nuestro caso, seguimos la nomenclatura Java:
  - Todos los nombres (de procedimientos, clases, variables, etc.) tienen que ser descriptivos
    - ¡Nada de abreviaturas!
  - Todos los nombres son un conjunto de palabras seguidas, sin espacios ni guiones entre ellas
    - Cada una (excepto la primera) empieza por letra mayúscula
    - El resto de letras son en minúsculas
    - Ej: variable «numeroAvisosUrgentes»
  - La primera letra de cada nombre:
    - Es mayúscula sólo si es el nombre de una clase
    - Es minúscula en todos los demás casos (procedimientos, variables, etc.)
  - Excepción: los nombres de las constantes tienen todas sus letras en mayúsculas
  - Los nombres de las clases y las variables son siempre sustantivos
  - Los nombres de los procedimientos son siempre verbos

Mínimo acoplamiento I

- Tenemos que seguir el “Principio de mínimo acoplamiento”
  - Una pieza de software es:
    - O bien un programa entero
    - O bien una parte de un programa
      - Un conjunto de líneas de código
      - Una clase completa
      - Un método completo
      - Etc.
    - En UML (Lenguaje Unificado de Modelado), una pieza de software se representa por un rectángulo con el nombre de la pieza de software escrito dentro de él
  - Una pieza de software “A” depende de una pieza de software “B” cuando “A” necesita que se haya ejecutado previamente “B” para que “A” funcione bien
    - Ej: “A” depende de “B” si “A” llama a un procedimiento de “B”
    - Ej: Un bucle que recorre un array de enteros para imprimirlo (pieza “A”) necesita que previamente se haya declarado y llenado dicho array de enteros (pieza “B”)
    - Cuando “A” depende de “B”, en UML se representa con una flecha desde “A” a “B”



  - Se dice que dos piezas de software “A” y “B” (“A” depende de “B”) están altamente acoplados cuando, al cambiar “B”, nos vemos forzados a cambiar “A” si queremos que “A” siga funcionando
    - Cuantos más cambios haya que hacer en “A”, mayor acoplamiento

Mínimo acoplamiento II

- Si el acoplamiento es alto, cuando queremos cambiar algo, lo tenemos que cambiar en varios sitios en el código.
- Esto es malo porque entorpece el mantenimiento debido a que:
  - Tardamos más en hacer el mantenimiento, al tener que cambiar el código en muchos sitios
  - Al tener que cambiar el código en varios lugares, aumentamos la probabilidad de un error humano (nos podemos equivocar o despistar al cambiar el código en muchos lugares)

Mínimo acoplamiento III

- Ejemplo: Un programa imprime los 10 primeros números naturales, entre otras operaciones.
  - Podemos dividir el código en al menos tres piezas de software relacionadas entre ellas
    - A: declarar un array de 10 componentes
    - B: llenar el array
    - C: imprimir el array

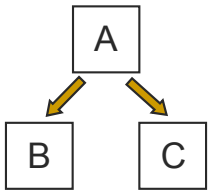
  - En la versión altamente acoplada, si queremos ahora cambiar el programa para imprimir los 20 primeros en vez de los 10 primeros, tendríamos que modificar el código en tres lugares.
    - Por lo tanto, ¿qué alternativa en pseudocódigo es mejor desde el principio de mínimo acoplamiento?

```
Main (...)
   int numeros[10];
   desde i=0 hasta 9
      numeros[i]=i+1
   ...
   desde i=0 hasta 9
      print(numeros[i])
```

```
#define N 10
Main (...)
   int numeros[N];
   desde i=0 hasta N-1
      numeros[i]=i+1
   ...
   desde i=0 hasta N-1
      print(numeros[i])
```

Mínimo acoplamiento IV

- Cuando hay dependencia entre dos piezas de software, siempre hay algo de acoplamiento entre ellas
  - Ej: si el procedimiento “A” llama al procedimiento “B”, si cambiamos el nombre o el tipo de los parámetros de “B”, posiblemente tengamos que cambiar algo en “A” para que pueda seguir llamándolo
- Por lo tanto, si no sabemos nada más, cuantas más dependencias haya entre las piezas de software que componen un programa, más acoplamiento suponemos que habrá
  - Ejemplo: ¿cuál de los dos programas, divididos cada uno de ellos en tres piezas de software, es mejor desde el punto de vista del principio de mínimo acoplamiento, si no sabemos nada más?



Máxima cohesión

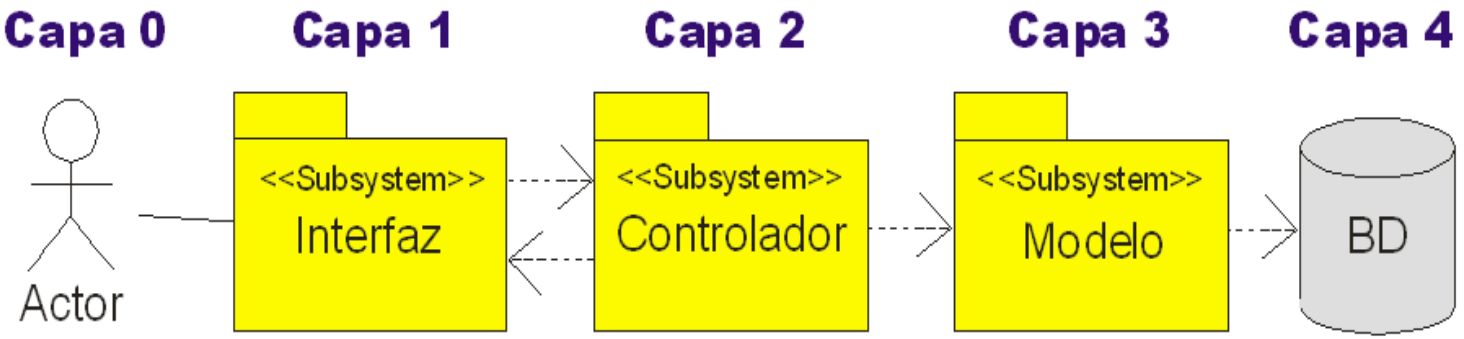
- Tenemos que seguir el “Principio de máxima cohesión”
  - Una pieza de software (generalmente una función, procedimiento, método o clase) está muy cohesionada cuando realiza sólo una tarea bien definida, y no varias
    - De este modo podremos reutilizar esa pieza en cualquier lugar en donde se necesite justamente esa tarea
  - Ej: un programa obtiene, de una base de datos de alumnos, los datos de un alumno concreto con el fin de imprimirlo por pantalla. ¿Cuál de las dos alternativas en pseudocódigo es mejor desde el punto de vista del principio de la máxima cohesión?

```
Main (...)
   int idJuan = ...
   Alumno juan = obtenerAlumno(idJuan)
   imprimirAlumno (juan)

Alumno obtenerAlumno (int idAlumno)
   comprobar si idAlumno existe en la BD
   si existe
      obtener sus datos de la BD
      devolver datos
   si no existe
      devolver error

void imprimirAlumno (Alumno unAlumno)
   si unAlumno==error
      print(“Juan no se encuentra”)
   si no
      ... // Obtenemos datos concretos del alumno
      print(...) // Imprimimos
```

```
Main (...)
   int idJuan = ...
   si existeAlumno(idJuan)
      Alumno juan = obtenerAlumno(idJuan)
      imprimirAlumno (juan)
   si no
      print ("Juan no se encuentra")

Alumno obtenerAlumno (int idAlumno)
   obtener datos del idAlumno de la BD
   devolver datos

bool existeAlumno(int idAlumno)
   comprobar si idAlumno existe en la BD
   si existe
      devolver true
   si no
      devolver false

void imprimirAlumno (Alumno unAlumno)
   ... // Obtenemos datos concretos del alumno
   print(...) // Imprimimos
```

Patrón MVC

- Hay que dividir el programa entre «interfaz» (también llamado «vista»), «control» y «modelo» (se llama paradigma, modelo o patrón «MVC»)
  - Además de MVC, existen más patrones interesantes, pero MVC es el más importante
  - Por lo tanto, no puede mezclarse vista con modelo en la misma clase y/o método
  - Si la división es clara, podemos cambiar fácilmente el interfaz de un programa sin afectar a su modelo (su núcleo). Esto favorece la reutilización del modelo en diferentes ambientes con diferente interfaz.
    - También podemos vender el modelo para ser reutilizado en otros programas con diferente interfaz, sin necesidad de tocar el modelo
  - El control sólo existe en programas complicados, y es una capa intermedia entre la vista y el modelo
  - Se minimizan las dependencias entre estas tres partes, y por lo tanto se minimiza el acoplamiento y se aumenta la cohesión que permite reutilizar cada parte por separado en otros proyectos. En UML se expresaría así:



División de los programas

- Hay que dividir bien los programas:
  - Dividir correctamente el programa en ficheros .cpp y .h
    - Los archivos .h contienen las declaraciones de clases, atributos y métodos, junto con los comentarios de cabecera
    - Los archivos .cpp contienen las definiciones de los métodos, junto con los comentarios internos de cada uno (de las variables locales y de las partes complicadas del código)
  - Dividir correctamente el programa en clases
    - El reparto de responsabilidades entre las clases es «natural». Cada clase es un «ser» que hace sólo lo que tiene que hacer
    - ¿Cómo hacerlo bien?
      - Primero, ante el enunciado del problema, subrayamos los sustantivos principales. En principio cada uno es una clase.
      - Ahora rechazamos las clases que sean demasiado pequeñas (uniéndolas a otras) y rechazamos las clases que sean demasiado grandes (diviéndolas en clases más pequeñas)
        - El tamaño de una clase se mide por el número de atributos y métodos que contiene
      - No olvidar el patrón MVC: cada clase debe ser únicamente uno de los tres tipos
  - Dividir correctamente cada clase en métodos
    - ¿Cómo? Siguiendo los principios de máxima cohesión y mínimo acoplamiento

Presentación

- Buena presentación de los programas
  - Imprescindible para trabajar en equipo, o incluso para que el programador original comprenda su propio trabajo después de meses o años de haberlo hecho
  - Dividir el código en partes lógicas, separadas por líneas en blanco
  - Comentar cada parte lógica al principio de la misma, diciendo para qué sirve
  - El código debe estar correctamente indentado
    - El compilador nos puede indentar automáticamente el código. Es aconsejable hacerlo a menudo.

Precondiciones I

- Una precondición de un método es una condición booleana que se debe cumplir antes de poder usar (llamar) a dicho método, si queremos que dicho método funcione
  - Ej: en el método obtenerAlumno() que vimos en el principio de máxima cohesión, podemos establecer la precondición de que exista dicho alumno que queremos recuperar de la base de datos
- La precondición está pensada para que la sepa el usuario del método (el que llama al método)
  - Por lo tanto hay que documentar la precondición en los comentarios del método, con lenguaje matemático y preciso siempre que sea posible
- Le decimos al usuario en qué casos funciona dicho método (cuando se cumple la precondición) y en qué casos no (cuando no se cumple la precondición)
  - Si no se cumple la precondición, el programador no asegura que el método funcione apropiadamente (se lava las manos)
  - El que se cumpla la precondición antes de llamar al método es responsabilidad del usuario de dicho método
- De este modo, el método se programa pensando en que la precondición se está cumpliendo
  - Esto facilita mucho la programación del método
  - Ej: obtenerAlumno() no tiene que preocuparse de que el alumno existe en la base de datos. Asume que sí existe. Y esto ayuda a que el método pueda tener máxima cohesión, como veremos posteriormente.

Precondiciones II

- Al inicio del cuerpo del método, deben también estar programadas con uno o varios “assert()”
  - El método assert() recibe como parámetro una expresión booleana en C que se evaluará como verdadero o falso
  - Si se evalúa como verdadero, no se hace nada y el programa continúa
  - Si se evalúa como falso, se interrumpe automática el programa entero, especificando el assert que lo hizo
- Los assert sólo se compilan (y por lo tanto funcionan) cuando compilamos en “modo debug”
  - Es el modo que utilizamos cuando se está construyendo o probando (versiones alfa o beta) un programa
  - Una vez que el programa es estable y lo ponemos en producción, compilamos en “modo release”, lo que significa que los asserts no se compilarán, y por lo tanto el programa se ejecutará más rápido
- Hay lenguajes (ej: Eiffel) que siempre compilan los assert, para que el código sea más fiable incluso en producción

Precondiciones III

- Consideremos el ejemplo que vimos en el principio de máxima cohesión
  - Ponemos la precondición tanto en los comentarios como en los assert() de obtenerAlumno()
  - ¿Quién es el responsable de que la precondición de obtenerAlumno() se cumpla antes de llamar a obtenerAlumno()? El usuario de obtenerAlumno(), que en este caso es el método main()
  - Si obtenerAlumno() no tuviera la precondición, obtenerAlumno() tendría que comprobar internamente (con un if) que el alumno existe, lo cual:
    - Disminuiría la cohesión de obtenerAlumno(), pues ahora este método haría dos tareas: comprobar que el alumno existe, y sacar sus datos de la BD
    - Plantearía nuevos problemas: si no existe el alumno, ¿de qué modo se devuelve un error? El método obtenerAlumno() sólo puede devolver un objeto de tipo "Alumno", no un error. Podríamos devolver un alumno vacío para indicar error, pero esto es antinatural y además incrementa el acoplamiento, pues el usuario (el método main) tendría que conocer y manejar el hecho de que obtenerAlumno() devuelva un alumno vacío

```
main (...)
   int idJuan = ...
   si existeAlumno(idJuan)
      Alumno juan = obtenerAlumno(idJuan)
      imprimirAlumno (juan)
   si no
      print ("Juan no se encuentra")

// Precondición: existeAlumno(idAlumno)
Alumno obtenerAlumno (int idAlumno)
   assert(existeAlumno(idAlumno))
   obtener datos del idAlumno de la BD
   devolver datos

bool existeAlumno(int idAlumno)
   comprobar si idAlumno existe en la BD
   si existe
      devolver true
   si no
      devolver false

void imprimirAlumno (Alumno unAlumno)
   ... // Obtenemos datos del alumno
   print(...) // Imprimimos
```

Complejidad

- Complejidad:
  - Hay que indicar, en los comentarios iniciales de cada método, la complejidad del mismo
  - Hay que indicarlo con O(…), de forma matemática y precisa
  - Hay que indicar tanto la temporal como la espacial
  - Hay que indicar el peor caso
  - Hay que indicar también el mejor caso y el caso medio
    - Sólo cuando el enunciado del programa lo pida