Departamento de Ingeniería Informática
Grado en Ingeniería Informática
Elisa Guerrero Vázquez
Esther L. Silva Ramírez

Metodología de la Programación

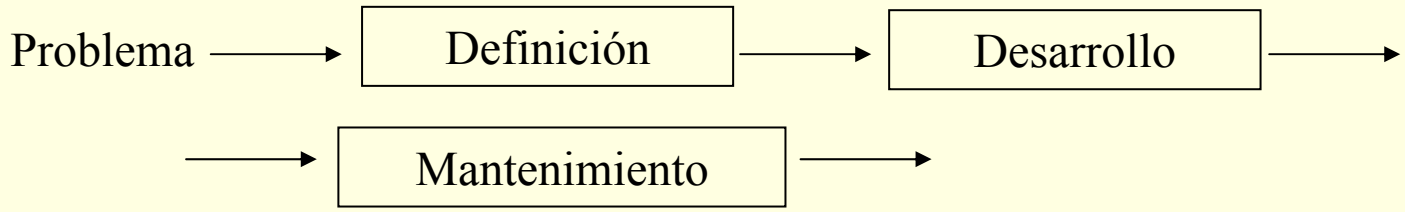
TEMA 3 – TEORÍA

DISEÑO MODULAR

1. Ciclo de vida del software

Desde un punto de vista general, el proceso de desarrollo del software contiene tres fases genéricas, independiente del paradigma de ingeniería elegido, que son:

- **Definición.** Se centra en *qué* hace el sistema.
- **Desarrollo.** Se centra en *cómo* se ha de realizar el sistema.
- **Mantenimiento.** Se centra en el *cambio* que va asociado al software desarrollado, debido a errores o a una posible mejora o ampliación del sistema.



La clasificación anterior es muy general, se pueden desglosar en actividades bien definidas, de esta forma las **etapas del ciclo de vida del software** son:

- **Análisis**, en esta etapa se analiza y define el problema, especificando el qué queremos. A su vez se distinguen dos fases:

  - Análisis del sistema (análisis y definición del problema).

  - Especificación de requisitos.

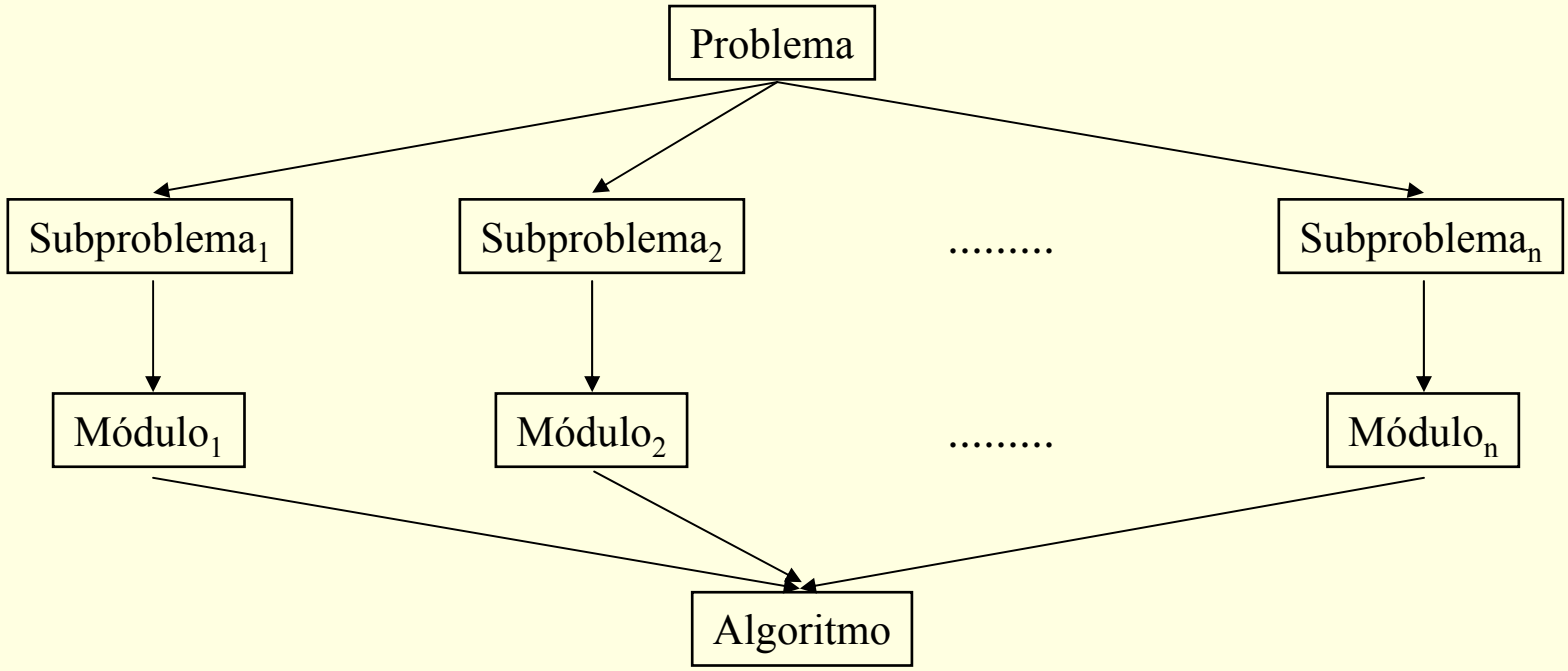
- **Diseño**, en esta etapa se determina cómo se van a realizar los requisitos recogidos y organizados.
- **Implementación** o codificación. En esta etapa, se traduce el algoritmo diseñado a un lenguaje de programación.
- **Pruebas**. En esta etapa se realizan pruebas del programa para detectar errores.
- **Mantenimiento.** Se modifica el programa para corregir errores o para realizar mejoras y adaptación del programa.

2. Introducción al diseño modular

- Características de un buen programa:
  - Correcto
  - Legible
  - Fácil de modificar y mantener
  - Eficiente
- Dificultad de escribir grandes programas para resolver problemas complejos.
  - La capacidad mental es limitada e insuficiente para tratar con precisión todos los elementos del problema al mismo tiempo.
  - El código resultante es más difícil de entender y por tanto, también es difícil de depurar, modificar y mantener.

2. Introducción al diseño modular

- Diseño modular:
  - Metodología de desarrollo de algoritmos basada en la descomposición del problema en subproblemas y en la resolución separada de cada uno de ellos mediante un módulo.
- La descomposición del problema conduce a dividir el algoritmo en componentes llamados *módulos*.
- Cada subproblema se resuelve por separado mediante un módulo y la solución global se obtiene combinando las soluciones parciales mediante la integración de los diferentes módulos en un algoritmo.



Módulo:

Unidad de organización de un algoritmo que realiza funciones específicas con un objetivo diferenciado del de otros módulos.

Está formado por una colección de definiciones de tipos, constantes, variables y funciones. Algunos de estos elementos son de uso exclusivamente interno (privados) y están ocultos respecto a cualquier declaración o acción ajena al propio módulo, mientras que otros elementos se ofrecen para uso externo (públicos) en otras partes del algoritmo.

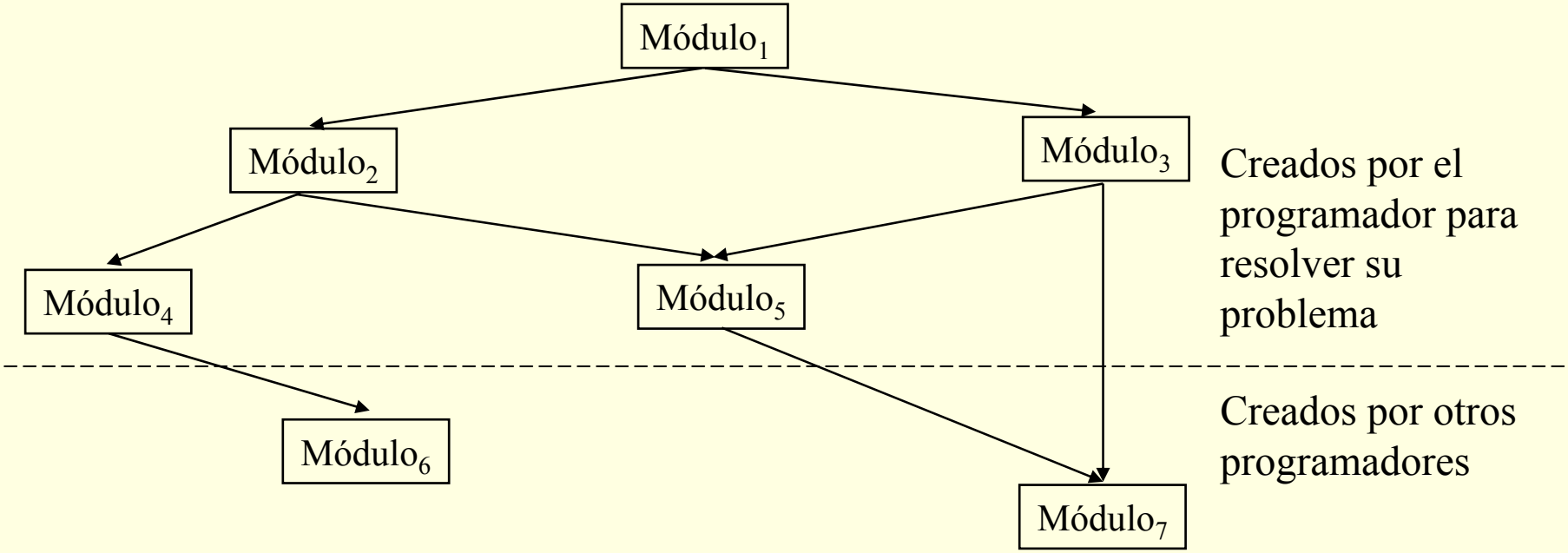
- Un módulo consta de una sección de importación y de otra sección de exportación.
  - **Sección de importación**: conjunto de elementos utilizados en un módulo y declarados o definidos en otros módulos.
  - **Sección de exportación**: conjunto de elementos que un módulo hace públicos para ser utilizados en otros módulos.

- El programador puede actuar como **creador** o como **usuario** de un módulo.
  - Como *usuario* **sólo conoce** los elementos exportados por el módulo:
    - Definición de los tipos de datos públicos.
    - Especificación de las funciones que actúan sobre estos tipos.
  - Como *usuario* **no sabe** ni puede deducir la implementación y por tanto tampoco puede modificarla.

- El programador puede actuar como **creador** o como **usuario** de un módulo.
  - Como *creador* **debe decidir**:
    - Elementos públicos: los necesarios en el exterior del módulo para poder utilizarlo.
    - Elementos privados: quedan ocultos en el interior del módulo y por tanto no deben ser necesarios para utilizar el módulo.

3. Jerarquía de módulos

- Un algoritmo es una colección de módulos organizados jerárquicamente, en la que unos utilizan elementos incluidos en otros.



- Una flecha entre dos módulos significa que el primero utiliza elementos del segundo.

4. Criterios de descomposición

- **Descomposición por tareas:** Se identifican las distintas operaciones a realizar con los datos y se implementa un módulo para cada una.

- **Descomposición por datos:** Se identifican los tipos de datos que intervienen en el problema y a continuación las operaciones asociadas a cada uno de ellos. Se crea un módulo para cada tipo de dato y sus operaciones.

5. Relación entre módulos

Para determinar de forma objetiva la relación entre módulos se definen dos criterios cualitativos:

- Cohesión: relación funcional que existe entre los elementos de un mismo módulo. Se organizan de tal manera que los que tengan una mayor relación a la hora de realizar una tarea pertenezcan al mismo módulo y los elementos no relacionados se encuentren en módulos separados.

- Acoplamiento: grado de interdependencia entre los módulos. Para obtener un buen diseño se debe intentar siempre minimizar el acoplamiento; es decir, hacer los módulos tan independientes unos de otros como sea posible. Para conseguir un acoplamiento mínimo ningún módulo tiene que preocuparse de los detalles de la construcción interna del resto de los módulos.

6. Esquema de un módulo

módulo nombre
    importa
      Lista de importaciones
      // relación de módulos de los que se importa algo
    fin_importa

    exporta
      Lista de exportaciones
      // relación de elementos exportables
    fin_exporta

    implementación
      declaraciones/definiciones de tipos, constantes y variables
      procedimientos
      funciones
    fin_implementación
fin_módulo

Ejemplo

Dado un texto acabado en un carácter de control determinado, suponiendo el texto no vacío, diseñar un algoritmo que cuente el número de palabras del texto que son anagramas de la primera palabra.

Se dice que una palabra *w* es un anagrama de la palabra *v*, si podemos obtener *w* cambiando el orden de las letras de *v*, es decir, *w* es una permutación de *v*. Por ejemplo, *vaca* lo es de *cava*.

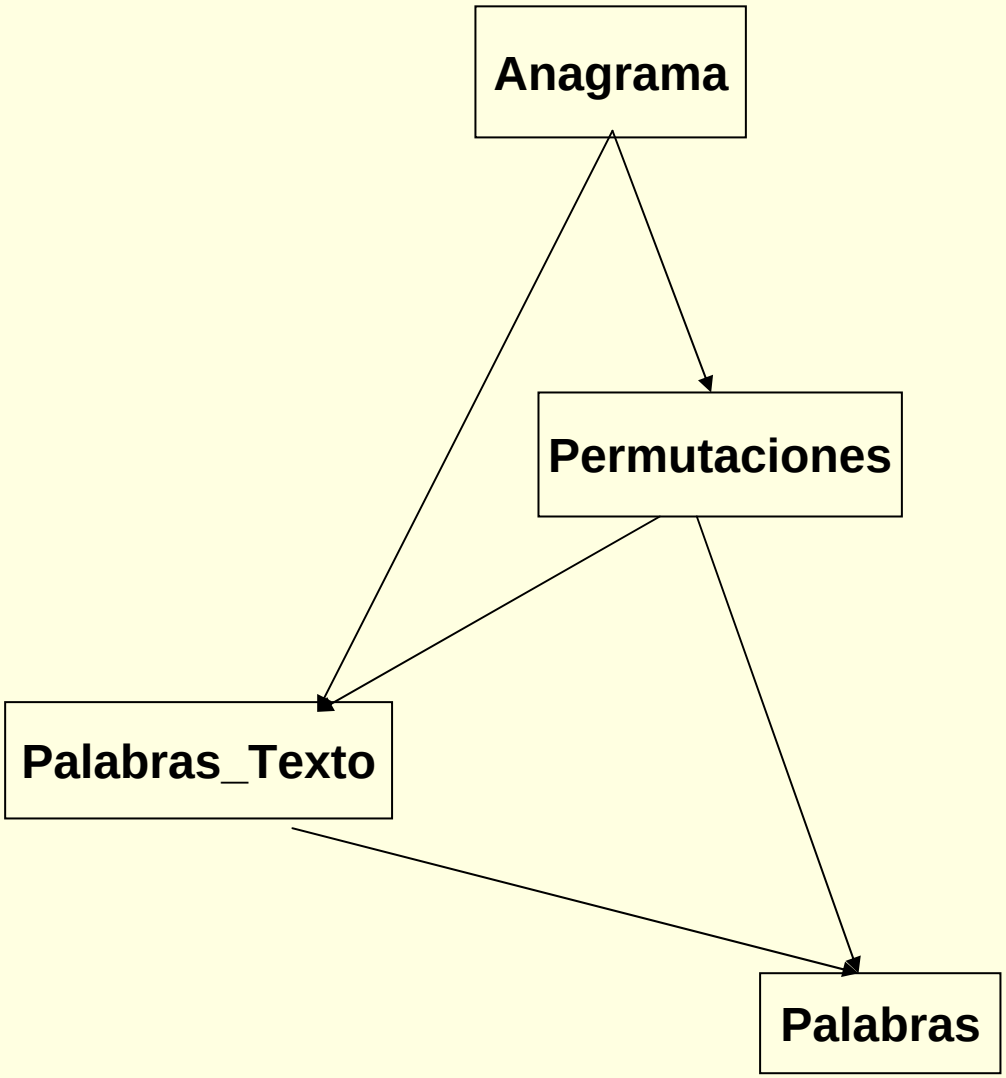
- Flora – Farol Brasil – Silbar Narciso – Cornisa
- Canarias – Sacarina Irónicamente – Renacimiento
- Bulliciosamente - Escabullimiento

Texto compuesto por palabras

zarin rizan cabello nizar carisma palabrería
material pata calar anudar raniz
maletín informática calabaza nariz

- Contar las palabras que son Anagramas de la primera palabra: ZARIN

```
Algoritmo Anagrama
inicio
  inicializar contador
  leer primera palabra
  mientras no fin texto hacer
                leer siguiente palabra
                si es permutación entonces
                        incrementar contador
                fin_si
  fin_mientras
  escribir(contador)
fin_algoritmo
```


Anagrama
Permutaciones
Palabras_Texto
Palabras

```
módulo Permutaciones
    importa
        Palabras
        Palabras_Texto
    fin_importa
    exporta
        entero función cuenta_permutaciones (E texto: txt)
        //Precondición: txt es un texto de tamaño máximo MAX_T,
        //representado mediante un  vector de caracteres acabado
        //en FIN_TEX.
        //Postcondición: Devuelve el número de palabras del texto
        //  que son anagramas de la primera palabra
    fin_exporta
```

```
implementación
entero función cuenta_permutaciones(E texto: txt)
var
    palabra: pal1, pal2
    entero: cursor, num_permut
inicio
    num_permut ← 0
    cursor ← 1
    leer_palabra (txt, cursor, pal1)
    mientras no fin_de_texto(txt, cursor) hacer
        leer_palabra (txt, cursor, pal2)
        si palabra_es_permut(pal1, pal2) entonces
            num_permut ← num_permut + 1
        fin_si
    fin_mientras
    devolver num_permut
fin_función
fin_implementación
fin_módulo
```

```
módulo Palabras_Texto
    importa
        Palabras
    fin_importa
        exporta
        const MAX_T, FIN_TEX
        tipo texto
        procedimiento leer_palabra (E texto: txt, E/S entero: cursor,
            S palabra: pal)
        //Precondición: txt es un texto de tamaño máximo
        //   MAX_T, representado mediante un vector de
        //   caracteres acabado en FIN_TEX y cursor es un entero que
        //   representa una posición en el texto.
        //Postcondición: pal es una palabra de tamaño
        //   máximo MAX_P, que contendrá la primera
        //   palabra del texto txt y la marca FIN_PAL. cursor
        //   contendrá la posición del primer carácter de la
        //   siguiente palabra de txt o FIN_TEX.
```

```
    lógico función  fin_de_texto (E texto: txt, E entero:cursor)
    //Precondición: txt es un texto de tamaño máximo MAX_T,
    //   representado mediante un vector de caracteres acabado en
    //   FIN_TEX y cursor es un entero que representa una posición
    //   en el texto.
    //Postcondición: Devuelve si cursor corresponde a la posición de
    //   la marca FIN_TEX.

fin_exporta

implementación
const
    MAX_T =300
    FIN_TEX =’$’
tipo
    vector[MAX_T+1] de caracter: texto
```

```
procedimiento leer_palabra (E texto: txt, E/S entero: cursor, S palabra: pal)
var
    entero: i
inicio
    i ← 1
    mientras txt[cursor] ≠ ' '∧ ¬fin_de_texto (txt, cursor) hacer
        pal[i] ← txt[cursor]
        i ← i + 1
        cursor ← cursor + 1
    fin_mientras
    pal[i] = FIN_PAL
    mientras txt[cursor] = ' ' hacer
        cursor ← cursor + 1
    fin_mientras
fin_procedimiento
```

```
lógico función fin_de_texto (E texto: txt, E entero:cursor)
inicio
    devolver txt[cursor]=FIN_TEX
fin_función

fin_implementación

fin_módulo
```

```
módulo Palabras
    exporta
        const  MAX_P, FIN_PAL
        tipo palabra

        lógico función palabra_es_permut (E palabra: pal1,
            E palabra: pal2)
        //Precondición: pal1 y pal2 son palabras de tamaño
        //  máximo MAX_P, representadas mediante un
        //  vector de caracteres acabado en FIN_PAL.
        //Postcondición: Devuelve si pal2 es una permutación
        //  de pal1 o no.
    fin_exporta
```

```
implementación
const
    MAX_P =20
    FIN_PAL =’#’
tipo
    vector[MAX_P+1] de caracter: palabra

lógico función palabra_es_permut (E palabra: pal1,  E palabra: pal2)
var
    lógico: permut
    entero: i, p
inicio
    //Buscamos, para cada carácter en pal1, si está en pal2. En caso
    // afirmativo se borra de ésta y  se continúa la comprobación. En
    // caso negativo se detiene el cómputo y se devuelve el valor
    // falso
    permut ← verdadero
```

```
    si longitud(pal1) ≠ longitud(pal2)  entonces
        permut ← falso
    si_no
        i ← 1
        mientras (i <= longitud(pal1)) y permut = verdadero hacer
            p ← pos_caracter (pal2, pal1[i])
            si p<=longitud(pal2) entonces
                borrar (pal2, p)
            si_no
                permut ← falso
            fin_si
            i ← i + 1
        fin_mientras
    fin_si
    devolver permut
fin_función
```

```
//Funciones y procedimientos no exportables.

//Cabecera: entero función longitud  (E palabra: pal)
//Precondición: pal es una palabra de tamaño máximo MAX_P,
//   representada  mediante un vector de caracteres acabado en
//   FIN_PAL.
//Postcondición: Llama a la función longitud_recursiva para devolver la
longitud de la palabra pal.

entero función longitud (E palabra: pal)
//Privada
inicio
     devolver  longitud_recursiva(pal,1)
fin_función
```

```
//Funciones y procedimientos no exportables.

//Cabecera: entero función longitud_recursiva  (E palabra: pal, E entero: i)
//Precondición: pal es una palabra de tamaño máximo MAX_P,
//   representada  mediante un vector de caracteres acabado en
//   FIN_PAL.
//Postcondición: Devuelve la longitud de la palabra pal.

entero función longitud_recursiva (E palabra: pal, E entero: i)
//Privada
inicio
    si pal[i]=FIN_PAL entonces
        devolver  i-1
    si_no
        devolver  longitud_recursiva(pal, i+1)
fin_función
```

//Cabecera: entero **función** **pos_caracter** (E palabra: pal, E caracter: c)

//Precondición: *pal* es una palabra de tamaño máximo MAX_P,

// representada mediante un vector de caracteres acabado en

// FIN_PAL y *c* es un carácter

//Postcondición: Realiza la llamada a la función recursiva para devolver la posición de la primera ocurrencia de *c* en *pal.* Si *c* no aparece en *pal*, devuelve la longitud de *pal* más 1.

entero **función** **pos_caracter**(E palabra: pal, E caracter: c)

//Privada. Puede hacerse pública incluyéndola en la sección de exportaciones

var

entero: n

Inicio

n ← longitud(pal)

devolver pos_caracter_recursiva(pal, c, 1, n)

fin_función

```
//Cabecera: entero función pos_caracter_recursiva(E palabra: pal, E caracter: c,
                                                  E entero: i, E entero: n)
//Precondición: pal es una palabra de tamaño máximo MAX_P,
//    representada mediante un vector de caracteres acabado en
//    FIN_PAL y c es un carácter
//Postcondición: Devuelve la posición de la primera ocurrencia de c en
//    pal. Si c no aparece en pal, devuelve la longitud de pal más 1.
entero función pos_caracter_recursiva (E palabra: pal, E caracter: c,
                                       E entero: i, E entero: n)
//Privada. Puede hacerse pública incluyéndola en la parte lista de exportaciones
inicio
    si i>n entonces      //no ha encontrado el carácter y ha llegado al final
        devolver  i
    si_no  si pal[i]=c  entonces   //ha encontrado el carácter
            devolver i
        si_no   // pal[i] ≠ c  sigue buscando dentro de la palabra
                devolver pos_caracter_recursiva(pal, c, i+1, n)
        fin_si
    fin_si
fin_función
```

```
//Cabecera: procedimiento borrar  (E/S palabra: pal, E entero: pos)
//Precondición: pal es una palabra de tamaño máximo MAX_P,
//    representada mediante un vector de caracteres acabado en
//   FIN_PAL, siendo pos∈ N ∧ 1 <= pos  <=longitud(pal)
//Postcondición: Suprime el contenido de la posición pos de pal
//    desplazando las siguientes una posición hacia la izquierda

procedimiento borrar (E/S palabra: pal, E entero: pos)
//Privada.
var
    entero: i
inicio
    desde i ← pos hasta longitud(pal) hacer
            pal[i] ← pal[i +1]
    fin_desde
fin_procedimiento
fin_implementación
fin_módulo
```

7. Ventajas del diseño modular

- Reduce la complejidad de los problemas
- Mejora la legibilidad de los programas
- Facilita la implementación
- Facilita la depuración
- Facilita la modificación y el mantenimiento de los programas
- Favorece la reutilización de módulos para resolver otros problemas.

8. Documentación

La documentación ayuda a leer y comprender mejor el programa haciendo más fáciles las tareas de mantenimiento (modificaciones) y las de depuración.

Suele distinguirse entre:

Documentación *interna* (la contenida en líneas de comentarios)

Documentación *externa*, que incluye análisis, diagramas de flujo y/o pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados. Dentro de ésta se distingue:

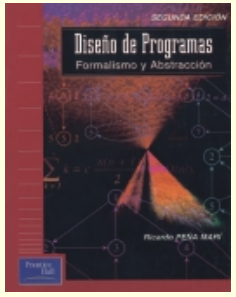
- Documentación del usuario (funciones del sistema).
  - Descripción funcional sobre lo que hace el sistema.
  - Documento que explique como instalar el sistema y adecuarlo para configuraciones particulares del hardware.
  - Manual introductorio que explique cómo iniciarse en el sistema y cómo salir de él.
  - Manual de referencia que describa con detalle las ventajas del sistema disponibles para el usuario y cómo se pueden usar.
  - Guía del operador, si fuera necesario.

- Documentación del sistema (diseño, implantación y pruebas del sistema).
  - Documento de definición de requisitos.
  - Documentación que describa cómo se descompone el problema en distintos subproblemas y los módulos asociados a cada uno de ellos, acompañados de su especificación.
  - Documentación por cada módulo, que describa lo que hace.
  - Un plan de pruebas que describa cómo se prueba cada módulo.
  - Un plan de prueba que muestre cómo se efectuó la integración de los distintos módulos y se probó todo junto.
  - Un plan de pruebas de aceptación, debe describir las pruebas que son necesarias pasar para que el sistema sea aceptado.

Bibliografía

- Castro Rabal, Jorge; Cucker Farkas, Felipe (1993). Curso de programación. McGraw-Hill / Interamericana de España, S.A.

- Bálcazar José Luis (2001). Programación Metódica. McGraw-Hill.



- Peña Marí, Ricardo; (1998) Diseño de Programas. Formalismo y Abstracción. Prentice Hall.