

## Documentación

### Introducción

La creación de nuevo sistema informático conlleva diversas fases que van más allá de la codificación del mismo. Un modelo tradicional de diseño establece cinco fases: análisis, diseño, codificación, pruebas y mantenimiento. En la primera se realiza un estudio de los requisitos del sistema y la viabilidad del mismo. La fase de diseño identifica y define los elementos que precisa el proyecto y la interconexión entre los mismos. Del resultado de estas dos fases se obtiene un plan que se utilizará como guía para la codificación del sistema. El producto resultante de la codificación se verá sometido a una batería de pruebas que aseguren la fiabilidad del mismo. Finalmente, el ciclo de vida de un sistema informático no termina en las pruebas, si no que se extiende mientras que esté en uso. Así, es necesario realizar y preveer un mantenimiento del mismos. Todas las fases deberían estar debidamente documentadas, permitiendo que en futuro, se puedan acometer cambios y mejoras en el sistema de manera sencilla. En este modelo clásico se contempla, además, que se pueden pasar por las distintas fases más de una vez, permitiendo retroceder a una fase anterior si se descubre una deficiencia que impida continuar. De esta manera, el producto final se obtiene mediante una serie de refinamientos sucesivos. Al finalizar cada una de las fases, se cumple un hito del proyecto, el cual, suele venir acompañado de un entregable. Este puede ser, por ejemplo, un nuevo documento o un listado de código, o una nueva versión de un entregable anterior.

En el presente documento nos centramos en explicar como se describirían las estructuras de datos y la algoritmia de un proyecto. Ambas podrían incluirse en entregables, por ejemplo, tanto en la fase de diseño como en la de documentación. Ahora bien, suponemos que se ha escogido un diseño estructurado para acometer el proyecto, y en este sentido irán las indicaciones. Existen otros paradigmas de programación, tales como orientación a objetos, que requieren un enfoque distinto en la elaboración de los documentos. Dentro del diseño estructurado, la herramientas que vamos a plantear permiten una descripción detallada del proyecto Existen otros modelos (ej, DFD, DEC...) que permiten realizar una descripción de más alto nivel, pero que quedan fuera de este documento.

### Estructuras de datos

Nicolas Wirth acuñó el famoso lema de la programación estructurada:

*Algoritmos + Estructuras de Datos = Programas*

En este apartado nos centraremos en describir las estructuras de datos, mientras que en el segundo lo dedicaremos a la parte algorítmica, ya que es ese en el orden en el que deberían aparecer explicadas en la documentación. Aunque durante la fase de diseño la elección de las estructuras de datos y algoritmos, de nuevo, consiste en un

proceso de refinamiento, a la hora presentar el resultado final se plantean primeramente las estructuras de datos, y a continuación, los algoritmos que se desprenden.

La definición de las estructuras parte de conjunto de tipos de datos básicos, y de mecanismos que nos permitan generar nuevos tipos de datos. Uno de nuestros objetivos es conseguir una descripción del proyecto independiente del lenguaje de programación escogido. Para ello, se proponen emplear una serie de tipos de datos genéricos que son comunes a la mayor parte de lenguajes estructurados.

Tipos básicos:

- entero
- real
- booleano
- carácter
- enumerado

Tipos compuesto:

- array (matriz)
- estructura
- cadena

La declaración de las estructuras de datos constará de dos partes. En la primera se definirán las constantes disponibles. En la segunda, los tipos de datos que se van a emplear. Cada constante, tipo o campo de un nuevo tipo irá acompañado de un comentario que explique la necesidad del mismo.

### **Ejemplo 1: Tres en raya**

Las tres en rayas es juego de dos jugadores. Se juega sobre un tablero de  $N \times M$ , y con dos tipos de fichas (tradicionalmente un X ó O). Cada jugador coloca en turnos sucesivos una ficha sobre alguna de las casillas vacías del tablero. El juego finaliza cuando o bien algún jugador consigue alinear tres de sus fichas en cualquiera de las direcciones posibles (vertical, horizontal o diagonal), o bien cuando no queda ninguna casilla vacía dónde colocar ficha.

A continuación se van a describir las posible estructuras de datos necesarias para un programa que implementara dicho juego. Primero se realiza una breve presentación, y a continuación una descripción detallada de las mismas.

Para representar el tablero se ha definido el tipo tablero, que consiste en un registro que contiene una matriz de  $N \times M$  de casillas, el número de casillas ocupadas y cuál de los jugadores posee el turno. Cada casilla puede estar vacía, ocupada por una ficha del usuario, o por una ficha del ordenador. Por último, el tipo de datos acción recoge las posibles acciones del usuario.

## Constantes

Nombre	Descripción	Valor por defecto
N	Número de filas que tiene el tablero	3
M	Número de columnas que tiene el tablero	3

## Tipos

Nombre <sup>1</sup>	Tipo	Descripción	Definición
Estado	enumerado	Posibles estados en los que se puede encontrar la partida	INICIO_PARTIDA, JUGANDO, PAUSADA FIN_PARTIDA
Accion	enumerado	Diferentes acciones que puede realizar el usuario	JUGAR, SALVAR, CARGAR, TERMINAR, CONTINUAR, PAUSAR
Casilla	enumerado	Posibles valores que puede almacenar una casilla del tablero. VACIA = si la casilla está libre; USUARIO = si la casilla contiene una ficha del usuario; ORDENADOR = ficha del ordenador	VACIA, USUARIO, ORDENADOR
Matriz	array	Tabla que contiene las casillas del tablero.	[1..N, 1..M] de Casilla
Tablero	estructura	Estructura que define el tablero y el estado del mismo. - casillas = Tablero de NxM casillas; - ocupadas = Número de casillas ocupadas en cada momento, es decir, casilla <> VACIA; - turno: VERDADERO si juega el ordenador y FALSO si juega el usuario	casillas: Matriz ; ocupadas: Entero turno : Booleano

<sup>1</sup> Todos los nuevos tipos de datos definidos se nombran empezando por mayúscula. Las constantes se escriben con todas las letras en mayúsculas.

## Pseudocódigo

Entendemos por pseudocódigo un lenguaje semiformal que facilita la descripción algorítmica de un programa. El pseudocódigo se entiende como lenguaje técnico, así que se espera que el lector posea conocimientos de programación (ej. analistas, desarrolladores). En este sentido, obedece a unas reglas mínimas conocidas por la comunidad. Ahora bien, no es necesaria emplear una validación formal, como puede ser el caso de un lenguaje de programación de alto nivel (ej. C ó Pascal), si no que se permite cierta flexibilidad a la hora de expresarse. Esto es así, en tanto en cuanto que el interprete del código no es una máquina si no un humano.

Otra ventaja del pseudocódigo es que permite independizar la descripción del programa de los lenguajes de programación. Un mismo pseudocódigo debería poder ser implementado mediante varios lenguajes. La flexibilidad que aporta el pseudocódigo permite al programador realizar una descripción bastante ajustada del diseño del programa, sin tener que ocuparse de los detalles de implementación. De echo, un mismo algoritmo se puede describir empleando distintos niveles de detalle. Así, se hace especialmente conveniente como herramienta de diseño descendente.

## Un posible pseudocódigo

Existen múltiples propuestas de pseudocódigo. Cualquiera que consiga expresar un programa de manera precisa nos valdría. Hemos elegido una propuesta en concreta basada en el libro de Cormen et al <sup>2</sup> que nos sirva de modelo para las explicaciones tanto en teoría como en prácticas. Ahora bien, cuando tengáis que emplear una descripción en pseudocódigo no es obligatorio seguir esta propuesta, sino que podéis, si os sentéis más cómodos, elegir otra que fuera igual de válida.

En el siguiente apartado se indican una serie de convenciones mínimas que definen la propuesta de pseudocódigo. Es posible que, en determinados programas, sea necesario expandir la propuesta para poder expresarse correctamente.

En el siguiente apartado se incluyen diversos ejemplos en los que se muestra el modo de empleo y la potencialidad del mismo.

## Convenciones del pseudocódigo

1. La indentación (o sangrado) se emplea para señalar el comienzo y fin de un bloque (véanse ejemplos). Esta regla también se aplica a la sentencia condicional **si...si no**.

---

<sup>2</sup> T. H. Cormen, C.L. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms. Second Edition. The MIT Press. 2003, pp. 19-20

2. Las sentencias de repetición **desde...hasta**, **mientras** y **haz...mientras**, y las estructura condicionales **si...si no** y **segun** tienen una interpretación similar a la que se da en C a las estructuras for, while, do-while, if-then-else, case (véanse ejemplos)
3. La asignación de un valor a una variable se realiza mediante el símbolo ' $\leftarrow$ '.  
Ej.  $i \leftarrow 30$   
Se puede asignar un mismo valor a varias variables en una única línea concatenando sucesivas asignaciones:  
Ej.  $i \leftarrow j \leftarrow 30$   
Asigna 30 tanto a i como a j.
4. Las variables que se empleen siempre se entiende locales al algoritmo que se esté describiendo. Las variables globales deben ser declaradas explícitamente (*global edad*), aunque no se recomienda su uso a no ser que sea estrictamente necesario
5. Los elementos de una tabla (*array*) se acceden especificando el nombre de la tabla seguida de el índice del elemento entre paréntesis cuadrados []. Ej.  $A[i]$  indica acceder al i-ésimo elemento de la tabla A. Con la notación ".." se puede especificar un rango de valores dentro de una tabla. Ejemplo, para indicar la subtabla formada por j elementos  $A[1], A[2], A[3] \dots A[j]$  se emplearía  $A[1..j]$
6. Los campos de los registros (estructuras) se acceden mediante el carácter punto.
7. Los parámetros de una función se pasan **por valor**: la función llamada recibe su propia copia de los parámetros, y si se realiza una modificación en el valor del parámetro, éste no afecta a en la función que ha realizado la llamada. Cuando se pasa un registro (estructura), se copia el puntero a los datos, pero no los campos. De esta manera, si x es un parámetro de la función llamada, la asignación  $x \leftarrow y$  y dentro de la función llamada no repercute en la función que ha realizado la llamada. En cambio, la asignación  $A[x] \leftarrow 3$ , si que es visible.
8. Los operadores booleanos se evalúan en cortocircuito. Esto es, cuando se evalúa la expresión "a y b" primero se evalúa a. Si a se evalúa como FALSO, entonces es imposible que la expresión entera se evalúe como VERDADERO, así que no es necesario evaluar b. En cambio, si a se evalúa como VERDADERO, se debe evaluar b para determinar el valor de la expresión. De esta forma se puede especificar una expresión como  $a = \text{NIL}$  y  $f[a]=b$  sin tener que preocuparnos por  $A[a] = b$  cuando x es NIL.

### Ejemplo 1: Ordenación por inserción

En este ejemplo se plantea como describir el algoritmo de ordenación por inserción. Se han dado dos posibles soluciones al mismo algoritmo, cada una con un grado de detalle diferente. Finalmente se muestran tres implementaciones distintas para los pseudocódigos planteados (una en Pascal, y dos en C).

Ambas propuestas comparten las siguientes entradas, y tienen la siguiente salida esperada.

**Entrada:**

$A[1..N]$  secuencia de elementos  $(a_1, a_2, a_3, \dots, a_n)$

**Salida:**

Permutación  $(a'_1, a'_2, a'_3, \dots, a'_n)$  de la entrada tal que  $a'_1 \leq a'_2 \leq a'_3 \dots \leq a'_n$  (donde  $\leq$  es una relación de orden entre los elementos)

### Ejemplo 1.1: Pseudocódigo descripción muy general

**desde**  $i \leftarrow 2$  **hasta** longitud(A) :

    insertar  $A[i]$  en  $A[1..i]$  manteniéndola ordenada.

### Ejemplo 1.3: Pseudocódigo descripción detallada

**Entorno:**

$i, j$ : números enteros

$x$ : elemento auxiliar

**desde**  $i \leftarrow 2$  **hasta** longitud(A) :

$x \leftarrow A[i]$

$j \leftarrow i$

**mientras** ( $j > 0$  y  $x < A[j-1]$ ) :

$A[j] \leftarrow A[j-1]$

        decrementa  $j$ ;

$A[j] \leftarrow x$ ;

### Ejemplo 1.4: Implementación en Pascal del pseudocódigo propuesto

```
for i = 2 to N do
begin
  x := A[i];
  j := i;
  while (j > 0) and (x < A[j-1]) do
begin
  A[j] := A[j-1];
  j := j - 1;
end;
A[j] := x;
end
```

### Ejemplo 1.5: Dos implementaciones en C del pseudocódigo propuesto

<pre>for (i=1; i &lt; n; i++) {     x = A[i];     for (j=i; j &gt; 0 &amp;&amp; A[j-1] &gt; x; j--)         A[j] = A[j-1];     A[j] = x; }</pre>	<pre>for (i=1; i &lt; n; i++) {     x = A[i];     j = i;     while((j &gt; 0) &amp;&amp; (A[j-1] &gt; x))     {         A[j] = A[j-1];         j = j - 1;     }     A[j] = x; }</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Ejemplo 2: Comprobar si un número es primo

Pseudocódigo de una función que determina si un número es o no primo. Este ejemplo permite mostrar como se realizan *if* anidados.

**Entrada:** n, número natural > 0

**Salida:** verdadero si es primo, falso en caso contrario

**Entorno:** i, número entero. esPrimo, booleano

```

si n <= 3 :
    esPrimo ← verdadero
si no
    si n es par :
        esPrimo ← falso
    si no
        i ← 3
        mientras i <= n/2 y i no es divisor de n :
            i ← i+2
        si i es divisor de n :
            esPrimo ← falso
        si no
            esPrimo ← verdadero
devuelve esPrimo

```

### **Ejemplo 3: Diseño descendente (Programa Tres en Raya)**

Este ejemplo muestra como emplear el pseudocódigo para describir un programa mediante un diseño descendente. Primeramente se definen las estructuras y tipos de datos necesarios que se van a emplear. La parte algorítmica se ha dividido en tres: programa principal, función *juega*, y procedimiento *mueveOrdenador*. Cada una describe una parte del programa con un nivel de detalle diferente, desde lo más genérico a los más concreto. La composición de estas tres partes (junto con otras que no sean incluido) dan lugar al diseño completo del programa.

#### **Programa principal**

El programa principal consta de un bucle que muestra el menú principal y recoge la acción a realizar por el usuario:

**Entorno:** tablero: Tablero,  
accion: Accion.

```
imprimeMenu(INICIO_PARTIDA)
```

```
mientras lee(accion) <> TERMINAR :
```

```
  segun accion :
```

```
    INICIAR: estado ← juega(INICIO_PARTIDA, tablero)
```

```
    CONTINUAR: estado ← juega (JUGANDO, tablero)
```

```
    SALVAR: guarda(tablero)
```

```
    CARGAR: abre(tablero)
```

```
  imprimeMenu(estado)
```

Destacar que el procedimiento *imprimeMenu* saca por pantalla un menú que varía según el estado en que se encuentre la partida.

### **Función juega**

Parte que resuelve la secuenciación de movimientos. Esta función es llamada desde el programa principal cuando el usuario quiere iniciar una partida.

**Entrada:** estado: de la partida.  
tablero: con la configuración de las fichas actual.  
**Salida:** estado de la partida (FIN\_PARTIDA o PAUSADA).

```
si estado = INICIO_PARTIDA :
  inicializar(tablero)
  estado ← JUGANDO
haz
  si juegaOrdenador(tablero.turno) :
    mueveOrdenador(tablero)
    si esTresenraya(tablero, ORDENADOR) :
      estado ← FIN_PARTIDA
      imprimeVictoria(ORDENADOR)
  si no
    lee(accion)
    si (accion = PAUSAR):
      estado ← PAUSADA
    si no
      mueveUsuario(accion, tablero)
      si esTresenraya(tablero,USUARIO) :
        estado ← FIN_PARTIDA
        imprimeVictoria(USUARIO)

  si estado = JUGANDO
    si sonTablas(tablero) :
      estado ← FIN_PARTIDA
      imprimeTablas
    si no
      cambia(tablero.turno)
mientras (estado <> FIN_PARTIDA y estado<>PAUSADA)
```

### Procedimiento mueveOrdenador

Este procedimiento se ha elegido como ejemplo de procedimiento que se emplea en la función *juega*. Faltaría describir alguna otra de las funciones relevantes del procedimiento juego (esTresenraya, sonTablas...) para completar el diseño.

**Entrada:** tablero con la configuración de las fichas actual y con al menos una casilla vacía.

**Salida:** mismo tablero con una nueva ficha colocada.

**haz**

$i \leftarrow \text{aleatorio}(1, N)$

$j \leftarrow \text{aleatorio}(1, N)$

**mientras** (tablero.casillas[i,j] <> VACIA)

tablero.casillas[i,j]  $\leftarrow$  ORDENADOR

tablero.ocupadas  $\leftarrow$  tablero.ocupadas + 1