

Programación estructurada.
Estructuras básicas.
Funciones y procedimientos

TEMA 25 (27 SAI)

ABACUS NT

Índice

- 1. Introducción.**
- 2. Programación estructurada.**
 - 2.1. Fundamentos.**
- 3. Teorema del programa estructurado**
- 4. Estructuras básicas.**
 - 4.1. El flujo de control de un programa**
 - 4.2. Estructura condicionales o alternativas.**
 - 4.3. Estructuras iterativas.**
- 5. Funciones y procedimientos.**
 - 5.1. Funciones.**
 - 5.2. Procedimientos**
 - 5.3. Parámetros Formales y Actuales**
 - 5.3.1. Paso por valor**
 - 5.3.2. Paso por referencia**
 - 5.3.3. Ámbito de un identificador**
- 6. Diseño de funciones y procedimientos.**
 - 6.1. Diseño estructurado.**
 - 6.2. Atributos de la calidad de un diseño.**
- 7. Técnicas de Diseño de Algoritmos**
- 8. Los lenguajes de programación actuales y su tendencia**
- 9. Conclusión**
 - 9.1. Relación del tema con el sistema educativo actual**
- 10. Bibliografía**

1. Introducción.

Con anterioridad a los años 60, los programas consistían en un conjunto de instrucciones sin estructura ninguna, lo que dificultaba enormemente su mantenimiento. Para solucionar esa problemática, a finales de los años sesenta, Dijkstra, junto con otros autores, como Chapin, o Warnier, desarrollan los principios de la programación estructurada, estableciendo que un **programa bien estructurado** debía cumplir con los siguientes principios:

- El código debe ser entendible sin necesidad de información adicional.
- El programa podrá ser desarrollado por partes fácilmente ensamblables.
- Las diferentes partes del programa pueden ser modificadas o cambiadas, sin que esto afecte al resto del programa.
- Los programas deben ser escritos utilizando las estructuras básicas: secuencial, alternativa y repetitiva.

En concreto, se buscaban metodologías de programación que permitiesen construir programas que fuesen más fáciles de mantener, de actualizar, de comprender.

Surgen así dos técnicas de programación:

La programación estructurada: La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora recurriendo únicamente a subrutinas y tres estructuras básicas: secuencia, selección e iteración

La programación modular: es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

La programación estructurada es un criterio complementario a la programación modular: la programación modular tiende a dividir un programa en partes más pequeñas, llamadas módulos y la estructurada se encarga de desarrollar estructuradamente cada una de esas partes.

2. Programación estructurada.

2.1. Fundamentos.

A finales de los años 1970 surgió una nueva forma de programar que no solamente daba lugar a programas fiables y eficientes, sino que además estaban escritos de manera que facilitaba su mejor comprensión, no sólo proveyendo ventajas durante la fase de desarrollo, sino también posibilitando una más sencilla modificación posterior.

A este respecto, Böhm y Jacopini demostraron en 1966 con el **teorema del programa estructurado** que un programa puede escribirse únicamente con estas tres estructuras básicas, sin necesidad de usar el **salto incondicional** (instrucción GOTO).

La programación estructurada aporta las siguientes ventajas respecto a las **técnicas de programación anteriores**:

- Los programas son más **fáciles de entender**.
- Reducción del esfuerzo en las **pruebas**.
- Reducción de los costes de **mantenimiento**
- Aumento de la productividad del programador.

3. Teorema del programa estructurado

La teoría de lenguajes de programación tiene varios hitos en su desarrollo, el presente teorema es uno de ellos. Su definición más acertada se centra en el trabajo de 1966 de Corrado Böhm y Giuseppe Jacopini, quienes plantean que **toda función computable puede ser implementada** solo tres tipos de estructuras de programación:

- Las instrucciones son o una instrucción simple o una **secuencia ordenada** de instrucciones (estructura secuencial).
- **Según** el resultado de una expresión lógica se ejecuta una instrucción u otra (estructura condicional o alternativa).
- **Mientras** una condición se mantenga verdadera ejecutar una instrucción (estructura iterativa, cíclica o de bucle).

La demostración se realizó sobre las los diagramas de flujo y P, el lenguaje diseñado por Böhm.

Implicaciones

Del Teorema del Programa Estructurado se establece que todos los algoritmos que se pueden desarrollar en una computadora pueden escribirse de siguiendo lo que luego se conocería como uno de los paradigmas dentro de la programación imperativa, la programación estructurada y con él se demostraba la capacidad expresiva de ésta para el desarrollo de soluciones computacionales.

En su demostración se probaba que todo diagrama de flujo puede transcribirse a uno estructurado y se establecía un método formal para realizar la conversión, aunque después el programador pudiera refinara.

También daba una idea de las tres estructuras que serían indispensables a la hora de diseñar un lenguaje estructurado: **secuencial, condicional y cíclica**; aunque es evidente

que puede extenderse a otras estructuras derivadas de estas que enriquezcan la expresión semántica y sintáctica del lenguaje en cuestión.

Lo que sí el artículo donde aparece el teorema no deja claro es cuándo es más pertinente el uso de la programación estructurada por sobre otros paradigmas.

Y como dato interesante, después de esta formalización se generó el fuerte debate sobre la existencia o no de la **instrucción de salto GOTO**, que ha llegado hasta nuestros días; sin embargo, existe el consenso acerca de que la misma no es necesaria en los lenguajes de alto nivel y que su presencia solo entorpece la legibilidad y elegancia del código, aunque todavía aparece la implementación de GOTO en muchos lenguajes modernos. El debate comenzó con Edsger Dijkstra quien escribió "La sentencia GoTo considerada dañina" (1968), una carta considerada hoy un clásico, como la mayoría de los trabajos del científico.

4. Estructuras básicas.

A continuación, vamos a desarrollar cada una de las estructuras propuestas por el **teorema del programa estructurado**, vamos a ver su correspondiente pseudocódigo y ordinograma:

4.1. El flujo de control de un programa

La expresión flujo de control hace referencia al orden en el que se ejecutarán las instrucciones de un programa, desde su comienzo hasta que finaliza. El flujo normal de ejecución es el secuencial. Si no se especifica lo contrario, la ejecución de un programa empezaría por la primera instrucción e iría procesando una a una en el orden en que aparecen, hasta llegar a la última, de manera **incondicional**:

Pseudocódigo	Ordinograma
<pre>Acción 1; Acción 2; Acción 3;</pre>	<pre> graph TD Start(()) --> Accion1[Acción 1] Accion1 --> Accion2[Acción 2] Accion2 --> Accion3[Acción 3] Accion3 --> End(()) </pre>

Algunos programas muy simples pueden escribirse sólo con este flujo unidireccional. No obstante, la mayor eficacia y utilidad de cualquier lenguaje de programación se deriva de la posibilidad de cambiar el orden de ejecución según la necesidad de elegir uno de entre varios caminos en función de ciertas condiciones, o de ejecutar algo repetidas veces, sin tener que escribir el código para cada vez.

4.2. Estructura condicionales o alternativas.

Permiten la **ejecución o no** de un grupo de instrucciones dependiendo de si se cumple o no una determinada **condición**. Existen **tres tipos** de instrucciones alternativas:

Alternativa simple

El clásico “if” que permite ejecutar o no una acción o bloques de acciones dependiendo de si se cumple o no, una determinada condición.

Pseudocódigo	Ordinograma
<pre>Si (Condición) entonces Acción 1 Fin si</pre>	<pre> graph TD Start(()) --> Cond{Condición} Cond -- Si --> Acc1[Acción 1] Acc1 --> Cond Cond -- No --> A((A)) </pre>

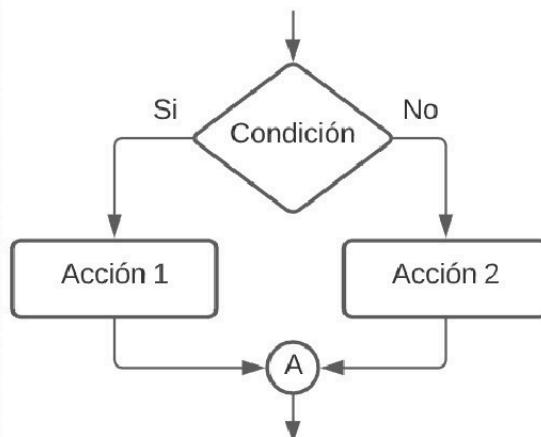
Alternativa doble

Al caso anterior, se le añade una nueva acción **si no** se cumple la condición.

Pseudocódigo	Ordinograma
--------------	-------------

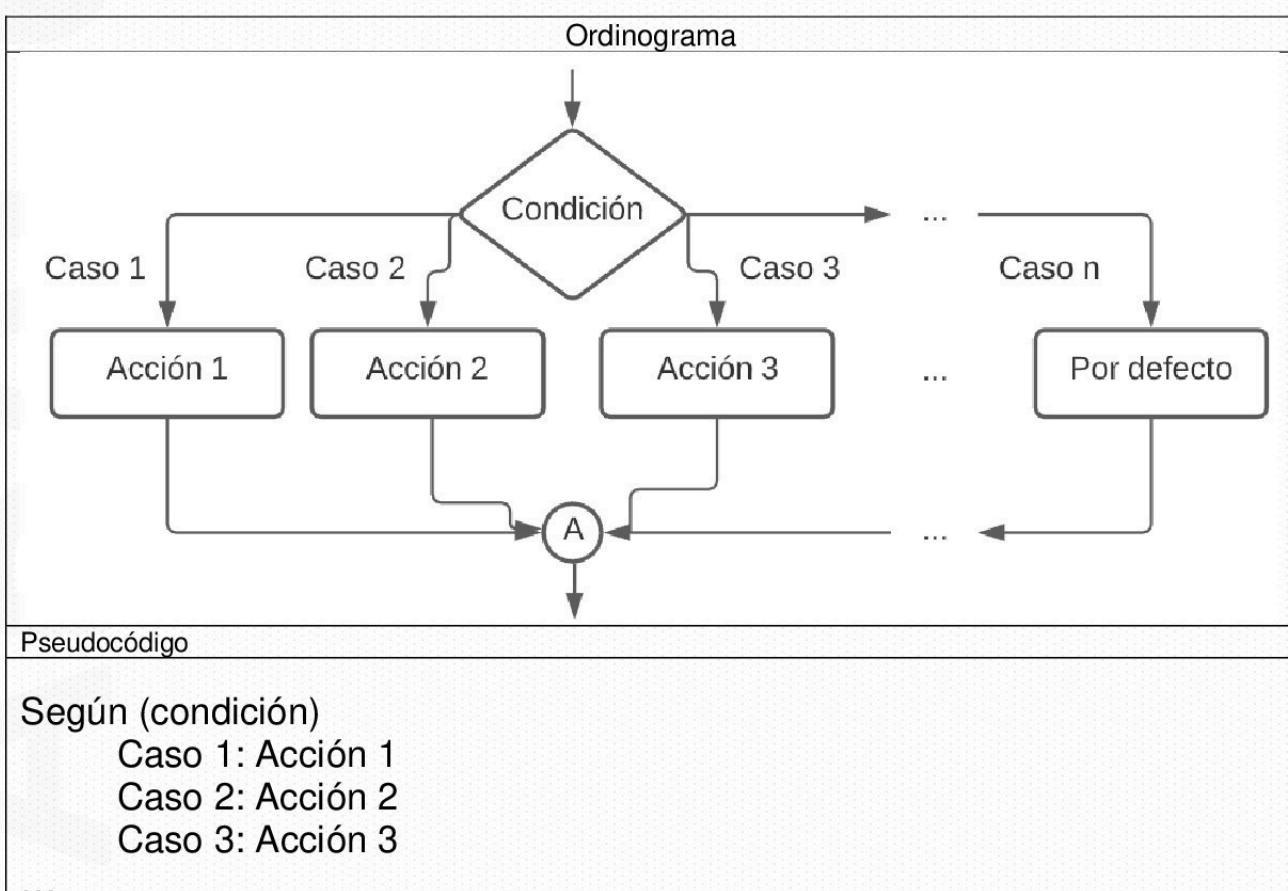
```

Si (condición) entonces
    Acción 1
Sino
    Acción 2
Fin si
  
```



Alternativa múltiple

Cuando las opciones que puede tomar la condición son múltiples, podemos anidar tantas estructuras si-sino como sean necesarias. Una mejor opción en aquellos lenguajes que lo permiten es implementar una estructura múltiple (conocida como switch o case) que permita seleccionar fácilmente una acción dependiendo del valor que tome la condición.



Por defecto:
Acción n
Fin segün

4.3. Estructuras iterativas.

Permiten la ejecución repetida de un grupo de instrucciones un número determinado de veces o hasta que se cumpla una determinada condición. Existen tres tipos:

Bucle Mientras

La instrucción mientras comprueba una condición y **mientras** se cumpla se ejecutará la acción (o bloque de acciones)

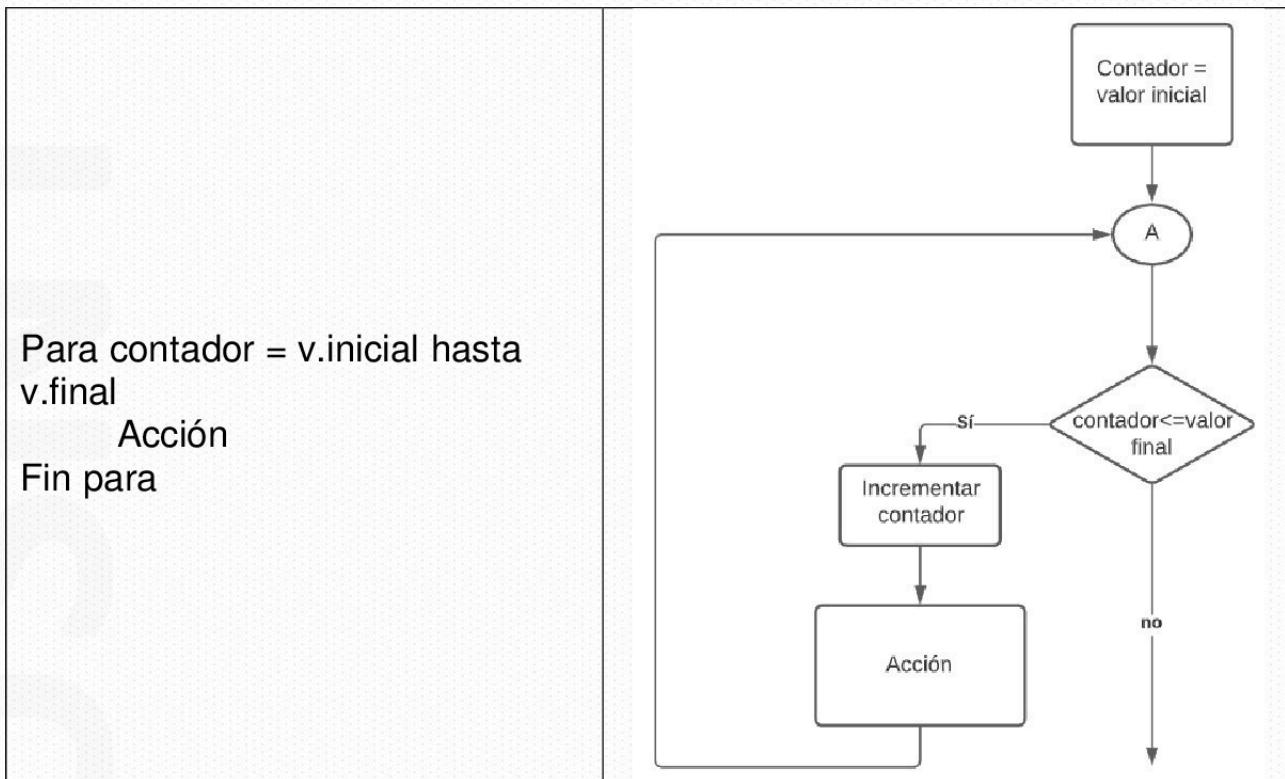
Pseudocódigo	Ordinograma
<pre>Mientras (Condición) Acción Fin Mientras</pre>	<pre> graph TD A((A)) --> Accion[Acción] Accion --> A Accion -- SI --> Condicion{Condición} Condicion -- no --> Fin </pre>

Bucle Para

El bucle para (*for* en todos los lenguajes) repite para n valores de la variable, comenzando por uno valor inicial, hasta un valor final e incrementando la variable. La variable puede ser incrementada, o decrementada en cualquier valor que se determine.

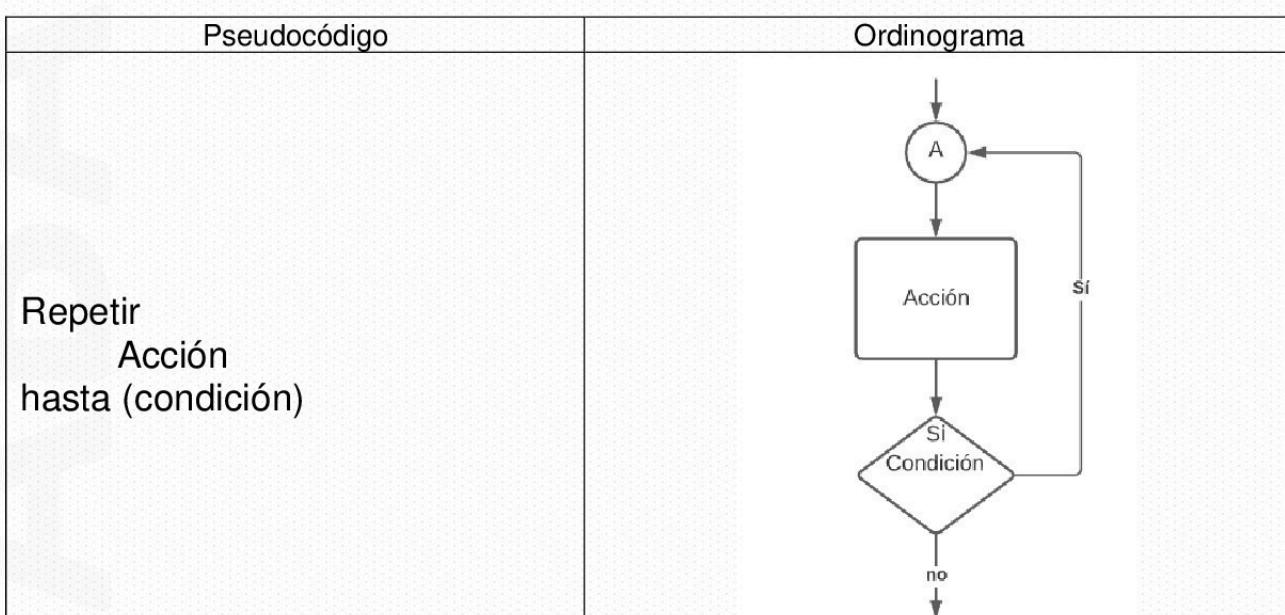
Este bucle en algunos lenguajes se utiliza no solo con variables numéricas, sino también con valores de rangos, conjuntos o incluso para recorrer las entradas de un directorio.

Pseudocódigo	Ordinograma
--------------	-------------



Repetir-hasta

El bucle **repetir-hasta** se diferencia del *mientras*, principalmente, en que ejecutará la acción al menos una vez hasta de comprobar la condición.



5. Funciones y procedimientos.

Al igual que la programación estructurada, las funciones y procedimientos surgieron con la idea de mejorar el mantenimiento de los programas, y aumentar la productividad de los programadores. Son utilizados para evitar la duplicidad de código y facilitar la legibilidad del mismo.

Un método para solucionar un problema complejo es dividir el problema principal en subproblemas, bajo el lema atribuido a Julio César, ***divide et vincis***.

Si los subproblemas son susceptibles de ser divididos de nuevo se repite el proceso hasta encontrar soluciones sencillas.

Este método también se conoce como **diseño top-down**, y consiste en ver un problema complejo como la suma de múltiples problemas más simples.

Normalmente las partes que componen dicho problema pueden implementarse independientemente mediante funciones o procedimientos.

5.1. Funciones.

Una función es un subprograma que recibiendo uno o más datos de entrada (parámetros) devuelve un único resultado que irá asociado al nombre de la función.

Una función necesita ser definida y declarada una vez y puede ser llamada tantas veces como se desee.

Existen 2 tipos de funciones:

Las que vienen definidas por el lenguaje de programación por ejemplo funciones matemáticas de cadenas de caracteres. Vienen incorporadas en el propio lenguaje (en librerías normalmente) y el usuario puede utilizarlas a través de llamadas.

Las definidas por el usuario, siendo éste el responsable de declararlas, definirlas e incluso crear sus propias librerías. Una vez creadas, cuando las necesite, sólo tiene que llamarlas.

La declaración de una función es lo primero que hay que realizar antes de definir y utilizar una función. Aquí se indica el nombre de la función, el tipo de valor returnedo y el número y tipo de los parámetros que utilizará. Los parámetros son variables que permite recibir o entregar valores entre el programa que hace la llamada y la función.

Los prototipos sirven para que el compilador del lenguaje pueda chequear si los parámetros y el valor de retorno de la función son los adecuados.

La definición de una función consta de dos partes:

Una **cabecera** formada por el tipo de dato que devuelve la función, el nombre de ésta y los parámetros (formales) de dicha función.

El **cuerpo** de la función: donde se definen las variables locales y las acciones o instrucciones a realizar.

La definición de una función tendrá el siguiente formato:

```
<tipo del resultado>función nombre_funcion (parametro1,  
parametro2, etc)  
[declaraciones locales]  
Inicio  
    <acciones> //cuerpo de la función  
fin
```

Donde:

Tipo de resultado es el tipo que devuelve la función

nombre_funcion es el nombre asociado a la función

parametro1 al parámetro N es la lista de parámetros formales o argumentos. Siempre es necesario reflejar el tipo de cada parámetro.

<Acciones> son las instrucciones que constituyen la definición de la función y que debe de contener una acción que asigne un valor al nombre de la función.

Los argumentos de la declaración de la función se denominan parámetros formales: son nombres de variables, nombres de otras funciones o procedimientos, que sólo se utilizan dentro del cuerpo de la función.

La llamada a una función se realiza a través de su nombre, seguido de los argumentos que se quieran introducir como datos de entrada.

```
nombre_funcion (lista de parámetros actuales)
```

Dónde:

Nombre de función es la función que llama

Lista de **parámetros** actuales son las constantes, variables, expresiones, valores de funciones, nombres de funciones o procedimientos.

Los argumentos utilizados en la llamada a la función se denominan parámetros actuales, pueden ser constantes, variables, valores de funciones o nombres de funciones o procedimientos.

La llamada a una función puede hacerse desde el programa principal o desde otro módulo del programa, pero siempre dentro de una expresión de la misma forma irá el dato que proporciona la función como resultado.

Cada vez que se llama a una función, se establece automáticamente una correspondencia entre los parámetros formales y los parámetros actuales. Es decir, se emparejan según su posición en la lista. Esto requiere que ambas listas tengan el mismo número de parámetros y que coincidan en el tipo.

Para realizar las acciones, pueden ser necesarias variables intermedias, las cuales serán definidas dentro de las acciones se las llamará variables locales. Estas variables sólo se utilizan en el ámbito de la función y no tienen significado en el programa principal.

5.2. Procedimientos

La estructura de un procedimiento es similar a la de una función, pero existen varias diferencias entre ellos:

- Los procedimientos pueden devolver más de un valor como resultado, o incluso ninguno, mientras que en las funciones siempre se devolvía uno.
- La devolución de resultados no se hace a través del nombre de procedimiento sino a través de los parámetros
- La llamada a un procedimiento se hace desde una instrucción concreta, no desde una expresión.

Tanto la entrada de información al procedimiento, como la devolución de los resultados desde el programa llamador, se realizarán a través de los parámetros.

La declaración de un procedimiento es similar a la de una función; las pequeñas diferencias existentes son debidas a que el nombre del procedimiento no se encuentra asociado a ningún tipo.

La declaración de un procedimiento será del tipo:

```
Procedimiento nombre_procedimiento (par1,  
par2,par3...parn)  
[declaraciones_locales]  
    Inicio  
    <acciones>  
    Fin
```

donde

nombre procedimiento es el nombre asociado con el procedimiento.

par1, hasta parn es la lista de parámetros formales o argumentos. Debe de reflejarse siempre el tipo de cada parámetro formal.

<Acciones> son las instrucciones que constituyen la definición del procedimiento.

En los procedimientos también podremos utilizar variables locales.

La llamada a un procedimiento tiene el siguiente formato:

[llamar a] Nombre Procedimiento (par1, par2, ...parn)

Donde debe existir la misma relación entre los parámetros formales y los actuales que en las funciones.

La palabra *llamar a* (call) es opcional y su existencia depende del lenguaje de programación.

5.3. Parámetros Formales y Actuales

Como hemos explicado anteriormente, se establece una correspondencia automática entre los parámetros formales y actuales cada vez que se llama a la función. Los parámetros actuales sustituirán y serán utilizados en lugar de los parámetros formales.

Existen dos métodos para establecer el paso entre parámetros: paso por valor y paso por referencia.

5.3.1. Paso por valor

Los parámetros formales correspondientes reciben una copia de los valores de los parámetros actuales, por tanto, los cambios que se produzcan en ellos por efecto del subprograma no podrán afectar a los parámetros actuales y no se devolverá información al programa llamador.

Se utiliza este método cuando se necesita transmitir información desde el programa llamador al subprograma (parámetros de entrada)

5.3.2. Paso por referencia

Lo que se pasa al subprograma es la dirección de memoria del parámetro actual. De esta forma, la variable pasada como parámetro actual es compartida, es decir, se puede modificar directamente por el subprograma.

Será el método adecuado cuando se necesite que:

- El subprograma reciba datos y los devuelva modificados (parámetros de entrada y de salida)
- Devolver resultados desde el subprograma al programa llamador (parámetros de salida, muy utilizados en los procedimientos)
- Se van a enviar como datos de entrada, estructuras que ocupan mucho espacio, por lo que es más aconsejable dar su dirección de memoria que enviar una copia del mismo.

La diferencia entre pasó por valor y paso por referencia se puede simplificar en lo siguiente:

En el paso por valor, los parámetros formales ocupan una zona de memoria distinta a los de los parámetros actuales.

En el caso de paso por referencia, ambos parámetros ocupan la misma dirección de memoria.

5.3.3. Ámbito de un identificador

Se podría definir el ámbito de un identificador, ya sea variable, función procedimiento, etc como la parte del programa desde donde éste puede ser identificado porque se conoce su existencia.

Atendiendo a este aspecto pueden dividirse en:

Locales: son las definidas dentro de un subprograma su ámbito es dicho subprograma (y sus subprogramas anidados), no siendo accesible fuera de él, incluso se podría definir una variable con el mismo nombre fuera del subprograma.

Globales: están definidas en el programa principal, por lo que su ámbito es todo el programa, incluidos los subprogramas, salvo que estos tengan variables con igual nombre, en cuyo caso tienen prioridad las segundas.

Con ambos tipos de variables pueden producirse los llamados efectos laterales. Esto sucede cuando un subprograma modifica una variable situada fuera de él, que no ha sido enviada como parámetro. Suele ocurrir al utilizar variables globales o locales en subprogramas anidados, y al causar efectos difícilmente controlables, se tiende a evitarlos.

6. Diseño de funciones y procedimientos.

6.1. Diseño estructurado.

El objetivo principal del diseño estructurado es describir la estructura del programa, así como las relaciones entre los elementos, denominadas módulos, que componen esta estructura. Para ello, **en Métrica v.3**, se propone el uso de diagramas de **cuadros de Constantine** o **diagramas de estructura**.

Estos diagramas cumplen las siguientes características:

- Los módulos son representados por **rectángulos**, y su nombre representa a la función que realiza.
- Las conexiones entre módulos son representadas mediante **flechas**
- Los módulos pueden **comunicarse** entre sí por medio de estructuras de datos o de control (flags)
- Un **módulo “predefinido”** es aquel que se encuentra disponible en la biblioteca del sistema o de la aplicación, y es representado con un rectángulo dentro de otro.

Actualmente (año 2020) con Metrica V3 aún utilizada por algunas empresas, las metodologías **Agile**, **XP** o **Scrum** hacen uso de representaciones similares.

6.2. Atributos de la calidad de un diseño.

Para medir la **calidad estructural** de un diseño se utilizan las siguientes métricas principalmente, las cuales son contrapuestas:

- **Acoplamiento:** Grado de interdependencia entre los módulos. Un buen diseño minimiza el acoplamiento al máximo.
- **Cohesión:** Mide la relación que existe entre los elementos de un mismo módulo, es decir, las relaciones intramodulares. Por el contrario, mientras mayor es la cohesión de un módulo, mejor está diseñado. Los autores Stevens, Myers y Constantine desarrollaron una escala para medir la cohesión, cuyos valores ordenados de mayor a menor van desde la cohesión funcional hasta la coincidental.

7. Técnicas de Diseño de Algoritmos

A la hora de resolver un problema, se nos plantean distintas formas de abordarlo para elaborar el algoritmo que lo solucione, lo que ha dado lugar a distintas técnicas:

Divide y Vencerás

Los problemas complejos se pueden enfrentar más eficazmente si los descomponemos en subproblemas que sean más sencillos de solucionar. En programación este método se llama **“Divide y Vencerás”** (*Divide et Vincere*, frase célebre de Julio César) y consiste en dividir un problema complejo en otros más simples. La subdivisión sucesiva de los subproblemas en otros aún más simples da lugar al **diseño Top-Down**.

Diseño Top-Down (Diseño Descendente)

Este diseño pasa por una serie de descomposiciones sucesivas del problema inicial, que va recibiendo un refinamiento progresivo.

La utilización de la técnica de diseño Top-Down tiene los siguientes objetivos básicos:

- Simplificación del problema y de los subprogramas de cada descomposición.
- Las diferentes partes del problema pueden ser programadas de modo independiente e incluso por diferentes personas.
- El programa final queda estructurado en forma de bloques o módulos, lo que hace más sencilla su lectura y mantenimiento.

Bottom-Up (Diseño Ascendente)

El diseño ascendente se refiere a la identificación de aquellos procesos que necesitan implementarse conforme vayan apareciendo, sin una planificación previa.

Cuando la programación se realiza internamente y haciendo un enfoque ascendente, es difícil llegar a integrar los subsistemas a un nivel tal que el desempeño global sea fluido.

Los problemas de integración entre subsistemas son sumamente costosos y muchos de ellos no se solucionan hasta que la programación alcanza la fecha límite para la integración, con lo cual se entregan sin depurar.

Aunque cada subsistema parece cumplir los requisitos, cuando se contempla el sistema en conjunto, adolece de ciertas limitaciones por haber tomado un enfoque ascendente:

- La accesibilidad del software es deficiente
- Se requieren datos superfluos e innecesarios.
- No se satisfacen los objetivos globales de la organización

Recursividad

La recursividad es una técnica de programación que consiste en resolver el caso más básico del problema mediante la resolución de uno o más casos base y llamadas sucesivas a la propia función que lo resuelve.

Normalmente es una técnica que es muy sencilla de programar, aunque hay que tener en cuenta que la cantidad de recursos que utiliza, ya que pueden desbordar la capacidad el equipo rápidamente.

Backtracking

Son algoritmos que utilizan tanto la técnica de divide y vencerás como la recursividad. Consiste en realizar una búsqueda sistemática por el espacio de soluciones hasta probar todas las opciones o demostrar que no hay solución. Para esto se separa la búsqueda en varias búsquedas parciales de naturaleza recursiva.

El método consiste en generar candidatos de soluciones posibles de acuerdo a un patrón dado; luego los candidatos son sometidos a pruebas de acuerdo a un criterio que caracteriza a la solución. Si un candidato no es aceptado, se genera otro; y los pasos dados con el candidato anterior no se consideran. Es decir, existe inherentemente una vuelta atrás, para comenzar a generar un nuevo candidato; por esta razón, este tipo de algoritmo también se denomina "con vuelta atrás"

Estos algoritmos recorren siempre un árbol o un grafo de soluciones, que si no existe se irá creando.

Ramifica y Acota

Es una mejora del backtracking que consiste en aplicar ciertas restricciones de tal forma que se puedan “podar” ramas del árbol de soluciones.

Resuelve el problema clásico de los movimientos del caballo en ajedrez o incluso un sudoku.

Algoritmos voraces

Funcionan en fases, en cada una de ellas se toma una decisión que es la mejor según los datos con que contamos en ese momento, pero sin considerar consecuencias futuras, por lo que no siempre se obtendrá la solución óptima, pero sí una buena solución.

Se basan en el uso de un óptimo local (Dijkstra) esperando que también el global sea óptimo. Un ejemplo es el camino más corto en un grafo.

Algoritmos heurísticos

Una heurística es una técnica diseñada para resolver un problema más rápidamente cuando los métodos clásicos son demasiado lentos, o para encontrar una solución aproximada cuando los métodos clásicos no logran encontrar una solución exacta.

8. Los lenguajes de programación actuales y su tendencia

Según la publicación “**2019 The top Programming Languages**” del IEEE Spectrum (el espectro del Instituto de ingeniería eléctrica y electrónica), de los diez lenguajes con más importancia en el sector productivo, 9 son multiparadigma y/o orientados a objetos, lo que muestra la gran importancia de la P.O.O actualmente. Estos lenguajes son:

1. **Python**: Desarrollado por Guido Van Rossum que liberó su código en 1991 bajo licencia Python License originalmente y GNU GPL en la actualidad, es un lenguaje de programación interpretado, multiplataforma, que soporta P.O.O, programación imperativa y funcional. Hace hincapié en la legibilidad del código, utilizando un tipado dinámico y conteo para el administrador de memoria. Debe su nombre a la afición de su desarrollador por los Monthly Python.
2. **Java**: Fue desarrollado en 1985 y está enfocado hacia la red, debido al auge de Internet en los años 90. Es independiente del ordenador donde se ejecute y para conseguir esta independencia utiliza una máquina virtual java (JVM) que nos permite utilizar código compilado en otra computadora distinta. Entre sus características destacamos:
 - Sintaxis similar a C++
 - Dispone de un recolector de basura
 - Elimina el uso de punteros, utilizando referencias

- Soporta programación multihilo
 - Dispone de mecanismos de detección de errores
 - Tiene amplias librerías de clases
3. **C:** Desarrollado en 1972 a partir de otro lenguaje llamado B, con la idea de conseguir un lenguaje de alto nivel, pero con posibilidad de acceso a bajo nivel. Esta propiedad hace que este lenguaje sea muy adecuado para la programación de sistemas.

Es adecuado para programadores expertos porque hace programas muy eficientes, por el contrario, su depuración puede ser bastante compleja. Tiene las siguientes características:

- Es débilmente tipado
 - Impone pocas restricciones al programador
 - Dispone de operadores a nivel de bit
 - Soporta el uso de punteros
 - Utiliza librerías
4. **C++:** Fue inventado en los laboratorios de AT&T en 1980. Inicialmente fue una gran mejora de C. Actualmente tiene un estándar, denominado ISO C++.

C++ soporta programación orientada a objetos incluyendo características como:

- Implementación de clases
 - Herencia múltiple
 - Acceso a los atributos y métodos
5. **R:** Es un entorno y lenguaje de programación con un enfoque al análisis estadístico.

Se trata de uno de los lenguajes de programación más utilizados en investigación científica, junto con Matlab, siendo además muy popular en los campos de aprendizaje automático (machine learning), minería de datos, investigación biomédica, bioinformática y matemáticas financieras. R es orientado a objetos, se puede integrar con bases de datos, tiene numerosas funciones de cálculo y funciones gráficas compatibles con LaTeX.

6. **JavaScript:**

Es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Se utiliza principalmente en su forma del lado del cliente (client-side), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web

JavaScript se diseñó con una sintaxis similar a C, aunque adopta nombres y convenciones del lenguaje de programación Java. Sin embargo, Java y JavaScript tienen semánticas y propósitos diferentes.

Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. Para interactuar con una página web se provee al lenguaje JavaScript de una implementación del Document Object Model (DOM).

Actualmente es ampliamente utilizado para enviar y recibir información del servidor junto con ayuda de otras tecnologías como AJAX.

7. C#:

Pronunciado C sharp, es un lenguaje de programación multiparadigma desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, de los lenguajes de programación diseñados para la infraestructura de lenguaje común (CLI).

Su sintaxis básica deriva de C/C++, utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.

8. Matlab:

Matrix Laboratory es un sistema de cómputo numérico que ofrece un IDE y lenguaje de programación llamado M que es interpretado y orientado a objetos. Este lenguaje permite operaciones de vectores y matrices, funciones, cálculo lambda y representación de gráficos 2D y 3D.

9. Swift

Swift es un lenguaje de programación **multiparadigma** creado por **Apple** enfocado en el desarrollo de **aplicaciones para iOS y macOS**. Fue presentado en la WWDC 2014 y puede usar cualquier biblioteca programada en Objective-C y llamar a funciones de C.

10. Go

Desarrollado por Google en 2009, la concurrencia, la sencillez, tipado estático, y el no utilizar excepciones son las principales características de Go. Es el lenguaje de programación de sistemas multiproceso más eficaz existente en la actualidad, inspirado en la sintaxis de C. Soporta orientación a objetos parcialmente, ya que no implementa mecanismos de herencia y las clases son declaradas como componentes separados de Interfaces y estructuras.

9. Conclusión

Los fundamentos de la programación estructurada siguen vigentes hoy en día, ya que su aplicación facilita la utilización de otras técnicas de programación. Es por ello por lo que otras técnicas o paradigmas de programación, como la programación modular o la programación orientada a objetos no rompen con sus principios más básicos.

El teorema del programa estructurado, propuesto por **Böhm-Jacopini**, demuestra que todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:

- Secuencia
- Instrucción condicional.
- Iteración (bucle de instrucciones) con condición al principio.

Utilizando el diseño top-down basado en la técnica divide y vencerás, podemos estructurar el programa en funciones y procedimientos, resultando por tanto un código mucho más legible y por tanto más fácil de actualizar y mantener.

9.1. Relación del tema con el sistema educativo actual

Este tema puede ser desarrollado en los siguientes módulos formativos (para atribución docente de PES)

- Bachillerato – Tecnologías de la Información y la Comunicación II (PES)
- GS- GS – DAW – Desarrollo Web en Entorno Servidor DAW/DAM – Programación (PES)

Aunque los profesores de SAI no tienen una atribución docente en ningún módulo con aplicación directa del tema, siempre es probable que se diseñe algún pequeño programa en clase, para lo que recurrir a la notación de pseudocódigo o de ordinograma sería un recurso esclarecedor.

10. Bibliografía

- Introduction to Java Programming and Data Structures, Comprehensive Version. Y. Daniel Liang. Pearson, 11^a edición. 2017.
- Java 9. Francisco Javier Moldes Teo. Anaya. 2017.
- Introducción a la computación. J, Glenn Brookshear, Pearson, 11^º edición. 2012
- Fundamentos de programación. Algoritmos, estructuras de datos y objetos. Luis Joyanes Aguilar, McGraw-Hill, 4^º edición. 2008.
- Langsam, Augenstein y Tanembaum: “Estructuras de Datos con C y C++”, Prentice-Hall 1997
- Prieto A., Lloris A. y Torres J.C.: Introducción a la Informática, 4^a ed (2006) McGraw-Hill
- Lenguajes de programación, principios y práctica. 2^a Ed (2004). Kenneth C.Louden