1. Caso Práctico

Unidad Didáctica V

Caso Práctico

*En una empresa de programación como **Sistemas Informáticos AndalucíaS. C. A.** (en adelante **SI Andalucía**), el término Programación Estructurada es bien conocido y utilizado. Más que un concepto se trata de un método de trabajo que garantiza una adecuada gestión de los proyectos. En un equipo de programación es especialmente útil, ya que se consigue una estructuración de las tareas que facilita considerablemente la distribución de tareas y la consecución de objetivos.*
Una empresa moderna no puede depender exclusivamente de sus trabajadores. Si, por cualquier motivo, se ve obligada a prescindir de uno de ellos, eso no debe resentir el proyecto al que estaba asignado. El modo como se trabaja debe garantizar la continuidad de un proyecto en estas situaciones y eso es lo que se consigue con la Programación Estructurada, en la que se basa el equipo de programación para establecer las pautas de trabajo para cada uno de sus miembros.

Unidad Didáctica V

Introducción

*Entre otras cosas la Programación Estructurada permite un fácil mantenimiento de los programas, así como actualizaciones a nuevas versiones. Y como hemos comentado anteriormente es imprescindible a la hora de que un equipo de programación se ponga a trabajar. En SI Andalucía, lo han tenido muy en cuenta a la hora de planificar el trabajo, aunque uno de sus miembros (**Víctor**) nunca lo había utilizado y como siempre era algo reacio. Ahora **Víctor** dice que no sabría trabajar de otro modo, que así es capaz de coger el programa que esté desarrollando cualquiera de sus compañeros y continuarlo con garantías. Además ha descubierto que de este modo le resulta más fácil programar. Es como si hubiera encontrado un mecanismo mediante el cual puede programar casi cualquier cosa, todos los programas los comienza igual y todos salen bien.*

*Actualmente el equipo de programadores de **SI Andalucía** está actualizando una aplicación de gestión de almacén para uno de sus clientes. Antes de aceptar el trabajo, **José** pidió estudiar el código de la aplicación para constatar que seguía las pautas de la Programación Estructurada. **Víctor** decía que era una pérdida de tiempo estudiar el código antes de aceptar el trabajo, pero **Carmen** le explicó que a veces, si el código no está estructurado es mejor hacer una nueva aplicación. **José** constató que se trataba de un código muy bien elaborado, siguiendo las líneas de la Programación Estructurada y que no sería muy costoso hacer las adaptaciones que requería el cliente, por lo que aceptó el trabajo y creó un nuevo proyecto para la empresa, al que asignó dos programadores; **Víctor** y **Carmen**.*

*Cuando **Víctor** comenzó a trabajar en este proyecto comprobó que cada una de las líneas del código de la aplicación estaba comentada, bien por el propio lenguaje o bien porque el programador había añadido los correspondientes comentarios que le permitían entender el motivo de utilizar tal instrucción o sentencia. De este modo le resultaba muy fácil utilizar el código existente y hacer las modificaciones necesarias para conseguir los cambios oportunos. Tras hacer las pruebas necesarias observó que los cambios realizados cumplían perfectamente lo que se esperaba y eso le animó a seguir el mismo método pensando que algún día alguien podría encontrarse con ese mismo código, en su misma situación.*

¿Te has preguntado alguna vez cuantas veces tendrás que modificar un programa o una aplicación después de haberlo considerado concluido? ¿Has pensado en la cantidad de programadores que se dedican a tiempo completo a modificar y adaptar los programas que otros han hecho anteriormente? Seguramente no.

Pero la realidad es que se estima que hoy en día más del 80% del coste del software (de cualquier aplicación) no se debe al desarrollo propiamente dicho, sino a su mantenimiento (actualización, modificación, adaptación, extensión,...)

Y a su vez se estima que más del 80% del coste de informatizar una determinada empresa no se debe a la adquisición y mantenimiento de los equipos hardware necesarios, sino al coste del software. ¿Qué conclusión crees que puede sacarse de todo esto? Una bastante clara:

Tenemos que esforzarnos en aplicar técnicas que reduzcan el tiempo y el coste de desarrollo del software, pero aún más tenemos que esforzarnos en que el software desarrollado sea fácil de mantener por cualquier programador, ya que probablemente la persona que realizará el mantenimiento no será la que lo desarrolló.

Esas técnicas que permiten reducir los tiempos y costes de desarrollo y mantenimiento deberán ir encaminadas a conseguir programas **claros y fáciles de entender**, lo más **autodocumentados** posible y en los que se use **código reutilizable**.

A lo largo de la historia de la programación tanto los lenguajes como las técnicas usadas han ido evolucionando rápidamente, y dando lugar a nuevos **paradigmas de programación** , nuevas técnicas que los lenguajes han tenido que ir incorporando para conseguir los objetivos de claridad y facilidad antes mencionados.

Pero los lenguajes de programación no son más que una forma de describir de manera rigurosa la solución de un problema, y la lógica que usan para ello no es más que un reflejo de la lógica humana, que es la que aplicamos al resolver cualquier problema. ¿Realmente la lógica humana ha cambiado con la misma rapidez que las técnicas y lenguajes de programación? Parece ser que no, que desde los romanos hasta ahora no hemos cambiado casi nada.

Por eso las técnicas y estructuras básicas subyacentes, usadas en todos los lenguajes y programas, aunque tengan diferencias, son bastante generales. Por eso las explicábamos en las unidades anteriores como algo independiente del lenguaje (las estructuras básicas de tratamiento las expresábamos en pseudocódigo, que es independiente de todos los lenguajes).

Autoevaluación

Respecto a la estimación de que más del 80% del coste de informatizar una determinada empresa no se debe a la adquisición y mantenimiento de los equipos hardware necesarios, sino al coste del software necesario. ¿Qué conclusión sacarías de esto?

- a) Tenemos que esforzarnos en que el software desarrollado sea fácil de mantener por cualquier programador, ya que probablemente la persona que realizará el mantenimiento no será la que lo desarrolló.
- b) Nuestra única preocupación como programadores, será desarrollar aplicaciones eficientes y baratas.
- c) Las afirmaciones a y b son correctas.
- d) Ninguna de las anteriores es correcta.

Comprobar

Respecto a la programación, podemos afirmar que:

- a) Las técnicas y estructuras básicas subyacentes, usadas en todos los lenguajes y programas, tienen bastantes diferencias, por lo cual son particulares y específicas del lenguaje de programación que utilicemos.
- b) Las técnicas y estructuras básicas subyacentes, usadas en todos los lenguajes y programas, aunque tengan diferencias, son bastante generales
- c) Las técnicas y estructuras básicas subyacentes, usadas en todos los lenguajes y programas, tienen bastantes diferencias, por lo cual son particulares y específicas del lenguaje de programación que utilicemos, por eso es mejor expresarlas en pseudocódigo, que es específico de un lenguaje en concreto.
- d) Ninguna de las anteriores es correcta.

Comprobar

Unidad Didáctica V

Reglas generales de "buena programación"

*Con este proyecto **Víctor** está descubriendo todo lo que le habían comentado sobre cómo se debe programar y que nunca había comprobado realmente. Pensaba que el programador de aquella aplicación debía ser una persona muy ordenada, con buen criterio a la hora de utilizar las palabras más adecuadas en los comentarios y con un profundo conocimiento del lenguaje de programación. Al hacerle estos comentarios a su compañera **Carmen**, ésta le desilusionó un poco al comentarle que esos métodos que a él le parecían de artista, eran lo habitual en un buen programador que basaba su trabajo en un buen algoritmo. Le dijo que ante todo debía ser muy ordenado y pensar que, si por cualquier causa la aplicación tiene que ser modificada, la mejor forma de hacerlo es sobre un código claro y bien estructurado. Como **Víctor** es difícil de convencer, **Carmen** tuvo que utilizar un ejemplo claro de código desarrollado de forma correcta frente a código mal desarrollado de algunos de los programas de la aplicación para la gestión de alumnos. Sin tener muchos conocimientos de programación, Víctor entendió en seguida a lo que se refería Carmen: se trata de tener un poco de sentido común y pensar que estás escribiendo el código para que un amigo lo entienda y pueda continuar con este trabajo.*

Ya hemos mencionado que existen esas normas, reglas y estructuras más o menos generales que siempre debemos tener presentes al programar, e incluso podemos intuir las que serán, ya que si están basadas en nuestra lógica deben ser de bastante sentido común, al menos en sus enunciados generales. En parte ya hemos hablado de ellas en las unidades anteriores. ¿Pero cuáles son las más importantes? ¿Qué ventajas nos aportan? ¿Son independientes unas de otras? ¿Debemos elegir entre cuál de ellas usar o todas se complementan?

La realidad es que gran parte de nuestro aprendizaje como programadores consiste en aprender a incorporarlas a nuestros programas, y a aplicarlas todas conjuntamente de forma coherente, cuando conviene en cada caso.

PARA SABER MÁS
Si te interesa el tema puedes ampliar conocimientos en el siguiente enlace, en el que encontrarás algunas sugerencias y reglas que te ayudarán como futuro programador.
Curiosidades sobre programación **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web)
Para descargar el programa Acrobat Reader pulsa aquí.

Unidad Didáctica V

Diseño descendente

Al diseñar un algoritmo partiremos de los conceptos y problemas más generales, que se irán descomponiendo en conceptos más simples, y continuaremos el proceso de "refinamiento paso a paso" hasta que lleguemos a conceptos y problemas de detalle, de fácil solución.

DEMO: Pulse para ver un ejemplo de un diseño descendente para gestionar el grupo de alumnos del módulo de PLE.

Por ejemplo:

BIEN HECHO:

```
Gestionar el grupo de alumnos del  módulo de PLE.

Inicio
Mostrar un menú con las opciones disponibles (matricular , dar de baja,
evaluar, listar, terminar el programa.)
Seleccionar una opción del menú.
Realizar las tareas correspondientes a la opción elegida.
Fin-Gestionar
     /**************************************************************************/
Mostrar menú de opciones disponibles
Inicio
         MOSTRAR OPCIONES:
                          1.Matricular un nuevo alumno
                          2.Dar de Baja un alumno
                          3.Evaluar a un alumno
                          4.Listado de todos los  alumnos
                          5.Salir del programa
         <sentencia_1 Mostrar_menú
          sentencia_2 Mostrar_menú
          ......
          sentencia_n Mostrar_menú >
Fin-Mostrar-menú
     /**************************************************************************/
Seleccionar la opción del menú
Inicio
         <sentencia_1 Seleccionar
          sentencia_2 Seleccionar
          ........
          sentencia_n Seleccionar >
Fin_Seleccionar
     /**************************************************************************/
Realizar las tareas correspondientes a la opción elegida.
Inicio
         Según Sea opción :
                  Caso 1: Matricular a un nuevo alumno
                  Caso 2: Dar de Baja a un nuevo alumno
                  Caso 3: Evaluar a un alumno
                  Case 4: Listado de todos los alumnos
                  Case 5: Salir del programa
         Fin-SegunSea
Fin-Realizar tareas
     /**************************************************************************/
         Matricular a un nuevo alumno.
         Inicio
Comprobar si quedan plazas libres.
Recoger los datos del alumno
Asignarle número de matrícula.
         Fin-Matricular
     /**************************************************************************/
         Dar de Baja un alumno
         Inicio
         <Sentencia_1 Dar_Baja
```

```
            Sentencia_2 Dar_Baja
                    ......
            Sentencia_n Dar_Baja >
          Fin_Dar_Baja
      /*************************************************************************/
          Evaluar a un alumno
          Inicio
          <Sentencia_1 Evaluar
           Sentencia_2 Evaluar
                    ......
           Sentencia_n Evaluar >

          Fin_Evaluar
      /*************************************************************************/
          Listado de todos los alumnos
          Inicio
          <Sentencia_1 Listado
           Sentencia_2 Listado
                    ......
           Sentencia_n Listado >

          Fin_Listado
      /*************************************************************************/
          Salir Del Programa
Inicio
          <Sentencia_1 Salir
           Sentencia_2 Salir
                    ......
           Sentencia_n Salir >

Fin_Salir
      /*************************************************************************/
          Comprobar plazas libres
          Inicio
          <Sentencia_1 Comprobar_Plazas
           Sentencia_2 Comprobar_Plazas
                    ......
           Sentencia_n Comprobar_Plazas >

          Fin_Comprobar_Plazas
      /*************************************************************************/
          Recoger datos del alumno
          Inicio
          <Sentencia_1 Recoger_Datos
           Sentencia_2 Recoger_Datos
                    ......
           Sentencia_n Recoger_Datos >

          Fin_Recoger_Datos
      /*************************************************************************/
          Asignarle Número de Matrícula
          Inicio
                   Comprobar el último número de matrícula asignado
                   Calcular el número de matrícula a partir del último asignado.
          Fin_Asignar_Numero
      /*************************************************************************/
          Comprobar el último número de matrícula asignado
          Inicio
          <Sentencia_1 Comprobar
           Sentencia_2 Comprobar
                    ......
           Sentencia_n Comprobar >

        Fin_Comprobar
             /*************************************************************/
          Calcular el número de matrícula a partir del último asignado.
          Inicio
          <Sentencia_1 Calcular
           Sentencia_2 Calcular
                    ......
           Sentencia_n Calcular >
          Fin_Calcular
      /*************************************************************************/
```

MAL HECHO:

Aunque el código obtenido aparentemente es más corto, es debido a que las sentencias no están escritas, sino indicadas con un nombre genérico que da el aspecto de modularidad. Pero están todas las tareas entremezcladas, y resultaría casi imposible distinguir dónde empieza y termina cada una o cómo se relaciona con las demás. El resultado es un código difícil de entender y difícil de desarrollar por más de una persona simultáneamente.

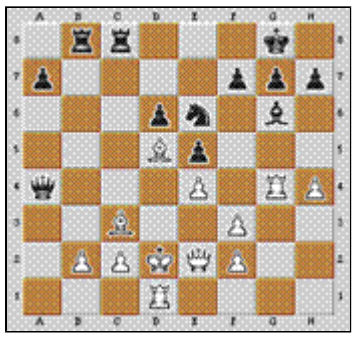
DEMO: Pulse para ver un ejemplo de diseño descendente mal hecho.

```
Gestionar el grupo de alumnos del  módulo de PLE
Inicio
        MOSTRAR OPCIONES:
                        1.Matricular un nuevo alumno
                        2.Dar de Baja un alumno
                        3.Evaluar a un alumno
                        4.Listado de todos los  alumnos
                        5.Salir del programa
        <sentencia_1 Mostrar_menú
         sentencia_2 Mostrar_menú
         ......
         sentencia_n Mostrar_menú >
        <sentencia_1 Seleccionar
         sentencia_2 Seleccionar
         ........
         sentencia_n Seleccionar >
Segun Sea opcion:
                Caso 1:
                        <Sentencia_1 Comprobar_Plazas
                         Sentencia_2 Comprobar_Plazas
                                ......
                         Sentencia_n Comprobar_Plazas >
                        <Sentencia_1 Recoger_Datos
                         Sentencia_2 Recoger_Datos
                                ......
                         Sentencia_n Recoger_Datos >
                        <Sentencia_1 Comprobar
                         Sentencia_2 Comprobar
                                ......
                         Sentencia_n Comprobar >
                        <Sentencia_1 Calcular
                         Sentencia_2 Calcular
                                ......
                         Sentencia_n Calcular >
                Caso 2:
                        <Sentencia_1 Dar_Baja
                         Sentencia_2 Dar_Baja
                                ......
                         Sentencia_n Dar_Baja >
                Caso 3:
                        <Sentencia_1 Evaluar
                         Sentencia_2 Evaluar
                                ......
                         Sentencia_n Evaluar >
                Case 4:
                        <Sentencia_1 Listado
                         Sentencia_2 Listado
                                ......
                         Sentencia_n Listado >
                Case 5:
                        <Sentencia_1 Salir
                         Sentencia_2 Salir
                                ......
                         Sentencia_n Salir >

        Fin-SegunSea
Fin-Gestionar
```

Unidad Didáctica V

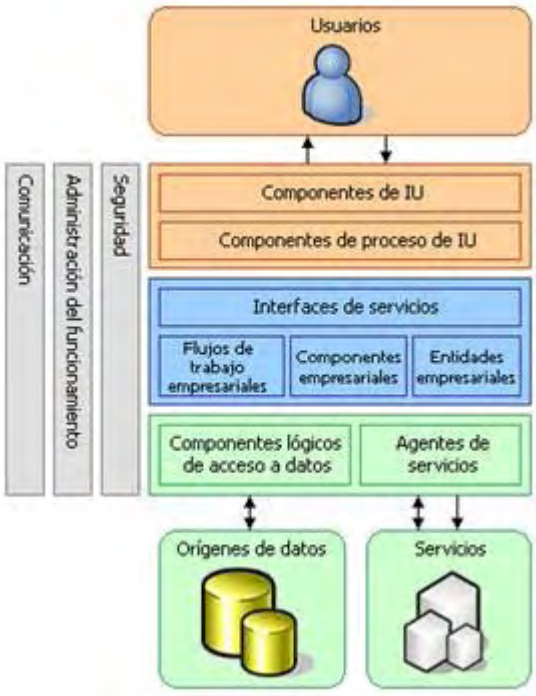
Diseño por módulos, o programación modular



Nos ocuparemos de la programación modular en la unidad 6, pero la mencionamos aquí porque ninguna de estas técnicas es del todo independiente de las demás.

- ¿Cómo afrontamos los problemas en nuestra vida diaria?
- Seguro que si tienes un problema difícil que solucionar lo primero que pensarás es en organizarte.
- Sería estupendo poder dividirlo en varios problemas independientes (módulos) que puedas ir resolviendo por separado.
- Y además sería estupendo que un amigo pueda ir ayudándote con la resolución de una parte a la vez que tú continúas resolviendo otra.
- Así el tiempo empleado seguro que será menor, y la dificultad de cada parte también.

Eso es lo que nos enseña a hacer la programación modular. Dividiremos nuestro problema en partes independientes que puedan abordarse por separado, y solucionarse de forma independiente, incluso por personas o equipos distintos.



Como ejemplo podría usarse el mismo que en el apartado anterior, ya que además de ir de lo general a lo concreto también estructura el código en módulos o subprogramas más o menos independientes. Aunque la modularidad en el diseño puede llegar más lejos cuando módulos distintos de un programa se guardan en ficheros distintos que incluso pueden estar almacenados en ordenadores distintos de una red o de Internet, o incluso separando totalmente la definición de las estructuras de datos de las aplicaciones que las usan, o usando una arquitectura de tres capas (presentación-negocio-acceso a datos) en la que cada capa de la misma aplicación incluso se programa en un lenguaje diferente, y por grupos de programadores diferentes.

PARA SABER MÁS:
En el siguiente enlace encontrarás información más detallada sobre qué es el modelo de tres capas en el diseño de una aplicación, que es ampliamente usado por la mayoría de las aplicaciones comerciales.

Desarrollo de una aplicación en tres capas. **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web)

Autoevaluación

Define en qué consiste la técnica de programación modular.

- a) Al diseñar un algoritmo partiremos de los conceptos y problemas de detalle, de fácil solución, para una vez solucionados, afrontar los problemas más generales.
- b) Dividir un problema en partes independientes que puedan abordarse por separado, y solucionarse de forma independiente, incluso por personas o equipos distintos.
- c) Solucionar el problema intentando agrupar las partes independientes que puedan abordarse por separado, para poder solucionarlo en conjunto.
- d) Ninguna de las anteriores es correcta.

Comprobar

Respecto a la programación modular, señala la afirmación correcta:

- a) La modularidad en el diseño puede llegar más lejos cuando módulos distintos de un programa se guardan en ficheros distintos que incluso pueden estar almacenados en ordenadores distintos de una red o de Internet.
- b) La programación modular nos permite desarrollar aplicaciones con una arquitectura de tres capas (presentación-negocio-acceso a datos) en la que cada capa de la misma aplicación incluso se programa en un lenguaje diferente, y por grupos de programadores diferentes.
- c) Las afirmaciones a y b son correctas
- d) Ninguna de las anteriores es correcta.

Comprobar

Unidad Didáctica V

Programación estructurada

Es sobre lo que nos centraremos en esta unidad. Escribir un programa que funcione es una cosa, pero que lo entienda con facilidad cualquier persona es otra distinta.

- ¿Podremos usar cualquier estructura de control de flujo?
- ¿Podremos dar saltos en el código de un lugar a otro sin que dependan de condiciones?

Si has visto programas antiguos escritos en **Basic**, esto era bastante frecuente, y las revistas antiguas de informática venían llenas de esos programas que funcionaban, pero que como tuvieran una errata ya no había quien los entendiera para poder hacerlos funcionar.

La programación estructurada fija su atención en las estructuras de control de flujo que se pueden usar imponiendo algunas limitaciones y restricciones para evitar generar algoritmos difíciles de seguir y entender, y por tanto difíciles y costosos de mantener.

Eso se consigue básicamente evitando **sentencias o estructuras de salto incondicional,** en las que en vez de ejecutar la siguiente sentencia o instrucción escrita en el programa, se salta a ejecutar otra sin que ello dependa de la comprobación de condición alguna. Sólo se permitirán por tanto estructuras de una entrada y una salida en el flujo del programa.

DEMO: Pulse para ver un ejemplo de un algoritmo que calcula la media de las notas introducidas por teclado. Tendrás la oportunidad de ver un flujograma con un diseño estructurado y otro con un diseño no estructurado.

Ejemplo

BIEN HECHO (ESTRUCTURADO)

```
Algoritmo MediaDeLasNotas
Var suma, nota, sumaNotas, notasIntroducidas, notaMedia : real

Inicio
        Leer nota
        Mientras nota >0 Y nota <> 11
                sumaNotas ← sumaNotas + nota
                notasIntroducidas ← notasIntroducidas + 1
                Leer nota
        Fin_Mientras
        notaMedia ← sumaNotas /  notasIntroducidas
        Escribir ("La nota media de las notas introducidas es", notaMedia)
Fin-Algoritmo
```

MAL HECHO (NO ESTRUCTURADO)

```
Algoritmo MediaDeLasNotas
Var suma, nota, sumaNotas, notasIntroducidas, notaMedia : real

Inicio
        Leer nota
        Mientras nota >0
                Si nota = 11 entonces
                        TerminarCicloMientras
                              /*Es un salto incondicional al final del bucle*/
```

```
      /* Aunque no aparece la parte Si No del condicional, de hecho el resto de sentencias
del ciclo mientras sólo se ejecutarán si la condición nota = 11 no se cumple,
ya que en otro caso se ejecutaría la sentencia de salto incondicional que haría
que el bucle terminara, con lo que tendríamos dos salidas posibles para el bucle:
La normal, tras comprobar la condición de control del bucle nota > 0,  si no se cumple
La interna, que hace que termine el bucle sin haber ejecutado todas las sentencias que
incluye, sin haber completado esa iteración./*

                sumaNotas ← sumaNotas + nota
                notasIntroducidas ← notasIntorducidas + 1

                Leer nota
        Fin_Mientras
        notaMedia ← sumaNotas /  notasIntroducidas
        Escribir ("La nota media de las notas introducidas es", notaMedia)
Fin-Algoritmo
```

Autoevaluación

¿Cómo piensas que se consigue evitar generar algoritmos difíciles de seguir y entender? Señala la afirmación correcta:

- a) Evitando, en la medida de lo posible, usar sentencias o estructuras de salto incondicional.
- b) Documentando correctamente todos los algoritmos, tanto interna como externamente.
- c) No usando bucles condicionales.
- d) Las afirmaciones a y b son correctas

Comprobar

Siguiendo los principios y objetivos de la programación estructurada, respecto a las estructuras, podemos afirmar que:

- a) Sólo se permitirán estructuras que contengan saltos incondicionales.
- b) Sólo se permitirán estructuras de una entrada y una salida en el flujo del programa
- c) Debemos usar, en la medida de lo posible, bucles condicionales
- d) Debemos usar, en la medida de lo posible, bucles repetitivos

Comprobar

Unidad Didáctica V

Relevancia de las estructuras de datos

La elección de una forma adecuada de representar los datos en cada caso influye de forma decisiva en la utilización que puede hacerse de los mismos y en la **eficiencia de los algoritmos.** Es importante seleccionar bien las estructuras de datos a usar.

- ¿Usarías variables de tipo real para almacenar la edad de los alumnos? No, ya que la edad no tiene decimales, y las variables reales ocupan más memoria.
- ¿Emplearías un array de reales para las notas de un alumno? Parece adecuado, ya que suelen ser números con decimales, y cada alumno tiene siempre la misma cantidad de notas.
- ¿Qué estructura de datos usaremos si necesitamos guardar nuestros datos de forma permanente para seguir usándolos en la próxima ejecución de nuestra aplicación? Un fichero es la opción adecuada.
- ¿Manejamos grandes cantidades de información sobre la que queremos realizar consultas elaboradas que relacionen entre sí distintos datos? Una base de datos parece ser lo que necesitamos.

Autoevaluación

Relaciona las siguientes variables con las estructuras de datos correspondientes:

a) Necesitamos guardar datos de forma permanente — Elige...

b) Número de hijos — Elige...

c) Notas de un alumno (suponemos siempre la misma cantidad) — Elige...

d) Gran cantidad de información sobre la que realizar consultas — Elige...

Comprobar

Unidad Didáctica V

Crear código reutilizable

Si inventaras una herramienta para apretar tornillos, y posteriormente en otro problema necesitaras apretar tornillos, ¿volverías a inventarla o usarías la que ya hiciste?

Por medio de técnicas como la **programación orientada a objetos**, que estudiaremos con detalle a través del lenguaje Java a partir de la unidad 14, se consigue facilitar **la reutilización del código, de forma que si un problema ya se ha resuelto, ya se ha comprobado su correcto funcionamiento y se ha codificado, pueda ser directamente utilizable, o a lo sumo con pocas adaptaciones, a nuevas situaciones sin tener que estar siempre "reinventando la rueda".**

Con ello además conseguimos que el código que generamos sea más fácilmente extensible, para incluir nueva funcionalidad, o para adaptarse a nuevas situaciones, pudiendo aprovechar el código anterior en su totalidad.

Autoevaluación

¿Qué entiendes por código reutilizable?:

- a) Usar código de programas que están en Internet.
- b) Usar software libre, por lo cual se puede utilizar las veces que se quiera.
- c) La reutilización del código que se ha desarrollado para solucionar un problema, que ya se ha comprobado su correcto funcionamiento y que se ha codificado, para dar solución a nuevas situaciones
- d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica V

Reglas de programación estructurada

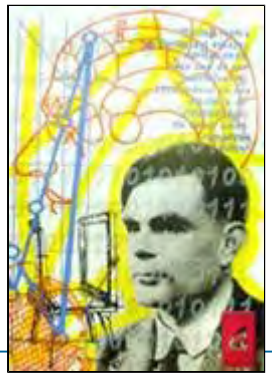
José *ha sido siempre considerado un buen programador por sus profesores y compañeros de estudios. Además, en la empresa es el más capacitado para analizar y planificar el desarrollo de una aplicación. Todo esto lo ha conseguido siendo un firme defensor de las reglas básicas de programación estructurada que transmite a todo aquél que trabaja a su lado. Su destreza como programador de aplicaciones está bien probada por aquéllas que funcionan diariamente como herramientas útiles para sus clientes.*

Como recordarás, en el apartado 2.3 hemos introducido el concepto de programación estructurada, pero sólo como primera aproximación.

- ¿Cuál será el fundamento de la técnica de programación estructurada?
- ¿Cuál es la idea central en programación estructurada?

Víctor siente admiración por esta capacidad y destreza de **José** para la programación y le gustaría alcanzar ese nivel de desarrollo, pero se desespera cuando el único consejo de **José** es que debe conseguir que el código elaborado siga las siguientes reglas:

- La idea central es : ***"Las estructuras de control del flujo de un programa sólo deben tener un punto de entrada y un punto de salida".***
- Existen caminos desde la entrada a la salida que se pueden seguir y que pasan por todas las partes del programa (no existen trozos de programa imposibles de ejecutar).
- Los beneficios que aporta son:
  - **Mejora en la legibilidad y claridad del código resultante** (programas más fáciles de comprender)
  - **Mejora la productividad de los programadores**, como consecuencia de lo anterior
  - **Disminuye los costes de la aplicación**, también como consecuencia de los dos puntos anteriores.
- **La ejecución de un programa estructurado progresa disciplinadamente, en vez de saltar de un sitio a otro de forma impredecible.**
- Su fundamento teórico es el Teorema de Böhm-Jacopini (1966) que puede enunciarse como sigue:



"Cualquier programa de ordenador puede diseñarse e implementarse usando únicamente las tres construcciones estructuradas, que son secuenciación - selección - iteración. (Sin usar por tanto estructuras de tipo salto incondicional o tipo goto."

- Por tanto todo programa estructurado sólo usará esas tres estructuras de control de flujo, que a su vez tienen una entrada y una salida.
  - **Secuencia**: Un conjunto de sentencias que se ejecutan en el orden en que están escritas, incluyendo llamadas a funciones o métodos.
  - **Selección**: Estructura de control condicional (if-then-else, case/switch)
  - **Iteración**: Estructura de control repetitiva o bucles (while, do-while, for)
- No existen bucles infinitos, es decir, ciclos en los que ninguna sentencia de los mismos cambie el valor de la condición de forma que se active la condición de salida.

Al principio **Víctor** no entiende muy bien las reglas (nunca le ha gustado seguir reglas), pero se da cuenta de su importancia cuando comprueba que su programa es mucho más claro y no hay sorpresas en la ejecución, como le ocurre habitualmente.

PARA SABER MÁS:
En el siguiente enlace podrás encontrar información sobre el Teorema de Böhm-Jacopini y sobre sus autores, así como de la trascendencia del mismo. Eso sí, en italiano.
Teorema de Böhm-Jacopini [Versión en Caché]

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web)

Para descargar el programa Acrobat Reader pulsa aquí.

Autoevaluación

Respecto a las Reglas de programación estructurada, señala la afirmación correcta:

- a) Las estructuras de control del flujo de un programa sólo deben tener un punto de entrada y un punto de salida
- b) No existen trozos de programa imposibles de ejecutar
- c) La ejecución de un programa estructurado progresa disciplinadamente, en vez de saltar de un sitio a otro de forma impredecible
- d) Todas las anteriores son correctas

Comprobar

En tu opinión, ¿qué mejoras aporta la programación estructurada? Señala la afirmación correcta:

- a) Programas más fáciles de comprender.
- b) Mejora en la legibilidad y claridad del código resultante.
- c) Disminuye los costes de la aplicación.
- d) Todas las anteriores son correctas

Comprobar

¿Qué entiendes por una estructura de control de flujo de selección? Señala la opción correcta:

- a) Un conjunto de sentencias que se ejecutan en el orden en que están escritas, incluyendo llamadas a funciones o métodos
- b) Es una estructura de control condicional
- c) Es una estructura que dirige la ejecución del programa a otras líneas del mismo interaccionando entre ellas.
- d) Estructura de control repetitiva o bucles.

Comprobar

¿Qué entiendes por una estructura secuencial de control de flujo? Señala la opción correcta:

- a) Un conjunto de sentencias que se ejecutan en el orden en que están escritas, incluyendo llamadas a funciones o métodos
- b) Es una estructura de control condicional.
- c) Es una estructura que dirige la ejecución del programa a otras líneas del mismo interaccionando entre ellas.
- d) Estructura de control repetitiva o bucles.

Comprobar

¿Qué entiendes por una estructura de control de flujo de iteración? Señala la opción correcta:

- a) Un conjunto de sentencias que se ejecutan en el orden en que están escritas, incluyendo llamadas a funciones o métodos
- b) Es una estructura de control condicional
- c) Es una estructura que dirige la ejecución del programa a otras líneas del mismo interaccionando entre ellas.
- d) Estructura de control repetitiva o bucles

Comprobar

Señala cual de las siguientes NO es una regla de programación estructurada:

- a) Las estructuras de control del flujo de un programa sólo deben tener un punto de entrada y un punto de salida
- b) No existen trozos de programa imposibles de ejecutar.
- c) No existen bucles infinitos.
- d) Se debe de usar la herencia y el polimorfismo

Comprobar

Unidad Didáctica V

Manejo de las estructuras de control de flujo en programación estructurada

*A **Carmen** lo que más le gusta de su trabajo es la programación. Sentarse ante el ordenador, escribir el código, probar los programas y depurarlos de errores. Ha adquirido cierta soltura en estas tareas y sabe que su secreto consiste en ajustarse a las reglas de la Programación Estructurada que tanto defiende su compañero **José**. Recuerda que al principio fue **José** precisamente quien le aclaró con varios ejemplos, algunas dudas que tenía sobre la utilización de determinadas sentencias condicionales y repetitivas con las que no lograba un código bien estructurado. **Carmen** se limita a reproducir estas estructuras de control de los ejemplos en todas las aplicaciones que así lo requieren.*

Recuerda que los tipos de estructuras de control de flujo ya han sido introducidos en la unidad 4. Aunque aquí no los vamos a repetir, sí debemos mirar algunos usos concretos de esas estructuras. Son situaciones que te vas a encontrar frecuentemente al programar, y conviene que te familiarices con su funcionamiento. Se trata de que se automatice la programación de situaciones usuales.

ESTRUCTURAS DE CONTROL DE FLUJO EN PROGRAMACIÓN ESTRUCTURADA.

Estructuras condicionales o selectivas anidadas.

Ciclos o bucles anidados.

Fin de bucle preguntando antes de iteración.

Fin de bucle conociendo número de iteraciones.

Fin de Bucle usando un valor de salida.

Unidad Didáctica V

Estructuras condicionales o selectivas anidadas

La estructura condicional permite decidir qué camino tomar entre dos posibles, pero seguramente habrás pensado que te puedes encontrar con muchas situaciones en las que debes elegir entre más de dos caminos posibles. ¿Cómo podemos conseguirlo usando estructuras condicionales, que equivalen a una pregunta de dos opciones? Evidentemente, preguntando más de una vez, y recordando lo que nos respondieron en la vez anterior.

Llevando esto al terreno de la programación, se consigue anidando varias estructuras condicionales tipo IF, lo que no contraviene los principios de la programación estructurada. De hecho el siguiente esquema es del todo equivalente a una estructura tipo CASE / SWITCH

```
Si condicion1 entonces
        Acciones1
Si No
        Si condicion2 entonces
                Acciones2
        Si No
                Si condicion3 entonces
                        Acciones3
                Si No
                        ...
                        Si No
                                AccionesN
                        Fin-Si
                ...
                Fin-Si
        Fin-Si
Fin-Si
```

Como además también podríamos tener estructuras IF en cada una de las partes "entonces" de cada condicional, y como además el nivel de anidamiento puede prolongarse tanto como se desee, el código puede hacerse difícil de entender, por lo que en sintonía con los principios de la programación estructurada debe usarse **indentación** para que los distintos márgenes indiquen de manera visual qué sentencias van dentro de cada parte de cada sentencia IF.

Debe tenerse en cuenta que la estructura interna tiene que estar totalmente dentro de la estructura externa, es decir, el if más interno tiene que estar totalmente dentro del if más externo (aunque si no es así, tendremos un error que nos lo avisará el compilador para que lo corrijamos).

Ejemplo de condicionales anidados: Algoritmo que ordene tres números de menor a mayor

```
Algoritmo OrdenaTresNumeros
Var a,b,c : entero
Inicio
        Leer a
        Leer b
        Leer c
        Si a<b entonces         /*condicional principal*/
                Si a<c entonces
                /*comienzo del condicional interno, en la parte entonces del principal,
                que debería terminar antes de llegar a la parte Si No del condicional princi
                        Si b<c entonces
                                Escribir ("Los número ordenados son ", a, ", ",b, " y
                        Si No
                                Escribir ("Los número ordenados son ", a, ", ",c, " y
                        Fin-Si
                Si No
                        Escribir ("Los número ordenados son ", c, ", ",a, " y ",b)
```

```
                Fin-Si          /*aquí termina el condicional interno*/
        Si No                   /*aquí comienza la parte Si no del condicional principal*/
                Si b<c entonces
                        Si a<c entonces
                                Escribir ("Los número ordenados son ", b, ", ",a, " y
                        Si No
                                Escribir ("Los número ordenados son ", b, ", ",c, " y
                        Fin-Si
                Si No
                        Escribir ("Los número ordenados son ", c, ", ",b, " y ",a)
                Fin-Si
        Fin-Si                  /*aquí termina el condicional principal*/
Fin-Algoritmo
```

Autoevaluación

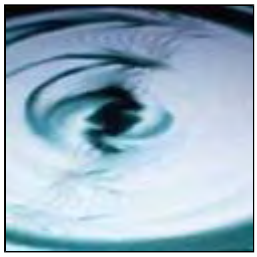
Cuando usemos condicionales anidados se debe tener en cuenta que:

- a) La estructura interna tiene que estar totalmente dentro de la estructura externa.
- b) Debe usarse indentación para que los distintos márgenes indiquen de manera visual qué sentencias van dentro de cada parte de cada sentencia IF.
- c) El if más interno tiene que estar totalmente dentro del if más externo.
- d) Todas las anteriores son correctas

Comprobar

Unidad Didáctica V

Ciclos o bucles anidados



¿Te imaginas lo que es repetir una o varias tareas entre las que se encuentra una que consiste justamente en repetir otra serie de tareas? Eso es justamente un **bucle anidado**. Imagina que la tarea es pasarle un cuestionario tipo test a todos los alumnos de un curso, y para cada alumno debemos formular y anotar la respuesta dada a cada pregunta.

Deberemos repetir la tarea de pasar el test con cada alumno distinto (bucle externo, mientras queden alumnos). Para cada alumno concreto, deberemos repetir la tarea de leer la pregunta y anotar la respuesta (bucle interno, mientras queden preguntas) con cada pregunta del cuestionario.

```
Algoritmo TestConBuclesAnidados
Var totalAlumnos, alumnosEncuestados, totalPreguntas, PreguntasHechas : entero
Inicio
        totalAlumnos ← 20
        totalPreguntas ← 10
        alumnoEncuestado ← 0
        Mientras alumnoEncuestado < totalAlumnos  Hacer      /*Ciclo externo*/
                alumnoEncuestado ← alumnoEncuestado + 1
                Escribir("Encuestando al alumno número", alumnoEncuestado)
                PreguntaHecha ← 0
                Mientras preguntaHecha < totalPreguntas Hacer       /*Ciclo interno*/
                        preguntaHecha ← preguntaHecha +1
                        Escribir ("Haciendo pregunta  número ", preguntaHecha, "al alu
                        Escribir ("Anotando respuesta del alumno", alumnoEncuestado, '
                                                        preguntaHecha)
                Fin-Mientras       /*Del ciclo interno*/
        Fin-Mientras       /*Del ciclo externo*/
Fin-Algoritmo
```

Al igual que ocurría con el condicional, la estructura del ciclo interno tiene que estar totalmente dentro de la estructura del bucle externo. En cada iteración del bucle externo se realizan todas las iteraciones del ciclo interno.

DEMO: Pulse aquí para ver un ejemplo de algoritmo utilizando un bucle: Escribir la tabla de multiplicar.

Ejemplo: Escribir la tabla de multiplicar clásica.

```
Algoritmo TablaDeMultiplicar
Var i,j, producto : entero
Inicio
        i ← 0
        producto ← 0
        Mientras i <= 10 Hacer
                Escribir ("Tabla del ", i, ":")
                j ← 0
                Mientras j <= 10 Hacer
                        producto ← i * j
                        Escribir(i,"x",j,"=",producto)
                        Escribir (cambio de línea)
                              /*Para que la tabla del siguiente número salga en otra línea*
                        j ← j + 1
                Fin-Mientras
                Escribir (cambio de línea)
                      /*Para que la tabla del siguiente número salga en otra línea*/
```

```
                    i ← i + 1
            Fin-Mientras
    Fin-Algoritmo
```

Unidad Didáctica V

Terminación de un bucle preguntando antes de cada nueva iteración

Sabemos ya que los bucles repiten un grupo de sentencias varias veces, dependiendo de una condición, pero ¿qué maneras tenemos de indicar si hay que repetir todo el bucle una vez más o de indicar que ya hemos terminado? Una de ellas es directamente preguntar, y será el usuario el que decidirá continuar o no.

DEMO: Pulse aquí para ver un ejemplo.

```
Algoritmo Operaciones
Var  a,b, resultado : real
         Opcion : entero
Inicio
         Repetir
                              /* En cada bucle se pide la introducción de dos números y se leen
                  Escribir ("Introducir dos números para operar con ellos")
                  Leer a
                  Leer b
                        /* Se escribe un menú con las opciones disponibles para operar con los
                  Escribir( " 1. Sumar a+b
                                                2. Restar a-b
                                                3. Multiplicar a*b
                                                4. Dividir a/b
                                                0. Salir del programa ")
      /* Se pregunta qué opción se elige y se recoge la respuesta en una variable llamada opcion
Si la respuesta es cero, cuando lleguemos al final del bucle la condición será cierta y no
volveremos a dar otra vuelta*/
                  Escribir (¿Qué operación deseas realizar ?(0-4)")
                  Leer opcion
      /*Se comprueba la opción elegida para realizar la operación correspondiente,
guardando el resultado en la variable de ese nombre*/
                  Según sea (opcion)
                           0: Escribir ("Programa terminado")
                           1: resultado ← a + b
                                    Escribir ("El resultado es ", resultado)
                           2: resultado ← a - b
                                    Escribir ("El resultado es ", resultado)
                           3: resultado ← a * b
                                    Escribir ("El resultado es ", resultado)
                           4: resultado ← a / b
                                    Escribir ("El resultado es ", resultado)
                           Otro caso: Escribir ("Operación incorrecta")
                  Fin-Según-Sea
      /*Se comprueba la opción que se eligió, y si es cero el bucle termina, no se hace ninguna
         Hasta (opcion = 0)
Fin-Algoritmo
```

Unidad Didáctica V

Terminación de un bucle indicando previamente el número de iteraciones a realizar

En el caso anterior preguntábamos si teníamos que terminar o no en cada vuelta, pero ¿tiene sentido preguntar si al empezar el bucle se sabe la respuesta, es decir, si sabemos al empezar cuántas vueltas tenemos que darle al bucle? Parece que no. Bastará con llevar la cuenta de las que llevamos con un **contador** para saber si debemos continuar o no. Eso es justamente lo que vamos a hacer en este ejemplo:

```
Algoritmo MediaNotas
Var   nota, suma: real
                numeroAlumnos, contador : entero

Inicio
        Escribir ("¿Cuántos alumnos hay en la clase?")
        Leer numeroAlumnos
        contador ← 0
        Mientras contador < numeroAlumnos Hacer
                Escribir ("Introduzca la nota del alumno ", contador)
                Leer nota
                suma ← suma + nota
                contador ← contador + 1
        Fin-Mientras
        Escribir ("La media de las notas de todos los alumnos es " , suma / numeroAlumno
Fin-Algoritmo
```

Unidad Didáctica V

Terminación de un bucle usando un valor de salida

¿Y si no queremos preguntar, pero tampoco sabemos el número de veces que debe repetirse una nueva iteración del bucle.? Podremos solucionarlo si ponemos un valor especial, que usamos como **centinela** para saber cuando terminamos.

Por ejemplo, si en cada vuelta debemos leer un número entero positivo, podríamos leer hasta que se introduzca un negativo.

Si por el contrario queremos repetir el bucle para una serie de personas, cuyo nombre se pida en cada vuelta, podríamos usar un valor raro, lo suficiente como para poder asegurar que nadie se va a llamar así. El centinela suele ser un valor del mismo tipo, pero fuera de rango. Por ejemplo podemos contar con que nadie se llamará "FIN".

Otro ejemplo: Si lo que esperamos es introducir el mes del año como un número, los valores válidos serán los números enteros del 1 al 12. Cualquier valor fuera de ese rango, por ejemplo 0, podría usarse como centinela.

Ejemplo: .

DEMO: Pulse para ver un ejemplo. Se trata de leer números correspondientes a un mes del año e indicar de qué mes se trata.

```
Algoritmo LeerMesDelAño
Var mes : entero
Inicio
        Repetir         /*Este es el bucle a repetir */
                Escribir ("Introduce  un número del 1 al 12 y te diré al mes que corres
                Leer mes
                Según Sea mes
                        Caso 1: Escribir ("El ", mes, " corresponde a Enero")
                        Caso 2: Escribir ("El ", mes, " corresponde a Febrero")
                        Caso 3: Escribir ("El ", mes, " corresponde a Marzo")
                        Caso 4: Escribir ("El ", mes, " corresponde a Abril")
                        Caso 5: Escribir ("El ", mes, " corresponde a Mayo")
                        Caso 6: Escribir ("El ", mes, " corresponde a Junio")
                        Caso 7: Escribir ("El ", mes, " corresponde a Julio")
                        Caso 8: Escribir ("El ", mes, " corresponde a Agosto")
                        Caso 9: Escribir ("El ", mes, " corresponde a Septiembre")
                        Caso 10: Escribir ("El ", mes, " corresponde a Octubre")
                        Caso 11: Escribir ("El ", mes, " corresponde a Noviembre")
                        Caso 12: Escribir ("El ", mes, " corresponde a Diciembre")
                        Caso 0: Escribir ("Terminando el programa... ")
                        Otro Caso: Escribir ("No es un mes válido ")
                Fin-Según-Sea
        Hasta  mes=0
        /      *Hasta que se introduzca el valor usado como centinela, que es cero.*/
Fin-Algoritmo
```

Unidad Didáctica V

Terminación de un bucle al agotar los datos de entrada

Hasta ahora hemos visto ejemplos en los que los datos se le iban suministrando al programa por parte del usuario, pero ¿es la única forma posible? Desde luego que no. Serán muchas las situaciones en las que los datos se obtendrán de alguna "fuente de datos" distinta, como pueda ser un fichero, o algún otro tipo de estructura de datos.

Será frecuente que queramos procesar todos los datos de esa estructura, (por ejemplo, actualizar a principio de año el sueldo de todos los trabajadores contenidos en el fichero de nóminas de la empresa). En este caso una buena opción es repetir un bucle que,

- **empezando por el primer elemento de la estructura** (el primer trabajador del fichero) **y**
- **mientras queden elementos por procesar,** (mientras queden trabajadores en el fichero),
- **procese ese elemento** (actualice el sueldo de ese empleado) **y**
- **pase al siguiente elemento de la estructura** (pase al siguiente trabajador del fichero).
- **Si al intentar pasar al siguiente elemento de la estructura de datos comprobamos que ya no quedan más por procesar, el bucle terminará** (Si al pasar al siguiente trabajador alcanzamos el fin del fichero, terminaremos el ciclo).

Para que esto sea posible, aunque no conozcamos el número de elementos de la estructura, debemos disponer de algún mecanismo, o de alguna "marca" que nos permita preguntar si hemos alcanzado el final o no. Por ejemplo, para los ficheros deberemos disponer de una **funciónEOF()** (End Of File) que nos indique si hemos alcanzado el final del fichero. La función podrá tener otro nombre, pero el funcionamiento será similar. Devolverá verdadero si estamos al final del fichero (más allá del último elemento del fichero, en la marca de fin de fichero), y falso en caso contrario.

Autoevaluación

¿Qué estructura utilizarías si deseamos que se ejecuten repetidamente una serie de instrucciones? Señala la opción correcta:

- a) No es posible ejecutar repetidamente una instrucción.
- b) Usando una estructura de control repetitiva o bucles (while, do-while, for).
- c) Usando la instrucción goto para ir al código a ejecutar
- d) Ninguna de las anteriores es correcta.

Comprobar

¿Qué estructura utilizarías si deseamos que se ejecuten repetidamente una serie de instrucciones un número de veces determinado? Señala la opción correcta:

- a) Repitiendo el código tantas veces como queremos que se repitan las instrucciones
- b) Usando una estructura de control repetitiva o bucles (while, do-while, for) y usando un contador
- c) Usando la instrucción goto para ir al código a ejecutar.
- d) No es posible ejecutar repetidamente una instrucción.

Comprobar

¿Qué estructura utilizarías si deseamos que se ejecuten repetidamente una serie de instrucciones, pero desconocemos el número de veces que deben ejecutarse? Señala la opción correcta:

- a) Usando una estructura de control condicional.
- b) Usando una estructura de control repetitiva o bucles (while, do-while, for) y poniendo un valor especial, que usaremos como centinela para saber cuando terminar.
- c) Usando la instrucción goto para ir al código a ejecutar
- d) Ninguna de las anteriores es correcta.

Comprobar

Programación estructurada

Unidad Didáctica V

Estructuras que no se corresponden con la programación estructurada

*Un fin de semana en casa, **Víctor** recuerda sus primeros programas, lo hacía en Basic y sonríe al comprobar la total ausencia de Programación Estructurada. Le viene a la cabeza lo que comentaría **José** al ver este código lleno de saltos y sin ningún tipo de comentario o estructura que permita adivinar las sentencias que componen un bloque de instrucciones. Estos eran programas que funcionaban, o al menos eso pensaba él en su momento, porque ahora encuentra errores continuamente y casi que le gustaría volver a hacerlos siguiendo las "reglas de **José**". Recuerda que en una ocasión su compañero le dijo que hay programadores que defienden el uso de estructuras que no se ajustan a la Programación Estructurada en situaciones muy concretas, pero que ese no era su caso, él siempre se esforzaba por prescindir de sentencias como Goto y hasta ese momento había conseguido evitarla.*

Hemos visto que las normas de programación estructurada tienen un claro propósito, que es hacer que los programas sean más claros, y por tanto más fáciles de entender, lo que redunda en más facilidad a la hora de modificarlos y mantenerlos.

Eso nos lleva automáticamente a una mayor productividad de los programadores, o lo que es lo mismo, a poder mantener más programas con menos programadores, o dedicando menos tiempo. Claramente con eso conseguimos un menor coste de mantenimiento, y por tanto aplicaciones más competitivas.

¿Pero siempre serán los programas más claros usando programación estructurada? ¿Debemos prohibir el uso de sentencias de salto incondicional tipo Goto?

Este tipo de sentencias permiten saltar, sin comprobar ninguna condición a otra sentencia distinta de la que está escrita a continuación en el programa. Esa sentencia incluso puede ser cualquier otra del programa que hayamos etiquetado para poder saltar hasta ella.

Ejemplo: Podemos simular una estructura selectiva, tipo "Según Sea" o "CASE", usando sentencias Goto (Ir a).

```
Si condicion1 entonces
        Goto L1        /*L1 es el nombre de la etiqueta que debemos colocar en la zc
                                de código a la que queremos saltar*/
Si condicion2 entonces
        Goto L2
Si condicion3 entonces
        Goto L3
Goto L4

L1: sentencias1       /*código para el caso de que condicion1 sea verdadera*/
        Goto L5
L2: sentencias2       /*código para el caso de que condicion2 sea verdadera*/
        Goto L5
L3: sentencias3       /*código para el caso de que condicion3 sea verdadera*/
        Goto L5
L4: sentencias4       /*código para el resto de casos. Aquí termina la selección*/

L5: sentencias5       /*Tras la selección el programa continuaría ejecutando estas se
```

El mismo resultado se puede conseguir con if anidados, o con una estructura selectiva múltiple, sin romper la regla de "una entrada - una salida" de la programación estructurada, con un código más fácil de comprender. Por ejemplo:

```
Si condicion1 entonces
```

```
            sentencias1       /*código para el caso de que condicion1 sea verdadera*/
    Si no
            Si condicion2 entonces
                    sentencias2        /*código para el caso de que condicion2 sea verdadera*/
            Si no
                    Si concicion3  entonces
                            sentencias3        /*código para el caso de que condicion3 sea ver
                    Si no
                            sentencias4       /*código para el resto de casos. Aquí termina la
                    Fin-Si
            Fin-Si
    Fin-Si
    sentencias5        /*Tras la selección el programa continuaría ejecutando estas sentencias*/
```

La mayoría de los lenguajes incluyen estructuras de control de flujo que rompen los principios de la programación estructurada, y en lenguajes primitivos, eran el único medio de conseguir determinados comportamientos, como llamadas a subprogramas.

PARA SABER MÁS:
Si quieres comprobar el exceso de complicaciones que presenta un programa en el que no se tiene en cuenta la programación estructurada, visita este enlace en el que puedes comprobar lo que algunos eran capaces de hacer para complicarse la existencia.
Nada de programación estructurada. **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web)

Como norma general se aconseja encarecidamente no usarlas, ya que suelen complicar la legibilidad del código. Pero según el lenguaje que se emplee, y el problema que se esté resolviendo, pueden resultar útiles e incluso mejorar la legibilidad, acortar el código o mejorar la eficiencia. Incluso en algunos lenguajes son necesarias para construir determinadas sentencias de control del flujo y que se comporten como cabe esperar según lo que establece el teorema de programación estructurada (una entrada - una salida).

En general, los usos "adecuados" de GOTO requieren un estilo de programación disciplinado, con transferencias de control del flujo siempre hacia delante y dentro de una región local del código. Como norma general, ya que siempre son evitables, los evitaremos. Aunque con las debidas precauciones, podemos admitirlos.

Cualquier uso de GOTO que incluya saltos hacia atrás en el código o saltos hacia delante pero a una región remota del código, complican su comprensión, y rompen la linealidad programa, por lo que no será admisible.

En definitiva, existe cierto debate entre distintos autores sobre la conveniencia o no de permitir el uso de GOTO (o sentencias de salto incondicional) Intentaremos aportar luz a ese debate.

PARA SABER MÁS:
Algunos comentarios sobre los usos del goto, y cómo afectan a la programación modular.
Usos de la Instrucción goto. **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web).

Unidad Didáctica V

Usos admisibles de sentencias GOTO o de salto incondicional

Ya hemos visto que se desaconseja el uso de esta estructura salvo contadas situaciones. ¿Cuáles serán esas situaciones? Sin ánimo de ser exhaustivos, vamos a presentar algunas de las situaciones posibles más frecuentes.

Básicamente hay tres casos en los que es hasta cierto punto "*admisible"* el uso de estructuras de salto incondicional (tipo GOTO):

- Salidas múltiples y "prematuras" de ciclos.
- Transferir el control al código manipulador de errores en lenguajes que no disponen de un manejo apropiado de **excepciones.**
- Mejorar la eficiencia en programas con un tiempo de respuesta crítico.

Por ejemplo, en Java, para construir una sentencia selectiva múltiple, tipo "Según Sea" (o tipo CASE), es necesario usar dentro de la sentencia switch la sentencia break que es una sentencia de salto incondicional, pero con la salvedad de que no nos lleva a cualquier punto del programa, sino al final del propio switch que se está ejecutando.

Ejemplo en Java: Se supone que n es una variable de tipo entero a la que ya se le ha dado un valor desde teclado.

```
...
switch (n){
        case 1: System.out.println("UNO");
                                break;
        case 2: System.out.println("DOS");
                                break;
        case 3: System.out.println("TRES");
                                break;
        case 4: System.out.println("CUATRO");
                                break;
        case 5: System.out.println("CINCO");
                                break;
        default: System.out.println("NO ES UN NÚMERO DEL 1 AL CINCO");
}
...
```

El resultado es que se escribirá con letra el valor del número si está entre uno y cinco, y si no está en ese rango, un mensaje indicándolo. Justamente el comportamiento estructurado que esperamos de esta sentencia. Pero ¿qué pasaría en Java si no escribiéramos **break** en cada caso? Que una vez que se entrara en uno de los casos, se ejecutarían todas sus sentencias y todas las de los casos siguientes hasta llegar a un break o hasta alcanzar el final del **switch.**

Sin break, si el valor de n fuese 1 la salida sería:

UNO
DOS
TRES
CUATRO
CINCO
NO ES UN NÚMERO DEL 1 AL CINCO

Sin break, si el valor de n fuese 3 la salida sería:

TRES
CUATRO

CINCO
NO ES UN NÚMERO DEL 1 AL CINCO

Sin break, si el valor de n fuese 5 la salida sería:

CINCO
NO ES UN NÚMERO DEL 1 AL CINCO

Etc.

Como vemos **break** es una sentencia que **salta incondicionalmente** al final de la estructura. Sin embargo, hace que toda la estructura se comporte como debe ya que ***no*** **nos lleva a cualquier sitio del programa, sino al fin de la sentencia *switch***.

Este ejemplo correspondería al primer apartado, a pesar de no tratarse de un ciclo, por similitud de la situación.

Unidad Didáctica V

Uso de Goto en salidas múltiples y "prematuras" de ciclos

Es frecuente tener varias condiciones para terminar un ciclo. Cuando esto ocurre, una buena idea es rediseñar el algoritmo de modo que no se requieran salidas múltiples. Si eso no es posible, (y no debemos llegar a esa conclusión demasiado pronto) se pueden manejar las salidas múltiples incluyendo una variable lógica (booleana) como bandera que identifique cada condición, y comprobar el estado de las banderas en cada iteración del ciclo.

Por ejemplo, a la hora de buscar un elemento en un vector, podemos seguir el siguiente código estructurado:

```
...
Var indice: entero
          encontrado: booleano
          tamañoVector:  entero
          X:  ...
          /*valor a buscar. Será del tipo de los elementos del vector*/
vector : array unidimensional de tamañoVector elementos de tipo...,
      /*que están en las posiciones de la cero a la "tamaño-1" del vector*/
...
indice=0
encontrado = falso

Mientras indice < tamañoVector Y encontrado=falso Hacer
      /*Mientras queden elementos y no lo hayamos encontrado*/
          Si vector[indice]=X entonces
                    encontrado=verdadero
                          /*bandera que permite comprobar si ha ocurrido la condición*/
          Si no
                    indice ← indice+1
                          /*Seguimos buscando en la siguiente posición del vector*/
Fin-Mientras

Si indice = tamaño
      /*Comprobamos si salimos del bucle por que llegamos al final sin encontrarlo*/
          sentenciasSiXNoEncontrado
Si no
      /*o por el contrario salimos porque ya lo habíamos encontrado*/
          sentenciasSiXEncontrado
Fin-Si
      /*Aquí continuaría el programa*/
```

Pero esta solución plantea tres "problemas":

- Se necesitan variables adicionales, lo que es menos eficiente.
- Se requieren pruebas adicionales a la salida del ciclo para verificar qué condición provocó la salida
- No es posible que el ciclo termine inmediatamente tras la ocurrencia de una condición sin completar la iteración en vez de al comienzo de la siguiente iteración. Con banderas, debemos terminar la iteración y comprobar la bandera al inicio de la siguiente. Terminar inmediatamente puede ser un requisito en algunos casos.

Una solución alternativa a la anterior para el manejo de salidas múltiples, razonable a pesar de usar goto sería la siguiente:

...

```
Var indice: entero
         tamañoVector:  entero
         X:  ...
/      *valor a buscar. Será del tipo de los elementos del vector*/
vector : array unidimensional de tamañoVector elementos de tipo...,
      /*que están en las posiciones de la cero a la "tamaño-1" del vector*/
...
indice=0

Mientras indice < tamañoVector Hacer       /*Mientras queden elementos */
         Si vector[indice]=X entonces
                  goto Etiqueta1       /*decimos donde está el código si se ha encontrado*/
         Si no
                  indice ← indice+1         /*Seguimos buscando en la siguiente posición del
Fin-Mientras
goto Etiqueta2        /* Si hemos llegado aquíes que no se encontró y no saltamos a Etiqueta1,
                                 luego debemos ejecutar las sentencias para el caso de no
Etiqueta1:
         sentenciasSiXEncontrado       /*se ejecutan si lo hemos encontrado*/
         goto Etiqueta3       /*Terminamos sin ejecutar las sentencias del caso "no encontrado
                                          que están detrás*/
Etiqueta2:
         sentenciasSiXNoEncontrado       /*o por el contrario salimos porque ya lo habíamos
Etiqueta3:
               /*Aquí continuaría el programa*/
```

Se mantiene el espíritu de linealidad y localidad del código, ya que a fin de cuentas el control sólo se transfiere hacia delante y a la vecindad de la salida del ciclo, por lo que la claridad no se ve excesivamente afectada. No obstante, esa pequeña complicación de la claridad no tiene sentido salvo que la eficiencia sea un factor crítico. En ese caso, evitar el uso de variables lógicas adicionales y evitar realizar pruebas adicionales a la salida del ciclo podría estar justificado. Si nuestro programa no trabaja con un tiempo crítico de respuesta, que es la situación más usual en las aplicaciones de gestión, no tendría justificación.

Unidad Didáctica V

Uso de Goto para mejorar la eficiencia en programas con un tiempo de respuesta crítico

En el apartado anterior hemos visto un uso admisible de goto que también estaba relacionado con la eficiencia en el uso de variables y en la limitación del número de condiciones a comprobar.

En este apartado vamos a tratar de la eficiencia, pero desde otro punto de vista. En la programación estructurada frecuentemente tenemos necesidad de repetir segmentos de código, o como alternativa, poner esos segmentos de código en subprogramas (también pueden ser llamados **métodos**, **procedimientos**, **funciones** o **subrutinas**) que serán llamados frecuentemente, incrementando así el coste implícito de mayor procesamiento al tener que ligar la llamada del método con el código que debe ejecutarse en su lugar.

Cuando se llama a un subprograma, todo el **entorno volátil del programa** en ejecución debe guardarse en memoria, y asignarle espacio en memoria a todas las variables y objetos que utilice el subprograma. Una vez que finaliza el subprograma, debe recuperarse de la memoria el **entorno volátil** y restaurarlo en la CPU (para que el procesador sepa por donde dejó el trabajo), continuando el programa principal por el mismo sitio que lo dejó antes, pero teniendo a su disposición los datos que haya podido generar el subprograma ejecutado.

Ese coste adicional en tiempo de procesamiento puede eliminarse usando Goto.

Ejemplo: Imaginemos el siguiente esquema de un programa estructurado, donde A y B representan trozos de código que deben ser repetidos, y que por tanto son incluidos como subprogramas a los que se llama para ser ejecutados, para no tener que duplicar la escritura de ese código. Es decir, A y B son subprogramas.

```
...
Si condicion1 entonces
        A
        B
Si no
        B
Fin-Si
Mientras condicion2 Hacer
        A
        B
Fin-Mientras
```

Primera Alternativa usando Goto, para evitar la repetición del código:

```
...
Si condicion1 entonces
        Etiqueta1:
                A
                goto Etiqueta2
Si no
        Etiqueta2:
                B
Fin-Si
Si condicion2 entonces
        goto Etiqueta1
Fin-Si
```

Segunda alternativa usando Goto, para evitar la repetición de código, aún más eficiente:

```
...
Si condicion1 entonces
        Etiqueta1:
                A
Fin-Si
B
Si condicion2 entonces
        goto Etiqueta1
Fin-Si
```

¿Debemos por tanto usar siempre goto con este fin? No parece aconsejable. Una vez más, el código resultante es menos claro, y sólo merecerá la pena cuando en tiempo de respuesta de nuestra aplicación sea tan crítico que tengamos que mejorar al máximo la eficiencia, disminuyendo tiempos de cualquier lugar que sea posible.

Pero además, en los casos en los que las restricciones de eficiencia aconsejen usar las versiones con goto, debe usarse documentación tanto **interna** con comentarios al programa como **externa**, para explicar porqué se ha escrito el código de esa manera. Además se aconseja incluir dentro de la documentación interna, como comentario, la versión de código estructurado equivalente para ayudar a la comprensión de un código que de otra forma sería oscuro para toda persona distinta del que lo programó, o incluso para el mismo programador pasado un tiempo.

Unidad Didáctica V

Uso de Goto para manejo de excepciones

Todavía existe otra posibilidad de que el flujo de ejecución de un programa no siga el orden lineal establecido en la escritura del programa.

Imagina que tenemos un programa que espera la introducción de un número entero desde el teclado, y que en lugar de introducir un valor válido le suministramos el valor ***xw34dfcvn-***@, por ejemplo. ¡El ordenador se vuelve loco intentando entender ese número entero! No hay forma de convertir eso en número entero. Efectivamente el resultado es que se produce un error "irrecuperable". Esto produce un **aborto del programa**, (hace que el programa aborte), terminando su ejecución de forma brusca en ese mismo instante. Eso no es una situación precisamente deseable, ¿verdad?.

El programador debe tener previsto la posibilidad de que ocurran esos "accidentes", llamados excepciones, y en el caso de que se produzcan, detectarlos y tratarlos adecuadamente para evitar que el programa aborte.

En lenguajes que no disponen de manejo adecuado de excepciones, eso se conseguía mediante directivas del compilador y usando goto para transferir el control al código manipulador de errores. La forma concreta de hacerlo, además de dificultosa, excede nuestros propósitos y necesidades.

Sin embargo, en lenguajes modernos como Java, que dispone de un eficaz manejo de excepciones, es bastante fácil.

Básicamente, cuando como programador detectó un trozo de código problemático, susceptible de producir una **excepción,** en lugar de evitar de forma costosa y complicada que se produzca, **dejo que ocurra y la capturo**.

En el momento que se produce la excepción, en vez de abortar, el control del flujo del programa pasa automáticamente al manejador de la excepción, que es un código identificado como tal. En ese código puedo establecer qué debe hacerse para recuperarse de ese error, o si eso no es posible, qué debe hacerse para terminar de forma ordenada (y prevista) el programa. La sintaxis en Java es la que figura a continuación, aunque el manejo de excepciones en Java se explicará con detalle cuando introduzcamos formalmente el lenguaje.

```
 try{

        <sentencias concódigo problemático>
}catch  (TipoExcepciónPrevista     identificadorExcepción){
        <sentencias de código que trata la excepción. Es el manejador de la misma>
}finally{
        <sentencias que se ejecutan siempre, tanto si hay excepción como si no>
}
```

Es como si dijésemos "intenta (try) ejecutar estas sentencias, y si por algo no puedes, captura (catch) la excepción que se ha producido y haz lo que te digo a continuación. Finalmente (finally) y con independencia del resultado, ejecuta también este otro trozo de código".

Hay prestigiosos especialistas en Java que sostienen que la cláusula finally es no sólo innecesaria, sino también inútil, por lo que normalmente no se usará.

Autoevaluación

Supón que estás desarrollando una aplicación en Java. En una de las funcionalidades

de tu aplicación, el usuario debe introducir su edad. Este valor introducido lo has definido como una variable de tipo entero.¿Cómo controlarías que el valor introducido no fuera real o de tipo carácter, lo cual produciría un error fatal de ejecución del programa?

- a) Mediante código, comprobar que el valor introducido es de tipo entero.
- b) Mediante bucles repetitivos con centinela.
- c) No es necesario controlarlo, ya que el compilador de Java depura estas situaciones.
- d) Manejando excepciones. El control del flujo del programa pasa automáticamente al manejador de la excepción, que es un código identificado como tal. En ese código puedo establecer qué debe hacerse para recuperarse de ese error o qué debe hacerse para terminar de forma ordenada el programa

Comprobar

Unidad Didáctica V

Uso inapropiado de estructuras de salto incondicional: sentencia return

La pregunta es ¿qué sentido tiene usar goto o algo similar para simular una estructura en vez de usar esa estructura directamente? En este caso, ninguno, y por eso este punto tiene un título que empieza por "uso inapropiado".

Como ejemplo vamos a ver un uso inadecuado de la sentencia return, que se usa en muchos lenguajes para indicar que termina la ejecución de un subprograma y que se retorna el control del flujo al punto en el que se hizo la llamada desde el programa principal. (También se usa return para indicar el **valor que devuelve el subprograma**). Esta sentencia, de acuerdo con los principios de la programación estructurada sólo debe ejecutarse como última sentencia del subprograma, ya que éste es un bloque de código que debe ejecutarse hasta el final (una entrada - una salida). Sin embargo, **podemos colocar una sentencia return en medio del código de un subprograma, para conseguir el mismo efecto que conseguiríamos con un if-then-else**

Imaginemos el ejemplo ya visto de un bucle para buscar un elemento en un vector. Podemos imaginar que este código está dentro de un subprograma, al que se le pasa como argumento el elemento "X" a buscar, y el vector de nombre "miVector" en el que hay que buscarlo, y que el subprograma devuelve Verdadero si lo encuentra y Falso en caso contrario.

```
Subprograma BuscarEnVector ( elemento X, vector miVector)
        Var indice: entero
                tamañoVector:  entero
            /*el parámetro X representa el valor a buscar. Será del tipo de los elementos de
        vector : array unidimensional de tamañoVector elementos de tipo...,
            /*que están en las posiciones de la cero a la "tamaño-1" del vector*/
        indice=0
        Mientras indice < tamañoVector Hacer       /*Mientras queden elementos */
                Si vector[indice]=X entonces        /*Si es el que buscamos*/
                 return verdadero        /*Devuelve verdadero para indicar que lo ha encont
                                                    termina el subprograma */
                indice ← indice+1         /*Si llegamos a esta sentencia es por que la conc
                                           anterior era falsa, y por tanto seguimos buscanc
                                           posición del vector. Aunque no aparece,realmente
                                           sentencia estuviese en la parte "Si no" del Si,
                                           depende de la condición. Pero no lo parece, ya c
                                           una cláusula "Si no"*/
        Fin-Mientras
        return falso      /*Si llegamos aquí es porque hemos recorrido todo el vector sin en
                                           el elemento X, por lo que tenemos que devolver f
Fin-Subprograma
```

El mismo resultado, pero en versión estructurada, con un solo return al final del subprograma, y con las sentencias que dependen de que no se cumpla la condición claramente en la parte "Si no", sería el siguiente:

```
Subprograma BuscarEnVector ( elemento X, vector miVector)
        Var indice: entero
                tamañoVector:  entero
                encontrado: booleana
            /*el parámetro X representa el valor a buscar. Será del tipo de los
        elementos del vector*/
        vector : array unidimensional de tamañoVector elementos de tipo...,
    /*que están en las posiciones de la cero a la "tamaño-1" del vector*/
        ...
        encontrado =falso
```

```
            indice=0
            Mientras indice < tamañoVector Y encontrado =falso  Hacer
                    /*Mientras queden elementos */
                    Si vector[indice]=X entonces
                          /*Si el de la posición actual es el que buscamos*/
                             encontrado = verdadero
                    Si no
                             indice ← indice+1       /*Comparamos con el siguiente*/
                    Fin-Si
            Fin-Mientras
            return encontrado
      Fin-Subprograma
```

Unidad Didáctica V

Algunas conclusiones

De todo lo dicho hasta ahora pueden extraerse algunas conclusiones importantes sobre lo que significa seguir un buen estilo de programación:

- La programación estructurada no pretende limitar la creatividad de los programadores, sino establecer **estándares** que faciliten su actividad y eviten el caos.
- Es deseable que todos los programadores de un proyecto sigan el mismo **estilo** de programación, para producir un código de calidad uniforme.
- El uso de proposiciones **Goto** debe **evitarse** en circunstancias normales.
- La profundidad de **anidamiento** de las construcciones de un programa no debe ser mayor de **cinco** en circunstancias normales.
- Los subprogramas no deben ser ni excesivamente pequeños, lo que aumentaría el coste de cada llamada en proporción al beneficio de no volver a escribir el código que contienen cada vez, ni demasiado grandes, lo que haría que fuesen difíciles de entender, y probablemente divisibles en más de un subprograma. Usualmente un **tamaño** entre 5 y 30 líneas de código es el apropiado.
- Apartarse de las circunstancias normales requiere un cuidadoso estudio de los pros y los contras, y normalmente deberá contar con la aprobación del líder o director del proyecto.

Autoevaluación

¿Cuál de las siguientes afirmaciones NO es una directriz de programación estructurada para conseguir un buen estilo de programación?:

a) La profundidad de anidamiento de las construcciones de un programa no debe ser mayor de cinco en circunstancias normales

b) Los subprogramas no deben ser ni excesivamente pequeños, lo que aumentaría el coste de cada llamada en proporción al beneficio de no volver a escribir el código que contienen cada vez, ni demasiado grandes, lo que haría que fuesen difíciles de entender, y probablemente divisibles en más de un subprograma. Usualmente un tamaño entre 5 y 30 líneas de código es el apropiado

c) El uso de proposiciones Goto debe evitarse en circunstancias normales.

d) Se han de usar estructuras condicionales siempre que sea posible, evitando los bucles repetitivos

Comprobar