aprenderaprogramar.com

Reflexiones finales sobre el control directo del flujo de programas y buenas prácticas de programación. (CU00183A)

Sección: Cursos

Categoría: Curso Bases de la programación Nivel I

Fecha revisión: 2024

Autor: Mario R. Rancel

Resumen: Entrega nº 82 del Curso Bases de la programación Nivel I

24

REFLEXIONES FINALES EN TORNO AL CONTROL DIRECTO DEL FLUJO DE PROGRAMAS

No queremos terminar el análisis del control directo del flujo de programas sin algunas consideraciones que estimamos de interés. Buscaremos algunas analogías con la vida real que puedan aportar algo de "visión de conjunto" respecto a qué supone el control directo de flujos.

En primer lugar pensemos en el alcohol: es una herramienta poderosa con distintas aplicaciones (médico-farmacéuticas, industriales, culinarias, ...). De su buen uso pueden derivarse efectos beneficiosos. Su mal uso puede generar efectos nocivos en distintos grados, incluso destructivos, relacionados con la pérdida de control de una persona sobre sí misma. En nuestro símil, las instrucciones de control directo de flujos nos pueden llevar a emborracharnos con cierta facilidad y a perder el control del programa.

Otra comparación, respecto a lo que supone para los programas, está relacionada con los conceptos de estructura y solidez. Por un lado los programas que usan las tres estructuras básicas normalmente estarán relacionados con "solidez", "linealidad", "orden". Se trataría de un edificio: con ascensores que trabajan ordenadamente (bucles). Los programas no asentados en estas estructuras pueden caer en "endeblez", "ramificación", "desorden". Se trataría de un árbol donde las ramas están enmarañadas y conectan unas con otras.

Una cuestión que queremos comentar, aunque sea de forma breve, es a qué nos hemos referido a lo largo de páginas anteriores cuando hablábamos de malfuncionamientos, falta de eficiencia, etc. y lo haremos viendo ejemplos de:

- Ramas u hojas muertas
- Código superfluo
- Disfunción de contadores.

❖ ***Ramas u hojas muertas.***

Todas las partes de un programa han de ser útiles para algo. Se admite que parte de un programa permanezca en una situación de "latencia": no funcionar prácticamente nunca, excepto en circunstancias excepcionales (por ejemplo la aparición de un error, o la desconexión de un servidor. Hechos que pueden pasar meses o incluso años sin que se produzcan). Lo que no se admite es una parte de un programa que no funciona nunca ni podrá funcionar nunca. Por analogía con el árbol decimos que hay una rama muerta (varias líneas o todo un bloque o conjunto de bloques no útiles) o una hoja muerta (una instrucción o una línea que no tienen actividad). No necesariamente una rama u hoja muerta aparecen a consecuencia del control directo de flujos. Vamos a ver ambas situaciones.

1º) Hoja muerta debido a control directo de flujos.

```
1. Inicio [Ejemplo aprenderaprogramar.com]
        2. Pedir Edad
        3. Si Edad < 18 Entonces
                IrA 8
           FinSi
        4. Si Edad >= 18 Entonces
                IrA 6
           FinSi
        5. Mostrar "Usted debe dirigirse a la calle Gran Vía, nº 43, oficina 3 – 2"
        6. Mostrar "Gracias por su visita"
        7. Finalizar
        8. Mostrar "Es requisito ser mayor de 18. Lo sentimos, pregunte en información"
9. Fin
```

La línea 5 no se ejecutará nunca debido a que el control directo de flujos lo impide. Esto implica que o bien el control directo de flujos está mal planteado o bien la línea 5 está ahí pero no debería existir. Este tipo de situaciones se suele presentar con programas muy largos o bien con programas que son correcciones de versiones anteriores donde el programador deja "olvidada" cierta parte del código que debía haber eliminado al mismo tiempo que hacía una de las correcciones.

2º) Rama muerta por diseño incorrecto.

```
1. Inicio [Ejemplo aprenderaprogramar.com]
        2. Mostrar "Por favor, introduzca un número" : Pedir Numero
        3. Numero = ABS(Numero)
        4. Si Numero < 0 Entonces
                Mostrar "Se ha transformado el número en su valor absoluto para poder operar con él"
                Numero = ABS(Numero)
           FinSi
        5. Mostrar "La raíz cuadrada del número es", SQR(Numero)
6. Fin
```

La línea 4 no se ejecutará nunca debido a que la condición que evalúa no será nunca verdadera (por haber transformado el número en la línea 3 en su valor absoluto). El programa "parece" que funciona pero obviamente no tiene ningún sentido alargarlo y complicar su comprensión innecesariamente. Los motivos por los que puede ocurrir, aparte de por un profundo nublamiento mental, son los mismos que en el caso anterior: programas muy largos en los que resulta difícil recordar el estado de una variable, o correcciones que no acaban completándose. El caso anterior podría tener la siguiente explicación:

1. El programador decide hacer valor absoluto del número para realizar los cálculos directamente.
2. Realiza una corrección: decide detectar si el número es negativo para advertir en ese caso al usuario de que se transforma en valor absoluto.
3. "Olvida" eliminar el valor absoluto que había establecido previamente.

Otro caso habitual de rama muerta es una parte de código entre un *Finalizar* y el *Fin* del programa que está inutilizada. El código detrás de un *Finalizar* ha de tener algún tipo de vínculo con el resto del programa ya que en caso contrario será código muerto.

❖ ***Código superfluo.***

Hace relación a toda parte de un programa que si bien tiene actividad, es innecesaria por repetida, redundante, complicada o alargada innecesariamente. Consideremos el siguiente caso, relativo a un proceso de extracción de datos y cálculo de su raíz cuadrada con final de datos definido por un señalero de valor – 99.

```
1. Inicio [Ejemplo aprenderaprogramar.com]
        2. i = 1
        3. Leer Dato(i)
        4. Mientras Dato(i) <> – 99 Hacer
                Desde j = 1 hasta i
                        Mostrar "La raíz del número es", SQR(Dato(i))
                        SalirDesde
                Siguiente
                i = i + 1
                Leer Dato(i)
           Repetir
5. Fin
```

Una vez más el programa "parece que funciona", pero es un despropósito. Si analizamos lo que hay, vemos un bucle *Mientras ... Hacer* para la extracción de datos y, anidado dentro de éste, un bucle *Desde ... Siguiente* de misión incierta. Existe en primer lugar una salida directa de este bucle a través de un *SalirDesde* que no es consecuencia de ninguna evaluación. Si esto es así, se produce la salida con una única pasada por el contenido del bucle. ¿Para qué vamos a querer un bucle de lectura única? Aparte de: ¿Para qué queremos ese bucle si el ya mencionado *Mientras ... Hacer* está resolviendo la iteración satisfactoriamente?

Si alguien dice "esto funciona" es como decir "el edificio no se cae". Habría que añadir: "de milagro. Es tan peligroso que no sólo no te van a dar licencia ni cédula de habitabilidad, sino que te lo van a precintar y demoler". Y es que el que aparezcan códigos de este tipo sólo es explicable como consecuencia del copia y pega sin sentido o del "construyo a lo loco y no poco a poco".

En general, antes de enfrentarnos a la corrección de un programa lleno de lindezas, será más provechoso apagar el ordenador y empezar de cero. Alargaremos la vida de nuestras neuronas. Otro ejemplo de código superfluo puede ser:

```
Si A > 10 Entonces
        Si B < 5 Entonces
                Mostrar "El teléfono es el 883221"
        FinSi
FinSi
```

En vez de:

```
Si A > 10 y B < 5 Entonces
        Mostrar "El teléfono es el 883221"
FinSi
```

Si el lenguaje de programación no nos permitiera la expresión resumida estaría justificado el anidamiento de *Si ... Entonces.* Pero en caso contrario estaremos escribiendo un código superfluo que además de serlo dificulta la comprensión del programa. Cuanto mayor es el grado de anidamiento de las estructuras más difíciles resultan de leer y comprender. ¿Para qué complicarnos la vida?, o ¿Para qué complicarle la vida a aquel que tenga que revisar el programa en el futuro?

❖ ***Disfunción de contadores.***

Ya hemos comentado diversos problemas relacionados con el uso de contadores, tanto al hablar de la instrucción *Desde ... Siguiente* como al hablar específicamente de contadores. Nos vamos a referir aquí en concreto a problemas derivados de los "saltos" asociados al control directo de flujos.

Supongamos una variante del ya visto caso de extracción de datos y cálculo de su raíz cuadrada con final de datos definido por señalero – 99, en esta ocasión con una cláusula para que en caso de que el número sea negativo ignorarlo y pasar al siguiente.

```
1. Inicio [Ejemplo aprenderaprogramar.com]
        2. i = 1
        3. Leer Dato(i)
        4. Mientras Dato(i) <> – 99 Hacer
                4.1 Si Dato(i) < 0 Entonces
                        IrA 4
                    FinSi
                4.2 Mostrar "La raíz del dato", i, "vale", SQR(Dato(i))
                4.3 i = i + 1
                4.4 Leer Dato(i)
           Repetir
5. Fin
```

¿Funcionará este algoritmo? Si tenemos "suerte" y no hay números negativos entre los datos extraídos, parecerá que funciona. Por el contrario, si se extrae un número negativo el ordenador se quedará bloqueado en el proceso consistente en ir de la línea 4.1 a la línea 4 indefinidamente. Y eso que "el programa estaba preparado".

La intención del programador parece ser continuar extrayendo el siguiente dato si uno resulta negativo. Lo que también parece es que para eso el contador tiene que avanzar y el dato extraerse... pequeño detalle. Es decir, bastaría con poner *i = i + 1 : Leer Dato(i) : IrA 4* en vez de directamente *IrA 4.* Pero ya puestos a ser inteligentes, ¿Por qué no escribir lo siguiente?

```
4.1  Si Dato(i) >= 0 Entonces
                Mostrar "La raíz de dato", i, "vale", SQR(Dato(i))
     FinSi
4.2  i = i + 1
4.3  Leer Dato(i)
```

Con una pequeña reflexión hemos ahorrado en código y tenemos una estructura más sólida y fácil de leer. La introducción de saltos indiscriminados nos hará propensos a "pequeños olvidos" (y grandes problemas). Más claro imposible.

Se tiene ahora un elemento más de juicio para valorar lo que decíamos en la presentación relativo a que todos los lenguajes informáticos eran válidos, con matices. Para empezar a programar uno de estos matices puede ser la "dependencia" que un lenguaje tenga de la instrucción *IrA*. El *Basic* que circulaba en los años 80 prácticamente "obligaba" a su uso, ya que el lenguaje no tenía otras formas de creación de bucles. Esto está completamente superado en el *Visual Basic* y resto de lenguajes modernos, donde el uso de *IrA* es prácticamente residual o nulo. Los lenguajes modernos tienen como base la programación estructurada. Asentarnos en buenas prácticas de programación estructurada será construirnos buenos cimientos como programadores.

Próxima entrega: CU00184A

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=28&Itemid=59