**UNIVERSIDAD DEL CEMA**
**Buenos Aires**
**Argentina**

Serie
**DOCUMENTOS DE TRABAJO**

**Área: Ingeniería Informática**

# REFACTORIZACIÓN DE CÓDIGO Y CONSIDERACIONES SOBRE LA COMPLEJIDAD CICLOMÁTICA

**Darío G. Cardacci**

**Septiembre 2016**
**Nro. 592**

**www.cema.edu.ar/publicaciones/doc_trabajo.html**
UCEMA: Av. Córdoba 374, C1054AAP Buenos Aires, Argentina
ISSN 1668-4575 (impreso), ISSN 1668-4583 (en línea)
Editor: Jorge M. Streb; asistente editorial: Valeria Dowding <jae@cema.edu.ar>

# REFACTORIZACIÓN DE CÓDIGO Y CONSIDERACIONES SOBRE LA COMPLEJIDAD CICLOMÁTICA

**Darío G. Cardacci**[*]

***Resumen***

La refactorización de software es una práctica que permite obtener código más legible y ordenado, lo que redundará en beneficios relacionados con aspectos económicos y el tiempo necesario para realizar actividades vinculadas a la obtención de software, como los relacionados con el testing y el mantenimiento en cualquiera de sus formas. A pesar de sus innegables aportes, se deben considerar ciertos aspectos fundamentales dependiendo el tipo de refactorización que se desee realizar. El presente articulo plantea parcialmente cuales podrían ser observados cuando la refactorización que se desee practicar conlleva la transformación de código estructurado a código orientado a objetos. En particular plantea como se modifica y comporta la métrica que monitorea la complejidad ciclomática, cuando en la refactorización propuesta se aplican mecanismos y relaciones válidas en la orientación a objetos e inexistentes en las formas estructuradas para el desarrollo de software. Con este precedente se puede continuar analizando la totalizada de transformadas posibles de un modelo a otro al refactorizar, con el objetivo de establecer qué prácticas resultan positivas respecto de la variable analizada y cuales no son convenientes utilizar.

## *Introducción*

En la actualidad se desarrollan piezas de software que deben coexistir con código desarrollado con anterioridad, en muchos casos décadas atrás. Al código heredado normalmente se lo denomina legacy(1). El transcurso del tiempo ha redefinido los escenarios de aplicación, las tecnologías que se imponen y la complejidad asociada a cada tipo de desarrollo. Estos factores operan como fuerzas movilizadoras para que surjan prácticas metodológicas que se adapten mejor a la realidad y aprovechen mejor los recursos disponibles.

En este sentido, mucho código escrito de manera estructurada se ha refactorizado(2) a código orientado a objeto, por considerarse que en determinados escenarios, este último se adapta mejor y brinda soluciones más eficaces y efectivas.

---

[*] Los puntos de vista expresados en este trabajo son de exclusiva responsabilidad de su autor, y no necesariamente expresan la posición de la Universidad del CEMA.

[1] "Legacy Code". Este tipo de código fuente en general se caracteriza por estar fuertemente relacionado con una tecnología de desarrollo cuyo soporte técnico es escaso, ya no existe o esta fuertemente ligado a un sistema operativo en particular. El término también se aplica a código insertado en software más nuevo con el objetivo de integrar u ofrecer soporte a una función creada en el pasado.

[2] Refactoring es un tipo de reestructuración de código que se define como "el proceso de cambiar el software de un sistema de manera que no altere su comportamiento externo pero mejorando su estructuración interna" (Fowler, 2000). Como cualquier reestructuración de código, el refactoring tiene como objetivo limpiar el código para que sea más fácil de entender y modificar.

***Aspectos a considerar en la refactorización de código estructurado a código orientado a objetos***

De acuerdo a lo expresado en la introducción esta forma de refactorización ha perseguido dos objetivos. El primero y más asociado al concepto mismo, es el de permitir que el código sea más legible. Esto redundará en mejorar el esfuerzo necesario para mantenerlo, aspecto no menor, considerando que gran parte del presupuesto asociado al ciclo de vida de un software se insume en mantenimiento. El segundo es trasladar el código estructurado a una forma orientada a objetos, de esta manera se pueden aprovechar las virtudes que esta última posee. Entre las ventajas más notorias se pueden mencionar: la herencia, el polimorfismo, la agregación, la implementación de interfaces y sobre todo el enfoque botton –up al momento de desarrollar, lo que permite avanzar integrando partes simples hasta lograr el sistema complejo.

Existen muchos aspectos que se pueden observar cuando se aplica esta forma de refactorización (de código estructurado a código orientado a objetos). A modo enunciativo se puede reflexionar sobre como quedaría el desarrollo en términos de líneas de código escritas, pues no siempre el código que es más legible es el que tiene menos líneas. La mayoría de las sugerencias sobre buenas prácticas de programación, recomiendan que al momento de escribir código se desdoblen las líneas que poseen varias instrucciones con el objetivo de obtener no solo una mejor legibilidad, sino que los sistemas de ayuda a la escritura de código puedan identificar claramente la instrucción y línea, cuando se produce un error sintáctico o de asignación de tipos.

Otro aspecto sobre el que se podría establecer una línea de razonamiento es el relacionado con cómo quedaría el acoplamiento y las métricas asociadas a él, como el grado de entrada y salida de cada módulo. Además de determinar esto, sería interesante establecer como afecta a la dinámica del desarrollo todos los elementos mencionados en conjunto.

Cómo se puede observar esta forma de refactorización donde no solo deseamos mejorar la legibilidad, sino que en la misma transmutamos el código de una forma estructurada a una orientada a objetos, podría tener muchas aristas a analizar. Dentro del espectro de análisis encontraremos aspectos favorables y otros que no lo son. Para visualizar todas las relaciones que se producen entres las variables y como se afectan se debería construir una matriz de análisis que permita observar claramente como se comportan al momento de llevar adelante la refactorización.

En este artículo en particular tomaremos uno de los aspectos denominado ***"Complejidad Ciclomática (Cyclomatic Complexity)"*** con la intención de dejar planteado que podría ocurrir con la misma en caso de llevarse adelante la práctica descripta. En este espacio no se intentará agotar todas las posibles implicancias debido a la extensión que conlleva dicho trabajo. Ese análisis detallado se lleva adelante en la tesis que el autor desarrolla referida al tema en particular y que le da origen al presente artículo.

La ***Complejidad Ciclomática*** es una métrica del software ampliamente difundida y aceptada, que permite obtener una medición cuantitativa de la complejidad lógica de un programa o pieza de software. Tiene la particularidad de haber sido desarrollada de manera que pueda aplicarse con independencia del lenguaje en que se ha programado.

Originalmente esta métrica fue propuesta por Thomas McCabe en 1976 y se basa en el diagrama de flujo determinado por las estructuras de control del código correspondiente a un programa. Del análisis

del mismo se puede obtener una medida cuantitativa de la cantidad mínimas de pruebas[3] que se deberían realizar para recorrer al menos una vez cada camino básico[4] del diagrama de flujo. Si se logra establecer un lote de pruebas que realice esto, habremos logrado ejecutar y probar cada sentencia del programa. Se asume que a mayor cantidad de caminos básicos mayor será la complejidad lógica[5] del programa.

Si bien el nombre de ***"Complejidad Ciclomática"*** es con el que se la conoce, la métrica no se concentra en contar ciclos o bucles sino en las decisiones (condiciones), por lo cual, en realidad lo que estaríamos midiendo, es la cantidad de decisiones simples[6] asumiendo que estas están directamente ligadas a la complejidad lógica del código.

Existen tres formas de calcular la complejidad ciclomática. Siendo:

CC = Complejidad Ciclomática.

A = Número de aristas del grafo (Una arista conecta dos vértices si una sentencia puede ser ejecutada inmediatamente después de la primera).

N = Número de nodos del grafo correspondientes a sentencias del programa.

P = Número de nodos predicados (Aquellos nodos cuyo grado de salida es dos. Normalmente representados en un diagrama de flujo por condiciones booleanas).

R = Número de regiones cerradas del grafo.

***Forma 1:***

CC = A – N + 2. Esta forma realiza un conteo de todas las aristas del grafo de nodos, luego cuenta la cantidad de nodos totales (nodos tradicionales + nodos predicados), finalmente sustrae la cantidad de nodos obtenida a la cantidad de aristas y le suma 2.

***Forma 2:***

CC = P + 1. Aquí se realiza un conteo de los nodos predicados solamente y se le suma 1.

***Forma 3:***

CC = R + 1. La tercera forma de cálculo propone identificar las regiones (cerradas) del grafo y le suma 1 para obtener el resultado.

---

[3] Se refiere a pruebas que recorren las instrucciones del sistema para determinar si la lógica expresada por las sentencias es la planificada y el algoritmo resuelve el problema planteado.

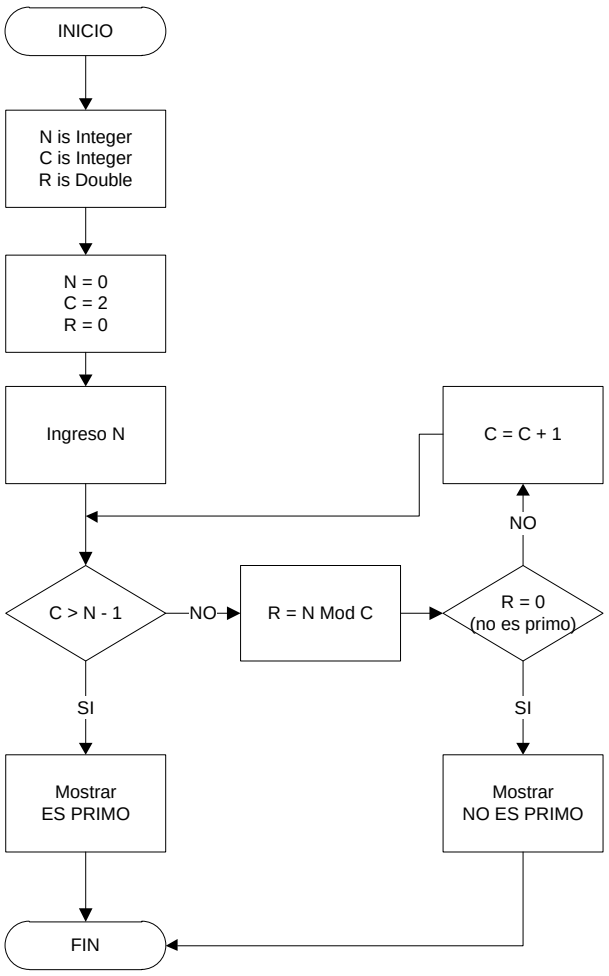
[4] Camino Básico: se denomina camino básico en un diagrama de flujo a cualquier camino que logre unir el punto de partida o inicio del mismo con el punto de finalización.

[5] Se debe comprender a la complejidad lógica en este contexto como la cantidad de posibles caminos básicos ofrecidos, para que a través de la selección de uno por la toma de múltiples decisiones se resuelve un problema.

[6] Las decisiones simples son aquellas que se sustentan en una decisión booleana cuyo resultado es verdadero o falso originándose a partir de ellos dos caminos alternativos en un grafo de flujo.

Analicemos un ejemplo concreto para clarificar el cálculo.

Se desea desarrollar un programa que permita ingresar un número positivo y determinar si es primo[7]. En caso afirmativo o negativo mostrar una leyenda que lo acredite.



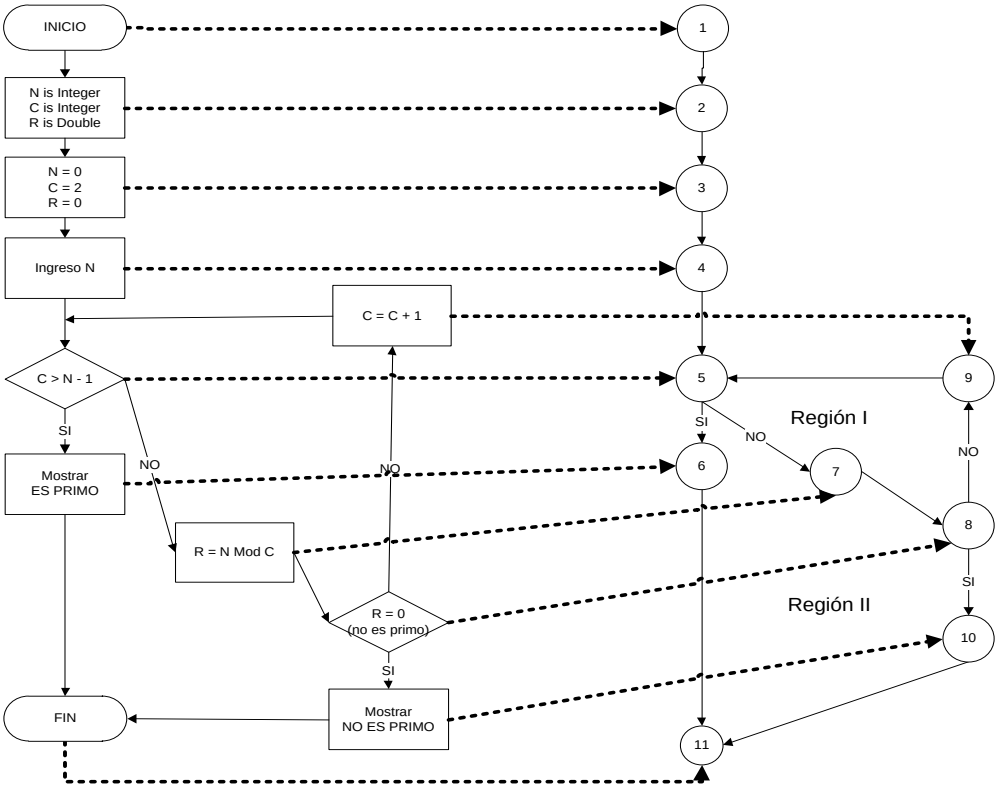

```
Private Sub EsPrimo
    Dim N As Integer = 0
    Dim C As Integer = 2
    Dim R As Double = 0

    N = InputBox("Ingrese un Número: ")

    Do Until C > N - 1
      R = N Mod C
      If R = 0 Then
        MsgBox("No es primo")
        Exit Sub
      Else
        C = C + 1
      End If
    Loop
    MsgBox("Es primo")
End Sub
```

[7] Un número primo es un número natural mayor que 1 que tiene únicamente dos divisores distintos: él mismo y el 1. Se entiende por divisor de un numero A al número B, siempre que existe un número entero C que satisface A = B * C.

Transformación del Diagrama de flujo a un grafo de nodos



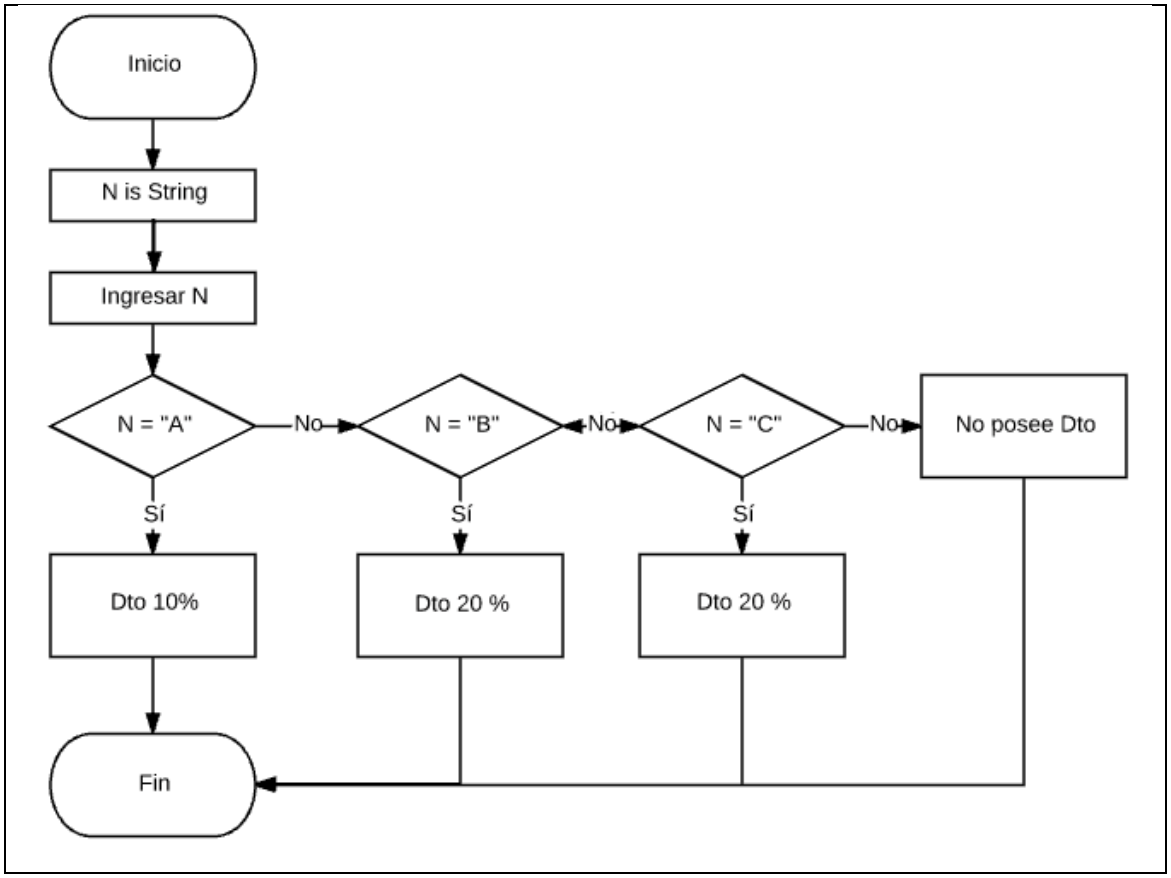

En el grafo de nodos se puede observar como los nodos 5 y 8 son los únicos que poseen un grado de entrada (aristas que le llegan al nodo) igual a uno y un grado de salida (aristas que salen del nodo) igual a dos. Esto es debido a que solo los nodos que representen un elemento de decisión del diagrama de flujo tendrán esa característica.

La complejidad ciclómatica del grafo de nodos es:

| Forma 1: | Forma 2: | Forma 3: |
|---|---|---|
| CC = A – N + 2<br>CC = 12 – 11 + 2<br>CC = 3 | CC = P + 1<br>CC = 2 + 1<br>CC = 3 | CC = R + 1<br>CC = 2 + 1<br>CC = 3 |

A continuación se analizará un código estructurado sencillo que permita determinar un descuento aplicable del 10%, 20% o 30% según el usuario indique que el descuento es del tipo A, B o C respectivamente. Se calculará la complejidad ciclomática y luego se refactorizará a código orientado a objetos para determinar si la complejidad ciclomática total se incrementa, se mantiene igual o disminuye.

```
Private Sub Descuento()
    Dim N As String
    N = InputBox("Ingrese A, B o C: ")
    If N = "A" Then
      MessageBox.Show("El descuento es del 10%")
    Else
      If N = "B" Then
        MessageBox.Show("El descuento es del 20%")
      Else
        If N = "C" Then
          MessageBox.Show("El descuento es del 30%")
        Else
          MessageBox.Show("No posee descuento")
        End If
      End If
    End If
  End Sub
```




Claramente se puede observar que la Complejidad Ciclomática es de 4. ¿Qué ocurrirá con la misma si se refactoriza el código a un forma Orientada a Objetos?

```
Public MustInherit Class Descuento
   Public MustOverride Sub CalculoDescuento()
End Class

Public Class Descuento10
   Inherits Descuento
   Public Overrides Sub CalculoDescuento()
      MessageBox.Show("El descuento es del 10%")
   End Sub
End Class

Public Class Descuento20
   Inherits Descuento
   Public Overrides Sub CalculoDescuento()
      MessageBox.Show("El descuento es del 20%")
   End Sub
End Class
```

```
Public Class Descuento30
   Inherits Descuento
   Public Overrides Sub CalculoDescuento()
      MessageBox.Show("El descuento es del 30%")
   End Sub
End Class


Private Sub DescuentoPolimorfico(pObjeto As Descuento)
      pObjeto.CalculoDescuento()
End Sub
```

Cómo se puede observar, en el código orientado a objetos expuesto se logra la misma funcionalidad. Para poder realizarlo se utiliza una clase abstracta denominada ***Descuento*** que declara un método virtual ***CalculoDescuento.*** El objetivo es que exista una subclase especializada por cada tipo de descuento e implemente el método virtual de forma polimórfica.

Las subclases ***Descuento10, Descuento20, Descuento30*** heredan de la clase ***Descuento*** e implementa el método como se mencinó en el párrafo anterior.

Luego desde el procedimiento ***DescuentoPolimorfico,*** que posee un parámetro ***pObjeto*** del tipo ***Descuento*** que es el mismo tipo que el de la clase base y como consecuencia directa de la herencia (que es una relación del tipo es-un), permitirá que le pasemos cualquier objeto del tipo ***Descuento10, Descuento20, Descuento30***.

Como cada sub clase posee el conocimiento del tipo de descuento, no será necesario evaluar (If .. Then) que tipo de descuento se debe realizar. Ni bien el usuario a través de la interfaz seleccione la opción A, B o C se instanciará la subclase específica sin necesidad de evaluar que ingresó el usuario o qué seleccionó el usuario.

Se puede observar que la complejidad ciclomática de esta función del software se redujo a cero. También en caso de modificarse el requerimiento y aparecer nuevas opciones, el código estructurado incrementará la complejidad ciclomática en 1 por cada nueva opción. En el código refactorizado utilizando técnicas de la orientación a objetos se irán agregando subclases que en su especialización definirán las nuevas posibilidades manteniendo la complejidad ciclomática en cero.

Podemos concluir en que al menos esta estructura de if … then secuenciales, al ser refactorizadas desde código estructurado a código orientación a objetos, mejora notablemente no solo la legibilidad del código sino la ***complejidad ciclomática*** que fue la variable que se deseaba observar.

Considerando que cualquier lógica de un sistema se puede expresar en términos primitivos como una forma if .. Then, se puede aseverar que cualquier programa se puede ver como ***Inicio*** – un conjunto de ***Procesos*** e ***if … Then*** y ***Fin.*** Esto deja planteada la posibilidad de revisar cualquier estructura de código estructurado y su comportamiento respecto a la ***complejidad ciclomática*** cuando se lo refactoriza a la forma orientada a objetos, de manera de establecer si siempre se verifica una disminución en la ***complejidad ciclomática***.