Acoplamiento y Cohesión

Jose Ramon Pascual June 15, 2019

«Divide y Vencerás» es una expresión de dudoso origen que se atribuye tanto a Julio César como a Maquiavelo, con poco honestas intenciones. El objetivo era reducir y fraccionar a un enemigo difícil de combatir, para enfrentarlo contra sí mismo, y para combatir las fracciones más fácilmente. Pero estoy convencido que en aquel ámbito no pensarían que la división de un problema podría plantear un problema de **Acoplamiento y Cohesión**.

En ciencias de la computación es conocido el paradigma de diseño algorítmico *Divide y Vencerás.* Este paradigma viene a resumir que, un problema de gran complejidad (o tamaño) se puede resolver recursivamente dividiéndolo, hasta donde sea posible y su solución sea trivial. La solución al problema original viene dada por la resolución recursiva de cada una de sus partes.

En ingeniería del software se aplica dicho principio para concebir el concepto de **modularización** del programa. Cuando se quiere enfrentar un problema complejo de desarrollo de software, se divide en partes mas simples. Estas partes se denominan **módulos** de forma genérica, y deben poderse analizar, diseñar, desarrollar y probar lo más independientemente posible del resto.

De esta manera, un programa complejo quedaría compuesto por varios módulos, que resuelven partes del programa mas simples, relacionados entre sí.

Las ventajas de la modularidad en un programa son evidentes. Sin embargo, la modularidad, y la relación entre los módulos de un programa, puede generar un nuevo problema en cuanto a lo fuertemente dependientes que sean unos de otros.

A menudo ocurre que al modificar un módulo de un programa deja de funcionar otro módulo que no parecía estar relacionado.

El motivo de esto es el **acoplamiento** entre dichos módulos.

Contenidos

¿Qué es el acoplamiento?

El acoplamiento es el grado en que los módulos de un programa **dependen** unos de otros.

Si para hacer cambios en un módulo del programa es necesario hacer cambios en otro módulo distinto, existe acoplamiento entre ambos módulos.

En Programación Orientada a Objetos, si una clase X usa una clase Y, se dice que X depende de Y. Esto es, X no puede realizar su trabajo sin Y, por lo tanto, existe acoplamiento entre las clases X e Y.

Como se observa, el acoplamiento es direccional, puede haber acoplamiento de la clase X con la clase Y, pero esto no implica que exista en sentido inverso.

Tipos de Acoplamiento

El acoplamiento se puede dar en muchas formas, y algunas de ellas no son fácilmente detectables. Por ejemplo, podemos tener código repetido en distintos módulos, por lo que para hacer un cambio en un fragmento habría que repetirlo en todos.

El mas frecuente es precisamente el **acoplamiento de contenido,** producido cuando un módulo utiliza propiedades, funcionalidades, métodos internos de otro módulo diferente.

Se dice **acoplamiento externo** al que ocurre entre unidades de software dependientes de componentes externos. Por ejemplo, librerías externas o dispositivos externos al programa.

También puede darse un **acoplamiento común** o global cuando hay zonas de memoria compartida entre distintos módulos, variables globales. Un cambio en el recurso, o el tipo, implicará cambios en todos los módulos que lo usan.

¿Qué es cohesión?

Cohesión no implica dependencia. Tal como dice el significado de la palabra en castellano, es el efecto de reunirse o adherise las cosas entre sí o la materia de la que está formada. Algo está cohesionado si tiene sentido, y una dirección común.

En ingenieria del software, algo tiene alta cohesión si tiene un alcance definido, unos límites claros y un contenido delimitado y perfectamente ubicado.

Si hablásemos de POO, una clase tendrá alta cohesión si sus métodos están relacionados entre sí, tienen un contenido claro y temática común, trabajan con tipos similares, etc. Todo bien encerrado dentro de la clase, y perfectamente delimitado.

Por ejemplo, si en nuestro programa tenemos la lógica de persistencia en base de datos encerrada en una sola clase, o en un paquete de clases, y no que cada clase por separado se gestione su propia persistencia, tendríamos alta cohesión.

En resumen, **un código altamente cohesionado tiende a ser mas autocontenido y con menos dependencias.**

Acomplamiento vs Cohesión

Acoplamiento y cohesión son dos conceptos que están relacionados entre sí. De forma natural, si en nuestro desarrollo buscamos una alta cohesión, obtendremos un bajo acoplamiento.

Por ejemplo, en el caso de la escritura en base de datos, **buscando la alta cohesión del programa**, tendríamos un paquete de clases de persistencia para el guardado de datos en una base de datos, independiente de las funcionalidades del resto del programa, y bien delimitados. Esa independencia de otros módulos estaría limitando, **estaríamos reduciendo el grado de acoplamiento** de esas clases con otros módulos del programa.

El riesgo de centrarse sólo en la cohesión será llegar a construir un macrocomponente que termine siendo poco cohesionado y haga demasiadas cosas no relacionadas con el objeto.

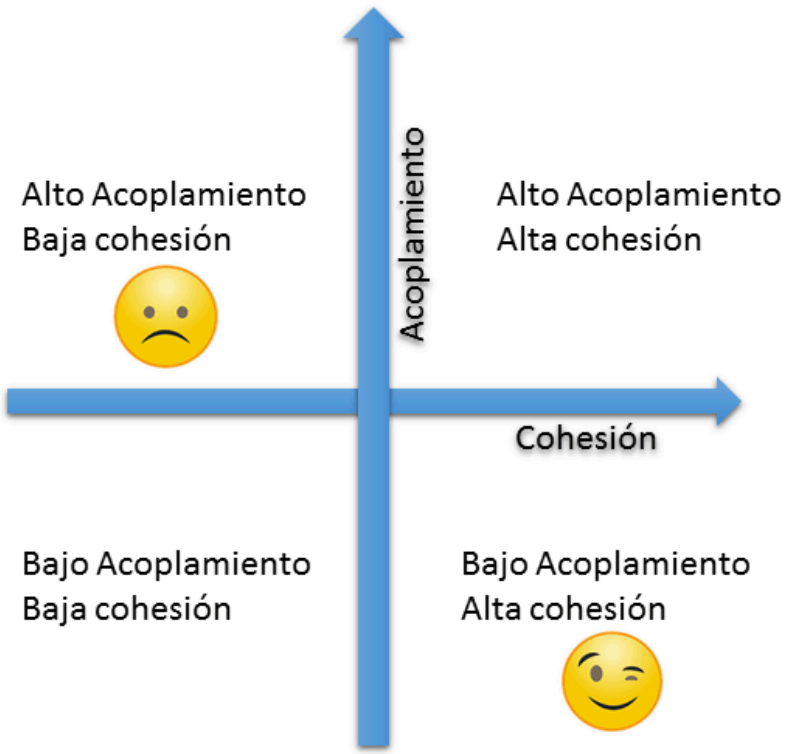
Centrarse sólo en el acoplamiento también puede tener el riesgo de atomizar la estructura. Componentes muy simples y pequeños, con la responsabilidad bien delimitada, pero que acabaran reutilizándose en distintos contextos, por lo que afectarán a la cohesión del programa y empezarán a acoplarse entre módulos.

Además, **una estructura altamente desacoplada será un código menos entendible**, menos legible, ya que para comprenderlo necesitamos explorar muchas referencias de distintos módulos.

Evidentemente, un programa modularizado no puede tener todos sus módulos independientes, puesto que serían distintos programas, pero siempre hay que intentar **minimizar el acoplamiento**.

¿Qué es lo ideal?

Teniendo en cuenta todo lo expuesto, hay que buscar las ventajas de cada uno y encontrar el balance que garantiza mayor simplicidad y coherencia. A modo gráfico, la tendencia que deberíamos buscar sería movernos siempre hacia el cuadrante inferior de la derecha, buscando bajo acoplamiento y alta cohesión.



Acoplamiento vs Cohesión

Un bajo acoplamiento nos garantiza:

- Mejorar la **mantenibilidad** de los módulos del software, facilitar los cambios en el software sin tener que revisar todos los módulos dependientes.
- Mejorar la **reutilización** de las unidades del software.

- Facilitar las **pruebas unitarias** de cada módulo, al ser más independientes.

Por otro lado, una alta cohesión nos permite:

- Tener un **código mas entendebile**, legible y coherente.
- Mejorar la **reutilización**, al tener todo lo relacionado con una cosa, en esa cosa, y no disperso entre módulos.
- Mejorar el **mantenimiento del software,** ya que todo está perfectamente localizado y los cambios en un módulo no afectarán a módulos externos.
- Facilita las **pruebas de caja negra**, ya que toda la funcionalidad relacionada con una cosa, está encerrada en esa cosa.

Existen técnicas y buenas prácticas para limitar el acoplamiento entre componentes, como la Ley de Demeter o el principio «Tell, Don't Ask«. Es difícil aplicarlo en el día a día del desarrollo, aunque es bueno tenerlo en consideración.

Métricas de Acoplamiento y Cohesión

Un aspecto importante a tener en cuenta, ahora que ya sabemos en qué consiste el acoplamiento y cohesión, es saber cómo medirlos.

Existen dos métricas importantes propuestas por Chidamber y Keremer en 1994, conocidas como **métricas CK** para la Programación Orientada a objetos: **CBO** y **LCOM**.

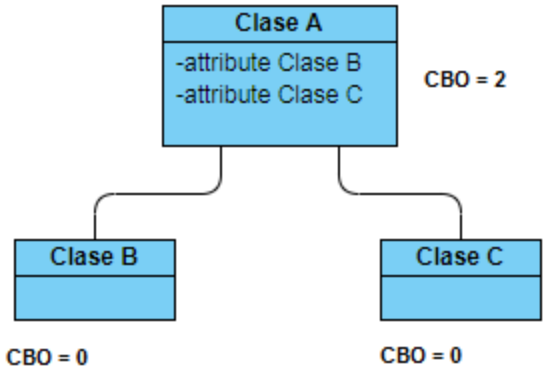
Métrica de acoplamiento entre clases CBO (CBO Coupling Between Object Classes)

Esta métrica calcula el nivel de acoplamiento entre clases, que pretende ser un indicador de esfuerzo necesario para el mantenimiento y testeo. Es evidente que el nivel de acoplamiento entre las clases de un desarrollo es inversamente proporcional a la facilidad de mantenimiento del desarrollo.

Definición del CBO de una clase

El CBO de una clase se define como el número de clases a las cuales está ligada, sin tener en cuenta la relación de herencia. La dependencia se da cuando una clase usa métodos o atributos de otra.

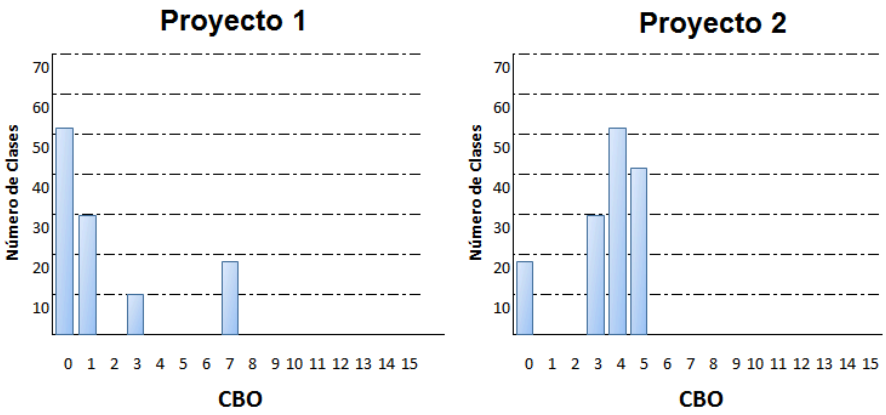
Los accesos múltiples a la misma clase computan como un acceso y sólo se cuentan las llamadas y referencias a variables. No computan atras referencias como: constantes, llamadas a APIS, manejadores de eventos, uso de tipos definidos o instalaciones de objetos. Pero, si una llamada es un método polimórfico, todas las clases consideradas se incluyen en el recuento.



CBO para tres clases asociadas

En resumen, dos clases estarán acopladas cuando los métodos declarados en una clase usen métodos o variables instancia definidos por la otra clase.

Con este cómputo se suele generar un histograma de análisis de un proyecto, para evaluar el esfuerzo de mantenibilidad/rehusabilidad del software. Para ello se obtienen todas las clases con sus valores de CBO, y el recuento de clases. Por ejemplo, para comparar la mantenibilidad de dos proyectos, obtenemos el siguiente histograma de CBO:



Histograma CBO de dos proyectos

En él podemos observar que el primer proyecto tiene un mayor número de clases sin dependencias, o independientes. También unas 30 clases con CBO de 1, y 10 con CBO de 3 , etc.

En el segundo proyecto observamos un mayor número de clases con 3, 4 y 5 dependencias. Por tanto, el proyecto 1 es mas reutilizable y mantenible que el proyecto 2.

Por otro lado, hay que asumir que un programa con muchas clases independientes tiene peor legibilidad. Es decir, es mas difícil entender su diseño, al haber pocas relaciones.

CBO es una métrica de acoplamiento, y como ya vimos, hay que buscar un mínimo el acoplamiento. Por tanto **lo recomendable es mantener un CBO tan bajo como sea posible**.

Cuanto menor sea el valor de CBO mayor independencia tendrá la clase y se podrá reutilizar y mantener mas fácilmente. Cuanto mayor sea el CBO, mas complicada será el mantenimiento y las pruebas.

Métrica de Falta de Cohesión (LCOM Lack of Cohesion)

La Métrica de Falta de Cohesión en los Métodos (LCOM) es una medida de lo cohesionada que es una clase, a partir del número de atributos comunes usados por diferentes métodos de la misma. Indica la calidad de abstracción de dicha clase.

LCOM se calcula a partir del **número de pares de métodos que no tienen atributos en común, menos el número pares que si los tienen**.

Es decir:

LCOM = NPNS – NPS

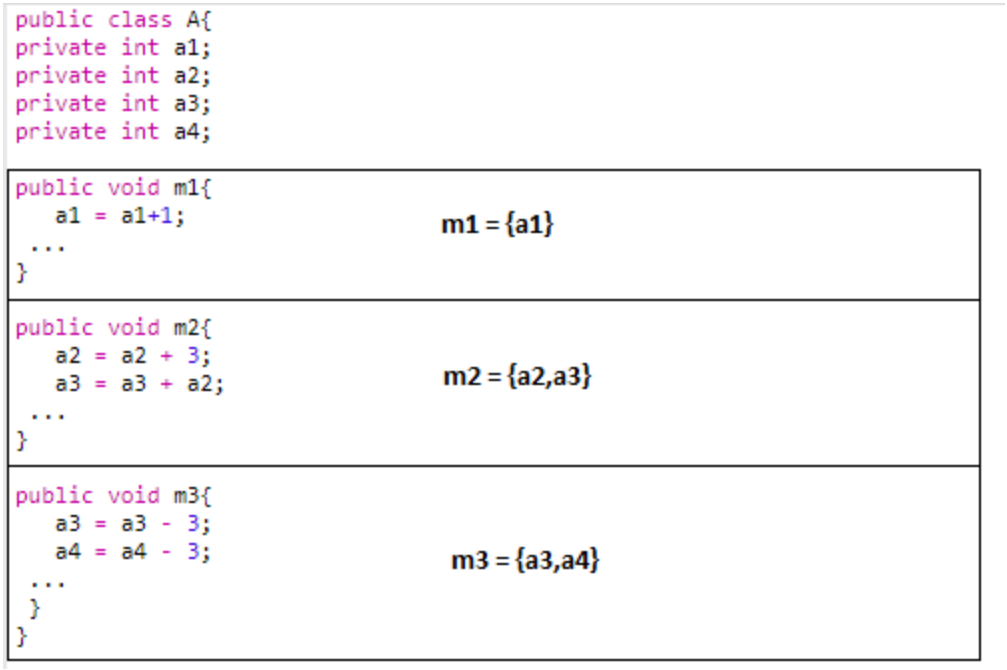
donde:

NPNS = Número de pares no similares.

NPS = Número de pares similares.

En caso de que NPS sea mayor que NPNS, el valor de LCOM es cero.

Veamos un ejemplo de cálculo de esta métrica:



Ejemplo de métrica LCOM sobre una clase A

La clase A tiene tres métodos, por lo que existen tres pares de métodos (m1,m2), (m2,m3), y (m1,m3).

Los métodos m2 y m3 tienen atributos en común (a3) por tanto, el valor de NPNS = 2, los pares (m1,m2) y (m1,m3). El valor de NPS = 1, en este caso el par (m2,m3).

LCOM = 2 -1 = 1

Un valor alto de LCOM implicará falta de cohesión. Esto viene a significar que los métodos de la clase tienen poca similitud, poca cohesión. Por tanto, un LCOM alto indicará que la clase está compuesta por elementos no relacionados, lo que **incrementará la complejidad y la probabilidad de errores**.

Una alta cohesión será sinónimo de encapsulación, de una clase bien definida, cohesionada y con sus elementos relacionados entre sí.