Diseño (1)

Alejandro Teruel

23 de abril de 2013

Contenido

Principios de diseño: Acoplamiento y cohesión experticia [falta desarrollar este concepto]. Dividir para conquistar.

Introducción

Un principio clásico de diseño de sistemas de cierta complejidad es el principio de *dividir para conquistar*:

- Descomponer (recursivamente) el sistema en subsistemas que puedan ser abordados por separado e integrados posteriormente con facilidad

Usualmente la integración es más fácil mientras más simples, reducidas y débiles sean las conexiones entre un subsistema y otro. El acoplamiento es el grado de interconexión entre dos elementos (objetos, módulos, componentes, subsistemas...).

Por otro lado es más fácil entender cada subsistema en la medida,no sóloque dependa poco del resto del sistema, sino que presente *cohesión*, es decir que las partes del subsistema contribuyan a un propósito claro común.

Los objetos y subsistemas de software están conectados por la información que se intercambian. Hay mayor dependencia entre objetos y subsistemas que se intercambian más información. Bertrand Meyer[1] expresa esta idea en la forma de una regla (La Regla del Acoplamiento Débil/Interfaces Pequeñas):

> Si dos módulos se comunican, deben intercambiar el mínimo posible de información.

Comunicar el mínimo posible de información, es un mecanismo de defensa ante posibles cambios a futuro, pues ayuda a contener esos cambios. Si tengo que cambiar la naturaleza o forma del intercambio de la información, tal cambio afectará más al resto del sistema, en la medida que se tengan más canales de comunicación y estos irradien más extensamente en un efecto que ha sido denominado *de cascada (ripple effect)*.

Adicionalmente es más fácil tomar un módulo con poco o débil acoplamiento para (re)utilizarlo en otro contexto o sistema.

Analicemos la idea de *acoplamiento* en forma más concreta para luego pasar a analizar *cohesión*.

Acoplamiento

Comenzaremos considerando la relación que existe elementos del software como rutinas o métodos de programación.

El acoplamiento de una rutina o método con su entorno se puede dar por tres mecanismos: el acceso y modificación de variables globales, la invocación y retorno del método o la lectura y escritura a una

1 Bertrand Meyer: Object Oriented Software Construction.

base de datos o alguna estructura persistente.

Acoplamiento a través de variables globales

Pasar información a través de variables globales constituye un acoplamiento fuerte. Potencialmente todos los métodos en el alcance de una variable global podrían estar accesando y modificando la variable. Estos accesos y modificaciones fácilmente pasan desapercibidos por el lector del código, puede adoptarse un protocolo implícito de uso y modificación que es fácil de olvidar o incumplir[2]. De hecho,la práctica muestra que el uso de variables globales termina siendo particularmente propenso a errores a medida que el software envejece y acumula modificaciones. Metafóricamente es un acoplamiento que se "mete dentro del código"; una invocación se ha comparado con un cable que se enchufa a la interfaz de una rutina, mientras que el uso de variables globales se ha comparado con un cable que se suelda dentro del cuerpo de una rutina: evidentemente es más fácil cambiar un cable que se enchufa al exterior de u objeto que uno que está soldado al interior del mismo. El uso de variables globales complica la verificación y el mantenimiento futuro del código y por ende, se recomienda evitar su uso en la medida de lo posible.

Acoplamiento con pase de datos

Es preferible el acoplamiento mediante el pase de parámetros en una invocación. Sin embargo no es suficiente con pasar parámetros, es importante pasar sólo la información que sea necesaria. Pasar más información que la necesaria complica el proceso de entender la función (¿por qué se pasan esos parámetros? ¿Son realmente innecesarios?) y puede tentar a un futuro mantenedor a usar información que aumente la dependencia del método sobre su entorno o, si son parámetros que se pasan por variable, a cambiar sus valores, comprometiendo aún más la legibilidad y la correctitud futura del programa. Pasar parámetros innecesarios "por si llegan a ser necesarios en el futuro" es una pésima práctica y ha sido fuente de inseguridades explotadas posteriormente por virus y *malware*.

Un número elevado de parámetros tampoco ayuda a la comprensión; muchos menos reducir artificialmente ese número empaquetando sus valores en estructuras u objetos ad-hoc, con poca o ninguna cohesión (**Acoplamiento con estampado**).

Hay que estar pendiente de valores u objetos "errantes", que se incluyen como parámetro en una cadena de invocaciones, y que pasan de un método a otro. Los datos errantes contribuyen claramente a efectos de cascada cuando ha de cambiarse su tipo o significado.

> Ojo: Un dato "errante" puede ser el resultado de un intento mal pensado de eliminar una variable global.

Ejercicio

¿Cuál es el diseño más adecuado para una función que regresa la raiz cuadrada de un número de punto flotante:

- *public float sqrt()*
- *public float sqrt(float x)*
- *public double sqrt(double x)*

2 Suponga que al probar un código que incluye nueve rutinas que pueden modificar una variable global, usted se da cuenta que el valor final de esa variable es incorrecto. ¿Cómo identifica el o los defectos que conducen a esa falla?

- *public double sqrt(float x; double xx; boolean doFloat)*
- *public FloatAndFloat sqrt(float x; double xx; boolean doFloat)*
- *public FloatAndDouble sqrt(float x; double xx)*
- *public FloatDoubleAndBoolean sqrt(float x; double xx)*

En Java, sqrt es un método estático de la clase Math que toma un parámetro tipo double y devuelve un valor tipo double. ¿Está de acuerdo que ello constituye la mejor opción? ¿Qué pasa si usted le pasa un valor positivo de tipo float? ¿Y si le pasa como parámetro un valor tipo double, pero negativo?

Acoplamiento con pase de control

Tampoco está bien visto que un parámetro se le pasa a un método sólo para indicarle qué debe hacer . Este es un ejemplo donde se está separando la detección de una situación y la activación real de acciones necesarias para atenderla. Evidentemente es más difícil leer, entender y mantener un programa donde ocurre este tipo de separación. Típicamente encontraremos este tipo de acoplamiento asociado a un parámetro tipo *bandera booleana*. Una excepción a esta regla ocurre cuando se tiene que recoger información sobre una situación compleja que requiere ser analizada por varios métodos para decidir, con base en esa información más completa, qué hacer. Hay por ende, una distinción sutil entre una bandera que *ordena* hacer una acción (acoplamiento por control) y una bandera que describe (quizás parcialmente) una situación. Muchas veces el caso del acoplamiento con pase de control se delata cuando resulta natural describir la bandera con un *verbo imperativo* (por ejemplo *levantarExcepciónX* o *DescartarEntrada*).

Acoplamiento híbrido

Peor aún es que un parámetro se use para pasar datos y control a la vez. Esto ocurría con cierta frecuencia en algunos software administrativos, donde se reservaban bloques de valores para activar acciones especiales. Por ejemplo un parámetro denominado "teléfono" puede transmitir números de teléfonos excepto para los valores por encima de cierto umbral deben ser interpretados como comandos para, digamos, controlar o revisar ciertas propiedades de las centrales telefónicas por las que pasa. En una época se diseñaban códigos, donde cada subsecuencia de caracteres o dígitos tenía un significado especial: por ejemplo podría concebirse un número de carnet para estudiantes en el que los primeros dos dígitos denotan los últimos dos dígitos del año de ingreso a la institución, otro dígito denota la sede de adscripción, otro dígito el sexo, otro la etnia y así sucesivamente. Esto demostró repetidas veces ser una pesadilla en el largo plazo -en el tiempo estas cuidadosa codificaciones tienden a olvidarse, por lo que en algún momento se expide un código sin pensar en su estructura, produciendo así efectos sorprendentes.

Ejemplo[3]

Una compañía de ventas por Internet reserva los númros de cuenta del 0000 al 9000 para clientes "normales", los números del 9001 al 9500 para clientes "especiales" a quienes les da treinta días adicionales de plazo para pagar y los números de cuenta del 9501 al 9700 para acciones especiales -por ejemplo 9501 significa enviar publicidad a la región 01 de ventas..

El mantenimiento de un sistema basado en una codificación de este tipo rápidamente se vuelve bizarro.

3 Tomado de Meilir Page-Jones: *The Practical Guide to Structured Systems Design.* Prentice-Hall, 1988.

Imagínese que un día se requiere crear un nuevo grupo de clientes a los que se les da un descuento especial. Siguiendo la "lógica" de la codificación debe apartarse un nuevo bloque de cuentas, digamos del 9701 al 9800 para tales clientes[4]. ¿Cómo manejamos clientes que, además de recibir los treinta días de plazo se les quiere conceder un descuento? ¿Apartamos otro bloque de códigos para clientes "super-especiales?

Peor aún, suponga que a la compañía le va viento y popa y un día se presenta su cliente "normal" número 9001. ¿Qué código le damos? Si le damos el 9001, podemos terminar asignando el mismo código a dos clientes diferentes. Si le damos un valor mayor, el supervisor que asigna el número se recordará que le está dando un estatus especial?

Acoplamiento externo

El acoplamiento externo se da cuando dos módulos comparten un formato de datos, un protocolo de comunicación o una interfaz impuesto externamente. Tal tipo de acoplamiento vuelve particularmente vulnerables ambos módulos a cambios en el formato, el protocolo o la interfaz.

Ejercicio:
Considere el siguiente método

public boolean *procesar(Pedido p).*

El primer componente de *p* contiene datos sobre la orden incluida en el pedido, la segunda una bandera booleana *expedita* que indica si es un pedido que debe expeditarse o no y la tercera otra variable (entera) que indica, mediante un código numérico, el tipo de *descuento* que debe aplicarse al elaborar la factura. ¿Qué tipo de acoplamiento presenta? ¿Qué nos sugiere sobre el diseño?

Lo primero que podría saltar a la vista es que hay un *acoplamiento por estampado* -a una orden, presumiblemente un objeto o alguna estructura compuesta- se le han juntado dos variables adicionales, de manera que, al menos a primera vista, luce ad-hoc. Si éste es el caso -y necesitamos más conocimiento sobre el dominio de la aplicación y el significado de "orden" y "pedido" en el entorno empresarial para estar seguro de ello- sería preferible rediseñar el método como:

public boolean *procesar(orden o;* ***boolean*** *expedita;* ***int*** *descuento*)

Más parámetros, pero mayor claridad incluso para entender la posible funcionalidad de *procesar.* El uso de la bandera *expedita* apunta hacia un posible *acoplamiento con pase de control.* De hecho es fácil pensar que la bandera hubiera podido llamarse *expeditar* (verbo en infinito). Tendríamos que revisar el cuerpo del método para saber si se trata de una bandera *descriptiva* o de *control*, pero de buenas a primera luce más a bandera de control.

Finalmente analicemos *descuento.* Para comenzar es un mal nombre para su propósito ya que no indica el monto o tasa de descuento (que es lo que naturalmente asumiríamos) sino que connota un código, por lo que en todo caso sería preferible que se denominara *codigoDescuento*. Sin embargo, el uso de esta variable levanta muchas posibles preguntas:

4 Nótese que esta solución puede involucrar cambiarle a algunos clientes ya establecidos su número de cuenta. Para implantar este cambio, la compañía puede verse obligada a dorarle la píldora a sus clientes, ya acostumbrados a un número de cuenta con una campaña promocional estilo "Usted se ha ganado una de nuestras cuentas doradas con jugosos descuentos". Pero, y si luego de varios meses, la compañía quiere quitar al cliente del grupo que recibe el descuento especial?

1. ¿Realmente se usará dentro del método o estamos en la presencia de un parámetro *innecesario* o un dato *errante*?
2. ¿El procesamiento de un pedido debe incluir el cálculo de su costo? ¿No sería mejor separar el impacto del descuento para mejorar la cohesión del método, eliminando así el parámetro *codigoDescuento* y reduciendo el acoplamiento de este módulo?
3. ¿Se estará manejando la variable *codigoDescuento* como una variable de control?
4. Se tiene un *acoplamiento externo*, ya que este método depende de algo en el exterior que asocia el código con el descuento a dar: un cambio en el formato de tal estructura probablemente involucrará un cambio en la programación del método (a menos que se trate de un dato errante).
5. El código luce potencialmente inseguro. ¿Estará definido un descuento para todo los valores de enteros que puede tomar *codigoDescuento*? En particular, ¿qué pasa si se invoca el método con un *codigoDescuento* negativo o que tome el valor del máximo entero representable?
6. Si se agrega un nuevo código de descuento, ¿habrá que cambiar el código del cuerpo del método?
7. El uso del tipo entero para *codigoDescuento* es cuestionable. Presumiblemente, sólo un subconjunto de enteros tiene asociado un descuento, ¿no sería mejor y más elegante utilizar un tipo ***enum***? Aquí entraríamos en un tema que desarrollaremos posteriormente, la *refactorización* de código., que está relacionada con la *calidad* de la programación.
8. El método, *procesar* regresa un valor booleano, presumiblemente indicativo si puedo procesar exitosamente el pedido o no. Sin embargo procesar un pedido presumiblemente debe tener otros efectos sobre el estado de la orden o pedido, sobre el inventario, sobre el *procesador* de pedidos. Esto sugiere que el método tiene otros acoplamientos con su entorno que no hemos visto aún.

Acoplamiento por base de datos

El acoplamiento vía el acceso y modificación de una base de datos o algún otro mecanismo de memoria persistente tiene sus propias reglas y su problemática bien características. En este caso se pueden presentar situaciones como:

- Es difícil averiguar quién colocó o modificó un dato en la base;
- Se separa, en el tiempo y en código, quienes escriben datos de quienes los leen .

Es importante destacar que justamente por estas dificultades es que los manejadores de bases de datos incluyen mecanismos de bitácora (*logs*) que permiten llevar un registro de las transacciones sobre la base de datos.

Acoplamiento en programación orientada por objetos

Los métodos y variables de instanciación públicos que define una clase definen un acoplamiento entre los objetos de esa clase y su entorno.

Acoplamiento a través de variables públicas de clases

Algunos autores recomiendan ocultar toda variable de instanciación *X*, es decir definirlas siempre como privadas, proporcionando en su lugar dos métodos, uno para accesarlas, *getX()*, y otros para modificarlas, *setX()*. Como casi toda regla de diseño, esta recomendación debe ser calificada de acuerdo con el contexto para alcanzar un balance razonable entre la facilidad para entender el código y su flexibilidad. ¿Por ejemplo por qué privatizar una *constante pública* para sustituir su acceso por una llamada a un método *get()?* En el caso de una variable, debemos recordar que la idea de fondo que justificaría su acceso mediante métodos *get()* y *set()* es facilitar flexibilidad a futuro. De este modo si es previsible que a futuro se modifique el *tipo* de esa variable, el uso de *get()* y *set()* pueden ayudar a contener tal cambio al traducir el valor de la variable de su tipo interno a su tipo externo y viceversa. Esto es particularmente pertinente en el caso que estemos devolviendo un componente de una variable con estructura -al vernos obligados a cambiar la estructura en el futuro (por ejemplo cambiar una lista a un arreglo o a una tabla *hash*), el uso de *get()* y *set()* contienen el cambio sin problema alguno, evitando que necesitemos cambiar el código externo a la clase que buscaba accesar el valor de la variable.

Acoplamiento de los métodos públicos de una clase con su entorno

Por analogía, los métodos públicos de una clase son como los parámetros de una rutina; acoplan a los objetos de la clase con su entorno. Por ende, en igualdad de condiciones, preferiremos que el número de métodos asociados a una clase no sean excesivos. Varios estudios sobre las capacidades cognoscitivas de las personas sugieren que el número máximo de ítemes a recordar o manejar está entre 5 y 9 (siete más o menos dos). Esto es válido en la medida que no se puedan agrupar los ítemes en grupos, pero en cada nivel de agrupamiento seguiría aplicando la heurística del *siete más menos dos*. Esta heurística la podemos ver en acción en los números telefónicos. Observe que cuando los números telefónicos sobrepasan 7, generalmente distinguimos un código de área, o una identificación de empresa de telecomunicaciones del resto de las cifras[5]. Adicionalmente tendemos a preferir, en igualdad de condiciones, que la semántica del método sea más sencillo. Esta preferencia frena la truculencia de intentar reducir el número de métodos, empaquetando dos en uno -lo que probablemente obligaría adicionalmente a introducir un acoplamiento por control en el nuevo método para poder indicar cuál método "primitivo" queremos invocar. En el fondo, el principio que impone límites a una reducción artificial de acoplamiento, es el principio de *cohesión* que veremos más adelante.

Acoplamiento entre los métodos de una misma clase

Los métodos de una clase usualmente accesan a las variables de instanciación de la clase. La relación entre tales métodos y variables recuerdan el mecanismo de variables globales, aunque limitado al interior del objeto. En general aceptamos que el acoplamiento que se da entre los elementos internos de un subsistema (por ejemplo el objeto) sea más fuerte que el acoplamiento que se dé entre un elemento interno y otro externo. De todos modos, el hecho que se puedan particionar a los métodos de una clase de acuerdo con las variables de instanciación a que necesiten accesar, puede ser un indicio de que la clase debe particionarse en varias.

Algunos métodos de una misma clase pueden estar acoplados externamente, es decir tienen que accesar

5 También es cierto que en la medida que dependemos más del directorio de número almacenados en el teléfono celular, hacemos menos esfuerzo por recordar el número. Los mecanismos de búsqueda de número (primeras letras del nombre, o apellido, apodo, etc. sustituyen entonces la estructura del número telefónico.

un parámetro cuyo formato, estructura o protocolo de acceso está definido externamente. Esta situación, es más aceptable si la definición del formato, estructura o protocolo está encapsulado dentro de la propia clase, de nuevo siguiendo la lógica de permitir mayor acoplamiento dentro de un subsistema que a su entorno.

El orden en que se definen los parámetros formales de una rutina o método también plantea, una estructura o protocolo de acceso que debe respectarse en el cuerpo del método. Por ejemplo, es usual que el primer elemento de una coordenada cartesiana corresponda al eje *X* y la segunda al eje *Y.* Invocar a una rutina que maneja tal tipo de coordenadas, para luego, en el cuerpo del método invirtir, por error, los valores de las coordenadas puede producir fallas graves (piense en un sistema de navegación GPS por ejemplo). Para facilitar la comprensión de código es muy aconsejable especificar el orden los parámetros en los diferentes métodos asociados a una misma clase, de forma consistente, siguiendo, en lo posible, la misma lógica y las convenciones usuales (el valor de la coordenada en el eje X siempre antes que el correspondiente a la Y , y éste antes que el de la Z). Esta consistencia reduce la posibilidad de que quienes programen invocaciones a los métodos cambien inadvertidamente el orden de los parámetros -defectos que es particularmente difícil de detectar cuando los parámetros cambiados tienen el mismo tipo. Las personas tienden a percibir lo que estamos acostumbrados a percibir [**video del mono] -aprovechemos esta fortaleza y evitemos ser espúreamente creativos programando contra nuestros hábitos y estructuras cognoscitivas.

Acoplamiento por herencia

Una clase que hereda de otra también presenta un acoplamiento asimétrico (el hijo está acoplado al padre pero en la programación orientada a objetos, el padre no conoce a sus posibles herederos). El tema de acoplamiento por herencia lo trabajaremos más adelante en el curso.

Ejemplo

Supongamos que se define una clase *Lámpara* un método *modificarEstado* que toma un parámetro cuyo valor determina si la lámpara se prende, se apaga o se vuelve intermitente. Observe primero que *modificarEstado* es un método con acoplamiento con pase de control. Se considera preferible rediseñar la clase para que presente tres métodos diferentes *prender(), apagar()* y *activarIntermitencia().* Note que tal rediseño nos permite también pensar en distinguir entre una superclase *Lámpara* con sólo dos métodos (*prender()* y *apagar()*) y una subclase de Lámpara (*LámparaIntermitente*) que agregue el método *activarIntermitencia()* a los dos métodos heredados.

La búsqueda de mayor flexibilidad de acoplamiento

> El propio mecanismo de invocación de rutinas de un lenguaje como Java fuerza un nivel de acoplamiento que otros lenguajes o mecanismos flexibilizan. Por ejemplo, posponer la resolución del nombre de la rutina al cuerpo de la misma (*binding*) permite mayor flexibilidad y reduce el acoplamiento. Este tema representa una diferencia fundamental entre los lenguajes de programación *compilados* y los *interpretados.* No en balde los lenguajes denominados *script languages,* como *Python, Perl* o *Lua,* que valoran la flexibilidad son interpretados y no compilados.
>
> Al invocar una rutina o método, también sería más flexible permitir que los parámetros puedan pasarse en formas diferentes a la forma secuencial en que fueron definidos. Para ello podemos pasar los parámetros actuales etiquetados con su nombre formal. Incluso podríamos omitir algunos parámetros, dejando que éstos tomen valores definidos por defecto. Esta flexibilización en el acoplamiento es típico de *middleware*, como *Corba* o *COM*,

> software desarrollado para facilitar la comunicación eficiente entre programas o procesos desarrollados en diferentes lenguajes de programación y que pueden estarse ejecutando incluso en máquinas diferentes en una red de computadores.
>
> La invocación de rutinas obliga a que el módulo que llame suspenda su ejecución hasta que el módulo invocado termine. Un acoplamiento más flexible lo proporciona el mecanismo de envío no-bloqueante de mensajes entre *procesos* tema fundamental en el cada vez más importante mundo de sistemas concurrentes y en red.
>
> Más adelante en el curso veremos cómo hay ciertos esquemas o *patrones* de diseño que buscan introducir menores niveles de acoplamiento entre los componentes de un software orientado por objetos.

Clases "enmorochadas" (*connascent clases*)

En 1996, Meilir Page-Jones introduje una extensión al concepto de acoplamiento (coupling) que denominó *connascence*[6], un sustantivo en Inglés bastante rebuscada que se refiere al nacimiento o producción conjunto de dos o más objetos, o al acto de crecer juntos. Podemos traducir el término al Castellano como "co-dependencia" o, en "venezolano", "enmorochamiento". La definición sería:

> Dos componentes están *enmorochados* si un cambio en uno de ellos obliga a modificar el otro para mantener la correctitud global del sistema.

Page-Jones define ocho niveles de enmorochamiento entre los que incluye lo que hemos introducido como, de menor grado a mayor grado, acoplamiento de nombre (tener que conocer el nombre definitivo de la rutina o componente a invocar) y acoplamiento de posición (tener que conocer el orden de los parámetros). Page-Jones también distingue como dos tipos de enmorochamiento en el acoplamiento externo, (1) enmorochamiento semántico o por significado y (2) enmorochamiento algorítmico.

Cohesión

La cohesión es posiblemente un término más difícil de precisar que el acoplamiento y está muy ligada a qué tan fácil sea comprender un elemento. En general el principio de la cohesión indica que un elemento (objeto, componente, subsistema) debe tener un propósito claro y coherente. Por ende, los componentes de un subsistema fuertemente cohesionados deben contribuir clara y directamente al propósito del subsistema.

Cohesión y acoplamiento no son conceptos ortogonales y un módulo que presente bajo acoplamiento con su entorno tenderá a mostrar alta cohesión y vice versa; sin embargo, pueden haber circunstancias en que la tendencia no se cumple y hay que buscar un balance entre ambos conceptos.

Por ejemplo ya vimos previamente cómo la búsqueda de un menos número de parámetros o métodos puede llevar a empaquetar varios parámetros o métodos en uno sólo, desmejorando la cohesión que existía previamente -aunque en tal caso también se desmejoró el nivel de acoplamiento al tener que introducir acoplamiento por pase de control.

Los niveles de cohesión más usuales son (de mayor fuerza a menor fuerza):

Cohesión informacional

Se considera la cohesión más fuerte que puede tener una clase. Para tenerla, la clase debe encapsular y ocultar los detalles internos de una estructura de datos o recurso y además cada uno de sus métodos

---

6 Una breve introducción al tema puede encontrarse en http://en.wikipedia.org/wiki/Connascence_%28computer_programming%29#Connascence_of_Meaning_.28CoM.29. Un video de Jim Weirich titulado *Grand Unified Theory of Software Design* (http://vimeo.com/10837903), cubre el mismo material. Para la fecha que se consultó (17 de abril 2013) sólo se podía ver el video en Inglésy sin subtítulos. Se recomienda con reservas recomienda ya que es un poco largo lamentablemente no llega a mostrar las láminas que contienen los ejemplos a que se refiere el expositor.

debe presentar cohesión funcional.

Cohesión funcional

Se considera la cohesión más fuerte que puede tener una rutina o método. Un elemento presenta cohesión funcional si tiene un sólo propósito claro y coherente. Matemáticamente puede especificarse como una función, en lenguaje informal su propósitivo no requiere conjunciones, disjunciones ni otro tipo de conectores gramaticales.

Cohesión secuencial

Un elemento presenta cohesión secuencial si agrupa componentes que se suceden en el tiempo -tipicamente la salida o resultado de un componente sirve de entrada o punto de inicio al siguiente componente. Por ejemplo un método que formatee y valide un registro dado presenta cohesión secuencial. Conectores como "y", "luego", "entonces" en la especificación informal de la rutina usualmente indican que estamos en presencia de este tipo de cohesión. Formalmente, la descripción natural suele obtenerse a partir de la composición de dos funciones -y el orden de la composición suele ser importante. Curiosamente a veces basta con agregar algunos componentes secuenciales para pasar de cohesión secuencial a funcional. Por ejemplo, un método decrito como *leer, formatear y validar un registro*, que luce superficialmente como un módulo con cohesión secuencial, puede retar tal tipificación y ubicarse como de cohesión funcional al notar que una descripción más acetada pudiera ser *obtener registro válido*.

Cohesión comunicacional

Un elemento presenta cohesión comunicacional agrupa componentes que trabajan sobre un elemento común de entrada, pero donde lo que hace un componente es conceptual y temporalmente independiente de lo que hace el otro. Así por ejemplo un método que genera un reporte del salario máximo pagado y su relativa frecuencia a la vez que calcule el salario promedio presenta cohesión comunicacional. La justificación aparente de esta decisión de diseño puede a veces plantearse en términos de aparente eficiencia: aprovechamos el mismo lazo para hacer ambas cosas. Sin embargo al programador de mantenimiento le va a costar entender el entremezclado lógico de código ya que agrupar las dos funciones no es natural. Nótese que la especificación matemática más natural sería en términos de dos funciones, cada una de las cuales se define sobre la misma entrada pero dando cada una dos resultados. Los resultados son los que se componen, no las funciones...

Cohesión temporal

Un elemento presenta cohesión temporal cuando el criterio de agrupación es que los componentes deben ejecutarse todos antes o después de algún evento. Se puede caer en este tipo de cohesión cuando se agrupan actividades de inicialización o terminación. Considero que la programación dirigida por eventos o excepciones puede constituir un contexto de excepción importante a la baja consideración que se pueda tener para este tipo de cohesión.

Cohesión lógica

Un elemento presenta una cohesión lógica si la afinidad entre sus componentes es que todos se enmarcan dentro de una misma categoría. Una biblioteca o librería de rutinas matemáticas presenta una cohesión lógica. Tal tipo de cohesión puede tener sentido para macro-componentes, pero es inaceptable en el contexto del diseño de clases o rutinas.

Cohesión casual

El elemento no presenta ninguna cohesión más allá de un agrupamiento arbitrario de código, cuyo

único propósito discernible pudiera ser evitar escribir dos o más veces el mismo código.

Ejercicio

Considere el diseño de una clase denominada *Contador* que presenta tres métodos públicos: uno que permite reinicializarlo en cero *reiniciar()*, otro para incrementarlo en uno *incrementar()* y otro para dibujarlo en pantalla, *mostrar(tamaño, color, mostrarConLetras,...).* ¿Qué tipo de cohesión considera Ud. que muestra este diseño?

Para pensar

Considere la clase *Fecha* y la clase *EncabezadoDeCorreoElectrónico.* Los objetos de tipo fecha incluyen un día, un mes y un año, mientras que los encabezados de correo electrónico incluyen un remitente, un destinatario, información sobre el camino que ha seguido el mensaje por el sistema electrónico y un mensaje. Diseñe ambas clases maximizando la cohesión que presenta. ¿Considera usted que ambas clases alcanzan el mismo nivel de cohesión?

Lecturas adicionales

- Meilir Page-Jones: *The Practical Guide to Structured Systems Design* (2° edición, Prentice-Hall, 1988) todavía conserva cierta frescura en su estilo aunque sólo presentaba cohesión y acoplamiento en el contexto de programas estructurados por funciones y procedimientos.
- Edward V. Berard: *Essays on Object-oriented Software Engineering: Volume I.* Prentice-Hall, 1993. Los capítulos 6 y 7 analizan de manera meticulosa y precisa la evolución y distinción de conceptos como encapsulamiento y ocultamiento de información (capítulo 6) y cohesión y acoplamiento, particularmente en el contexto de programación orientada por objetos (capítulo 7). La presentación hecha en este documento debe mucho al capítulo 7 y aprovecha varios de los ejemplos que plantea. Se recomienda con reservas, ya que la discusión puede requerir de un lector con más experiencia en el tema.

Nota

Este escrito está basado en el capítulo 2 de la *Guía N° 1* escrita como parte del material de apoyo para la asignatura *CI-3711 Sistemas de Programas* de la Universidad Simón Bolívar por Alejandro Teruel (Versión Enero 1993).