Introducción a la Programación Orientada a Aspectos

Juan Manuel Nieto Moreno *
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla, España
nulain@yahoo.es

ABSTRACT
La Programación Orientada a Aspectos (POA) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminado las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así, el término POA es usado en [1] para referirse a varias tecnologías relacionadas como los métodos adaptivos, los filtros de composición, la programación orientada a sujetos o la separación multidimensional de competencias.

1. INTRODUCCIÓN

Muchas veces nos encontramos, a la hora de programar, con problemas que no podemos resolver de una manera adecuada con las técnicas habituales usadas en la programación procedural o en la orientada a objetos. Con éstas, nos vemos forzados a tomar decisiones de diseño que repercuten de manera importante en el desarrollo de la aplicación y que nos alejan con frecuencia de otras posibilidades. Por otro lado, la implementación de dichas decisiones a menudo implica escribir líneas de código que están distribuidas por toda, o gran parte, de la aplicación para definir la lógica de cierta propiedad o comportamiento del sistema, con las consecuentes dificultades de mantenimiento y desarrollo que ello implica. En inglés este problema se conoce como *tangled code*, que podríamos traducir como código enredado. El hecho es que hay ciertas decisiones de diseño que son difíciles de capturar con las técnicas antes citadas, debiéndose al hecho de que ciertos problemas no se dejan encapsular de igual forma que los que habitualmente se han resuelto con funciones u objetos. La resolución de éstos supone o bien la utilización de repetidas líneas de código por diferentes componentes del sistema, o bien la superposición dentro de un componente de funcionalidades dispares. La programación orientada a aspectos, permite, de una manera comprensible y clara, definir nuestras aplicaciones considerando estos problemas. Por aspectos se entiende dichos problemas que afectan a la aplicación de manera horizontal y que la programación orientada a aspectos persigue poder tenerlos de manera aislada de forma adecuada y comprensible, dándonos la posibilidad de poder construir el sistema componiéndolos junto con el resto de componentes.

La programación orientada a objetos (POO) supuso un gran paso en la ingeniería del software, ya que presentaba un modelo de objetos que parecía encajar de manera adecuada con los problemas reales. La cuestión era saber descomponer de la mejor manera el dominio del problema al que nos enfrentáramos, encapsulando cada concepto en lo que se dio en llamar objetos y haciéndoles interactuar entre ellos, habiéndoles dotado de una serie de propiedades. Surgieron así numerosas metodologías para ayudar en tal proceso de descomposición y aparecieron herramientas que incluso automatizaban parte del proceso. Esto no ha cambiado y se sigue haciendo en el proceso de desarrollo del software. Sin embargo, frecuentemente la relación entre la complejidad de la solución y el problema resuelto hace pensar en la necesidad de un nuevo cambio. Así pues, nos encontramos con muchos problemas donde la POO no es suficiente para capturar de una manera clara todas las propiedades y comportamientos de los que queremos dotar a nuestra aplicación. Así mismo, la programación procedural tampoco nos soluciona el problema.

---

* Este documento es parte del proyecto final de carrera de Juan M. Nieto, tutelado por Antonia M. Reina del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

Entre los objetivos que se ha propuesto la POA están, principalmente, el de separar conceptos y el de minimizar las dependencias entre ellos. Con el primer objetivo se persigue que cada decisión se tome en un lugar concreto y no diseminada por la aplicación. Con el segundo, se pretende desacoplar los distintos elementos que intervienen en un programa. Su consecución implicaría las siguientes ventajas:

- Un código menos enmarañado, más natural y más reducido.
- Mayor facilidad para razonar sobre los conceptos, ya que están separados y las dependencias entre ellos son mínimas.
- Un código más fácil de depurar y más fácil de mantener.
- Se consigue que un conjunto grande de modificaciones en la definición de una materia tenga un impacto mínimo en las otras.
- Se tiene un código más reusable y que se puede acoplar y desacoplar cuando sea necesario.

2. ¿NECESITAMOS ASPECTOS?

En muchas situaciones la programación orientada a objetos deja de ser útil y se convierte en un proceso insostenible. Por un lado, hay modelos orientados a objetos que están muy especializados en un determinado dominio de problemas y que no son convenientes para los demás. También hay cuestiones a resolver que no están vinculadas con ningún problema en particular, sino que nos las encontramos por toda la aplicación, afectando a gran parte de ella y que nos obligan a escribir el mismo código por muchos sitios diferentes para poder solucionarlas. Este es el código que hace que nuestra aplicación orientada a objetos bien diseñada vaya siendo cada vez menos legible y delicada.

Por otro lado, están las interfaces de los objetos, clara debilidad en la evolución de los mismos, ya que deben ser definidas desde el principio. Mientras las necesidades del problema no cambien, el diseño con interfaces funciona bien. Pero eso no es lo habitual, sino que los negocios cambian, lo hace el dominio o bien aparecen nuevas necesidades. En tal caso, hay que cambiar las interfaces y ello supone también cambiar mucho código en todas las clases que las implementan. Para realizar tales cambios, el programador se ve obligado a analizar detalladamente la implementación actual, navegando por las diferentes clases para obtener una visión global del problema al que se enfrenta. Veremos en el ejemplo que acompaña a este documento en qué pueden ayudarnos los aspectos y si ciertamente la orientación a aspectos traerá tantos beneficios como los más optimistas auguran.

3. HISTORIA

Muchos observan la programación orientada a aspectos como el siguiente paso en la evolución de la ingeniería del software e intentan hacer ver como ésta supondrá, como lo hizo en su momento la POO, un cambio importante en el diseño de aplicaciones. No obstante, recordemos que fue sobre 1960 cuando apareció la primera implementación usable de los conceptos de orientación a objetos con el lenguaje Simula-68, mientras que no fue hasta 1980, veinte años después, cuando de verás se empezaron a usar de manera comercial en proyectos reales, dichas técnicas, es decir, con mucho dinero por medio. Fue entonces el *boom* de las telecomunicaciones y el crecimiento de C++ frente a C y COBOL.

El concepto de POA fue introducido por Gregor Kiczales y su grupo, aunque el equipo Demeter [3] había estado utilizando ideas orientadas a aspectos antes incluso de que se acuñara el término. El trabajo del grupo Demeter estaba centrado en la programación adaptiva, que puede verse como una instancia temprana de la POA. Dicha metodología de programación se introdujo alrededor de 1991. Consiste en dividir los programas en varios bloques de cortes. Inicialmente, se separaban la representación de los objetos del sistema de cortes; luego se añadieron comportamientos de estructuras y estructuras de clases como bloques constructores de cortes. Cristina Lopes propuso la sincronización y la invocación remota como nuevos bloques [4]. No fue hasta 1995 cuando se publicó la primera definición temprana del concepto de aspecto, realizada también por el grupo Demeter y que sería la siguiente:

"Un aspecto es una unidad que se define en términos de información parcial de otras unidades".

Por suerte las definiciones cambian con el tiempo y se hacen más comprensibles y precisas. Actualmente es más apropiado hablar de la siguiente definición de Gregor Kiczales:

"Un aspecto es una unidad modular que se dispersa por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa".

4. UN PROCESADOR DE IMÁGENES

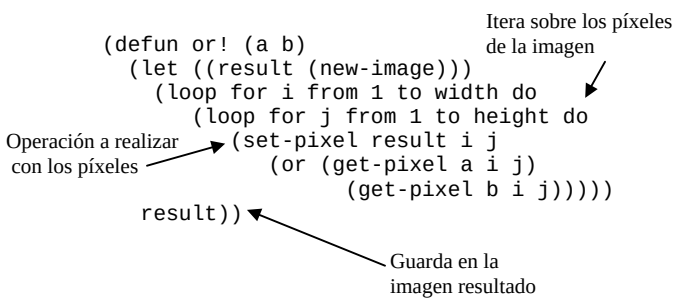
El siguiente ejemplo está sacado del trabajo publicado en 1997 para la *European Conference on Object-Oriented Programming* (ECOOP) celebrada en Finlandia [2]. Supongamos una aplicación de procesamiento de imágenes en blanco y negro. El dominio del problema son las imágenes que deben pasar por una serie de filtros para obtener el resultado deseado. Se podrían hacer tres implementaciones: una que fuese fácil de entender pero ineficiente, una eficiente pero difícil de entender y una tercera, basada en POA, que a la vez sería fácil de entender y eficiente.

Figura 1 Procesador de imágenes

Los objetivos entonces a cumplir para el desarrollo del procesador de imágenes van a ser que sea fácil de desarrollar y de mantener, así como que haga un uso eficiente de la memoria. Los motivos son evidentes: con el primero conseguiremos de forma rápida y libre de fallos futuras mejoras del sistema; el segundo se debe al tamaño de las imágenes, que suele ser elevado, por lo que conviene minimizar el uso de memoria y los requerimientos de almacenamiento.

4.1 DEFINICIÓN DEL PROBLEMA

Aún con la tradicional programación procedural podemos implementar de una forma clara, concisa y apropiada para el dominio del problema nuestro procesador de imágenes, consiguiendo el primer objetivo. De esta forma los filtros pueden ser definidos como procedimientos que toman una o varias imágenes de entrada y producen una imagen de salida. Se tendrían así un conjunto de procedimientos para los filtros más básicos y que servirían para definir los filtros más complicados en base a éstos. Así por ejemplo, un filtro `or!` que tomara dos imágenes y que devolviera una resultado de aplicar esta operación sería implementado en Common Lisp como sigue:



Empezando con este sencillo filtro `or!` y junto con otras primitivas, el programador puede construir filtros más complejos en base a ellos. Así por ejemplo podemos hacer:

– Un filtro que se quede con la parte superior de una imagen:[1]

```
(defun down! (a)
  (let ((result (new-image)))
    (loop for i from 1 to width do
      (loop for j from 1 to height-1 do
   (set-pixel result i j
           (get-pixel a i j+1))))
   result))
```

– Un filtro que tome dos imágenes y devuelva otra con la diferencia entre ellas:

```
(defun remove! (a b)
    (and! a (not! b)))
```

– Un filtro que se quede con la parte superior de una imagen:

```
(defun top-edge! (a)
     (remove! a (down! a)))
```

– Un filtro que se quede con la parte inferior de una imagen:

```
(defun botton-edge! (a)
     (remove! a (up! a)))
```

– Un filtro que se quede con los píxeles negros de las partes superior e inferior:

```
(defun horizontal-edge! (a)
     (or! (top-edge! a)
          (botton-edge! a)))
```

Como puede verse únicamente los filtros más básicos hacen uso de las estructuras `loop` para iterar sobre los píxeles de las imágenes. Los filtros de más alto nivel, tal como `horizontal-edge!`, se expresan en términos de los filtros básicos. El código resultante es fácil de leer, comprender, depurar y extender. Se cumple entonces el primero de los objetivos.

4.2 OPTIMIZANDO LA MEMORIA

La implementación anterior no tiene en cuenta nuestro segundo objetivo de optimización de los recursos de memoria. Cada vez que se llama a un procedimiento, se itera sobre una serie de imágenes y se produce una nueva imagen resultante. Se crean demasiadas nuevas imágenes que en muchas ocasiones sólo sirven de resultados intermedios, lo que conlleva excesivas referencias a memoria y petición de espacio para almacenar, que implica además fallos de caché, fallos de página y todo ello un bajo rendimiento.

[1] La sintaxis de este ejemplo no es Lisp estándar con objeto de que sea más legible.

Para solucionarlo se puede tomar una perspectiva más global del problema, tomando aquellos resultados intermedios como entradas de otros filtros y programar una versión que sintetice los bucles de cada uno de los filtros de manera apropiada para implementar la funcionalidad original, creando el menor número posible de imágenes intermedias. Así tendríamos el siguiente código para el mismo procedimiento anterior `horizontal-edge!`:

```
(defun horizontal-edge! (a)
 (let ((result (new-image))
   (a-up (up! a))
   (a-down (down! a)))
  (loop for i from 1 to width do
   (loop for j from 1 to height do
     (set-pixel result i j
       (or (and (get-pixel a i j)
                (not (get-pixel a-up i j))
           )
           (and (get-pixel a i j)
                (not (get-pixel a-down i j))
           )
  ))))
 result))
```

Comparado con el original, la lógica de este método es mucho más confusa y como se dice en la terminología inglesa, el código está enredado (*tangled*). Lo que se ha hecho es incorporar en un solo método todos los anteriores, eliminando así algunas de las iteraciones, fusionando las operaciones `and!` y `or!`.[2] Y aunque se ha mejorado la eficiencia, esto ha conllevado la pérdida de la clara estructura original.

En un pequeño ejemplo como este, aún podríamos sobrellevar el deterioro de la claridad que ha supuesto esta modificación. Pero en aplicaciones reales la complejidad debida a este tipo de optimizaciones termina por ser un verdadero inconveniente a la hora de seguir desarrollando o manteniendo el código. Este ejemplo proviene de una aplicación real de reconocimiento de caracteres, donde este método es parte de un subcomponente importante. Implementándolo de una manera sencilla, pero ineficiente, se necesitan 768 líneas de código, mientras que la versión optimizada, con la fusión de los bucles, manteniendo resultados intermedios, asignación de memoria en tiempo de compilación y estructuras de datos intermedias para mejorar las prestaciones requiere 35213 líneas de código, que además están totalmente enredadas. Esta implementación es extremadamente difícil de mantener, ya que pequeños cambios en la funcionalidad requieren entender todo el enredamiento de código que se ha empleado. Pero es la única que es práctica debido a las prestaciones. Las figuras 2 y 3 muestran dos diferentes diagramas de la versión sin optimizar del filtro `hotizontal-edge!` La figura 2 presenta una descomposición funcional que se corresponde directamente con el dominio del problema. La 3 es un diagrama de flujo, donde las cajas representan los filtros básicos y las aristas el flujo de datos entre ellos en tiempo de ejecución. Abajo, la caja etiquetada con `a`, es la imagen de entrada.

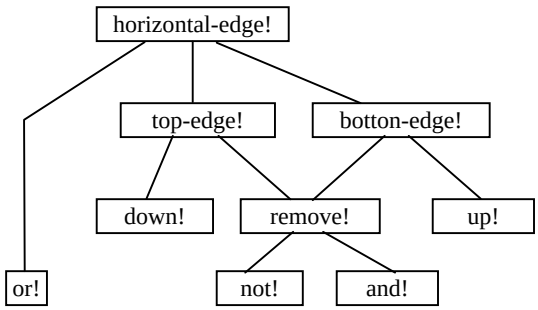


Figura 2 Descomposición funcional de horizontal-edge!

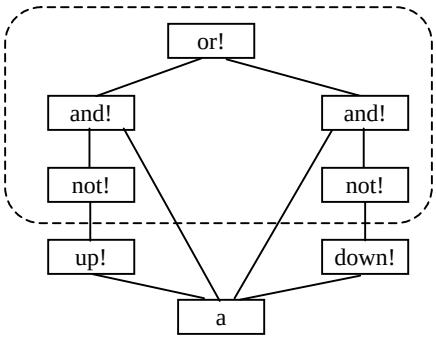


Figura 3 Diagrama de flujo de horizontal-edge!

4.3 CÓMO SE ENTRELAZA EL CÓDIGO

La figura anterior nos da otro punto de vista para entender porqué se enreda el código en la aplicación del ejemplo. A la izquierda se ha presentado la estructura jerárquica de la funcionalidad del filtro. A la derecha, un diagrama de flujo de los datos en la versión original sin optimizar de `horizontal-edge!` En el diagrama, las cajas y líneas muestran los filtros básicos y el flujo de datos entre ellos. La caja ovalada de línea discontinúa muestra la parte que se fusionó en un único bucle en la versión optimizada.

Como se ve, la caja ovalada no incorpora todas las acciones de `horizontal-edge!`. De hecho, no se corresponde con ninguna de las unidades funcionales jerárquicas de la izquierda. Mientras que las dos propiedades que se quieren implementar, la funcionalidad y la fusión de los bucles, ambas provienen de los mismos filtros básicos, tienen que combinarse de diferente forma cuando se fusionan los filtros. La funcionalidad sigue un orden jerárquico a la manera tradicional. La fusión de los bucles se realiza con aquellos filtros cuya estructura cíclica tiene la misma forma y que además, son vecinos

[2] Si se fusionaran también los loops de up! y down! sería entonces cuando pasaría a ser totalmente ilegible. Veremos como la versión con POA sí los fusionará.

directos en el diagrama de flujo de datos. Cada una de estas reglas de composición es fácil de entender viendo cada una de los dibujos anteriores por separado. Pero la relación de composición entre ambas es bastante más complicada y es difícil de observar la manera de componerse de una en el dibujo del otro.

La confusa relación anterior es la que provoca que el código se entremezcle y se debe al sencillo mecanismo de composición que nos ofrecen los lenguajes actuales: llamadas a procedimientos. Este es de utilidad para implementar versiones ineficientes, pero una vez hecho ésto, debido a las diferentes reglas de composición que requieren una intención y otra, nos obligan a combinar el código de una manera artificiosa y como se diría, a mano, terminando en lo que ya hemos dado en llamar código entrelazado o *tangled* en inglés.

En general, cuando se quieren coordinar dos propiedades que afectan de diferente forma al código, es cuando se dice que ambas se entrelazan. Debido a que los lenguajes basados en procedimientos, ya sean orientados a objetos, funcionales o procedurales soportan un solo mecanismo de composición, el programador tiene que hacer la composición manualmente, conllevando esto a la complejidad del código y al código enredado. Esto nos lleva a la definición de dos términos que ayudarán a entender la definición de aspecto. Respecto a una aplicación y a su implementación usando un lenguaje basado en procedimientos, una propiedad se implementará como:

- *un componente*, si puede ser encapsulado de forma clara en un procedimiento, como pueden serlo los objetos, métodos, procedimientos, API... Se entiende de forma clara, está bien localizada, es de fácil acceso y que se deja componer como sea necesario. Los componentes suelen ser unidades de la descomposición funcional del sistema, como filtros de imágenes, la cuenta de un banco o los componentes de las interfaces gráficas.

- *un aspecto*, si no se puede encapsular de forma clara en un procedimiento. Los aspectos no suelen ser unidades funcionales que obtengamos al descomponer el sistema, sino más bien, propiedades que afectan al rendimiento o el comportamiento de los componentes. Como ejemplos de aspectos, podemos citar: patrones de acceso a memoria, sincronismo de objetos concurrentes, control de errores y manejo de excepciones, utilización de caché...

Con estos términos claros podemos ahora establecer cual es el objetivo de la programación orientada a aspectos: proporcionar al programador una técnica para, de una forma clara, separar componentes y aspectos unos de otros[3], dotando de mecanismos que hacen posible abstraer estos para después componerlos dando resultado al sistema final. Mientras que expresado con las mismas palabras, diremos que el objetivo de los lenguajes basados en procedimientos es proporcionar al programador una técnica para separar componentes de componentes (no hay posibilidad de definir aspectos), dotando de mecanismos que hacen posible abstraer estos para después componerlos dando resultado al sistema final.

4.4 CÓMO HACERLO CON POA

Ya que hemos visto la necesidad y utilidad de separar los componentes de los aspectos, podemos pasar a definir una nueva implementación del procesador de imágenes basado en POA. El objetivo de esta sección será explicar cómo es la estructura de una implementación basada en POA, pero no explicarla completamente.

La estructura de una aplicación cuya implementación se basa en POA es análoga a la estructura de una que se base en un lenguaje procedural. Podemos decir que esta segunda consiste en: (i) un lenguaje, (ii) un compilador para dicho lenguaje y (iii) un programa escrito en el anterior lenguaje. Por otro lado, una aplicación orientada a aspectos consiste en: (i.a) un lenguaje de componentes, con el que programar los componentes; (i.b) uno o más lenguajes de aspectos con los que programar los aspectos; (ii) una herramienta para combinar los aspectos y los componentes (*aspect weaver*); (iii.a) un programa que implemente los componentes usando el lenguaje de componentes; (iii.b) uno o más programas que implementen los aspectos usando los lenguajes de aspectos. El proceso de entrelazado de código dependiendo del lenguaje y la herramienta de entrelazado, puede hacerse en tiempo de ejecución (*run-time weaving*) o en tiempo de compilación (*compile-time weaving*).

4.5 EL LENGUAJE DE COMPONENTES

Para este ejemplo utilizaremos un lenguaje de componentes y uno de aspectos. El de componentes es similar al usado para desarrollar la versión sin aspectos anterior, sólo que se le han introducido unos cambios que permiten ver de forma clara cómo trabajan los aspectos. Así los cambios son los siguientes:

- Los filtros no son procedimientos explícitamente.
- Los bucles se definen de otra forma, de manera que la estructura del bucle sea mucho más explícita.

Con tales cambios, el mismo filtro anterior `or!` se escribe de la siguiente forma:

[3] Componentes de componentes, aspectos de aspectos y componentes de aspectos.

```
(define-filter or! (a b)
  (pixelwise (a b)
             (pixel-from-a pixel-from-b)
             (or pixel-from-a pixel-from-b)
  )
)
```

El constructor `pixelwise` es un iterador, que en este caso itera sobre cada uno de los elementos de las imágenes `a` y `b`, asignando en `pixel-from-a` y `pixel-from-b` cada uno de los elementos que recorre y aplicándoles después la operación que sigue, sea en este caso la or, devolviendo finalmente una imagen con el resultado. De igual forma se construyen los otros filtros que implementan las funciones de agregación, diferencia y desplazamiento de píxeles necesarias para la aplicación. El hecho de introducir este constructor para los bucles posibilita que a continuación, el lenguaje de aspectos sea capaz de detectar, analizar y fusionar los bucles de forma mucho más fácil.

4.5 EL LENGUAJE DE ASPECTOS

El diseño del lenguaje de aspectos para este ejemplo se basa en la observación y las conclusiones que se tienen del gráfico del diagrama de flujo de datos, ya que ahí se entiende cómo se pueden fusionar los bucles. El lenguaje de aspectos es un sencillo lenguaje procedural que permite simples operaciones sobre los nodos del diagrama de flujo de datos. Así el programa de aspectos puede detectar los bucles que pueden fusionarse y proceder a fusionarlos. El siguiente fragmento de código es una parte de la aplicación donde se lleva a cabo la fusión de los bucles que tiene la misma estructura y que están seguidos en el diagrama de flujo de datos. Primero comprueba las parejas de nodos del grafo conectadas por una arista que tengan la misma estructura `pixelwise`, esto es la que tiene un `or` o un `and`. Aquellas que encuentre las fusiona en un único bucle, con la misma estructura y que combina de manera adecuada las entradas, las variables del bucle y el cuerpo de los dos bucles originales. Véase el siguiente fragmento:

```
(cond ((and (eq (loop-shape node) ‘pointwise)
            (eq (loop-shape
input)‘pointwise))
       (fuse loop input ‘pointwise
             :inputs (splice …)
             :loop-vars (splice …)
       :body (subst …))))
```

Describir las reglas de composición y fusión para los cinco tipos de bucles que se dan en la aplicación real requiere del orden de una docena de cláusulas similares para indicar cuando y cómo realizarla. Es por ello que no es posible pretender que un compilador haga por sí solo esta clase de optimizaciones, así como por ejemplo compartir resultados intermedios o mantener el uso de la memoria en tiempo de ejecución dentro de unos límites. De momento, los compiladores no han llegado a ese estado.

4.6 PROCESO DE ENTRELAZADO

La aplicación encargada del proceso de combinación de los componentes y los aspectos, en la aplicación real desarrollada en *Xerox Palo Alto Research Center* y que sirve aquí de ejemplo produce como salida un programa en C, tomando como entrada los componentes y los aspectos. El proceso requiere de tres fases como se ilustra en la figura 4.

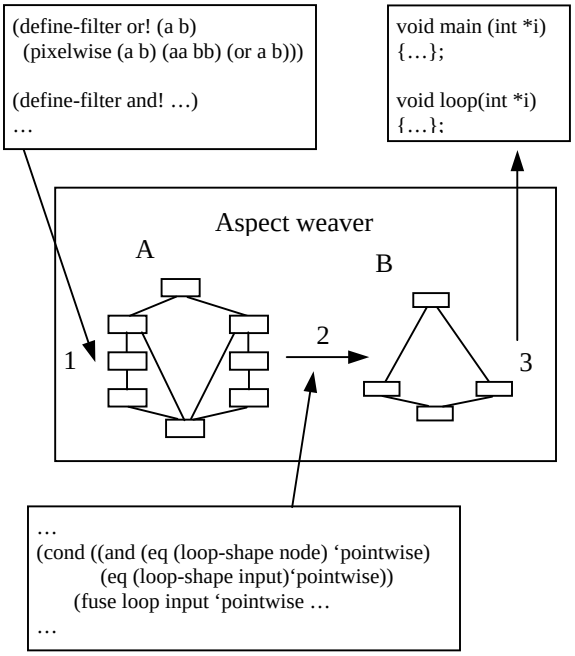


Figura 4 Proceso en tres fases del entrelazador de aspectos (aspect weaver)

En la primera fase el entrelazador de aspectos genera un grafo de flujo de datos con el programa de componentes. En este grafo, los nodos representan filtros básicos y las aristas el flujo de imágenes entre un filtro y otro. Cada nodo contiene un único constructor de bucles como los presentados antes. Así por ejemplo, el nodo etiquetado con A en la figura contiene el siguiente constructor, donde #<...> hace referencia a las aristas que llegan al nodo:

```
(pointwise (#<edge1> #<edge2>) (i1 i2)
   (or i1 i2))
```

En la segunda fase, se utiliza el programa de aspectos para editar el grafo fusionando nodos y ajustando el cuerpo de los filtros de manera adecuada. El resultado es un grafo con menos nodos y donde cada uno de los nuevos nodos tiene ahora más operaciones básicas entre píxeles que antes de esta fase. Por ejemplo, el nodo etiquetado con B, que corresponde a la fusión de cinco bucles del original grafo, tiene el siguiente cuerpo:

```
(pointwise (#<edge1> #<edge2> #<edge3>)
           (i1 i2 i3)
   (or (and (not i1) i2) (and (not i3) i2))))
```

Finalmente, en la tercera de las fases, un sencillo generador de código recorre el nuevo grafo de los nodos fusionados y genera una función en *C* para cada nodo, así como una función `main` que llama a estas funciones en el orden apropiado, pasándoles los resultados a cada una de la función del nodo que le preceda. Esta generación de código es sencilla puesto que cada nodo contiene únicamente un constructor de bucles donde el cuerpo está formado únicamente por funciones básicas entre píxeles.

Un aspecto fundamental de este sistema es que el entrelazador no es necesariamente un compilador muy complejo e "inteligente". Al usar POA hemos dejado que sea el programador quien defina todas las decisiones sobre las estrategias implementación haciendo uso de un lenguaje apropiado de aspectos. El entrelazador no necesita tomar ninguna decisión inteligente. Pedir al programador que se encargue de los aspectos de implementación podría parecer un paso atrás en el arte de programar. Sin embargo, no es tanto así. Cuando por ejemplo un programador está definiendo un aspecto que afecta al uso de la memoria, usar POA supone definir las estrategias de implementación a un nivel apropiado de abstracción, mediante un lenguaje de aspectos adecuado. No se requiere entrar en los detalles de implementación, ni se requiere trabajar directamente con el código confuso enredado. A la hora de evaluar los beneficios de una versión de una aplicación orientada a aspectos es importante compararla con ambas la versión clara e ineficiente y la eficiente pero compleja.

4.7 RESULTADO FINAL

La aplicación real del ejemplo presentado es algo más complicada. Además del aspecto aquí presentado hace uso de dos más: uno para compartir la ejecución de operaciones básicas comunes, reduciendo el volumen de éstas; y otro para asegurar que a la misma vez se tienen en memoria el menor número de imágenes posibles, para optimizar el uso de memoria. Los tres están escritos con el mismo lenguaje de aspectos.

Como hemos visto en el ejemplo, la implementación mediante POA ha conseguido los objetivos propuestos al principio. Se puede razonar fácilmente sobre el código de la aplicación, por lo que es además fácil de mantener y ha sido fácil de desarrollar, mientras que al mismo tiempo es bastante eficiente. Al mantenerse limpio el código de los componentes, es fácil para el programador entenderlos y ver la forma en la que se combinan unos con otros. Igualmente la definición de los aspectos es clara y es sencillo entender como se combinan y su efecto sobre los componentes. Si se introdujeran cambios en los componentes o en el aspecto de fusión estos se reflejarían de manera sencilla sin más que volver a utilizar el entrelazador de código y dejarle hacer su trabajo. Lo laborioso que es escribir los detalles de la implementación se ha eliminado del trabajo del programador, por lo que no tiene que enfrentarse a esos pequeños detalles y minuciosidades que tan complicadas hacen las versiones eficientes.

La versión real de esta aplicación desarrollada en *Xerox Palo Alto Research Center* tiene 1039 líneas de código, contando los componentes y los tres aspectos. El entrelazador de aspectos, incluyendo un componente de generación de código reusable, tiene 3520 líneas[4]. La versión programada a mano requiere 35213 líneas, lo que es una diferencia significativa en tamaño, pensando además que en esta el código es mucho más complejo y delicado, debido a que está todo mezclado: la funcionalidad y las optimizaciones. Eso sí, la versión a mano es más rápida aunque menos eficiente en espacio de memoria. En cualquier caso, la versión orientada a aspectos es unas 100 veces más rápida que la versión sin optimizar.

5. CONCLUSIONES

La separación de conceptos es una herramienta de ingeniería del software que reduce la complejidad de las aplicaciones a niveles manejables para las personas y permite a los desarrolladores centrarse en problemas concretos, ignorando otros que en determinado momento no sean tan importantes. Para hacer un uso efectivo de la separación de conceptos es preciso en cada momento ser capaz de identificar, encapsular, modularizar y manipular diferentes dimensiones o niveles conceptuales de abstracción en cada una de las fases de vida del software, para no verse afectado por los efectos negativos que tiene la dispersión del código, el efecto dominó que pueden suponer cualquier modificación o la necesidad de reestructuración debido a la aparición de nuevos requisitos. Los conceptos de corte suelen aparecer en la fase de captura de requisitos y suelen tomar diferentes formas durante el proceso de desarrollo.

Hasta antes de la POA, los paradigmas de programación se han basado en una única y dominante manera de descomposición del software (clases, funciones, reglas...) La POA persigue la eliminación de lo que se conoce como tiranía de descomposición del software. Actualmente existen diferentes implementaciones que, con más o menos éxito, persiguen este objetivo. La que actualmente goza de mayor difusión es AspectJ [5], una

[4] Realmente la parte del núcleo tiene 1959 líneas. El resto es del componente reusable de generación de código.

extensión orientada a aspectos de Java, en la que se distinguen entre clases, que encapsulan los requisitos funcionales del sistema, y aspectos, que encapsulan los requisitos de corte no funcionales. Otra solución es la aportada por IBM, Hyper/J [6], que soporta la separación e integración de diferentes dimensiones, entendiéndose éstas como diferentes niveles conceptuales de abstracción. Una de ellas sería la dimensión de las clases, donde los requisitos funcionales tendrían cabida. Otra podría ser la de los requisitos no funcionales, que en AspectJ identificaríamos como aspectos (sincronismo, concurrencia,...) y otra podría incluir las restricciones impuestas a la aplicación que desarrollamos (tiempos de ejecución, entornos de funcionamiento,...) AspectJ ha apostado por encapsular las dos dimensiones hasta ahora mejor identificadas. Hyper/J nos deja la puerta abierta a nuevas dimensiones.

6. REFERENCIAS

1. T. Elrad, R. E. Filman, and A. Bader. *Aspect-Oriented Programming*. Commun. ACM, 2001.
2. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John. Irwin, *Aspect-Oriented Programming*, Xerox Palo Alto Research Center, 1997.
3. Karl J. Lieberherr. Adaptive Object-Oriented Software. The Demeter Method. Northeastern University Boston.
4. Cristina Lopes, *A Language Framework for Distributed Programming*, Ph.D.thesis, Northeastern University, noviembre 1997.
5. *The AspectJ™ Programming Guide*, the AspectJ Team, Xerox Parc Corporation
6. Peri Tarr, Harold Ossher. *Hyper/J™ User and Installation Manual*, IBM Research, 2000.