# Programación Orientada a Aspectos

Ing. José Joskowicz

El presente trabajo ha sido realizado en el marco de la asignatura “Nuevas Técnicas de Desarrollo de Software en Ingeniería Telemática”, del Doctorado de Ingeniería Telemática de la Universidad de Vigo, España.

*“Se trata de permitir que los programadores escriban programas que, en todo lo que sea posible, se parezcan a su diseño”*

Adaptado de la página de Gregor Kiczales [1]

# Indice

# Abstract

La “Programación Orientada a Aspectos” es una propuesta reciente que brinda un mayor grado de abstracción en el desarrollo de software, permitiendo realizar de manera clara y eficiente una clara “separación de incumbencias”.

El presente trabajo resume los conceptos principales de la Programación Orientada a Aspectos. Se incluye una introducción al tema, una breve reseña histórica, las definiciones formales necesarias, la descripción de algunos lenguajes orientados a aspectos y las características destacables de la implementación y el diseño de las aplicaciones orientadas a aspectos.

# 1. Introducción

Generalmente, el desarrollo de una aplicación involucra varias tareas que se deben realizar. Hay tareas que pueden considerarse “principales”, y que son típicamente detalladas por los usuarios como parte del análisis funcional de requerimientos. Pero, adicionalmente, existen también tareas que pueden considerarse “servicios comunes”, generalmente no detalladas en el análisis funcional. Ejemplos de estos “servicios comunes” pueden ser la necesidad de generar registros de auditoria (logging), accesos a bases de datos, temas relacionados con las seguridad, temas relacionados con la concurrencia del acceso a cierta información, etc. Es habitual que esta clase de “servicios comunes” deba ser realizada en forma similar pero independiente en diversas partes del código.

Cada una de estas tareas es considerada una “incumbencia” (*“concern”*, en inglés), en el entendido que al código que la implementa le debería “incumbir” solamente esa tarea. La “separación de incumbencias” es, por lo tanto, deseable en cualquier desarrollo de software.
Sin embargo, en los lenguajes de programación típicos (ya sea procedurales u orientados a objetos), es difícil, o imposible, separar claramente las incumbencias de los “servicios comunes” de las incumbencias “principales”, teniendo como consecuencia, por lo tanto, que las tareas de los “servicios comunes” queden dispersas dentro del código de las incumbencias principales. Esto es conocido como “incumbencias transversales” (*“crosscutting concerns”*, en inglés).

Como se mencionó anteriormente, las incumbencias son los diferentes temas, asuntos o *aspectos* de los que es necesario ocuparse para resolver un problema determinado. Separando las incumbencias, se disminuye la complejidad a la hora de tratarlas, y se gana en claridad, adaptabilidad, mantenibilidad, extensibilidad y reusabilidad.

Los conceptos y tecnologías conocidas con el nombre de “Programación Orientada a Aspectos" (POA) buscan resolver el problema de la “separación de incumbencias”, de una manera sistemática, clara y eficiente.

POA es conocida también como AOP, por las siglas en ingles “*Aspect-Oriented Programming”* o AOSD, por *Aspect-Oriented Software Development*

## 2. Breve reseña histórica

El término “separación de incumbencias” fue introducido en la década de 1970, por Edsger W. Dijkstra [2]. Significa, simplemente, que la resolución de un problema dado involucra varios aspectos o incumbencias, los que deben ser identificadas y analizados en forma independiente.

La “Programación Adaptativa” fue el concepto predecesor de la “Programación Orientada a Aspectos”. Las ideas originales de la “Programación Adaptativa” fueron introducidas a principio de la década de 1990 por el grupo Demeter [3], siendo Karl Lieberherr uno de sus ideólogos.

Los conceptos de POA fueron introducidos en 1997 por Gregor Kiezales y su grupo [4].

Actualmente, si bien hay ya varias implementaciones de lenguajes orientados a aspectos (como se verá más adelante en éste articulo), el nuevo paradigma aún continua en desarrollo, y todavía está abierta la investigación hacia varias áreas que recién están siendo exploradas, como por ejemplo llevar los “aspectos” a las etapas de diseño.

## 3. Definición de *Aspecto*

La definición formal de “Aspecto” ha evolucionado desde su concepción hasta el momento. Una definición inicial, aunque todavía no se manejaba el término “aspecto”, fue introducida por Karl Lieberherr en [3]. Adaptando la definición al término actual:

> “Un aspecto es una unidad que se define en términos de información parcial de otras unidades”

Gregor Kiezales y su grupo brindan una primera definición de “aspecto” en [4]:

> “Una propiedad que debe ser implementada es

> un ***componente***, si puede ser claramente encapsulada dentro de un procedimiento generalizado (por ejemplo, un objeto, un método, un procedimiento, una API). Se entiende que un elemento es claramente encapsulado si está bien localizado, es fácilmente accesible y resulta sencillo componerlo...
>
> un ***aspecto***, si no puede ser claramente encapsulado en un procedimiento generalizado. Los aspectos tienden a no ser unidades de la descomposición funcional del sistema, sino a ser propiedades que afectan la performance o la semántica de los componentes en forma sistemática..."

La definición actual de aspecto, de Gregor Kiezales de mayo de 1999 es la siguiente:

> "Un ***aspecto*** es una unidad modular que se disemina (*"cross-cuts"*) 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 implementación es una unidad modular del programa que aparece en otras unidades modulares del programa".

Se desprende de esta definición, que los ***aspectos*** de una aplicación son aquellos módulos que generan "incumbencias transversales", es decir, los módulos que están diseminados por el resto de las unidades funcionales.

Identificando los aspectos, y aplicando las técnicas desarrolladas en la POA es posible, por lo tanto, realizar adecuadamente la "separación de incumbencias".

Es de resaltar que la definición de ***aspecto*** no hace referencia al tipo de programación en la que se implemente (orientada a objetos o procedural), por lo que el concepto, como tal, aplica a ambos.

En una primera instancia puede ser fácil asociar a a***spectos*** los "servicios comunes", tal como se detalló en la introducción, incluyendo *aspectos* de auditoria (logging), *aspectos* de seguridad, *aspectos* de concurrencia, etc. Sin embargo, el concepto es mucho más genérico. Las herramientas que soportan POA manejan tanto las "clases" como los "aspectos" como unidades funcionales naturales, lo que permite total generalidad en el uso de los aspectos. Gregor Kiezales, haciendo referencia a Adrian Colyer (quien participa del proyecto AspectJ en ecplipse.org) indica [5]:

> "Cuando uno llega por primera vez a POA, los ejemplos y los tipos de incumbencias que se le presentan tienden a ser muy ortogonales con la aplicación principal (o "no funcionales", si se prefiere éste término). Me

refiero a los temas clásicos de “tracing”, “logging”, manejo de errores, etc... los que a veces llamamos “aspectos auxiliares”

Cuando se utilizan herramientas que soportan POA de la misma manera a la que estamos acostumbrados que se soporte POO (programación orientada a objetos), comenzamos a ver no solo aspectos auxiliares, sino también lo que llamamos “aspectos principales” a través del código....”

# 4. Implementación de *Aspectos*

En los lenguajes “tradicionales” (ya sea orientados a objetos o procedurales), se implementan las funciones principales mediante objetos, métodos o procedimientos. Tal como se definió anteriormente, esto representa a los ***componentes***. Los lenguajes orientados a aspectos agregan una nueva “dimensión”, encargada de concentrar la programación de conceptos tales como persistencia, gestión de errores, registros de auditoria, sincronización, comunicación entre procesos, o, en forma más general, lo que se ha definido previamente como ***aspectos***.

Los lenguajes orientados a aspectos definen una nueva unidad de programación de software para encapsular las funcionalidades diseminadas por todo el código (los ***aspectos***). Sin embargo, componentes y aspectos deben interactuar. En la ejecución del programa, finalmente, los aspectos deberán estar insertados dentro de los componentes. Para ello será necesario definir claramente como será ésta estrategia de inserción. En la terminología de POA, este proceso se denomina “***entretejido***”, ya que puede pensarse que aspectos y componentes deben “entretejerse” para formar finalmente un código ejecutable.

Para poder realizar este entretejido entre aspectos y componentes, es necesario definir o declarar ciertos ***“puntos de enlace”***. Los puntos de enlace son una clase especial de interfaz entre los aspectos y los módulos del lenguaje de componentes. Son los lugares del código en los que éste se debe modificar, incorporando los comportamientos adicionales especificados en los aspectos.

El encargado de realizar la inserción de los aspectos en los puntos de enlace es conocido como ***“tejedor”*** (***“weaver”***, en inglés). El tejedor se encarga de “tejer” o “entremezclar” los diferentes mecanismos de abstracción y composición que aparecen en los lenguajes de aspectos y componentes en los puntos de enlace.

Para tener un programa orientado a aspectos necesitamos definir los siguientes elementos [6]:

- Un lenguaje para definir la funcionalidad básica. Este lenguaje se conoce como *lenguaje base*. Suele ser un lenguaje de propósito general, tal como C++ o Java. En general, se podrían utilizar cualquier lenguaje.

- Uno o varios lenguajes de aspectos. El lenguaje de aspectos define la forma de los aspectos.
- Un tejedor de aspectos. El tejedor se encargará de combinar los lenguajes. El proceso de mezcla puede hacerse en el momento de la compilación, o puede ser retrasado para hacerse en tiempo de ejecución.

En los lenguajes tradicionales, es suficiente disponer de un compilador o intérprete (o máquina virtual) que, partiendo del lenguaje escrito en alto nivel, genere código entendible por la máquina.

En los lenguajes orientados a aspectos, además del compilador, es necesario tener un "tejedor", que combine el código de los componentes con el código de los aspectos. Ejemplos de lenguajes orientados a aspectos serán vistos más adelante.

Los componentes y los aspectos se pueden "tejer" en forma estática (o sea, en tiempo de compilación), o en forma dinámica (es decir, en tiempo de ejecución).

El entretejido estático consiste en modificar el "código fuente" de los componentes, en los puntos de enlace, insertando código definido de los aspectos. La principal ventaja de este tipo de entretejido es que evita que el nivel de abstracción que se introduce con la POA derive en una degradación de la performance de la aplicación. Adicionalmente, varios controles de errores pueden hacerse al compilar.

El entretejido dinámico requiere que los aspectos existan y estén presentes tanto en tiempo de compilación como en tiempo de ejecución. Una manera de conseguir esto es que, tanto los aspectos como los componentes se modelen como objetos y se mantengan en el ejecutable. Un tejedor dinámico será capaz de añadir, adaptar y remover aspectos de forma dinámica durante la ejecución. Pueden utilizarse mecanismos de herencia en forma dinámica para agregar el código de los aspectos a los componentes (o clases). La ventaja del entretejido dinámico radica en su potencialidad, ya que podrían tomarse decisiones en base a criterios dinámicos en la configuración de los aspectos. Las desventajas radican en afectar directamente la performance de la aplicación y en la complejidad de la depuración de errores, ya que varios de éstos podrían ser detectados únicamente en tiempo de ejecución.

# 5. Diseño y desarrollo de aplicaciones Orientadas a *Aspectos*

Hasta hace poco tiempo, la orientación a aspectos se centró principalmente en la implementación y codificación (desarrollo), pero en los últimos tiempos cada vez surgen más trabajos para llevar la separación de incumbencias a nivel de diseño.

Varios trabajos ([7], [8], [9], [10]) proponen utilizar UML (Unified Modeling Language) como lenguaje de modelado, ampliando su semántica con los mecanismos que el propio lenguaje unificado tiene para tales efectos y consiguiendo así representar la funcionalidad básica separada de los otros aspectos.

Desarrollar un sistema basado en aspectos requiere entender qué se debe incluir en el lenguaje base, qué se debe incluir dentro de los lenguajes de aspectos y qué debe compartirse entre ambos lenguajes. El lenguaje de los componentes debe proveer la forma de implementar la funcionalidad principal y asegurar que los programas escritos en ese lenguaje no interfieran con los aspectos. Los lenguajes de aspectos tienen que proveer los medios para implementar los aspectos deseados de una manera intuitiva, natural y concisa.

El desarrollo de una aplicación basada en aspectos requiere de tres pasos [11]:

1. Descomposición de aspectos y componentes: Descomponer los requerimientos para distinguir aquellos que son componentes de los que son aspectos

2. Implementación de las incumbencias: Implementar cada incumbencia por separado (aspectos y componentes)

3. Recomposición: Definir las reglas que permitan combinar los aspectos con los componentes

Como se mencionó, la actual tendencia es a detallar la primer etapa de descomposición de componentes y aspectos en el momento de diseño, y no postergarlo a la etapa de desarrollo. La segunda etapa, de implementación, se realiza programando componentes y aspectos, cada uno en su lenguaje. Las reglas para la recomposición se declaran generalmente como parte del lenguaje de aspectos.

# 6. Lenguajes orientados a *Aspectos*

La idea central de la POA es permitir que un programa sea construido describiendo cada concepto (o incumbencia) separadamente. El soporte para este nuevo paradigma se logra a través de una nueva clase de lenguajes, llamados lenguajes orientados a aspectos (LOA), los cuales brindan mecanismos para capturar y declarar aquellos elementos que se diseminan por todo el sistema (***aspectos***).

Una definición para tales lenguajes sería: Los LOA son aquellos lenguajes que permiten separar la definición de la funcionalidad “principal” de la definición de los diferentes aspectos.

Los LOA deben satisfacer varias propiedades deseables, entre las que se pueden mencionar [12]:

- Cada aspecto debe ser claramente identificable.
- Cada aspecto debe auto contenerse.
- Los aspectos deben ser fácilmente modificables.
- Los aspectos no deben interferir entre ellos.
- Los aspectos no deben interferir con los mecanismos usados para definir y evolucionar la funcionalidad principal, como la herencia.

Se distinguen dos enfoques diferentes en el diseño de los lenguajes orientados a aspectos: los lenguajes orientados a aspectos de dominio específico y los lenguajes orientados a aspectos de propósito general.

Los *LOA de dominio específico* han sido diseñados para soportar algún tipo particular de Aspectos, como por ejemplo la concurrencia, sincronización o distribución. Este tipo de lenguajes suelen tener un nivel de abstracción mayor que el lenguaje base, y permiten representar los conceptos específicos del aspecto a un nivel de representación mas elevado.
Algunos de estos lenguajes necesitan imponer restricciones en el lenguaje base, de manera de poder garantizar que las incumbencias que son tratadas en los aspectos no pueden ser programadas en los componentes, evitando de esta manera inconsistencias o funcionamientos no deseados. Por ejemplo, si el lenguaje de aspectos se especializa en la concurrencia o sincronización, puede requerir que sean deshabilitadas las primitivas del lenguaje base que puedan ser utilizadas para estas funciones (un ejemplo de este tipo de LOA es COOL, el cual se describe más adelante)

Los *LOA de propósitos generales* han sido diseñados para soportar cualquier tipo de Aspectos. Este tipo de lenguajes no pueden imponer restricciones en el lenguaje base. Generalmente tienen el mismo nivel de abstracción que el lenguaje base, y soportan las mismas instrucciones o primitivas, ya que, en principio, cualquier código debería poderse escribir en los Aspectos desarrollados con estos lenguajes (un ejemplo de este tipo de LOA es AspectJ, el cual se describe más adelante).

Los LOA de propósitos general tienen la clara ventaja de ser, tal como su nombre indica, generales, y por lo tanto, capaces de ser utilizados para desarrollar con ellos cualquier tipo de aspecto. Sin embargo, tienen también una desventaja. No garantizan la separación de funcionalidades [6]. Al no poder restringir las instrucciones o primitivas en la programación de los componentes, no puede garantizarse que las tareas que deberían programarse como aspectos no sean programados dentro de los componentes. Esto queda librado al programador.
Por el contrario, los LOA de dominio específico fuerzan a programar las tareas de aspectos dentro de éstos, ya que en el lenguaje base se restringe el uso de las instrucciones relacionadas con las funcionalidades de aspectos.

A continuación se detallan las características destacables de algunos de los LOA disponibles:

## 6.1. COOL

COOL (COOrdination Language) es un lenguaje de dominio específico, desarrollado por Xerox, cuya finalidad es tratar los aspectos de sincronismo entre hilos concurrentes. El lenguaje base que utiliza es Java, pero en una versión modificada, en la que se eliminan los métodos *"wait"*, *"notify"* y *"notifyAll"*, y la palabra clave *"synchronized"* para evitar que se produzcan inconsistencias al intentar sincronizar los hilos en el aspecto y en los componentes (clases en este caso).
En COOL, la sincronización de los hilos se especifica de forma declarativa y, por lo tanto, más abstracta que la correspondiente codificación en Java.

## 6.2. RIDL

RIDL (Remote Interaction and Data transfers aspect Language) es un LOA de dominio específico que maneja la transferencia de datos entre diferentes espacios de ejecución.
Un programa RIDL consiste de un conjunto de módulos de "portales". Un portal es el encargado de manejar la interacción remota y la transferencia de datos de la clase asociada a él, y puede asociarse como máximo a una clase.

## 6.3. MALAJ

MALAJ (Multi Aspect LAnguage for Java) [13] es un LOA de dominio especifico, focalizado en la sincronización y reubicación.

MALAJ sigue la misma filosofía que COOL y RIDL, indicando que la flexibilidad ganada con los LOA de propósito general, pueden potencialmente llevar a conflictos con los principios básicos de la POO. Por esta razón, MALAJ propone un LOA de dominio especifico, donde varios aspectos puedan ser resueltos, cada uno especializado en su propio "incumbencia".

Tal como lo indica su nombre, el lenguaje base es Java, pero en una versión restringida, en la que se ha eliminado la palabra reservadas "synchronized" así como los métodos *"wait"*, *"notify"*, y *"notifyall"*, en forma similar a lo que sucede con COOL.

El propósito final de los creadores de MALAJ es cubrir un gran espectro de aspectos específicos, mas allá de las dos mencionadas anteriormente.

## 6.4. AspectC

AspectC es un LOA de propósito general que extiende C. Es similar a AspectJ (ver más adelante), pero sin soporte para la programación orientada a objetos.
El código de aspectos interactúa con la funcionalidad básica en los límites de una llamada a una función, y puede ejecutarse antes, después, o durante dicha llamada.
Como el lenguaje C es estático, el tejedor de AspectC es también estático.
Dado que C es utilizado en un gran numero de aplicaciones, es muy interesante poder disponer de un LOA que lo tenga como lenguaje base. Un ejemplo de su utilización en le desarrollo de sistemas operativos es presentado en [14].

## 6.5. AspectC++

AspectC++ [15] es un LOA de propósito general que extiende el lenguaje C++ para soportar el manejo de aspectos. Sintacticamente, un aspecto en este lenguaje es muy similar a una clase en C++. Sin embargo, además de funciones, un aspecto puede definir "avisos" (*"advice"*). Luego de la palabra clave "advice", una expresión de corte (*"pointcut expresion"*) define el punto donde el aspecto modificará al programa (es decir, los "puntos de enlace" o *"join points"*) [16]. Pueden utilizarse expresiones de corte para identificar un conjunto de puntos de enlaces. Se componen a partir de expresiones de corte y un conjunto de operadores algebraicos. La declaración de los avisos es utilizada para especificar código que debe ejecutarse en los puntos de enlace determinados por la expresión de corte. Los avisos pueden insertarse antes, después o durante la ejecución de los métodos donde se insertan.

El tejedor de AspectC++, ac++, transforma programas escritos en AspectC++ a código de C++, por lo que puede ser utilizado con cualquier compilador de C++, como g++ o Microsoft C++ de VisualStudio.NET.

## 6.6. AspectJ

AspectJ es un LOA de propósito general [17], que extiende Java con una nueva clase de módulos que implementan los aspectos. Los aspectos cortan las clases, las interfaces y a otros aspectos.
En AspectJ, un aspecto es una clase, exactamente igual que las clases Java, pero con una particularidad, que pueden contener unos constructores de corte, que no existen en Java. Los cortes de AspectJ capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, y excepciones de señales y gestión. Los cortes no definen acciones, sino que describen eventos. En forma similar a AspectC++, en AspectJ se definen cortes ("*pointcuts"*), avisos y puntos de enlace [18].

# 7. Ejemplos de aplicación

Varios ejemplos de aplicación de las técnicas de POA se han presentado, en diversos trabajo. Se citarán a continuación algunos de ellos. Los detalles de cada ejemplo pueden ser consultados en las referencias.

## 7.1. Código "enredado": Filtros de imágenes

Uno de los primeros ejemplos fue presentado por Gregor Kiezales y su grupo, con la introducción de la POA [4]. El ejemplo consiste en el desarrollo de una aplicación de procesamiento de imágenes en blanco y negro, en la que una imagen de entrada debe pasar por una serie de filtros para producir una imagen de salida. En dicho artículo se menciona tres implementaciones de la misma aplicación: Una con código fácil de comprender pero ineficiente, otra eficiente pero con código difícil de comprender, y una implementada con POA que es a la vez eficiente y con código fácil de comprender.
En este ejemplo, para lograr filtros de imágenes complejos, se definieron previamente filtros simples, los que concatenados, lograban la funcionalidad de un filtro complejo. Esta solución permitía tener un código fácilmente entendible (cada filtro simple era fácil de comprender), pero para llegar a la imagen final, se generaban varias imágenes intermedias lo que redundaba en una baja eficiencia del programa. La generación de un único código para el filtro complejo era posible, pero el mismo quedaba poco entendible (muy "enredado", según la terminología del artículo)
El compromiso "código entendible" vs "eficiencia", estaba dado por la facilidad de comprensión en los bucles iterativos de los filtros de imágenes, frente a la generación de varias imágenes intermedias para poder llegar a la imagen final, al concatenar varios filtros simples para obtener uno mas complejo. La implementación con POA, en este caso, permitió programar declarativamente la manera de "unir" varios bucles sencillos concatenados (y por lo tanto, poco eficientes) en un único bucle, eficiente.

A los efectos de cuantificar la mejora obtenida, este trabajo presenta la siguiente ecuación:

$$\text{Factor de mejora} = \frac{\text{(tamaño del código "enredado") – (tamaño del código de los componentes)}}{\text{(tamaño del código de los aspectos)}}$$

En este caso, el factor de mejora fue evaluado en 98, lo que demuestra la gran ventaja de POA. Sin embargo, se menciona en el mismo artículo que es muy difícil de cuantificar los beneficios de una determinada herramienta, y que en todo caso, esto debe hacerse estudiando un numero importante de casos.

## 7.2. Aspectos de concurrencia: Manejo de colas circulares

Antonia Mª Reina Quintero presenta en [6] un ejemplo de la gestión de una cola circular. En esta cola circular, los elementos se van añadiendo por atrás, y se van borrando en la parte de delante. El potencial problema a resolver son los aspectos de concurrencia. Si no es tenida en cuenta la concurrencia y varios hilos solicitan ejecutar métodos de la cola circular, pude darse el caso de que la cola haya cubierto su capacidad, y se sigan haciendo peticiones de inserción de elementos, o al contrario, es decir, que la cola esté vacía y se hagan peticiones de extracción de elementos. Para prevenir esto, el hilo que hace la solicitud debería esperar a que se extraiga algún elemento, en el primer caso, o a que se inserte alguno en el segundo. Es decir, se necesita código para tener en cuenta la sincronización.

El ejemplo presentado en este articulo realiza la programación necesaria en Java, primero sin tener en cuenta los aspectos de sincronización. Luego le añade las líneas de código necesarias para gestionar este aspecto, lo que permite apreciar claramente cómo estas se diseminan por todo el programa, quedando un código poco claro. Después el mismo ejemplo se implementa con COOL, y por último con Aspect J.

Tal como se concluye en el artículo referenciado, en la resolución de este ejemplo, la estrategia de sincronización seguida queda más clara y más intuitiva utilizando COOL que utilizando AspectJ, ya que con el primero la programación de los aspectos es en forma declarativa. Sin embargo, con COOL no se pueden resolver aspectos no vinculados al sincronismo, por lo que AspectJ sigue siendo, en ese sentido, más flexible.

## 7.3. Aspectos de “logging”: Protocolo TFTP

Asteasuain y Contreras, en su tesis de licenciatura [12], presentan un ejemplo de la aplicación de AOP de aspectos de “logging” en el desarrollo de un servidor de TFTP (Trivial File Transfer Protocol). En el desarrollo inicial del protocolo, realizado en Java, el código es confuso y complejo. Esto se debe a ciertos aspectos, que no tienen que ver con el protocolo en sí, sino a cuestiones extras, o agregados. El más importante, es el aspecto de logging, encargado de llevar las trazas, informar de las excepciones y manejar las estadísticas de las conexiones, entre otras tareas. El código necesario para el logging resulta disperso por todas las clases y muchas veces está duplicado.

Aplicando AspectJ se logran separar los aspectos de logging, generando un código mas claro, y mas pequeño. En el articulo mencionado, el código sin aspectos se implementa en 1001 líneas de código, mientras que el código con aspectos fue implementado en 714 líneas de código.

### 7.4. Aspectos de "timing": Un sistema de telecomunicaciones

Cristina Lopes, en su página personal [19], presenta un ejemplo de POA para un sistema de telecomunicaciones. Este sistema debería manejar clientes, llamadas y conexiones. La operación básica de este sistema está relacionada a las llamadas que hacen los clientes, incluyendo, por ejemplo, conferencias en las que varios clientes pueden unirse a una llamada establecida. Sobre la operación básica, existen funciones de temporización que lleven registro de la duración de las conexiones de cada cliente. Y adicionalmente, podrían existir funciones de facturación que carguen a las cuentas de los clientes las llamadas o conferencias que realizan, de acuerdo al tiempo de utilización y al tipo de llamada.
Una manera de implementar este sistemas seria con POO (Programación orientada a objetos), donde el "cliente", las "llamadas" y las "conexiones" serian candidatos naturales para los objetos, los que podrían implementarse con clases. Las funciones de temporizacion podrían también implementarse con clases. Sin embargo, aun "encapsulando" las funciones de temporizacion en clases, debería existir código en las clases "básicas" que invoquen a las funciones de temporizacion en los puntos apropiados (por ejemplo, cada vez que se establece una nueva conexión, se llama a la clase que implementa un temporizador). En esta solución, los puntos en los que se invocan a las funciones de temporizacion son muy importantes, y extraían seguramente dispersos dentro de varios objetos "básicos". Sin embargo, las especificaciones de que cosas temporizar y/o cobrar, podrían variar en forma independerte a cualquier otra funcionalidad de la aplicación.
Utilizando algún lenguaje de POA, podrían capturarse estas funciones en módulos propios. Por ejemplo, podría existir un aspecto de temporizacion que encapsule toda la información necesaria acerca de esta función. De esta manera, puede concentrase la especificación de las funciones de temporizacion en un único modulo (aspecto), preservando la integridad y claridad del resto de la operación básica del sistema.

## 8. Conclusiones

Los primeros artículos de POA tienen menos de 10 años, por lo que puede decirse que es una tecnología relativamente nueva. Sin embargo, numerosos artículos han sido escritos desde sus comienzos, y varios lenguajes de programación han sido desarrollados para soportar esta nueva tecnología. Son de destacar especialmente AspectJ y AspectC++, los que aparecen actualmente como firmes candidatos a permanecer y ser utilizados cada vez por más programadores.

AOP se está extendiendo hacia todos las etapas del diseño de software, especialmente hacia las etapas previas al desarrollo, es decir, el análisis. Esto indica que los conceptos introducidos en POA están penetrando cada vez mas en los procesos de desarrollo de software.

Sin embargo, no puede decirse que la tecnología este “madura” y totalmente aceptada por la comunidad informática. Hay visibles avances en el “mundo Java”, pero por ahora incipientes en el “mundo .Net”, sobre los que se están desarrollando en la actualidad una cantidad importante de programas. Dado que .Net es “multilenguaje”, uno de los problemas a resolver es el soporte de multilenguaje en el código base, por parte de las herramientas de aspectos. Las variantes de POA estáticas (es decir, las que generan código al momento de compilación), suelen estar muy relacionadas al lenguaje base (ya que insertan instrucciones desarrolladas en ese mismo lenguaje en tiempo de compilación). Esto hace pensar que las implementaciones en .Net deban tener entretejido dinámico, lo que lleva seguramente a modificar la máquina virtual o el soporte de tiempo de ejecución de los lenguajes [20].

Es de esperar que los conceptos de la POA se asiente en los próximo años, y su implementación sea de uso habitual, sea cual sea el lenguaje base utilizado.

# 9. Referencias


1 Gregor Kiczales´s Home Page
http://www.cs.ubc.ca/~gregor/

2 On the role of scientific thought
prof. Dr. Edsger W Dijkstra
Burroughs Research Fellow
30th August 1974
http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

3 Adaptive Object-Oriented Software - The Demeter Meted
Karl Lieberherr
College of Computer Science, Northeastern University Boston, 1996

4 Aspect-Oriented Programming
Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin

5 Interview with Gregor Kiczales
TheServerSide.com, Julio 2003
http://www.theserverside.com/talks/videos/GregorKiczalesText/interview.tss

6 Visión general de la Programación Orientada a Aspectos
Antonia Mª Reina Quintero
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
Diciembre, 2000

7 Extending UML with Aspects: Aspect Support in the Design Phase
Junichi Suzuki, Yoshikazu Yamamoto
3rd Aspect-Oriented Programming (AOP) Workshop at ECOOP'99

8 From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach
Mohamed M. Kandé, Jörg Kienzle and Alfred Strohmeier
Software Engineering Laboratory
Swiss Federal Institute of Technology Lausanne
CH - 1015 Lausanne EPFL
Switzerland

9 A UML Profile for Aspect Oriented Modeling
Omar Aldawud, Tzilla Elrad, Atef Bader
OOPSLA 2001 workshop on Aspect Oriented Programming

10 UML Profile Definition for Dealing with the Notification Aspect in Distributed Environments
José Conejero, Juan Hernández, Roberto Rodríguez
6th International Workshop on Aspect-Oriented Modeling
March 14, 2005, Chicago, Illinois, USA

11 I Want My AOP (Part 1, 2 and 3)
Ramnivas Laddad
Java World, January 2002
http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html

12 Programación Orientada a Aspectos, análisis del paradigma
Fernando Asteasuain, Bernardo Ezequiel Contreras
Tesis de Licenciatura
Departamento de Ciencias e Ingeniería de la Computación, UNIVERSIDAD NACIONAL DEL SUR, Argentina, Octubre de 2002

13 Malaj: A Proposal to Eliminate Clashes Between Aspect-Oriented and Object-Oriented Programming
Gianpaolo Cugola, Carlo Ghezzi, Mattia Monga, Gian Pietro Picco
Politecnico di Milano, Dip. di Elettronica e Informazione

14 Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code
Yvonne Coady, Gregor Kiczales, Mike Feeley and Greg Smolyn
Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Internation Symposium on the Foundations of Software Engineering (FSE-9), 2001

15 The home of Aspect C++
http://www.aspectc.org/

16 AspectC++: an AOP Extension for C++
Olaf Spinczyk, Daniel Lohmann, Matthias Urban
Software Developer's Journal 5/2005

17 AspectJ project
http://www.eclipse.org/aspectj

18 The AspectJ Programming Guide
http://www.eclipse.org/aspectj/doc/released/progguide/index.html

19 Aspect-Oriented Programming and Software Development
Cristina Lopes
http://www.ics.uci.edu/~lopes/aop/aop.html)

20 Programación Orientada a Aspectos (AOP)

Nicolás Kicillof
MSDN - Microsoft