Programación Orientada a Aspectos: Metodología y Evaluación

Fernando Asteasuain, Bernardo E. Contreras,
Elsa Estévez, Pablo R. Fillottrani
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Av. Alem 1253 – (8000) Bahía Blanca
Argentina
{fa,bec,ece,prf}@cs.uns.edu.ar

Resumen

La Ingeniería de Software tradicional carece actualmente de mecanismos adecuados para abstraer, y encapsular conceptos que no forman parte de la funcionalidad básica de los sistemas, tales como debugging, sincronización, distribución, seguridad, administración de memoria, y otros. El resultado de esta insuficiente abstracción es una notable disminución de la calidad del software final. Una de las alternativas más prometedoras para resolver este problema es la Programación Orientada a Aspectos (POA). En este trabajo se analiza la aplicación de la POA. Se realiza la reingeniería de una implementación orientada a objetos introduciendo aspectos. Se definen métricas para evaluar las importantes ventajas de incorporar aspectos a la programación tradicional. Se las aplica a un caso de estudio, y se muestra que los resultados obtenidos confirman la eficacia, y la utilidad de la POA. Se presenta un caso de estudio que consiste en la adaptación considerando aspectos del protocolo para transferir archivos Trivial File Transfer Protocol(TFTP).

Palabras claves: Ingeniería de Software, Programación Orientada a Aspectos, métricas orientadas a aspectos, AspectJ, desarrollo orientado a aspectos.

1 Introducción

La Ingeniería de Software ha evolucionado desde sus comienzos. Con esta evolución se introdujeron conceptos tales como el agrupamiento de instrucciones a través de procedimientos y funciones, módulos, bloques estructurados, tipos de datos abstractos, genericidad, y herencia entre otros. Estos conceptos proveen formas de asbtracción y constituyen el medio para lograr una programación de alto nivel. Asímismo, los progresos más importantes se han obtenido aplicando estos conceptos junto con tres principios estrechamente relacionados entre sí: abstracción, encapsulamiento, y modularidad.

La Programación Orientada a Objetos (POO) introdujo un avance importante forzando el encapsulamiento y la abstracción, por medio de una unidad que captura tanto funcionalidad como comportamiento, y estructura interna. A esta entidad se la conoce como clase, y su principal característica es que enfatiza datos y algoritmos. A través de POO, o de otras técnicas de abstracción de alto nivel, se logra un diseño y una implementación que satisface la funcionalidad básica, y que presenta niveles de calidad aceptable. Sin embargo, existen conceptos entrecruzados que no pueden encapsularse dentro de una unidad funcional, debido a que atraviesan todo el sistema, o varias partes de él. Algunos de ellos son: sincronización, manejo de memoria, distribución, chequeo de errores, seguridad, y redes.

Las técnicas de implementación actuales tienden a implementar los requerimientos usando metodologías de una sola dimensión, forzando a que todos los requerimientos sean expresados en esa única dimensión. Esta dimensión resulta adecuada para la funcionalidad básica, pero no para los otros requerimientos, los cuales quedan diseminados a lo largo de la dimensión dominante. Es decir, que mientras el espacio de requerimientos es de n-dimensiones, el espacio de la implementación es de una sola dimensión. Dicha diferencia produce un mapeo deficiente de los requerimientos a sus respectivas implementaciones. Los dos síntomas más significativos de este problema son el *Código Mezclado (Code Tangling)* y el *Código Diseminado (Code Scattering)* [1]. El *Código Mezclado* se presenta cuando en un mismo módulo de un sistema de software conviven simultáneamente más de un requerimiento. Esto hace que en el modelo existan elementos de implementación de más de un requerimiento. El *Código Diseminado* se produce cuando un requerimiento está esparcido sobre varios módulos. Por lo tanto, la implementación de dicho requerimiento también queda diseminada sobre esos módulos.

La combinación de estos síntomas afecta tanto al diseño como a la implementación de software [1]. La implementación simultánea de varios conceptos tiene dos efectos negativos. El primero es una baja correspondencia entre un concepto y su implementación. El segundo es una menor productividad de los desarrolladores, ya que se distraen del concepto principal. Por otro lado, la implementación de varios conceptos en un mismo módulo lleva a un código poco reusable, de baja calidad, y propenso a errores. Esto se debe a que alguno de los tantos conceptos puede ser subestimado. Por último, la evolución es más dificultosa debido a la insuficiente modularización. Los futuros cambios en un requerimiento implican revisar y modificar cada uno de los módulos donde esté presente ese requerimiento.

Por las razones expuestas se concluye que las técnicas tradicionales no soportan una completa separación de conceptos. Esto es, no permiten una modularización adecuada que facilite separar la implementación de determinados aspectos, distintos a la funcionalidad básica. Esta situación tiene un impacto negativo en la calidad del software, ya que esta característica es clave para producir un software comprensible y evolucionable.

Como respuesta a este problema nace la Programación Orientada a Aspectos (POA). La POA permite a los desarrolladores escribir, ver, y editar un aspecto diseminado por todo el sistema como una entidad por separado, de una manera inteligente, eficiente e intuitiva. La POA es una nueva metodología de programación que aspira a soportar la separación de las propiedades para los aspectos antes mencionados. Esto implica separar la funcionalidad básica de los aspectos, y los aspectos entre sí, a través de mecanismos que permitan la abstracción y la composición de los mismos, a fin de poder implementar todos los requerimientos del sistema.

En este trabajo se estudia la aplicación de la POA. Se compara la implementación de un producto de software con y sin aspectos. Se presenta la metodología empleada para convertir una implementación tradicional, en una que considera aspectos. Para realizar una comparación más exhaustiva, se definen métricas. Las métricas son necesarias en todo proyecto tecnológico, ya que la idea de medición hace los conceptos mas visibles, y por lo tanto mas comprensibles y controlables [2]. Se introducen métricas que permiten medir el impacto de la aplicación del paradigma. A continuación, se aplican las métricas al caso de estudio presentado, y se analizan los valores obtenidos. Finalmente, se mencionan trabajos relacionados, se incluyen las conclusiones, y el trabajo futuro.

2 Fundamentos de la POA

Los tres elementos constitutivos principales de la POA son [3]:

- Un lenguaje para definir la funcionalidad básica, conocido como *lenguaje base* o *componente*. El mismo puede ser un lenguaje imperativo, o no, como por ejemplo C++, Java, Lisp, ML.
- Uno o varios *lenguajes de aspectos*, para especificar el comportamiento de los distintos aspectos. Algunos ejemplos son *COOL*, para espeficicar sincronización, y *RIDL* para especificar distribución.
- Un *tejedor* de aspectos, del inglés *weaver*, que se encarga de combinar los lenguajes en tiempo de ejecución o de compilación.

Los lenguajes orientados a aspectos definen una nueva unidad de programación de software para encapsular aquellos conceptos que cruzan todo el código. Al momento de *tejer* los componentes y los aspectos para formar el sistema final, es evidente que se requiere una interacción entre el código que provee la funcionalidad básica, y el código de los aspectos. Claramente, esta interacción no es la misma que ocurre entre los módulos del lenguaje base, donde la comunicación está basada en declaraciones de tipo, y llamadas a procedimientos y funciones. Por esta razón, la POA define una nueva forma de interacción, provista a través de *puntos de enlace*.

Los *puntos de enlace* brindan la interfaz entre aspectos y componentes. Son lugares dentro del código donde es posible agregar el comportamiento adicional que caracteriza a la POA. Dicho comportamiento adicional es especificado en los aspectos, y puede agregarse antes, después, o en el lugar del punto de enlace. Por ejemplo, dentro de la POO algunos *puntos de enlace* pueden ser: llamadas a métodos, creación de objetos, acceso a atributos, etc.

Por último, resta mencionar al encargado principal en el proceso de la POA, el *tejedor*. Éste debe realizar la parte final y más importante: *tejer* los diferentes mecanismos de abstracción y composición que aparecen tanto en los lenguajes de aspectos, como en los lenguajes de componentes, guiado por los *puntos de enlace*.

La estructura general de una implementación basada en aspectos difiere de la estructura de una implementación tradicional. Una implementación tradicional consiste de un lenguaje, un compilador, o intérprete para ese lenguaje, y finalmente, un programa escrito

en ese lenguaje que implemente la aplicación. En cambio, una implementación basada en la POA presenta en primer término, un lenguaje base que permite implementar la funcionalidad básica. Luego, uno o más lenguajes de aspectos para la implementación de los mismos, y un *tejedor* de aspectos encargado de la combinación de los lenguajes. Por último, se requiere el programa escrito en el lenguaje base, que implementa los componentes funcionales, y uno o más programas de aspectos que implementan a estos. Gráficamente, se pueden comparar ambas estructuras, como queda reflejado en la figura 1.

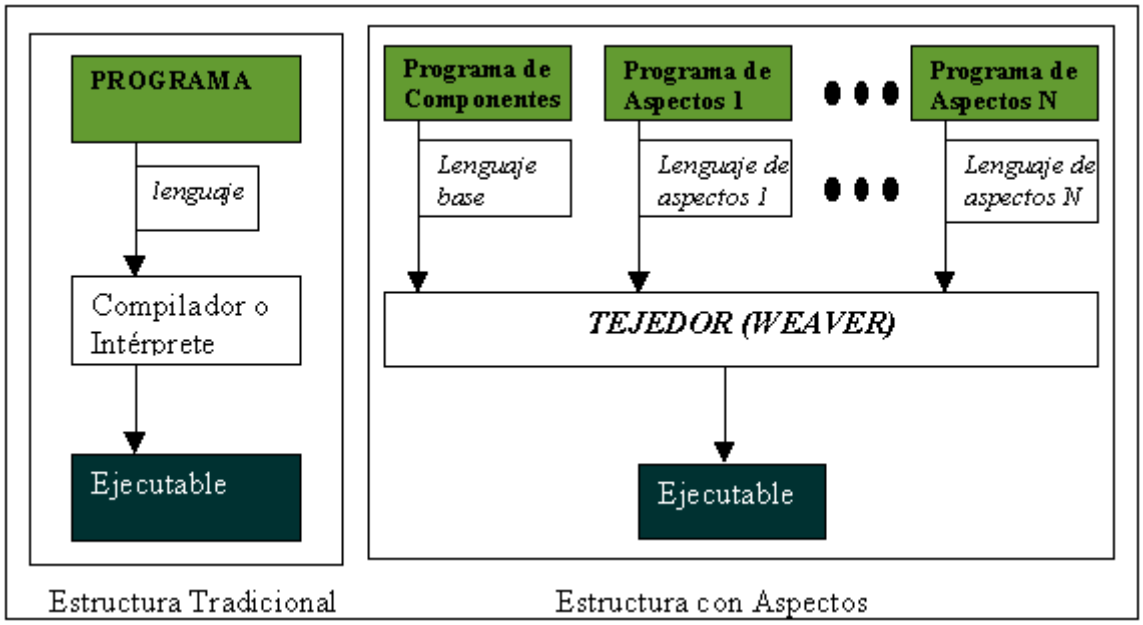

Figura 1: Estructura tradicional y Estructura aplicando POA

En la sección siguiente se presenta el caso de estudio, y su implementación con y sin aspectos. En base a éste, y con los resultados de las métricas definidas, se analizan los beneficios de la aplicación del paradigma.

3 Aplicación de la POA

En este traabjo se plantea un caso de estudio, que considera la implementación del protocolo TFTP (Trivial File Transfer Protocol) con y sin aspectos, con el objetivo de aplicar y evaluar la POA. La versión sin aspectos es la versión TFTP 0.8 de Mark Benvenuto, realizada en Java, de distribución libre, y que puede obtenerse de su página [5]. Nuestro trabajo consistió de una reingeniería de este producto, logrando una implementación en AspectJ [6], una extensión de Java para soportar la programación orientada a aspectos. La funcionalidad básica coincide con la provista por la versión original del autor. Sin embargo, se debieron introducir algunas modificaciones específicas para el tratamiento de los aspectos. La comparación de ambas implementaciones permite obtener relevantes conclusiones y evaluar con resultados concretos la verdadera dimensión de las ventajas de la POA, y de su principal herramienta AspectJ.

TFTP es un protocolo para transferir archivos entre procesos. Es una alternativa a FTP (File Transfer Protocol) con menos sobrecarga, más simple pero con seguridad mínima. Para mayor información sobre el protocolo consultar las RFC [10].

En las próximas dos subsecciones se analizan las dos implementaciones del caso de estudio. En la primera se describe la implementación en Java, y en la siguiente en AspectJ. A continuación, se comparan ambas implementaciones, y en la sección siguiente se miden los resultados mediante la definición de métricas.

3.1 Implementación en Java

En la implementación del protocolo existen dos enfoques principales: el del cliente, y el del servidor. Por cuestiones de extensión, en este trabajo sólo se analiza el punto de vista del servidor. Esto se debe a que el estudio de esta rama de análisis es suficiente para mostrar claramente los resultados obtenidos con la aplicación de las métricas. Por razones similares, dentro del punto de vista del servidor, sólo se analizan las acciones correspondientes a la escritura de archivos. Esto es, un pedido de lectura por parte del cliente. El objetivo principal durante el desarrollo del trabajo fue mantener el modelo lo más sencillo posible, y simultáneamente poder observar mejor el impacto de la utilización de la POA.

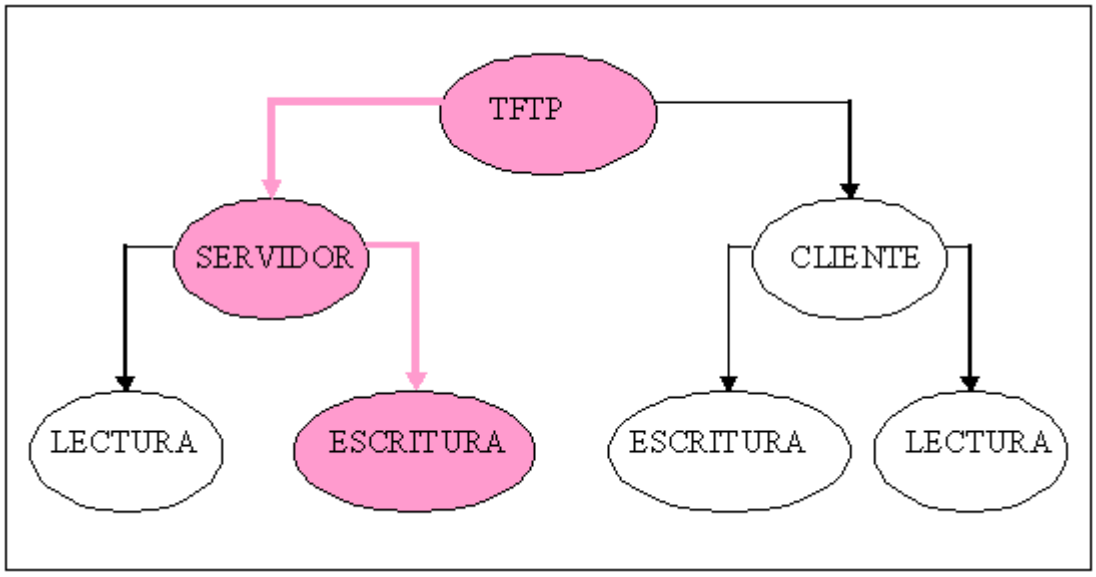

Figura 2: Rama de análisis

La figura 2 muestra el camino de análisis elegido. Dicho camino está marcado con color dentro del árbol que modela la implementación del protocolo TFTP. El código en Java que implementa el camino seleccionado se encuentra detallado en [3].

Al analizar el código en Java se observa que si bien la lógica del protocolo es simple, el código resultante es confuso y complejo. Esto se debe a ciertos aspectos, que no tienen relación directa con el protocolo, sino a cuestiones extras, o agregados.

El aspecto más importante y que más "ensucia" el código es el aspecto de logging: encargado de llevar las trazas, informar las excepciones, manejar las estadísticas de las

conexiones, entre otras tareas. Todo el código necesario para el logging resulta desparramado por todas las clases, esto es, entrecruza las clases, obstruyendo el entendimiento de la lógica del programa. Este código sucio, que muchas veces resulta código duplicado, incluye sentencias, mensajes, como también atributos relativos al logging dentro de las clases que implementan el protocolo. Las graves consecuencias de este tipo de código ya se mencionaron en este trabajo, y entre otras podemos mencionar la dificultad de entendimiento y de lectura, menor robustez, dificultades a la hora de enfrentar cambios o modificaciones, y disminución de la calidad.

3.2 Implementación en AspectJ

AspectJ permite abstraer y encapsular concretamente el aspecto de logging a través del constructor lingüístico *aspect* [11]. Con esto se logra una funcionalidad básica limpia con código sólo relativo al protocolo, y en forma separada, todas las acciones de logging, claramente especificadas dentro de una única unidad.

A modo de ejemplo, se considera en la figura 3 una porción de código escrita en Java, y en las siguientes dos figuras la implementación del mismo comportamiento en AspectJ.

```
if(debug > 0)
   logWriter.println("Receiving Packet");
try{
 sock.receive(ACKp);
    }catch( InterruptedIOException iioe) {
        logWriter.println("Error Receiving Packet: " + iioe);
        logWriter.println("Continuing to wait for Packet");
        continue;  }
...
%Código para procesar el paquete
...
```

Figura 3: Ejemplo de *Código Mezclado* en Java

En la figura 3, se muestra una de las acciones del protocolo correspondiente a la sentencia para recibir un paquete. La misma se ve ensuciada por las actividades propias del logging.

En AspectJ, el código se divide en dos partes: la funcionalidad básica, que consiste en recibir el paquete para luego procesarlo; y el aspecto de logging, que encapsula las actividades del logging. En la figura 4 se detalla la funcionalidad básica, mientras que en la figura 5 se detalla la implementación de este aspecto.

En este código resulta una funcionalidad básica que se refiere únicamente al protocolo TFTP, y las acciones de logging ya no resultan mezcladas, debido a que el aspecto logra abstraerlas por separado. Es importante destacar que estos beneficios resultan de aplicar el paradigma sólo a una porción del sistema. Las ganancias son notablemente mayores al aplicar el paradigma sobre el sistema completo.

```
 sock.receive(ACKp);
 %Código para procesar el paquete
 ...
```

Figura 4: Funcionalidad básica

```
Aspect Logging{
...
pointcut receivePacket(DatagramPacket packet):
call(DatagramSocket.receive(DatagramPacket))&& args(packet);

before(DatagramPacket packet):receivePacket(packet)&& anyTftp(){
   if(debugMode > 0)
      logStream.println("Receiving Packet");
 }
after(DatagramPacket packet) throwing(InterruptedIOException iioe ):
                  receivePacket(packet){
    logStream.println("Error Receiving Packet: " + iioe);
    logWriter.println("Continuing to wait for Packet");  }
}
```

Figura 5: Código del aspecto

Para implementar el resto del sistema en AspectJ se sigue la misma estrategia que se aplicó a la porción de código analizada anteriormente. La estrategia de desarrollo orientada a aspectos que definimos en este trabajo consiste de tres pasos:

1. Identificar un aspecto en el sistema. Por ejemplo, el aspecto de logging en la implementación del protocolo TFTP.

2. Para el aspecto identificado considerar los siguientes pasos:

   (a) Identificar un *punto* o momento en particular donde se necesitan las acciones relacionadas con el aspecto. Por ejemplo, al recibir o enviar un paquete, al leer o escribir un archivo, al crear un socket, al obtener información de un paquete.

   (b) Capturar ese punto como un corte de AspectJ. Por ejemplo, el siguiente corte captura el momento de enviar un paquete.

   ```
   pointcut sendPacket(DatagramPacket packet):
            call(DatagramSocket.send(DatagramPacket)) && args(packet);
   ```

   (c) Identificar las acciones, relativas al aspecto en cuestión, que ocurren antes y después de ese *punto* o momento. Por ejemplo, avisar que se está enviando un paquete antes de comenzar el envío, señalar la finalización del envío, señalar que ha ocurrido una excepción.

(d) Codificar esas acciones como avisos "antes" y "después". Por ejemplo, el siguiente aviso "antes" avisa que se está por enviar un paquete.

```
before(DatagramPacket packet):sendPacket(packet){
        logStream.println("Sending  Packet ");
              }
```

(e) Mientras existan *puntos* o momentos de interés, ejecutar los pasos 2(a), 2(b), 2(c), y 2(d).

3. Mientras existan aspectos en el sistema, ejecutar el paso 2.

El código en AspectJ que implementa el protocolo se encuentra detallado en [3].

3.3 Comparación de ambas implementaciones

Al utilizar aspectos la ventaja más destacable es que el código que implementa la funcionalidad básica es mucho más limpio y entendible. Otra ventaja importante es una mayor facilidad para realizar cambios y modificaciones, como también acciones para remover o agregar comportamiento. Esto se debe a que todas las acciones de logging resultan encapsuladas en el aspecto, y son claramente identificadas. En cuanto al desarrollo, el mismo se vuelve mucho más intuitivo y natural, dada la naturaleza declarativa de los *puntos de enlace*, y al comportamiento adicional que permite definir AspectJ.

En la próxima sección se evalua la aplicación del paradigma a través de métricas orientadas a aspectos que cuantifican los resultados de la utilización de la POA.

4 Métricas para POA

A fin de cuantificar los beneficios de la POA, y AspectJ se definen y se aplican métricas orientadas a aspectos en el caso de estudio. Las métricas tradicionales no son de significativa utilidad aplicadas sobre programas basados en este nuevo paradigma. Esto se debe a que no incorporan en sus cálculos el efecto de una programación orientada a aspectos. Por lo tanto, en este trabajo se definen dos métricas orientadas a aspectos, las cuales incluyen los conceptos relacionados al paradigma de aspectos, reflejando de una manera más fiel y precisa el desempeño de la POA.

En primer lugar se define la métrica *Beneficio_POA*, que mide cuál es el beneficio de introducir aspectos. Para ello, define una relación de esfuerzo de desarrollo entre una implementación sin considerar aspectos, y otra donde se aplica el paradigma. Está definida en función de dos factores claves de un proyecto: el tamaño del producto, y el esfuerzo de desarrollo. Considera el tamaño de una implementación sin y con aspectos, y pondera estos valores con un factor de desarrollo. Para esto último, se tomó en cuenta el factor de desarrollo propuesto por Rich Rice [7]. Este factor establece que para un sistema pequeño el tiempo de desarrollo promedio en Java es de 10 horas/programador, mientras que en AspectJ es de 2 horas/programador. A fin de considerar el tamaño de ambas implementaciones, se evalua el tamaño físico de los archivos de las clases y del aspecto, medidos en Kb. La métrica se define como un cociente entre los valores aplicados a ambas

implementaciones. Un valor de resultado mayor que uno significa un impacto positivo de la utilización de aspectos. La definición de la métrica es

$$Beneficio\_POA = \frac{Tamaño\_sin\_Aspectos * Factor\_de\_Desarrollo\_Java}{Tamaño\_con\_Aspectos * Factor\_de\_Desarrollo\_AspectJ}$$

donde

$$Tamaño\_Con\_Aspectos = Tamaño\_Funcionalidad\_Básica + Tamaño\_Aspecto$$

Para aplicar esta métrica a nuestro ejemplo, consideramos el tamaño físico de las clases especificados en la siguiente tabla.

Clases	**Con aspectos**	**Sin aspectos**
Tftpd	6,14 Kb	10,1 Kb
TftpServerConnection	5,09 Kb	6,35 Kb
TftpWriteConnection	8,49 Kb	12,1 Kb
Aspect logging	8,30 Kb	-

Luego,

$$Tamaño\_Sin\_Aspectos = (10,1 + 6,35 + 12,1)Kb = \mathbf{28,55Kb}$$

$$\begin{aligned} Tamaño\_Con\_Aspectos &= Tamaño\_Funcionalidad\_Básica + Tamaño\_Aspecto \\ &= (6,14 + 5,09 + 8,49)Kb + 8,30Kb \\ &= 19,72Kb + 8,30Kb = \mathbf{28,02Kb} \end{aligned}$$

reemplazando estos valores en la fórmula de la métrica se obtiene

$$Beneficio\_POA = \frac{28,55Kb * 10hp}{28,02Kb * 2hp} = \frac{285,5}{56,04} = \mathbf{5,09}$$

Como ya se ha mencionado, la utilización de aspectos produce un código de funcionalidad básica más "puro", debido a que el comportamiento de los conceptos entrecruzados queda encapsulado en los aspectos. Para definir el porcentaje de código que se reduce al introducir aspectos, se define la segunda métrica, llamada *Limpieza*. La misma, mide el porcentaje de código que se depura de la funcionalidad básica al introducir aspectos. Utiliza el tamaño de las clases, medido en Kb. La definición es como se detalla a continuación

$$Limpieza = \frac{(Tamaño\_Sin\_Aspectos - Tamaño\_Funcionalidad\_Básica) * 100}{Tamaño\_Sin\_Aspectos}$$

Aplicando esta métrica en el caso de estudio, resulta el siguiente valor:

$$Limpieza = \frac{(28,55 - 19,72)Kb * 100}{28,55Kb} = \mathbf{30,93}$$

Por último, se compara el tamaño de las clases que implementan las funcionalidades básicas de las dos alternativas, expresado en líneas de código(LDC).

Clases	**Con aspectos**	**Sin aspectos**
Tftpd	199 ldc	318 ldc
TftpServerConnection	176 LDC	212 LDC
TftpWriteConnection	339 LDC	471 LDC
Tamaño total	714 LDC	1001 LDC

5 Trabajos Relacionados

R. Laddad [1], enumera algunas de las principales ventajas de la POA. En principio, ayuda eficazmente a superar los problemas causados por el *Código Mezclado* y *Código Diseminado*. En cuanto a la modularización, la POA logra separar cada concepto con mínimo acoplamiento, resultando en implementaciones modularizadas aún en la presencia de conceptos que se entrecruzan. Esto lleva a un código más limpio, menos duplicado, más fácil de entender y de mantener. A su vez la separación de conceptos permite agregar nuevos aspectos, modificar y / o remover aspectos existentes fácilmente, lo que permite una mejor evolución, y mayor reusabilidad. Durante el diseño, el desarrollador se ve beneficiado ya que puede retrasar las decisiones sobre requerimientos actuales o futuros, debido a que permite, luego, implementarlos separadamente e incluirlos automáticamente en el sistema. Esto último resuelve el dilema del arquitecto sobre cuántos recursos invertir en la etapa de diseño, cuándo la cuestión planteada es "demasiado diseño".

Las desventajas del paradigma surgen del hecho de que la POA está en su infancia. En el análisis de Guezzi et.al.[12] sobre la POA se mencionan tres falencias. La primera remarca posibles choques entre el código funcional, expresado en el lenguaje base, y el código de aspectos expresado en los lenguajes de aspectos. Usualmente, estos choques nacen de la necesidad de violar el encapsulamiento para implementar los diferentes aspectos, sabiendo de antemano el riesgo potencial que se corre al utilizar estas prácticas. La segunda es la posibilidad de choques entre los aspectos. El ejemplo clásico es tener dos aspectos que trabajan perfectamente por separado, pero al aplicarlos conjuntamente resultan en un comportamiento anormal. La tercera, trata con posibles choques entre el código de aspectos y los mecanismos del lenguaje. Uno de los ejemplos más conocidos de este problema es la *anomalía de herencia* [9]. Dentro del contexto de la POA, el término puede ser usado para indicar la dificultad de heredar el código de un aspecto en la presencia de herencia.

Con respecto a métricas para el desarrollo orientado a aspectos, a nuestro entender existen escasas publicaciones sobre el tema. Zhao [8] propone métricas específicas para cuantificar el flujo de control en software orientado a aspectos. Las mismas se pueden utilizar para medir la complejidad de un programa orientado a aspectos.

6 Conclusiones y Trabajo Futuro

A partir de los resultados obtenidos en las métricas presentadas podemos concluir que la utilización de aspectos fue ampliamente exitosa. Primero, la métrica *Beneficio_POA* arrojó el valor 5,09 que al ser mayor que 1 se traduce en un impacto positivo. Segundo, el valor 30,93 calculado por la métrica *Limpieza* indica que el código original se ha limpiado casi en un 31%. Es importante notar que la cantidad de líneas de código de la funcionalidad básica se redujo en 300 líneas de código, que representa el 30% del total, confirmando el valor obtenido en la métrica de *Limpieza*.

Por otra parte, es válido preguntarse cuál es el costo de trabajar con AspectJ. Esta cuestión es lo que hace tan atractiva a la POA, y en particular a AspectJ, y la respuesta es que el costo es mínimo. Si el programador ya cuenta con conocimientos sólidos en Java, capacitarse en AspectJ es una tarea trivial, debido a que este último lenguaje es una extensión del primero.

Aprender a trabajar con aspectos en AspectJ es sumamente sencillo, dada la naturaleza declarativa de los mismos. Una característica importante de los aspectos es que son fácilmente "enchufados" y " desenchufados" (plug-in, plug-out) a la funcionalidad básica. Esto le permite al programador ir experimentando con aspectos, sin tener que modificar la funcionalidad básica, resultando en un aprendizaje gradual y efectivo. Todo lo que hemos experimentado y analizado, puede resumirse en una sola afirmación, que establece por sí misma la importancia de este nuevo paradigma *"con muy poco esfuerzo, se obtienen ganancias importantes en la calidad, no sólo del producto final, sino también en el proceso que desarrolla el producto"*. Los resultados encontrados en la bibliografía, y los que hemos obtenido en este trabajo son más que prometedores, y nos hacen pensar que la POA es una de las ramas con mayor futuro dentro de la Ingeniería de Software.

En base a la experiencia lograda con el desarrollo del caso de estudio del presente trabajo, es posible agregar otras tres ventajas. En primer término, al separar la funcionalidad básica de los aspectos, se aplica con mayor intensidad el principio de dividir y conquistar. También hay que destacar que el ambiente de desarrollo es un ambiente de N-dimensiones, donde es posible implementar el sistema con las dimensiones que sean necesarias, y no en una única dimensión "sobrecargada". Por último, la POA enfatiza el principio de mínimo acoplamiento y máxima cohesión.

Nuestro análisis, nos permite observar también, que los lenguajes orientados a aspectos actuales, no cuentan con mecanismos lingüísticos suficientemente poderosos. Estos mecanismos son necesarios para respetar por completo todos los principios de diseño, como por ejemplo, el encapsulamiento.

La POA es un nuevo paradigma que aún adolesce de madurez y formalidad. Debido a esto, plantea nuevas líneas de investigación tales como: analizar cómo aplicar aspectos en las otras etapas del ciclo de vida del software, en el análisis, en el diseño, en el testing y en la documentación; investigar sobre la formalización de los aspectos; observar cómo es la integración de los aspectos con las aproximaciones, métodos, herramientas, y procesos de desarrollo existentes; desarrollar herramientas orientadas a aspectos que respeten los principios de diseño, entre otras.

Nuestro próximo paso es investigar cómo incluir aspectos en el diseño de un sistema de software. El diseño es una parte crítica de cualquier proceso, y es necesario investigar la

forma de incluir los aspectos durante esta etapa. La mayoría de los trabajos de investigación relacionados, buscan maneras convenientes de extender UML, para que este lenguaje de diseño sea capaz también de expresar aspectos. Debido a que UML es el lenguaje de diseño con mayor respaldo dentro de la Ingeniería de Software, y es considerado como un estándar, buscaremos los medios para expresar aspectos en UML.

Referencias

[1] *"I want my AOP"*. Ramnivas Laddad. Part 1,2,3 from JavaWorld, Enero-Marzo-Abril 2002.

[2] *"Software Metrics"*. Norman Fenton, Lawrence Pfleeger. PWS Publishing Company, 1997.

[3] *"Programación Orientada a Aspectos: Análisis del Paradigma"*. Asteasuain Fernando, Bernardo Ezequiel Contreras. Tesis de Licenciatura en Ciencias de la Computación. Universidad Nacional del Sur, noviembre de 2002.

[4] *"Aspect-Oriented Programming"*. Gregor Kickzales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Proceedings, European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. Junio 1997.

[5] Página de Mark Benvenuto: http://www.cs.columbia.edu/~markb/

[6] Página de AspectJ: http://www.aspectcj.org

[7] *"Untangling Code"*. Claire Tristram. in the January/February 2001 issue of the Technology Review.

[8] *"Towards a Metric Suite for Aspect-Oriented Software"*. Jianjun Zhao. Technical Report SE-136-25, Information Processing Society of Japan (IPSJ), Marzo 2002.

[9] *"Analysis of inheritance anomaly in object-oriented concurrent programming languages"*. S. Matsuoka, A. Yonezawa. in Research Directions in Concurrent Object-Oriented Programming (G. Agha, P.Wegner, and A. Yonezawa, eds.),pp. 107-150, Cambridge, MA: MIT Press, 1993.

[10] *Request For Comment 783 y 1350: The TFTP Protocol (Revision 2), junio 1981 - junio de 1982*. http://www.ietf.org/

[11] *"Getting Started with AspectJ"*. Gregor Kickzales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G.Grisnold. Communications of the ACM (CACM) Vol. 44 Nro.10, Octubre 2001.

[12] *"Language Support for Evolvable Software: An Initial Assessment of Aspect-Oriented Programming"*. Gianpaolo Cugola, Carlo Ghezzi, Mattia Monga. Proceedings of the International Workshop on the Principles of Software Evolution, IWPSE99, Julio 1999.