Tesis de grado en Ingeniería en Informática

Utilización de programación orientada a aspectos en aplicaciones enterprise

Tesista: Nicolás Martín Paez

Directora: Lic. Rosita Wachenchauzer

Fecha: Noviembre 2007

FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Resumen

Las aplicaciones enterprise son aplicaciones que dan soporte a los procesos de negocio de una organización. Como tales, se caracterizan por manejar grandes volúmenes de información persistente, la cual es accedida en forma concurrente por varios usuarios mediante diversas interfaces de usuario. Estas aplicaciones deben evolucionar de forma rápida ante los cambios del negocio, provocando el menor impacto posible. Esta necesidad de rápida evolución obliga a que estas aplicaciones deban cumplir con ciertos atributos técnicos y de calidad como escalabilidad, mantenibilidad, transaccionalidad, etc. La implementación de dichos atributos resulta ser de difícil modularización cuando se utilizan las técnicas tradicionales de programación.

Es aquí donde la programación orientada a aspectos entra en juego. Esta técnica de programación surgida a fines de los 90' y con importante crecimiento en los últimos años, busca facilitar la modularización de aquellas incumbencias transversales que resultan de difícil modularización con las técnicas tradicionales de programación.

El objetivo del presente trabajo es analizar la utilización de la programación orientada aspectos en las aplicaciones enterprise, proponiendo soluciones basadas en aspectos para las problemáticas características de dichas aplicaciones y mostrando su uso mediante el desarrollo de una aplicación de referencia.

Agradecimientos

A mi mamá Silvia y a mi hermano Germán por su apoyo incondicional y su aliento continuo.

A Coty, por su paciencia infinita durante esta larga carrera.

A Rosita, directora de este trabajo, por su confianza en mi y su paciencia.

A Carlos por sus comentarios constructivos.

A Marcio, a Fran, a Caro y a todo el equipo de Nardolandia, por las experiencias compartidas.

A los profesores, compañeros, amigos y colegas, que de una u otra forma han colaborado en este trabajo.

Índice

Capitulo I: Introducción

Problemática

La evolución de la informática y de las telecomunicaciones en las últimas dos décadas ha colocado a los sistemas informáticos en un rol preponderante dentro de las organizaciones, llegando incluso en algunos casos a producir cambios radicales en el negocio. Esta situación se ha visto potenciada por el auge de internet, dando origen a nuevos modelos de negocio, donde los sistemas informáticos forman parte del corazón del negocio. Entre las organizaciones más representativas de estos cambios, pueden citarse los sitios de remate y compra en línea, como eBay y Amazon.

En este contexto, los departamentos de sistemas se han visto forzados a responder de forma inmediata a los cambios del negocio, lo cual ha planteado un interesante desafío a quienes deben diseñar las aplicaciones, ya que decisiones erróneas en el diseño, pueden resultar muy costosas en el mediano y largo plazo cuando una aplicación deba evolucionar. Estas aplicaciones que dan soporte a los procesos de negocio de una organización han sido denominadas aplicaciones enterprise (AE).

El desafío que proponen estas aplicaciones enterprise a los profesionales de sistemas también ha tenido repercusión en los ámbitos académicos, donde ha impulsado nuevas áreas de investigación relacionadas a la ingeniería de software.

Como respuesta a este nuevo escenario planteado por las aplicaciones enterprise, es que en los últimos años han surgido nuevas tendencias, como los métodos de desarrollo ágil y la orientación a servicios. Entre estas nuevas tendencias se encuentran las técnicas avanzadas de separación de incumbencias, las cuales pretenden ofrecer herramientas para lograr una mejor modularización de la aplicaciones. Esta temática ha despertado el interés de varios referentes de la industria y del mundo académico, dando origen a la comunidad de desarrollo de software orientado a aspectos (Aspect-oriented Software Development, AOSD)[AOSD]. Una de las técnicas más popularizadas dentro de este área de investigación es la programación orientada a aspectos (Aspect-oriented Programming, AOP). Múltiples trabajos han propuesto el uso de AOP para la resolución de ciertas problemáticas que afectan a las aplicaciones enterprise como seguridad, manejo de excepciones y persistencia. Los trabajos más destacados en este sentido son los de Rod Johnson y Renaud Pawlak.

Objetivo

El objetivo de la presente tesis es analizar la utilización de la programación orientada a aspectos en el desarrollo de aplicaciones enterprise, repasando y extendiendo lo propuesto por Johnson y

Pawlak, pero centrandonos tecnológicamente en la plataforma .NET.

A lo largo del trabajo se analizarán las problemáticas de las AE en las que el uso de AOP podría resultar beneficioso. Luego del análisis, se planteará el caso de una aplicación hipotética para mostrar como implementar sobre plataforma .NET las propuestas desarrolladas.

Alcance

El trabajo está centrado en las dos problemáticas que se consideran distintivas de las aplicaciones enterprise: los atributos técnicos (y de calidad) y la lógica de negocio. Queda explícitamente fuera de alcance el análisis de las cuestiones relacionadas a la capa de presentación e interacción con el usuario como así también la influencia de AOP en el proceso de desarrollo de software.

Organización de la tesis

El presente trabajo está estructurado en una serie de capítulos, cuyo contenido se describe resumidamente a continuación:

1. **Introducción**: describe la problemática enfrentada y objetivos del presente trabajo.
2. **Programación orientada a aspectos**: introduce al lector en el paradigma de la programación orientada aspectos, presentado los conceptos centrales, sus beneficios y limitaciones. Al mismo tiempo se presentan algunas herramientas destacadas que implementan el paradigma y se hace un breve resumen del estado del arte en la materia.
3. **Aplicaciones enterprise**: trata sobre las aplicaciones enterprise, caracterizándolas y describiendo las arquitecturas y tecnologías más utilizadas en la actualidad para su implementación.
4. **Hacia las aplicaciones enterprise con AOP**: prepara el terreno para comenzar a atacar el objetivo principal de la tesis, se identifica la utilidad que podría tener AOP en el desarrollo de las AE y se analizan algunos mitos y leyendas sobre AOP. El capítulo finaliza describiendo la arquitectura de base sobre la que se planteará el resto del trabajo.
5. **Incumbencias técnicas**: analiza las incumbencias técnicas características de las AE. Para cada incumbencia identificada se mencionan las soluciones actuales y se propone -cuando aplica- una solución basada en AOP.
6. **Incumbencias del negocio**: analiza las incumbencias del negocio sobre una aplicación concreta, proponiendo el uso de AOP para el modelado de ciertas incumbencias de negocio.
7. **La foto completa**: describe de forma completa la problemática de una AE, para la cual se plantea una solución tradicional y otra solución basada en AOP.

8. **Conclusiones**: describe los resultado del trabajo de investigación, las conclusiones extraídas del mismo y algunas posible líneas de investigación.

Adicionalmente a los capítulos mencionados, el trabajo ofrece cierto contenido adicional en forma de apéndices para aquellos lectores que deseen profundizar sobre temas que no hacen al cuerpo principal de conocimiento de la tesis.

Notas para el lector

A continuación se hacen algunas aclaraciones que pueden resultar de utilidad para el lector del presente trabajo.

Notaciones

A lo largo del trabajo se mostrará código en lenguaje C# y Java, por ser los principales lenguajes utilizados en la actualidad para el desarrollo de aplicaciones enterprise.

A la hora de mostrar código AOP, se utilizará una notación genérica, con el fin de que resulte clara e intuitiva para el lector.

Para comunicar cuestiones de diseño se utilizará la notación propuesta por el lenguaje unificado de modelado, pero dado que no hay una extensión única estandarizada para la representación de aspectos, se ha decidido utilizar una notación ad-hoc.

Traducciones

Se ha decidido no traducir algunos términos, principalmente aquellos referentes a terminología específica de AOP. Esto se debe a que existe escasa documentación en castellano, y aventurarse a una traducción no resulta apropiado.

Herramientas utilizadas

Para la redacción de este trabajo se utilizó el paquete de oficina Open Office. Los diagramas fueron realizados con Rational Software Modeler.

La aplicación de referencia fue desarrollada utilizando el entorno de desarrollo Microsoft Visual Studio 2005 Team Developer Edition. Los gráficos de la aplicación de referencia fueron realizados con Paint.NET.

Material adicional

Tanto el presente texto, como el código fuente del caso de estudio y algunos materiales complementarios utilizados y/o generados durante la elaboración de esta tesis puede encontrarse en la siguiente ubicación: http://www.fi.uba.ar/~npaez.

Capitulo II: Programación orientada a aspectos

Introducción

El problema

Desde sus comienzos los lenguajes de programación han evolucionado proveyendo cada vez mejores técnicas de modularización. Sin duda la programación orientada a objetos (POO) ha sido una de las técnicas de modularización más potentes, pero a pesar de ello siguen existiendo ciertas **incumbencias** (concerns) que ni la POO ni las técnicas precedentes de programación han logrado modularizar de manera efectiva. La dificultad de modularizar estas incumbencias, se debe a que las técnicas tradicionales de programación proveen la posibilidad de descomponer el problema de acuerdo a una única dimensión (la dominante), permitiendo un efectiva modularización de las incumbencias de dicha dimensión a costa de sacrificar una buena modularización de las incumbencias de las demás dimensiones. Este problema ha sido denominado "la tiranía de la descomposición dominante" [Tarr99] y como consecuencia del mismo las incumbencias de las dimensiones no dominantes adquieren una naturaleza transversal que les otorga el nombre de ***incumbencias transversales*** (cross-cutting concerns, CCC).

Como consecuencia directa de estos CCC aparecen dos fenómenos no deseados. El primero de ellos es denominado ***código disperso*** (code scattering) y se da cuando existen invocaciones a un método esparcidas por toda la aplicación.

El segundo de estos fenómenos es el llamado ***código mezclado*** (code tangling) y se da cuando una clase tiene código que nada tiene que ver con su esencia, lo cual disminuye su cohesión y viola uno de los principios básicos del diseño de objetos: el principio de única responsabilidad.

Entre las consecuencias de estos dos fenómenos se cuentan acoplamiento, dificultad de mantenimiento, menor claridad del código y todas las demás consecuencias de una pobre modularización.

Para comprender mejor esta problemática veamos un ejemplo. Supongamos una simple aplicación de banca electrónica donde los clientes de un banco pueden consultar el estado de sus cuentas, realizar transferencias de fondos y administrar sus datos personales. Adicionalmente consideremos los siguientes dos requisitos, consecuencia de las políticas operacionales del banco.

- Los datos del cliente y de sus cuentas sólo pueden ser modificados por usuarios con los

correspondientes permisos.

- Toda modificación de datos realizada en la aplicación debe ser auditada dejando constancia de la modificación realizada y del momento de la modificación.

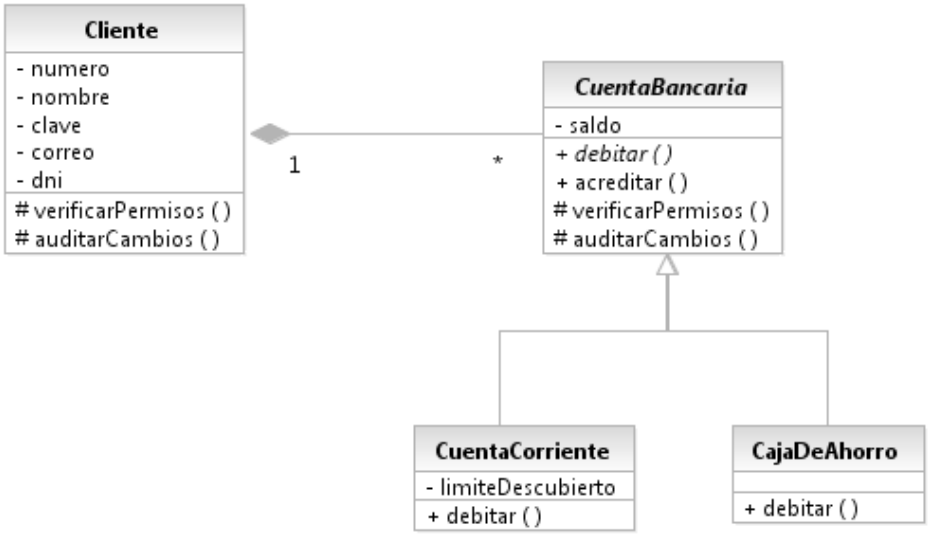

Figura 1: Diagrama de clases del dominio de banca electrónica

La figura 1 muestra un posible diagrama de clases simplificado para el mencionado contexto. Como es de esperar si utilizamos un enfoque orientado a objetos, tendremos una clase *Cliente* y una jerarquía de clases *CuentaBancaria* que contempla los distintos tipos de cuenta.

Tanto la clase *Cliente* como *CuentaBancaria*, tienen métodos *verificarPermisos* y *auditarCambios*, que claramente exceden la esencia de cada clase. Tal como está planteada esta solución, los dos métodos en cuestión están expresando la existencia de dos incumbencias transversales: autorización y auditoria[1].

Hasta el momento, estas incumbencias transversales no representan mayor problema, pero supongamos que una vez terminada la aplicación, nos piden algunos cambios.

- Modificar la forma en que se auditan los datos debido a que se requiere más detalle de información.
- Contemplar un nuevo tipo de cuenta, *PlazoFijo*,con una política distinta de permisos.

Por como está planteada esta solución, estos cambios implicarían la modificación de las clases *Cliente* y *CuentaBancaria* para reflejar los cambios en la auditoria. Al mismo tiempo debería agregarse una nueva clase *PlazoFijo*, la cual también debería contemplar las cuestiones de la verificación de permisos y auditoria.

Bien, aplicados estos cambios, la solución sigue funcionando, pero hay algunas cuestiones a

1 Si bien la solución planteada puede no ser la más óptima, su simplicidad permite poner de relevancia la existencia de las incumbencias transversales.

considerar.

- A pesar que ninguno de los requisitos de cambio mencionaba explícitamente al cliente, la clase *Cliente* debió ser modificada.
- El agregado de un nuevo concepto del dominio (plazo fijo), nos implicó no sólo la creación de una nueva clase, sino que además se debió contemplar cuestiones adicionales como lo son la auditoria y la autorización.
- Los dos puntos anteriores ponen de relevancia una vez más, la naturaleza transversal de la autorización y la auditoria.

Habiendo entendido las desventajas de la solución planteada en cuanto al manejo de las incumbencias transversales uno podría hacer uso de algunas técnicas de diseño, como patrones o prácticas de refactoring, con el fin de lograr una solución que maneje de mejor forma las incumbencias transversales. Una de las técnicas de factible aplicación es la orientación a aspectos.

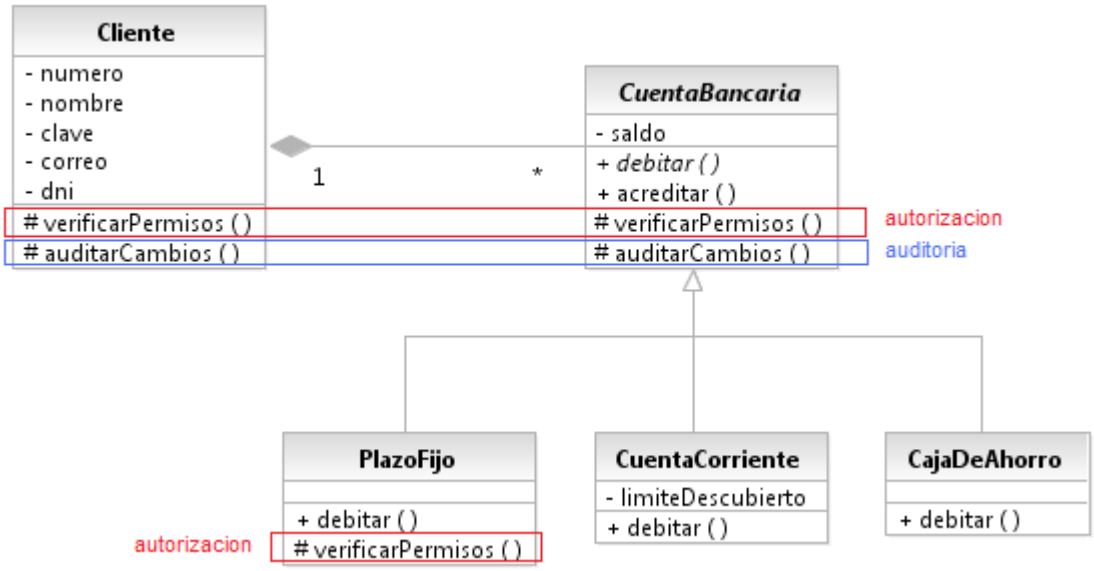

Figura 2: Incumbencias transversales en las clases del dominio de banca electrónica

Desarrollo de software orientado a aspectos

El desarrollo de software orientado a aspectos constituye un área de investigación centrada en lograr una efectiva modularización de incumbencias. Dentro de este área existen varios enfoques la mayoría de los cuales son presentados en [Filman04]. Entre estos enfoques se encuentran Adaptative programming, Composition Filters, Multidimensional Seperation of Concerns, Subject-Oriented Programming y Aspect-Oriented Programming, entre otros. Cada uno de estos enfoques ofrece distintos mecanismos para la separación y composición de incumbencias.

Conceptualmente existen dos paradigmas para la separación de las incumbencias, uno simétrico y otro asimétrico [Harrison02].

El paradigma ***simétrico*** plantea, que para lograr una efectiva separación de incumbencias, es necesario romper con la tiranía de la descomposición dominante. Para ello todas las incumbencias son modeladas como componentes de primera clase, todos con la misma estructura y donde ninguno es más básico que otro. Podría decirse que el estandarte de este paradigma es el enfoque planteado por la técnica denominada ***Multidimensional Separation of Concerns***.

El paradigma ***asimétrico*** por su parte, hace una diferenciación entre las incumbencias dominantes (también llamadas centrales o funcionales) las cuales modela mediante componentes base que luego son compuestos con las incumbencias no dominantes (también llamadas aspectos, no centrales o secundarias). Como se vio en el ejemplo de la sección anterior, las incumbencias no centrales en un paradigma asimétrico, tienen generalmente una naturaleza transversal. El exponente más representativo de este paradigma es la ***programación orientada a aspectos*** (aspect oriented programming, AOP).

El término programación orientada a aspectos surgió de un trabajo de Gregor Kiczalez y su equipo en el centro de investigación de Xerox, en Palo Alto, hacia fines de los 90' [Kiczales97]. El principal objetivo de AOP es lograr una efectiva modularización de los CCC de manera ***transparente*** (oblivious) y soportando distintas formas de ***cuantificación***. Desde el punto de vista de AOP, una aplicación consiste en un conjunto de incumbencias dominantes fácilmente modularizables (componentes base), más un conjunto de incumbencias transversales de difícil modularización (aspectos). La mezcla de los componentes base y los aspectos da como resultado la aplicación final. Dicha mezcla se realiza por medio de un proceso denominado ***entretejido*** (weaving). La asimetría entre los componentes base y los aspectos pone de manifiesto en enfoque asimétrico de AOP en lo que a separación de incumbencias respecta.

Es importante destacar las dos propiedades de AOP mencionadas en el párrafo anterior: la transparencia y la cuantificación.

La transparencia tiene que ver con la posibilidad de introducir aspectos en el código base de forma transparente, de manera tal que quien vea el código base no pueda predecir la existencia de los aspectos. Esta propiedad puede resultar polémica, pero es justamente una de las propiedades que distinguen a AOP de las técnicas predecesoras.

Por su parte, la cuantificación brinda la posibilidad de expresar enunciados (statements) del tipo:

*En todo **programa P**, cuando se dé la **condición C**, ejecutar la **acción A**.*

El programa P no es más que un programa tradicional constituido por componentes base, mientras que la acción A es parte de un aspecto que será entretejido con el programa P en el punto indicado por la condición C. La cuantificación puede ser estática, si la condición C trata sobre la estructura del programa P, o bien dinámica, si la condición C tiene que ver con algún suceso de la ejecución de P. Al mismo tiempo, la cuantificación estática puede ser de caja negra (cuantificación

sobre los elementos de la interfase pública del programa P) o de caja blanca (cuantificación sobre la estructura interna del programa P).

Elementos del paradigma

Joinpoint

Como ya lo hemos mencionado, un ***aspecto*** es una abstracción que permite modularizar una incumbencia transversal. Los puntos en la ejecución de los componentes base donde es posible el entretejido de aspectos son denominados ***joinpoints.*** Algunos ejemplos de joinpoints son: una llamada a un método, la creación de una instancia, el manejo de una excepción, la ejecución de un ciclo, el retorno de un método, la asignación de un valor a una variable, la modificación de un atributo, entre otros. El modelo de jointpoints de una herramienta AOP, define los tipos de cuantificaciones que dicha herramienta soporta y por consiguiente la potencia de la herramienta.

```
1 public class Foo
2 {
3     private int cantidadDeLlamadas;
4     public Foo()
5     {
6         try
7         {
8              System.out.println("Foo.new()");
9         }
10        catch(IOException e)
11        {
12             RuntimeException re = new RuntimeException(e);
13             throw re;
14        }
15    }
16    public void doFoo()
17    {
18        System.out.println("Foo.doFoo()");
19        cantidadDeLlamadas += 1;
20    }
21    public static void main(String[] args)
22    {
23        Foo foo = new Foo();
24        foo.doFoo();
25    }
26 }
```

Figura 3: código de ejemplo con varios joinpoints

En el fragmento de código de la figura 3 existen los siguientes joinpoints.

Número de línea	*Joinpoint*
5	Ejecución del método Foo.new()
10	Manejo de una excepción del tipo IOException
12	Llamada al método RuntimeException.new(Exception)

Número de línea	*Joinpoint*
13	Lanzado de una excepción del tipo RuntimeException
17	Ejecución del método Foo.doFoo()
19	Modificación del atributo Foo.cantidadDeLlamadas
23	Llamada al método Foo.new()
24	Llamada al método Foo.doFoo()

Poincut

A un conjunto definido de joinpoints se lo llama ***pointcut***. A diferencia de los jointpoints, los pointcuts, son definidos por el programador. Algunos pointcuts simples podrían ser todas las ejecuciones del constructor de una determinada clase, todas las llamadas al método *Pow* de la clase *Math* que reciba como parámetro un 5 o el lanzado de una excepción de la clase *RuntimeException*. Todas las herramientas AOP brindan un conjunto de designadores de pointcuts que en conjunto con el uso de expresiones regulares permiten la definición de los pointcuts. La siguiente tabla muestra algunos designadores de pointcuts de aspectj.

Designador de pointcuts	*Tipo de Joinpoints*
Call(X)	Llamadas a métodos / constructores de la clase X
Handler(X)	Manejadores de excepciones de la clase X (bloques catch)
Cflow(X)	Joinpoints en el contexto del jointpoint X
Target(X)	Joinpoints cuyo objetivo involucre a instancias de clase X
args(X)	Joinpoint que reciban argumentos de la clase X
Annotation(X)	Joinpoint que contengan la anotacion X
NOTA: En la mayoría de las herramientas X suele ser una expresión regular	

Tabla 1: Designadores de pointcuts de AspectJ

Advice

El código del aspecto que se ejecuta asociado a un determinado pointcut es denominado ***advice***. Conceptualmente los advices contienen el comportamiento del aspecto. Los advices puede ser básicamente de 3 tipos.

- Before: se ejecutan previos a la ejecución del joinpoint asociado.

- After: se ejecutan después de la ejecución del joinpoint asociado.
- Around: se ejecutan antes y después del joinpoint asociado, pudiendo incluso reemplazar la ejecución del joinpoint.

Declaraciones de intertipo

Otro elemento de AOP, muchas veces pasado por alto, es la ***declaración de intertipos*** (también llamada introducción o mixin) que permite extender la estructura de un componente base mediante el agregado de miembros (métodos y atributos) e interfases.

La declaración de un intertipo forma parte de un aspecto y es dicho aspecto el que define cuales serán los componentes base que se extenderán con el intertipo. Hay que destacar que las extensiones realizadas por un intertipo no son visibles[2] para los componentes base, ni tampoco para el propio componente que ha sido extendido.

Pongamos un ejemplo para entender esta situación: supongamos que para poder persistir objetos en una base de datos es necesario que todo objeto implemente la interfase *ObjetoPersistente*, la cual define un método para obtener el identificador del objeto a utilizarse como clave primaria en la base de datos.

Utilizando AOP podría definirse un aspecto que con una declaración de intertipo se encargue de las siguiente cuestiones.

- Hacer que todas las clases que deban persistirse implementen la interfaz *ObjetoPersistente.*
- Agregar la implementación del método requerido por la interfaz *ObjetoPersistente* en todas las clases que deban persistirse.

Ahora, dado un objeto que al que se le ha agregado el mencionado aspecto para implementar la interfaz *ObjetoPersistente*, si algún objeto quisiera tratarlo como un *ObjetoPersistente*, debería previamente castearlo a dicho tipo, ya que lo agregado por las declaraciones de intertipos no es directamente visible para los componentes base.

Aspecto

Habiendo introducido todos los elementos característicos de AOP estamos en condiciones de definir el concepto de alto nivel que conjuga todos los elementos, el aspecto.

Un aspecto es una entidad que modulariza una incumbencia transversal mediante la definición de un conjunto de pointcuts, advices y declaraciones de intertipo.

Como veremos más adelante dependiendo de la herramienta utilizada, puede que la definición de un aspecto se haga enteramente en un archivo fuente, tal como ocurre con una clase en java, o bien que los advices e introducciones de definan en un archivo fuente y los poincuts en un archivo de

2 No son visibles sin hacer un casteo explícito.

configuración de la aplicación final.

Solución AOP al ejemplo planteado

Habiendo introducido los elementos característicos del paradigma, estamos en condiciones de plantear una solución AOP a la problemática de banca electrónica.

Como muestra la figura 4, se remueven los métodos *auditarCambios* y *verficarPermisos* de las clases de dominio reubicando la lógica de estos métodos en los aspectos *Auditor* y *Autorizador*.

La figura 5 muestra la definición del aspecto Auditor con un pointcut que identifica todas las llamadas a setters[3] de las clases *Cliente*, *CuentaBancaria* y descendientes sus descendientes. Asociado a dicho pointcut define una advice de tipo after, dentro del cual debería codificarse la registración de los cambios.

La figura 6 muestra la definición del aspecto *Autorizador* con dos pointcuts: uno que identifica todas las llamadas a métodos de las clases *PlazoFijo* y otro que identifica las llamadas a métodos de las *Cliente*, *CuentaBancaria* y descendientes de estas, que no estén incluidas en el poincut anterior. Asociado a cada pointcut hay un advice de tipo before, dentro del cual debería codificarse la verificación de permisos.

Para que los aspectos previamente descriptos puedan cumplir con su cometido es necesario que los mismos tengan acceso a la información de contexto, la cual suele estar disponible mediante artefactos provistos por el lenguaje AOP.

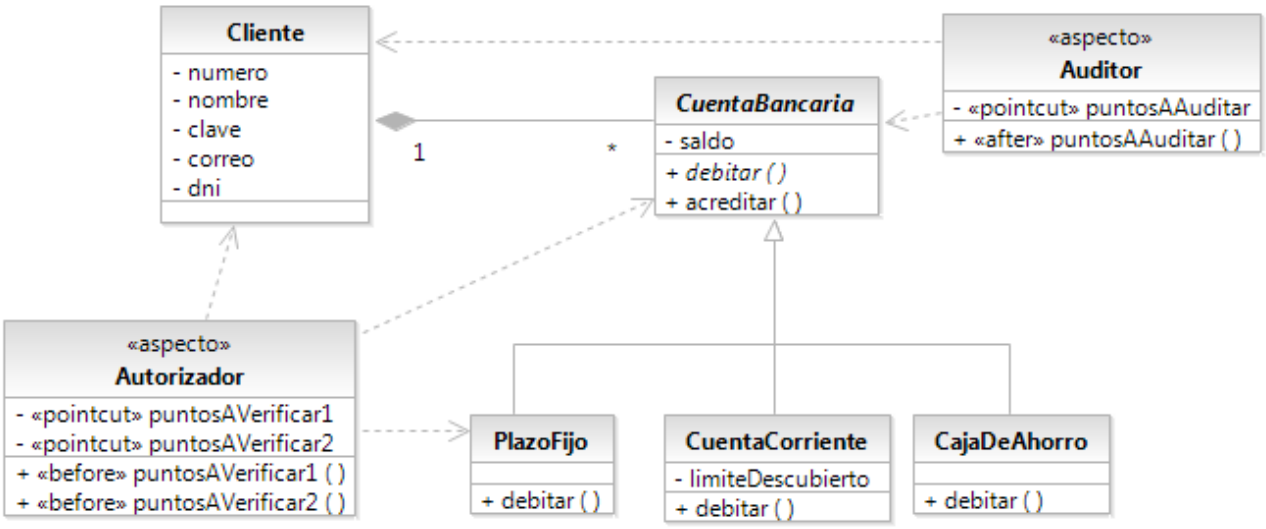

Figura 4: Diagrama de la solución AOP del dominio de banca electrónica

3 Se suele llamar setters a los métodos que permiten la modificación de atributos de una clase.

```
public aspect Auditor {

      pointcut puntosAAuditar(): call(* Cliente+.set*) ||
                                 call( * CuentaBancaria+.set*);

      after(): puntosAAuditar(){
            /*
             * Aquí iria la registración de los cambios, los cuales se
             * podrían acceder usando la información de contexto
             */
      }
}
```

Figura 5: Fragmento de código del aspecto Auditor

```
public aspect Autorizador {

      pointcut puntosAVerificar1(): call( * PlazoFijo.*(..));

      pointcut puntosAVerificar2(): call(* Cliente+.*(..)) ||
                                 call( * CuentaBancaria+.*(..))
                                 && !puntosAVerificar1();

      before(): puntosAVerificar1(){
            /*
             * Aquí iria la verificación de permisos para todas las clases
             * excepto PlazoFijo
             */
      }

      before(): puntosAVerificar2(){
            /*
             * Aquí iria la verificación de permisos
             * para la PlazoFijo
             */
      }
}
```

Figura 6: Fragmento de código del aspecto Autorizador

Herramientas AOP

Tipos de herramientas

Básicamente y muy a grandes rasgos podemos decir que existen dos tipos de herramientas AOP, diferenciadas principalmente por el modelo de entretejido, el cual puede ser estático o dinámico.

Las herramientas AOP de ***entretejido estático*** son los compiladores, pre compiladores y post compiladores. Estas herramientas toman como entrada los aspectos y los componentes base en forma de código fuente o binario y generan como salida código binario con los aspectos y los componentes base ya entretejidos. En general los componentes base se escriben en algún lenguaje orientado a

objetos, mientras que los aspectos se escriben en nuevos lenguajes con soporte de aspectos que generalmente son una extensión al lenguaje orientado a objetos de los componentes base. Este es el caso de AspectJ[4][aspectj], Aspect.NET [aspectnet] y AspectC++ [aspectc++], entre otros.

Las herramientas AOP de ***entretejido dinámico*** son generalmente frameworks desarrollados en un lenguaje orientados a objetos y están basados en proxies dinámicos. La definición del aspecto consta de dos partes. Por un lado se define el advice, extendiendo una clase base provista por el framework de aspectos. Por otro lado se realiza la definición de los pointcuts, la cual suele hacerse por medio de metadata, que muchas veces reside en archivos XML. Por lo general si un aspecto consta de más de un advice, es necesario crear una clase por cada advice.

Estos frameworks proveen una clase que funciona como una fábrica de objetos y que es el punto de acceso a la funcionalidad del framework. La figura 7 muestra las clases típicas de un framework AOP, hay que destacar que la clase *Proxy* suele ser creada en tiempo de ejecución. Esta fábrica de objetos interpreta los pointcuts definidos en la metadata y en base a eso a la hora de crear un objeto detecta si deben agregarse aspectos al objeto creado, en cuyo caso crea un proxy con el conjunto de advices que aplicarán al objeto en cuestión y devuelve una referencia al proxy en lugar de una referencia directa el objeto pedido. El diagrama de secuencia de la figura 8 ilustra este proceso.

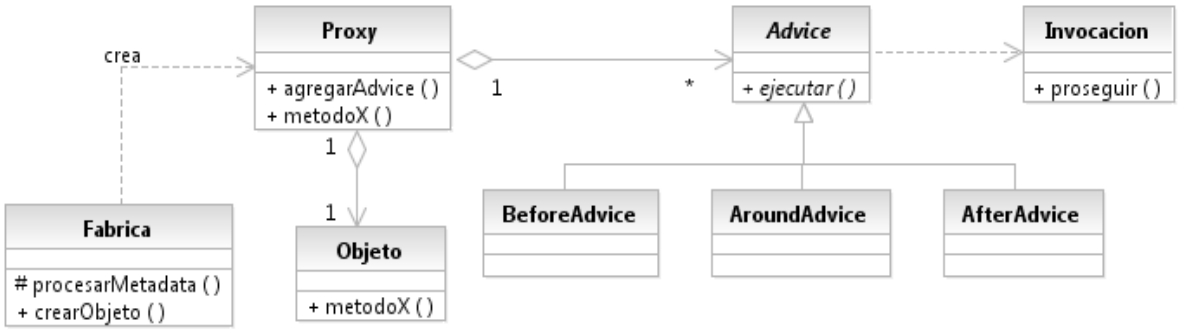

Figura 7: Clases típicas de un frameworks AOP.

4 A partir de la versión 1.2 AspectJ también soporta otras formas de weaving.

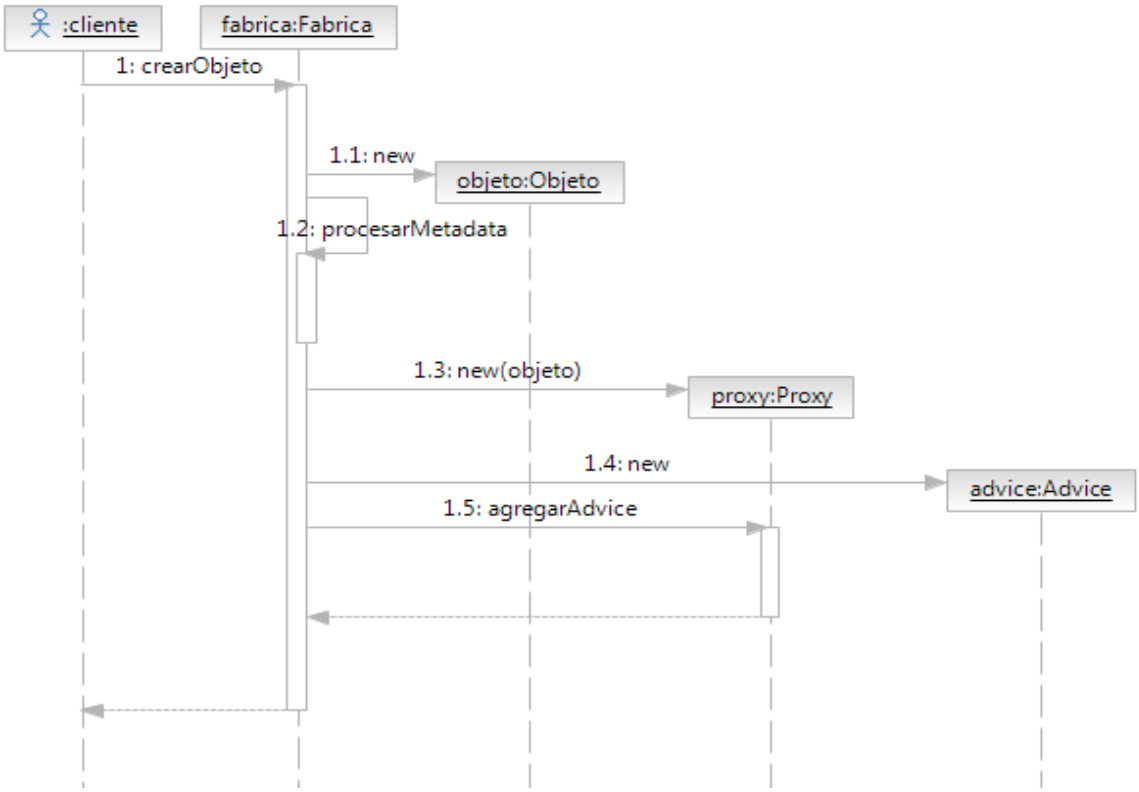

Figura 8: Creación de un objeto aspectizado por un framework AOP

De esta forma todas las llamadas al objeto con aspectos son interceptadas por el proxy que da intervención a los advices, los cuales actúan antes y/o después de la ejecución del objeto en cuestión. Ejemplos de herramientas de este tipo son: SpringFramework [spring], JbossAOP [jbossaop], Loom.NET [loom] y Naspect [naspect].

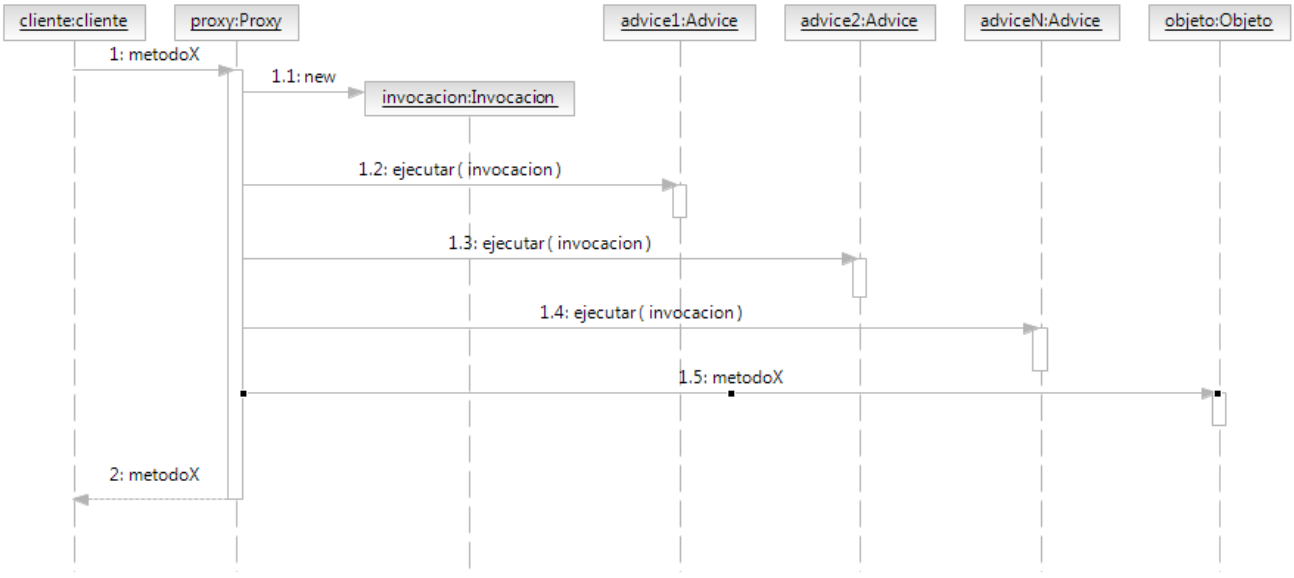

Figura 9: Secuencia de ejecución de un advice de tipo before

Hay un serie de cuestiones que caracterizan a cada tipo de herramienta.

- Mientras que las herramientas de entretejido estático brindan la posibilidad de detectar

errores en la definición de los pointcuts durante el desarrollo (tiempo de compilación), el uso de herramientas de entretejido dinámico obliga a esperar hasta el momento de ejecución para detectar dichos errores.

- Dado que la gran mayoría de herramientas de entretejido dinámico son frameworks, su curva de aprendizaje es mucho más amena que la de las herramientas de entretejido estático, ya que estas suelen requerir del aprendizaje de un nuevo lenguaje.
- Analizando las herramientas existentes, pareciera que las de entretejido estático son más completas desde el punto de vista conceptual que las de entretejido dinámico.
- Desde el punto de vista de implementación, resulta mucho más simple implementar una herramienta de entretejido dinámico que una de entretejido estático.

Si bien hasta el momento hemos hecho referencia a sólo dos tipos de entretejido -estático y dinámico- en tecnologías basadas en máquinas virtuales, existen distintas variantes de estos tipos de entretejido. En el caso de Java, el entretejido dinámico puede implementarse en tiempo de ejecución o también en tiempo de carga, para lo cual se reemplaza el cargador de clases de la máquina virtual, por un cargador de clases que realiza el trabajo adicional del entretejido. En estos casos el cargador de clases reemplaza lo que hemos denominado fábrica de objetos.

Entre las herramientas AOP existentes en la actualidad se destacan Aspectj, Jboss-AOP y SpringFramework, aunque más allá de estas tres existe una cantidad importante de herramientas casi todas ellas extensiones a lenguajes OO [aspectc++], [Loom.NET], [aspectDNG], [Naspect], [aspect#].

Sin duda, gran parte del camino de la evolución de AOP lo ha marcado AspectJ, siendo la primer herramienta estable y la de mayor difusión en la actualidad. Al mismo tiempo hay que destacar el aporte de SpringFramework, promoviendo el uso de AOP en ambientes enterprise. Es por esto que a continuación haremos una breve reseña de estas dos herramientas.

AspectJ, el génesis

AspectJ [aspectj] surgió como consecuencia del trabajo de Gregor Kiczales y su equipo en XPARC. A partir de la creciente actividad de la comunidad formada en torno a AspectJ, en 2001 aspectJ pasó a la órbita del proyecto Eclipse [eclipse]. Tal vez por ser la primer herramienta AOP, ha marcado en gran parte el rumbo de la comunidad AOSD. AspectJ es una extensión a Java.

Desde el punto de vista de la implementación, en un comienzo fue un pre compilador, pero actualmente es un compilador que genera Java Byte Code totalmente compatible con la JVM de Sun Microsystems. En el año 2005, AspectJ se fusionó con aspectwerkz, incorporando varias funcionalidades de este último relacionadas al entretejido dinámico. La mayoría de los libros sobre AOP en la actualidad tratan sobre AspectJ.

Spring Framework, el promotor

SpringFramework es un desarrollo de código abierto creado por Rod Johnson para facilitar el desarrollo de aplicaciones enterprise inicialmente en Java y actualmente también en .NET. Este framework consta de varios módulos, uno de los cuales es SpringAOP. SpringAOP utiliza entretejido dinámico y puede utilizarse independientemente de los otros módulos. Un punto interesante de SpringFramework, es que varios de sus módulos hacen uso del modulo AOP.

En el caso de la implementación Java, ha tenido un gran nivel de adopción convirtiéndose en el estándar de facto para algunas compañías. La versión 2.0 a dado un paso interesante hacia la integración con AspectJ.

En cuanto a la implementación .NET, el módulo AOP de Spring, es la implementación AOP más completa, estable y utilizada de AOP en la actualidad.

Estado del arte

Con sus escasos 10 años AOP es aún paradigma en desarrollo, un hecho que da cuenta de la juventud de AOP es que al momento de redacción de este trabajo no existen más de 20 libros sobre el tema[5].

Usos de AOP

Si bien en la actualidad casi no hay dudas respecto de los conceptos de AOP, todavía se sigue debatiendo si AOP es un paradigma, una técnica de programación o una estrategia de diseño.

Otra cuestión que aún no está clara, tiene que ver con los usos de AOP, si bien desde sus comienzos AOP fue concebida para modularizar CCC, no hay un acuerdo sobre si es válido su uso para cualquier CCC. Hay autores que sólo consideran válido su uso para CCC no funcionales, al tiempo que otros proponen su uso para todo CCC. Tampoco faltan los que proponen el uso de AOP para cuestiones más allá de la modularización de CCC.

Los rumbos de la comunidad

Podría decirse que en la actualidad se distinguen tres líneas de trabajo en la comunidad AOP.

La primera de ellas y la más antigua, está relacionada con las herramientas de programación. Dentro de esta línea están los grupos de desarrollo de las herramientas previamente mencionadas, que trabajan en la mejora de las mismas. Por otro lado, existen grupos trabajando en nuevas herramientas de distinta índole, intentando superar algunas limitaciones de las herramientas actuales.

La segunda, relacionada a la ingeniería de software y denominada “early aspects”, trabaja sobre la identificación y especificación de aspectos en etapas tempranas del ciclo de desarrollo de software

5 Fuente: amazon.com

como captura de requerimientos, modelado de negocio y arquitectura. El desarrollo de esta línea de investigación es fundamental para facilitar la adopción de AOP en ambientes industriales. Dentro de esta línea de investigación, hay trabajos sobre herramientas de modelado, varias de las cuales han propuesto distintas extensiones a UML, pero lamentablemente ninguna ha predominado sobre las demás.

Finalmente la tercera, trabaja sobre posibles aplicaciones de AOP dentro de distintas áreas del desarrollo de software. Es justamente dentro de esta línea de trabajo donde se encuadra esta tesis.

Capitulo III: Aplicaciones enterprise

Introducción

Ante todo una aplicación enterprise, es una aplicación que da soporte a algún proceso de negocio. Entre las características de estás aplicaciones se destacan [Fowler03]:

- Manejo de ***grandes volúmenes de información*** relacionada al negocio que debe ser persistida. Esta información podría incluir datos de stock de productos, clientes, proveedores, transacciones comerciales, etc.
- Dicha ***información suele ser accedida en forma concurrente*** por varios usuarios.
- La necesidad de los usuarios de acceder a la información obliga a que estas aplicaciones cuenten con ***varias interfaces de usuario distintas*** que permitan presentar la mencionada información de la manera más conveniente para cada usuario. Tomando como ejemplo la información de las ventas del mes de una compañía, seguramente la misma será visualizada en forma distinta por un empleado del sector de ventas que por un gerente.
- Es muy común que estas aplicaciones deban ***integrarse con otras aplicaciones*** de la organización, desarrolladas en muchos casos con diferentes tecnologías. Por ejemplo en el caso de los bancos es muy común la presencia de aplicaciones COBOL que interactúan con aplicaciones en tecnologías más actuales como java y .NET.
- El manejo simultáneo de las problemáticas antes mencionadas, sumado a la complejidad propia de las reglas de negocio, reviste a estas aplicaciones de una ***complejidad*** que se expande desde los requerimientos hasta la operación, pasando por el diseño, la implementación y la prueba. Al mismo tiempo esta complejidad suele crecer en forma exponencial con el tamaño de la aplicación.

Las características previamente mencionadas sobre las aplicaciones enterprise, son comunes a una gran cantidad de aplicaciones. Dependiendo de la problemática particular de cada aplicación, cada una de las características mencionadas toma distinta relevancia.

A grandes rasgos podríamos decir que existen 3 grandes grupos de aplicaciones enterprise.

El primer grupo está constituido por las aplicaciones de gestión general. Dentro de este grupo se encuentran los ERPs (del inglés, Enterprise Resource Managers) y los CRMs (del inglés, Customer Relationship Managers). Estas aplicaciones generalmente constituyen el núcleo de sistemas de una organización brindando soporte a la gran mayoría de sus procesos, obligando a todos los demás

sistemas a interactuar con ellos, directa o indirectamente. Ejemplos concretos son: SAP, Peoplesoft, Siebel y Financials a nivel mundial y Géminis, Tango y Calipso a nivel nacional.

El segundo grupo, está formado por aplicaciones de gestión particular que complementan a las aplicaciones del primer grupo y se encargan de alguna problemática puntual de la organización. Generalmente, estás aplicaciones son utilizadas sólo por un grupo de usuarios de la organización, por ejemplo todos los pertenecientes al sector de logística. En este grupo bien podrían ubicarse aplicaciones de recursos humanos y SFA (del inglés, Sales Force Automation).

Finalmente, en el tercer grupo se encuentran las aplicaciones de B2C (del inglés, Business to Customer). Estas aplicaciones a diferencia de las anteriores, son aplicaciones de internet, abiertas al público, por lo que la cantidad de usuarios puede ser varios órdenes de magnitud mayor que en los otros dos grupos mencionados. En el caso de las denominadas empresas .COM, estas aplicaciones son la base del negocio, mientras que en las empresas más tradicionales (no .COM) estas aplicaciones representan un complemento al negocio, ofreciendo un canal alternativo de contacto con los clientes. Como ejemplo de aplicaciones .COM podemos mencionar Amazon y Mercado Libre.

El siguiente cuadro resume algunas características distintivas de estos tipos de aplicaciones.

Característica	***Gestión general***	***Gestión puntual***	***B2C***
Cantidad de usuarios	Proporcional a la organización	Proporcional a un sector	Mundial
Volumen de información	Alto	Bajo	Alto
Necesidad de escalar	Media	Baja	Alta
Interface de usuario	Productiva	Productiva	Atractiva

Tabla 2: Características distintivas de los distintos tipos de AE

El hecho de que estas aplicaciones estén estrechamente ligadas a procesos del negocio, impone sobre las mismas la obligación de cumplir con ciertos atributos de calidad y requisitos técnicos (al conjunto de estos dos grupos se los suele llamar requisitos no funcionales).

Atributos de calidad

En muchos casos los atributos de calidad pasan inadvertidos para los usuarios de la aplicación, pero su ausencia suele impactar negativamente tanto en el negocio como en percepción que tienen los usuarios de la aplicación. Un claro ejemplo de esto es el tiempo de respuesta: mientras que la aplicación responda dentro de los parámetros de tiempo esperados por los usuarios, será raro escuchar a usuario alguno haciendo referencia al tiempo de respuesta de la aplicación, pero si, por el contrario, la aplicación tarda demasiado en responder a las interacciones de los usuarios, serán comunes las quejas

respecto de este hecho.

A continuación veremos los atributos de calidad característicos de las aplicaciones enterprise.

Disponibilidad

Según [Bass03], la ***disponibilidad*** está relacionada con los desperfectos (failure) del sistema y sus consecuencias. Un desperfecto se produce cuando el sistema deja de proveer servicio de acuerdo con las especificaciones. Además, dicho desperfecto debe ser observable por los usuarios del sistema, ya sean humanos u otras aplicaciones. Entre las incumbencias de este atributo se encuentran el cómo se detectan los desperfectos, cuán frecuentemente ocurren, qué sucede con el sistema cuando se produce un desperfecto, cuánto tiempo puede estar el sistema fuera de operación y cómo pueden prevenirse, entre otros. Resulta importante hacer la distinción entre desperfectos y fallas (faults). Un falla se convierte en un desperfecto cuando es observable para un usuario.

Típicamente la disponibilidad es definida como

d = tiempo medio entre fallas / (tiempo medio entre fallas + tiempo medio de reparación)

El tiempo calendarizado que el sistema está fuera de servicio no es considerado al calcular la disponibilidad.

Desempeño (performance)

El desempeño es uno de los atributos más complejos, debido a su relación con los demás atributos y a la gran cantidad de factores que abarca.

Según [Bass03] el desempeño tiene que ver con el ***tiempo de respuesta***, o sea, el tiempo que le insume al sistema responder a los eventos externos (estímulos del usuario, llamadas a funciones, procesamiento de pedidos de otras aplicaciones, etc.).

Al mismo tiempo, para Martin Fowler [Fowler03] la cuestión no es tan simple y sostiene que el desempeño puede verse como el tiempo de respuesta del sistema o también como la cantidad de trabajo que el sistema puede realizar en un cierto lapso de tiempo (***throughput***).

Dos medidas relacionadas al desempeño son:

eficiencia = desempeño / cantidad de recursos

capacidad = máximo throughput

Escalabilidad

La escalabilidad es una medida de como el agregado de recursos afecta al desempeño del sistema. Un sistema es escalable, si el agregado de recursos produce una mejora en el desempeño del sistema. En base a esto la escalabilidad podría medirse como:

escalabilidad = diferencia de desempeño/ cantidad de recursos agregados

Hay dos tipos de escalabilidad: vertical y horizontal. La primera, consiste en aumentar la potencia de los recursos existentes, mientras que la segunda, en agregar más recursos.

Confiabilidad (Reliability)

En [Buschmann95] se define la confiabilidad como la habilidad del sistema para mantener su funcionalidad a pesar de fallas, uso incorrecto y situaciones inesperadas. Es posible distinguir dos aspectos de la confiabilidad:

Por un lado, la ***tolerancia a fallas*** busca asegurar el correcto comportamiento ante fallas y la capacidad interna del sistema de auto corregirlas.

Por otro lado, la ***robustez***, está relacionada con la protección de la aplicación ante el uso incorrecto, las entradas de datos degenerados y la ocurrencia de errores inesperados, permitiendo que la aplicación permanezca en un estado definido. A diferencia de la tolerancia a fallas, la robustez no espera que el sistema tenga la capacidad de reponerse ante la ocurrencia de errores, sino que sólo pretende garantizar que la aplicación permanezca en un estado definido.

Mantenibilidad (modifiability)

La mantenibilidad tiene que ver con el costo de cambio y tiene dos incumbencias:

- ¿qué puede cambiar? Los cambios pueden ser de distinta índole, funcionalidad brindada por la aplicación, plataforma sobre la que aplicación existe (hardware, sistema operativo, etc), ambiente en el cual la aplicación se ejecuta (aplicaciones con las que interactua, protocolos de comunicación, etc) y atributos de calidad, entre otros. A su vez los cambios pueden ser agregados, supresiones o modificaciones de algún aspecto.
- ¿cuándo puede ser hecho el cambio y quién debe hacerlo? Tiempo atrás todo cambio en una aplicación, requería de un programador que trabajase sobre código fuente con la consiguiente, compilación y despliegue. En la actualidad los cambios pueden ser realizados por usuarios, administradores o programadores. Al mismo tiempo los cambios pueden hacerse durante la ejecución, la configuración, el despliegue o la programación.

Todo cambio tiene asociado un costo de tiempo y dinero, los cuales pueden ser cuantificados.

La mantenibilidad estará determinada en gran medida por la forma en que sea descompuesta la aplicación y por las técnicas de programación utilizadas.

Reusabilidad

El reuso es una de los temas más recurrentes en el desarrollo de software dado que teóricamente permite reducir el costo y el time to market. De acuerdo a [Buschmann95] la reusabilidad puede analizarse desde dos vistas complementarias.

Por una lado la reusabilidad como la capacidad de ***reutilizar*** artefactos de software ya existentes como bibliotecas, frameworks y componentes.

Por el otro, la reusabilidad como la posibilidad de crear artefactos para que puedan ***ser reutilizados*** por otros, en situaciones similares.

Verificabilidad (testability)

Cuanto más grande y complejo es un sistema, más importante y dificultosa es su prueba y mayores son las consecuencias de no hacerla correctamente. La verificabilidad refiere a la facilidad con la que puede demostrarse el correcto funcionamiento del software, mediante su prueba. Si bien de acuerdo a [Bass03], la prueba representa un 40% del costo total del sistema, en el caso de las AE, el no hacerla puede implicar un costo mucho mayor, debido a los perjuicios que el funcionamiento incorrecto provoque al negocio.

Usabilidad

Según Len Bass [Bass03], la usabilidad brinda una medida de cuán fácil es para los usuarios del sistema completar una tarea. Al mismo tiempo [Hohmann03] sostiene que tradicionalmente la usabilidad ha sido asociada a la interfaz de usuario, lo cual tiene perfecto sentido ya que la impresión que tienen los usuarios sobre cuán usable es el sistema es través de la interfaz de usuario. Para la usabilidad es importante que la interfaz de usuario sea clara y fácil de usar. Adicionalmente hay cuestiones de usabilidad que están más allá de la interfaz de usuario como por ejemplo brindar al usuario la posibilidad de cancelar operaciones, deshacer operaciones o reutilizar datos previamente ingresados.

Una de las razones más importantes para cuidar la usabilidad es el dinero. Está por demás probado que la inversión en usabilidad se recupera muy rápido y con creces en la vida de cualquier aplicación. Algunos de los beneficios de la usabilidad son los que se enumeran a continuación.

- Reducción de los costos de entrenamiento.
- Reducción de los costos de soporte.
- Reducción en el costo de los errores.
- Incremento en la productividad de los usuarios.
- Incremento en la satisfacción del cliente.

Seguridad

La seguridad es una medida de la habilidad del sistema para evitar usos no autorizados; al mismo tiempo de permitir el uso a quienes legítimamente tienen permiso para usarlo.

Todo intento de quebrar la seguridad es considerado un ataque o amenaza. Estas amenazas

pueden tomar distintas formas, encontrándose entre las más comunes, el acceso no autorizado a datos/funcionalidades y el intento de impedir el acceso a usuarios legítimos. La seguridad puede ser caracterizada por una serie de propiedades.

- Autenticación: es el mecanismo que permite identificar a los usuarios de la aplicación. Generalmente es implementada en la interfaz de usuario para poder accionar los mecanismos de autorización, auditoria y personalización. Para llevar a cabo la autenticación es necesario que el usuario provea algún tipo de credencial, típicamente su nombre de usuario y una contraseña.
- Autorización: es la incumbencia que identifica las acciones que tiene permitidas realizar cada usuario.
- Auditoría: es la propiedad de la aplicación de registrar toda actividad realizada, de manera de poder saber quién y cuándo realizó cada acción.

Interoperabilidad

En muchos casos las aplicaciones no están aisladas, sino que forman parte de un todo en el cual deben interactuar con otros sistemas. En este contexto la interoperabilidad es la capacidad de una aplicación de poder ***interactuar con aplicaciones independientemente de su tecnología***. Para lograr este cometido suelen utilizarse soluciones basadas patrones de diseño y estándares abiertos.

Consideraciones sobre los atributos de calidad

Más allá de la definición e identificación de los atributos de calidad de un sistema, existen una serie de cuestiones no menores a considerar que se mencionan a continuación.

Al intentar implementar estos atributos de calidad es posible que nos encontremos que mientras que algunos atributos se complementan entre sí, otros resultan ser incompatibles. Un caso de atributos complementarios podría ser disponibilidad y escalabilidad, mientras que como atributos incompatibles podrían mencionarse desempeño y confiabilidad.

Otra cuestión a considerar, es la forma de medir los atributos de calidad. Algunos como el desempeño, pueden medirse sin grandes complicaciones mientras la aplicación está ejecutándose, midiendo el tiempo de respuesta. Pero para otros atributos, más abstractos, como la usabilidad, la forma de medición no es tan intuitiva.

Requisitos técnicos

Adicionalmente a los atributos de calidad, las AE cuentan con ciertos atributos técnicos, algunos de las cuales son consecuencia de la implementación de los atributos de calidad mientras que otros surgen directamente a partir de necesidades del negocio.

Persistencia

La persistencia de la información es una de las problemáticas centrales de estas aplicaciones y por ello es que las arquitecturas actuales plantean una capa de la aplicación encargada exclusivamente de la persistencia

Sin duda alguna, el estándar de facto en la actualidad para la persistencia de información en AE, son las bases de datos relaciones, mientras que para la codificación de la lógica de negocio se utilizan tecnologías de objetos. Esto nos pone en la situación de almacenar objetos en tablas, problemática que por su recurrencia ya ha sido caracterizada con un nombre propio: ***diferencia de impedancia*** (impedance mismatch) [Ambler04]. En este contexto hablar de persistencia implica:

- Definir el mapeo de datos entre objetos y tablas.
- Ddefinir las clases que se encargarán de llevar a cabo el mapeo previamente definido, lo que implicará la manipulación de sentencias SQL.
- Colocar las llamadas necesarias a las clases previamente definidas para persistir los objetos en la base de datos y recuperarlos de la misma una vez persistidos.

A estos puntos mencionados, se les debe sumar la complejidad que implica el manejo del ciclo de vida de los objetos, tanto el memoria como en la base de datos. Existen múltiples trabajos que han tratado esta problemática [Brown], [Ambler02], [Ambler00] y [Fowler03].

Transaccionalidad

Una transacción es una secuencia de trabajo, que abarca una serie de operaciones y que cuenta con un comienzo y un fin claramente delimitados [Fowler03]. Tanto al comienzo de la transacción, como al final de la misma, el sistema está en estado válido, o sea, si durante la ejecución de la transacción, algo falla o la transacción es cancelada, el sistema deberá retornar al estado previo al comienzo de la transacción.

En las aplicaciones enterprise existen dos tipos de transacciones: de negocio y del sistema. Las transacciones de negocio son aquellas que tienen sentido para el negocio y por consiguiente para los usuarios de la aplicación. Por su parte las transacciones del sistema suelen ser más granulares que las transacciones de negocio y generalmente no tienen sentido directo para el usuario. Una problemática corriente en las AE, es lidiar con la diferencia de granularidad entre ambos tipos de transacciones, ya que por lo general una transacción de negocio abarca varias transacciones del sistema.

Las transacciones son el principal mecanismo utilizado en las AE para el manejo de la concurrencia.

Las transacciones -independientemente de su tipo- se caracterizan por cuatro propiedades.

- ***Atomicidad***, cada operación debe completarse exitosamente para que la transacción pueda

completarse, de lo contrario todo el trabajo debe deshacerse, esto nos lleva a una situación de todo o nada.

- ***Consistencia***, todo recurso involucrado debe estar en un estado consistente tanto al comienzo como al final de la transacción.
- ***Aislamiento*** (isolation), el resultado de una transacción no debe ser visible a ninguna otra transacción hasta que la misma no sea completada.
- ***Durabilidad*** (durability), el resultado de una transacción debe ser hecho en forma permanente.

Las transacciones que cumplen con estas cuatro propiedades se conocen como transacciones ACID (Atomicity, Consistency, Isolation, Durability).

Poniendo las transacciones en el contexto de las AE, un usuario realiza acciones en la aplicación, donde cada acción del usuario involucra un conjunto de operaciones por parte de la aplicación, las cuales generalmente forman parte de una misma transacción.

Distribución

A lo largo del tiempo las AE han pasado por distintos modelos de procesamiento, desde las arquitecturas mainframes de procesamiento centralizado con terminales bobas para entrada y salida de datos, pasando por arquitecturas cliente-servidor y llegando en la actualidad hasta las arquitecturas distribuídas, donde la misma aplicación se ejecuta en varios nodos físicos. En algunas ocasiones la distribución es consecuencia de que el negocio se encuentra físicamente distribuído, mientras que en otras ocasiones la distribución es necesaria para poder brindar ciertos atributos de calidad como ser alta disponibilidad.

Es interesante hacer notar que el hecho de que muchas AE sean distribuídas ha generado la creencia de que la distribución es una condición necesaria para que una aplicación sea considerada enterprise. Esto es incorrecto, ya que la distribución es una consecuencia de los requisitos impuestos por el negocio sobre las aplicaciones enterprise y de ninguna manera una condición necesaria ni mucho menos suficiente para determinar el carácter enterprise de una aplicación.

Concurrencia

Como ya hemos mencionado, las AE suelen tener varios usuarios concurrentes, lo cual provoca que la aplicación no solo deba soportar con un buen desempeño varias sesiones de usuarios, sino que ademas deba controlar el acceso concurrente a datos del sistema, asegurando su consistencia todo el tiempo. Puede darse que dos o más usuarios intenten modificar un mismo registro al mismo tiempo, lo cual podría ocasionar inconsistencias en los datos de la aplicación. Para evitar estas situaciones existen varias estrategias de diseño, las cuales se encuentran en su gran mayoría documentadas en [Fowler03].

Monitoreo

Esta funcionalidad es necesaria para tener visibilidad sobre el comportamiento de las aplicaciones en el ambiente productivo, ya que en la mayoría de los casos el acceso a dicho ambiente se encuentra por demás restringido.

Generalmente, el monitoreo provee información útil para las siguientes cuestiones.

- Salud de la aplicación: ¿está la aplicación funcionando correctamente?, ¿hay algún pico de procesamiento?, ¿está la aplicación corriendo en un hardware adecuado?
- Asegurar el cumplimiento de los acuerdos de los nivel de servicio comprometidos.
- Conocimiento del negocio: ¿cuales son las funcionalidades más utilizadas de la aplicación?, ¿en qué momento los usuarios utilizan más la aplicación?, ¿es posible optimizar algún proceso del negocio?, ¿hay algún cuello de botella en el proceso de negocio?

Adicionalmente durante el desarrollo el monitoreo sirve como una herramienta más para la depuración de la aplicación.

Arquitectura

En lo que a arquitectura lógica respecta, la arquitectura en capas (layers)[Buschmann95] se ha convertido en un estándar de facto para estas aplicaciones. Dicha arquitectura propone estructurar la aplicación agrupando sus componentes en capas. Los componentes dentro de una misma capa deben ser cohesivos y tener el mismo nivel de abstracción. Al mismo tiempo cada capa desconoce la capa superior mientras que hace uso de las funcionalidades provistas por la capa inferior. En [Trowbridge03] se mencionan algunas variantes y refinamientos de este patrón muy comunes en las aplicaciones enterprise, pero sin duda la variante más adoptada es la de tres capas. Dicha variante propone descomponer la aplicación en una capa de presentación, una de lógica de negocio y una de acceso a datos. Entre las ventajas que provee la división en capas se encuentran el desacoplamiento de las mismas y la posibilidad de distribuirlas físicamente. Relacionado a la distribución física de las capas hay que mencionar que generalmente suele hacerse utilizando funcionalidades provistas por los servidores de aplicaciones.

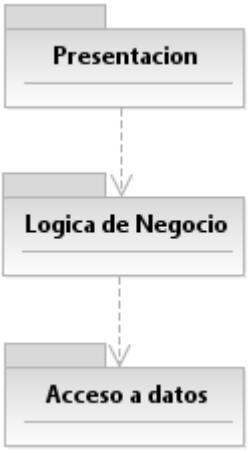

Figura 10: Estructura genérica en capas de AE

La capa de presentación se encarga de la interacción con el usuario, mientras que la de lógica de negocio como su nombre lo indica contiene todas las reglas del negocio. Finalmente la capa de acceso a datos, encapsula la interacción con el repositorio de información, generalmente una base de datos relacional. En algunos casos se propone una capa transversal denominada de infraestructura u operacional que agrupa aquellos componentes de uso común en todas las capas, como ser: manejo de excepciones, monitoreo y seguridad, entre otros.

Existen variantes específicas de la arquitectura de 3 capas para las distintas tecnologías [Jezierski03], [Johnson02], [Ashmore04] y [Nilsson06].

Bajando el nivel de abstracción, dentro de cada capa es un estándar en la actualidad la utilización de programación orientada a objetos. Esto se debe a que la POO provee potentes mecanismo de encapsulamiento y modularización que resultan muy convenientes para el manejo de la complejidad inherente de las AE. Junto con la POO es muy común por esto días el uso de patrones de diseño.

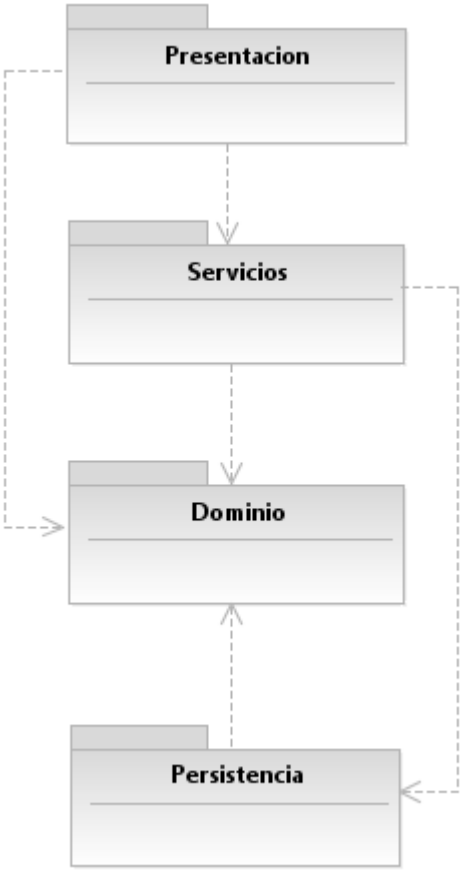

Figura 11: Variante muy difundida de arquitectura en capas

Patrones de diseño

Desde hace ya varios años, es muy común en todo desarrollo de software orientado a objetos, el uso de patrones de diseño. De acuerdo a Christofer Alexander [Alexander77]:

"...cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces."

Si bien este autor se refería a patrones de ciudades y edificios, su definición fue tomada por Erich Gamma y compañía, quienes escribieron el libro fundacional de patrones de diseño de software [Gamma95]. Algunas cuestiones interesantes sobre los patrones son las siguientes.

- Los patrones no se inventan, sino que se descubren.
- La definición de un patrón consta 4 elementos: un nombre, un problema al cual se supone el patrón aplica, una solución que indica como resolver el problema y un conjunto de consecuencias, surgidas de la aplicación del patrón.
- El uso de patrones permite establecer un lenguaje común para la comunicación y al mismo tiempo eleva el nivel de abstracción de la conversación.

- Existen tres tipos de patrones de diseño, dependiendo de su nivel de abstracción. Los patrones de arquitectura, son patrones de alto nivel que tratan de como estructurar los módulos de la aplicación. Los patrones de diseño propiamente dichos, tratan de diseño de clases. Finalmente los denominados idioms, tratan de patrones particulares de un lenguaje o tecnología.

Los patrones enunciados en la publicación original de Gamma [Gamma95] son de uso general. Con el correr del tiempo han ido surgiendo familias de patrones específicos de cada dominio. En particular, el libro de Martin Fowler [Fowler03] se ha convertido en una de las publicaciones de referencia en lo que a patrones enterprise respecta. Veamos algunos de los patrones más comúnmente utilizados en cada capa.

En la capa de presentación suele utilizarse el patrón ***Model-View-Controller*** (o algunas de sus variantes como MVP, front-controller y page-controller). Este es un patrón de alto nivel que propone una descomposición estructural de la aplicación en tres partes, tal como su nombre lo refleja: Modelo, Vista y Controlador.

En cuanto a la capa de negocio, existen básicamente 3 patrones para organizar la lógica: ***Transaction Script***, ***TableModule*** y ***Domain Model***.

En muchos casos es común encapsular la lógica de negocio utilizando aplicando el patrón ***service layer***, lo que deriva en que la capa de negocio sea dividida en dos subcapas: una capa de de servicios de negocio y otra de lógica de negocio. Los servicios de negocio actuan como una fachada sobre la lógica subyacente, realizando chequeos de seguridad y administrando el contexto transaccional.

En la capa de acceso a datos suelen utilizarse los patrones como ***TableDataGateway*** y ***Datamapper***. El primero se suele utilizar cuando la lógica de negocio está organizada en base a Transaction Scripts, mientras que el uso del segundo es más común cuando el negocio está basado en Domain Model.

Algunos otros patrones de uso común son ***Data Transfer Object***, ***Dependency Injection*** y ***Value Object***.

Capa	*Patrones*
Presentación	Model-View-Controller, Front Controller, Page Controller
Lógica de negocio	Domain Model, Transaction script, TableModule, Service layer
Acceso a datos	Datamapper, Active record, TableDataaway

Tabla 3: Patrones de uso común en Aplicaciones Enterprise

En uno de los apéndices hay más información sobre los patrones aquí mencionados.

Orientación a servicios

Hasta aquí hemos visto las generalidades de las aplicaciones enterprise y sus características técnicas más importantes en la actualidad, pero más allá de esto, es fundamental no perder de vista el contexto organizacional en el cual deben encajar estas aplicaciones.

Es moneda corriente que los procesos de negocio de una organización involucren a más de una aplicación, necesitando de la coordinación de un conjunto de aplicaciones. Este fenómeno ha dado origen a un nuevo paradigma denominado orientación a servicios.

Acorde a este nuevo paradigma, cada aplicación está constituida por un conjunto de servicios, donde cada servicio implementa una función del negocio, posibilitando que la implementación de un proceso de negocio pase a ser una orquestación de servicios.

Para que una aplicación pueda considerarse orientada a servicios, debe cumplir con ciertos principios de diseño. Según [Erl05], no existe una definición oficial de dichos principios, pero a pesar de ello existe un conjunto de principios ampliamente aceptados entre los que se encuentran los siguientes.

- Todo servicio expone un contrato formal para su interacción.
- Los servicios están desacoplados entre sí.
- Los servicios son abstractos a la lógica subyacente.
- Los servicios son autónomos.
- Los servicios no tienen estado.

De acuerdo a [Endrei04] la orientación a servicios provee los siguientes beneficios.

- Posibilidad de reutilización de los activos existentes.
- Reducción de costos a partir de un mayor reuso.
- Mayor facilidad de integración.
- Mejor Time-To-Market.

Tecnologías

En cuanto a tecnologías, actualmente el terreno está claramente divido entre Java y Microsoft.NET. Si bien estas dos tecnologías parecen ser antagónicas, resultan ser muy similares. Ambas tecnologías están basadas en máquinas virtuales que soportan el paradigma de programación orientada a objetos y proveen al menos un lenguaje de programación, un entorno integrado de desarrollo y una extensa biblioteca de clases.

Si bien ambas tecnologías pueden utilizarse para diversos tipos de aplicaciones, se encuentran fuertemente orientadas al desarrollo de aplicaciones de enterprise. Dentro de este contexto, existen en estas plataformas componentes denominados servidores de aplicaciones. Estos servidores son

aplicaciones que permiten alojar otras aplicaciones a las cuales les brindan ciertos servicios entre los cuales pueden mencionarse: soporte transaccional, manejo de sesiones, distribución de objetos, etc. Además de estos servicios, los servidores de aplicaciones permiten alcanzar ciertos atributos de calidad como ser escalabilidad y disponibilidad, pero al mismo imponen ciertas restricciones sobre la forma de programación y la arquitectura de las aplicaciones.

La figura 12 muestra los principales bloques constitutivos de estas dos tecnologías (hay bloques que han sido omitidos por no contar con uno análogo en la otra tecnología).

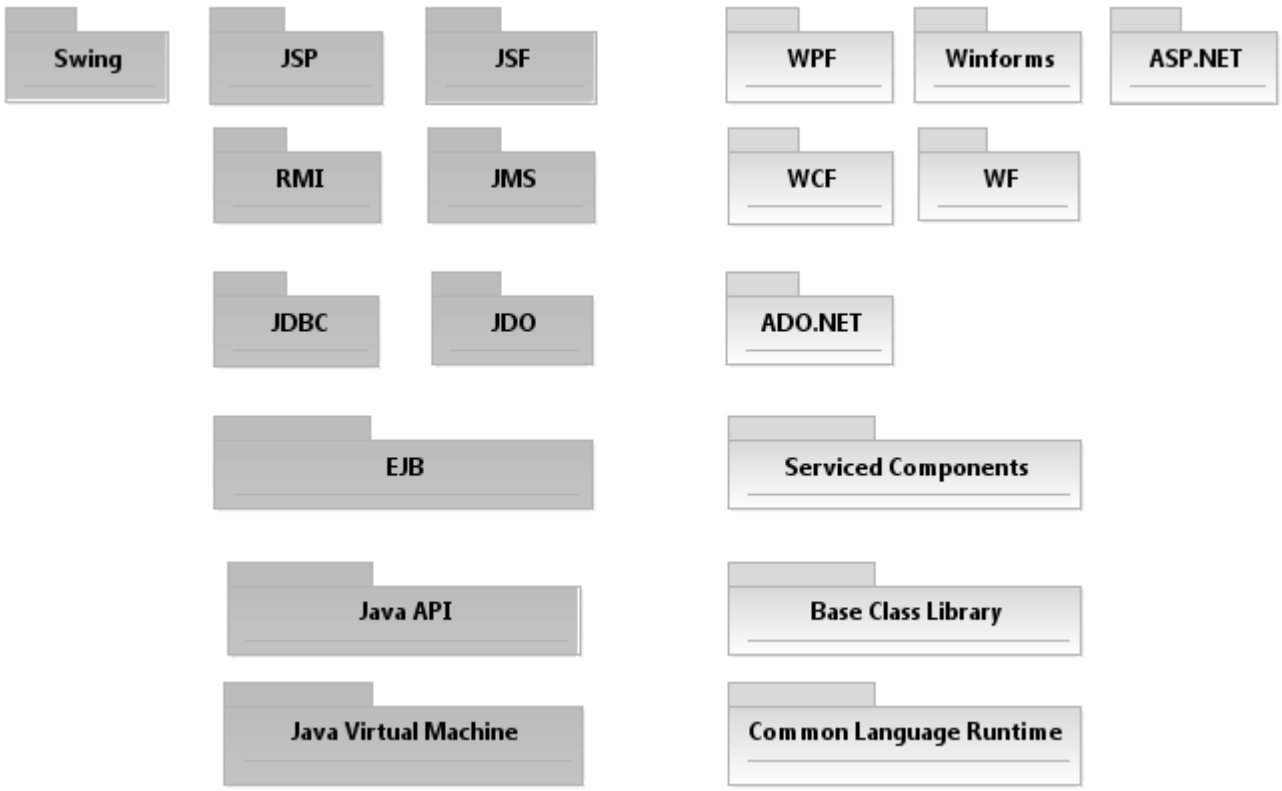

Figura 12: Principales bloques de las tecnologías Java y .NET

Capítulo IV: Hacia las aplicaciones enterprise con AOP

Introducción

Dado que el objetivo de AOP es facilitar la modularización de los CCC, es de esperar que su aporte a las AE sea en ese sentido, es por ello que comenzaremos este capítulo identificando las incumbencias transversales más comunes en las AE. Luego veremos como es que las mismas son atacadas en la actualidad en las AE y finalmente veremos como sería el proceso de comenzar a utilizar AOP para atacarlas.

Antes de entrar de lleno en el tema de este capítulo es necesario hacer algunas aclaraciones preliminares.

Si bien es posible hablar de incumbencias transversales en las distintas etapas del proceso de desarrollo (análisis, diseño, implementación, etc.) tal como lo plantean múltiples trabajos [Sousa][AORE][EarlyAspects] sólo nos concentraremos en las incumbencia transversales a nivel de implementación (código fuente).

Asimismo nos encontramos que las incumbencias tranversales a nivel de implementación pueden variar en base a la tecnología utilizada. Un ejemplo de esto es la administración de memoria, la cual representa un clara incumbencia transversal en C y C++, hecho que no ocurre en Java y .NET, donde la administración de la memoria está a cargo de la plataforma de ejecución. Dado que las tecnologías predominantes en el desarrollo de aplicaciones enterprise en la actualidad son Java y .NET, nuestro análisis de las incumbencias transversales será alrededor de estas dos tecnologías.

Incumbencias transversales

A simple vista en toda aplicación nos encontramos con dos tipos de incumbencias:

- ***incumbencias dominantes***, que generalmente se logran modularizar sin mayores inconvenientes utilizando las técnicas tradicionales de programación, ya que constituyen la dimensión dominante del problema. En el caso de las aplicaciones enterprise estas incumbencias pertenecen a la capa de lógica de negocio y por ello las denominaremos incumbencias del negocio. Dado que generalmente estas incumbencias son particulares de cada aplicación, su análisis lo postergaremos hasta el capítulo 6 donde analizaremos un caso concreto.

- ***incumbencias no dominantes***, relacionadas generalmente a los atributos de calidad y requisitos técnicos de la aplicación. Dado que estas incumbencias no pertenecen a la dimensión dominante, su modularización puede resultar dificultosa utilizando las técnicas tradicionales de programación. Si bien existen bibliotecas específicas que implementan cada una de estas incumbencias, resulta que las llamadas a dichas bibliotecas se encuentran dispersas por todo el código, generando los conocidos fenómenos de código mezclado y código disperso, evidenciando una naturaleza transversal. En el caso de la AE, muchas de estas incumbencias son llamadas servicios enterprise. Estas incumbencias serán tratadas en el capítulo 5 y en general nos referiremos a ellas como incumbencias técnicas.

En el caso de la AE, existen en la actualidad varias propuestas para la modularización de las incumbencias no dominantes, muchas de las cuales cuentan con un importante grado de adopción. En la siguiente sección analizaremos algunas de estas propuestas.

Intentos de modularización

Servidores de aplicaciones y componentes

Tanto java como .NET ofrecen como parte de sus arquitecturas de referencia componentes denominados servidores de aplicaciones que brindan un entorno de ejecución y una serie de servicios para las aplicaciones que son desplegadas en ellos. Los servidores de aplicaciones brindan soluciones a varias de las problemáticas enterprise mencionadas en el capitulo 3 (distribución, escalabilidad y alta disponibilidad, entre otros) y es por ello que en la actualidad casi todas las AE dependen de algún servidor de aplicaciones.

Un caso particular de servidor de aplicaciones son los servidores de componentes, los cuales brindan algunos servicios enterprise adicionales como manejo de transacciones y distribución de componentes, entre otros. Al mismo tiempo permiten que los componentes desplegados en ellos hagan uso de los mencionados servicios en forma declarativa, disminuyendo así el esfuerzo de desarrollo y la cantidad de código relacionado a infraestructura.

En el caso de java, los servidores de componentes son conocidos como contenedores EJB, mientras que en el caso de .NET el servidor de componentes es Component Services (también conocido como COM+).

Si bien con el uso de servidores de componentes se facilita la implementación de algunas incumbencias transversales, existen algunas limitaciones. Una de las desventajas que tiene el uso de servidores de componentes es que suelen imponer un modelo de programación bastante intrusivo que suele dificultar la prueba de los componentes y aumenta de manera considerable el esfuerzo de despliegue de la aplicación. Otras de las desventajas mencionadas en [Johnson04] es que los servicios

brindados por los servidores de componentes son limitados, en el sentido de que no es posible agregar servicios adicionales desarrollados por el programador.

Frameworks de aplicación

Otra herramienta de uso muy común el desarrollo de AE son los frameworks de aplicación (*application frameworks, AF)*. Un framework es un sub-sistema, parcialmente completo que define la arquitectura de una familia de sistemas, proveyendo los bloques básicos para su creación y especificando los puntos de extensión y adaptación [Buschmann95]. Un framework para un dominio específico de aplicaciones es un framework de aplicación. Algunos de los beneficios del uso de estos frameworks son los siguientes.

- ***Productividad***: los AF estandarizan la forma de resolver los problemas, permitiendo que a la hora de resolver un nuevo problema no deba empezarse desde cero.
- ***Mayor foco en la lógica central de la aplicación***: los equipos de desarrollo pueden concentrarse en la funcionalidad principal de la aplicación, dejando en manos del AF las cuestiones técnicas no funcionales que casualmente suelen ser incumbencias transversales no dominantes.
- ***Predictibilidad***: las soluciones basadas en la arquitectura predefinida por el AF, reducen el riesgo y aumentan la predictibilidad del proceso de desarrollo y de la aplicación final.

Entre los servicios que suelen brindar los AF para AE se encuentran:

- Configuración de la aplicación y de sus objetos.
- Manejo de transacciones.
- Simplificación de la interacción con la capa de acceso a datos.
- Manejo de errores.

A modo de ejemplo veamos algunos AF para desarrollo de aplicaciones enterprise.

Enterprise Development Application Framework (EDAF)

EDAF es un AF desarrollado por el grupo de Prácticas y Patrones de Microsoft Corporation en el año 2004, con el objetivo de facilitar el desarrollo de aplicaciones enterprise orientadas a servicios y basadas en estándares de la industria. El desarrollo de EDAF fue guiado por los siguientes principios de diseño:

- Separación de la interfaz de servicios de la implementación interna de los mismos.
- Separación de la lógica de negocio de las incumbencias no funcionales como auditoria y monitoreo, entre otros.
- Separación de la lógica de negocio de la tecnología de comunicación, permitiendo que un mismo servicio de negocio pueda ser accedido por vía de distintas tecnologías de

comunicación.

La figura 13, muestra la arquitectura propuesta por EDAF, la cual consiste en una variante de la arquitectura de 3 capas, donde la capa de presentación interactúa con la interfaz de servicio. La capa de interfaz de servicio actúa como frontera, rechazando pedidos inválidos o no autorizados. Luego hay una capa que contiene lógica transversal, como auditoría y transaccionalidad y opera antes (y después) de la ejecución de la lógica de negocio.

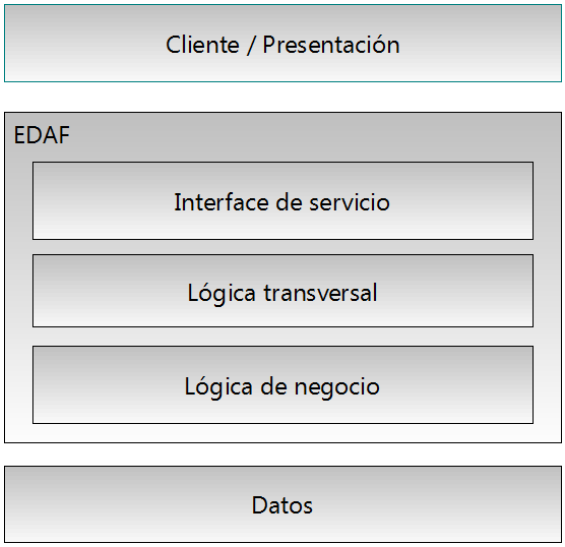

Figura 13: Estructura de capas propuesta por EDRA

A grandes rasgos el procesamiento de un pedido consisten en:

1. El flujo comienza con un mensaje enviado por la capa cliente para la ejecución de una acción de negocio.
2. El mensaje es manejado por un adaptador asociado al canal de comunicación. El adaptador crea un contexto para el procesamiento del mensaje y lo pasa al pipeline correspondiente.
3. El pipeline, consiste en una cadena de handlers y un objetivo. Una vez ejecutados los handlers, que generalmente realizan operaciones relacionadas a incumbencias transversales, se da paso a la ejecución del objetivo, que siempre consiste en una acción de negocio.

Los handlers, actúan como interceptores en el proceso de ejecución, y pueden ejecutarse antes y/o después del objetivo. Si bien EDAF provee ciertos handlers de uso común, también es posible desarrollar los propios. Estos handlers están pensados para manejar la lógica transversal de la aplicación.

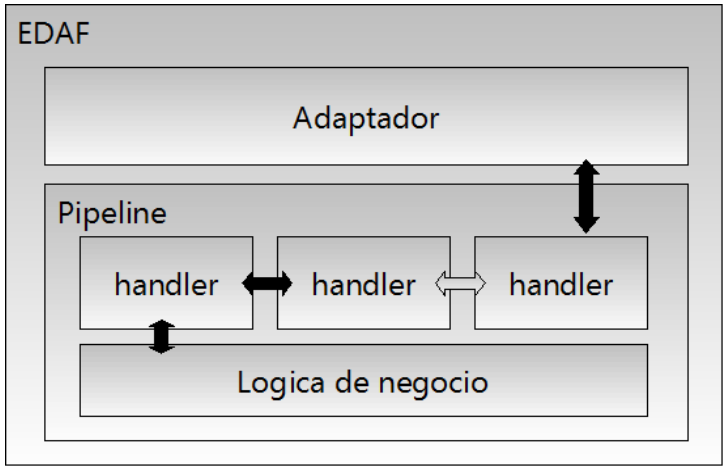

Figura 14: Proceso de ejecución de una petición EDRA

La estrategia plateada por EDAF para el manejo de las incumbencias transversales, utilizando handlers en el pipeline de ejecución, fue luego adoptada por ***Windows Communication Foundation*** [wcf] que es uno de los bloques fundacionales de la versión 3.0 de la plataforma .NET. WCF, al igual que EDAF, provee un conjunto de handlers (en el caso de wcf son llamados behaviors) que implementan ciertas incumbencia típicas como auditoría, seguridad y transaccionalidad. Al mismo tiempo,brinda un modelo de programación extensible, permitiendo que el programador pueda desarrollar sus propios handlers.

Si bien la utilización de un pipeline de ejecución con handlers para aplicar funcionalidades transversales, tiene una gran similitud con lo que hacen las herramientas AOP basadas en proxies dinámicos, tanto EDAF como WCF tienen ciertas limitaciones para el manejo de incumbencias transversales, en comparación con las herramientas AOP.

En primer lugar, el enfoque de EDAF/WCF brinda solución para incumbencias transversales a nivel de servicios[6], pero no para las incumbencias transversales de grano más fino que pueden afectar a los objetos que son utilizados dentro de los servicios. En términos de AOP: con EDAF/WCF es posible aplicar aspectos a servicios pero no a objetos.

En segundo lugar, aunque no menos importante, EDAF/WCF no brinda la posibilidad de expresar genéricamente a qué servicios se pretende aplicar aspectos, sino que es necesario enumerarlos uno a uno. En términos de AOP: el mecanismo de cuantificación de EDAF/WCF es muy limitado en comparación con los modelos de pointcuts provistos por las herramientas AOP.

Por último con WCF no es posible extender la estructura de los servicios. En términos AOP, EDAF/WCF no soporta declaraciones de intertipo.

Si bien en los párrafos anteriores se hizo referencia explícita a WCF, los mismo problemas aplican en mayor o menor medida a todas las herramientas que usan un modelo handlers no orientado a

6 En este caso el término servicio hace referencia a un objeto sin estado, que brinda funcionalidad de grano grueso.

aspectos.

Contenedores livianos

En el año 2004 Rod Johnson publicó el libro "J2EE Development without EJB" [Johnson04] en el cual afirma que sólo un pequeño porcentaje de las AE requieren realmente de la utilización de un servidor de componentes[7]. A partir de esta afirmación, Johnson propone y detalla a lo largo del mencionado libro el uso de contenedores livianos como una alternativa factible para la gran mayoría de las AE.

Los contenedores livianos son un tipo particular de AF que se caracterizan por los siguientes motivos.

- Promover la orientación a objetos y las buenas prácticas de diseño.
- Proponer un modelo de desarrollo no intrusivo, imponiendo mínimos requisitos a los objetos que contienen.
- Proveer una serie de funcionalidades (servicios enterprise) para los objetos que contienen, entre las que se encuentran el manejo del ciclo de vida y resolución de dependencias.
- Brindar soporte de ciertas características AOP.

Johnson sostiene que mientras que los servidores de componentes ofrecen una cantidad limitada de funcionalidades enterprise e imponen un modelo de programación intrusivo, utilizando contenedores livianos en conjunto con AOP, es posible contar con prácticamente los mismas funcionalidades, con un modelo de programación no intrusivo y con la posibilidad de extender dichas funcionalidades o incluso agregar nuevas.

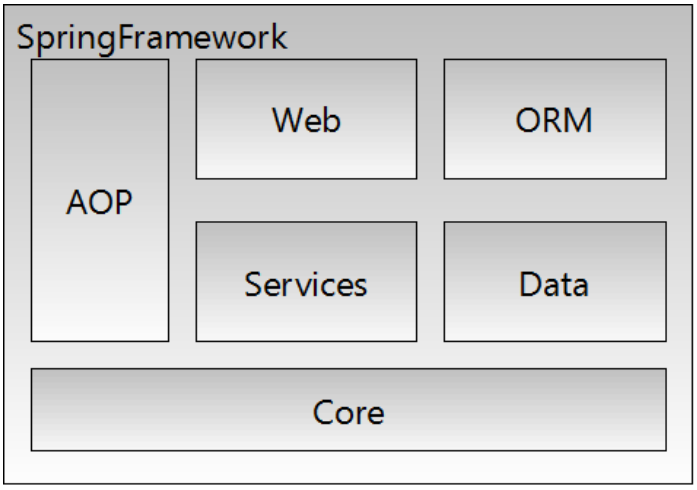

Figura 15: Módulos de Spring Framework

Estas ideas de Johnson tuvieron un gran impacto en la comunidad de desarrolladores Java, provocando un adopción masiva de contenedores livianos, en particular de Spring Framework. Fue tal

7 Johnson se refiere en particular a contenedores EJB

la magnitud este impacto que su influencia se vio reflejada en las siguiente versión de las especificaciones de los contenedores EJB. Adicionalmente hay que destacar que en la actualidad Spring.AOP (uno de los módulos de SpringFramework) se encuentra entre las herramientas AOP más utilizadas.

Si bien los contenedores livianos surgieron como una alternativa a los servidores de componentes java (contenedores EJB, considerados contenedores pesados) en la actualidad su uso se ha extendido también a .NET, al punto que tanto SpringFramework como PicoContainer, surgidos en java, cuentan hoy en día con versiones para plataforma .NET.

Entre los contenedores livianos actuales se destacan Spring Framework (java y .NET) y Windsor Castle (.NET), ambos proveen varias funcionalidades típicas en AE, que pueden ser utilizadas declarativamente. La mayoría de estas funcionalidades han sido implementadas integrando bibliotecas específicas mediante el uso de interceptores. Hay que mencionar que a pesar de que estos AF brindan soporte de AOP, en muchos casos se opta por utilizar explícitamente interceptores en lugar de las abstracciones ofrecidas por AOP.

A pesar del extendido uso de contenedores livianos, el uso de sus capacidades AOP es escaso, lo cual puede que se deba al desconocimiento de los programadores del paradigma AOP, que incluso habiendo escuchado de AOP, prefieren el uso directo de interceptores en lugar de trabajar con las abstracciones definidas por AOP.

Bibliotecas enterprise

Las bibliotecas, a diferencia de los frameworks, no imponen restricciones sobre la forma de desarrollar, o sea, cuando uno utiliza un framework, está tomando una estructura base en la cual "enchufa" componentes propios siguiendo las reglas impuestas por el framework. Por su parte el uso de una biblioteca brinda la posibilidad de estructurar la aplicación como uno guste y simplemente invocar (o "enchufar") la funcionalidad brindada por la biblioteca cuando así se lo requiera. Al mismo tiempo las bibliotecas implementan funcionalidades mucho más puntuales que los frameworks.

La contra de cualquier biblioteca es que si bien su funcionalidad se encuentra bien modularizada, las llamadas a la biblioteca pueden llegar a dispersarse por todo el código de la aplicación.

Existen en la actualidad varias bibliotecas de gran utilidad en el desarrollo de AE.

En particular en los desarrollos con tecnología .NET, se destaca el uso de "Microsoft Enterprise Library", la cual es un biblioteca de código abierto desarrollada por el equipo de Prácticas y Patrones de Microsoft, que agrupa un conjunto de bibliotecas utilitarias las cuales implementan cada una, una incumbencia común en AE como ser logging, caching o manejo de excepciones.

La última versión de la Enterprise library ha incorporado un nueva biblioteca llamada Policy

Injection Application Block, que permite insertar interceptores (llamados handlers) en las llamadas a ciertos objetos para implementar incumbencias transversales y de esta forma permitir el uso de otras bibliotecas, evitando el código disperso y entremezclado que estas suelen generar. Si bien en cierto punto el Policy Injection Application. Block, podría llegar a brindar las mismas funcionalidades que una herramienta AOP basada en proxies dinámicos, el nivel de abstracción provisto por el mismo es mucho más rudimentario, ya que carece de la semántica AOP. Al mismo tiempo un punto destacable es que trae una serie de interceptores ya implementados entre los que se encuentran Logging, Caching, manejo de excepciones y autorización

AOP Alliance

En el año 2003 un conjunto de personalidades relacionadas al movimiento de la orientación a aspectos, entre las que se encontraban Gregor Kiczales, Rod Johnson y Renaud Pawlak, fundaron el proyecto AOP Alliance [aopalliance]. La motivación de estos individuos se debió a la existencia de múltiples herramientas AOP, que por haber surgido principalmente de ambientes experimentales, resultaban tener ciertos puntos de incompatibilidad al intentar ser utilizadas en ambientes específicos (particularmente ambientes J2EE). Es por esto que el proyecto definió los siguientes objetivos.

- Evitar la reprogramación de las herramientas AOP, permitiendo su reuso.
- Simplificar la adaptación de las herramientas AOP existentes para un ambiente determinado (típicamente J2EE).
- Simplificar el reuso de aspectos contando con una API común.
- Simplificar la implementación de herramientas de desarrollo que deseen integrarse con AOP.

El proyecto ha tenido un éxito relativo, ya que a 4 años de su fundación, en el sitio oficial [aopalliance], solo pueden encontrarse un borrador del white paper fundacional [Pawlak03] y la especificación de una API AOP en Java, para ser implementada por herramientas AOP basadas en proxies. Si bien esta API ha sido implementada por varias herramientas (incluso con tecnologías distintas a java) no todas la han implementado.

Una falencia de la API definida por AOP Alliance es que no es lo suficientemente abstracta, ya que hace mención explícita reiteradamente a interceptores que no son un concepto propio de AOP, sino una mera cuestión de implementación, de la cual el programador debería estar ajeno.

Mitos y leyendas

Habiendo puesto de relevancia la necesidad de contar una herramienta que permita una mejor modularización de las incumbencias transversales y antes de exponer cómo es que con AOP puede lograrse, es necesario hacer algunas aclaraciones preliminares y dar por tierra con algunos mitos y prejuicios sobre AOP, varios de los cuales han sido tratados por Ramnivas Laddad [Laddad06].

AOP considered harmful

Una de las objeciones más importantes al uso de AOP se encuentra documentada en el polémico artículo "AOP considered harmful" [Stoerzer04] el cual compara el uso de AOP con el uso de la clausula GOTO, repasando los argumentos planteados por Dijkstra en su famoso escrito "Go to Statement considered harmful"[Dijkstra68]. Dijkstra planteaba que el uso de la claúsula GOTO dificultaba el entendimiento del flujo de ejecución del sistema por parte de los programadores.

Según Stoerzer los advices son en cierto sentido similares a procedimientos/métodos, pero al mismo tiempo tienen ciertas propiedades que al igual que el GOTO, dificultan el entendimiento de la secuencia de ejecución del sistema.

- En primer lugar, el código que expone los joinpoints no tiene conocimiento de los advices que se ejecutarán en dichos joinpoints (propiedad de transparencia). Si bien en ocasiones los entornos de desarrollo tienen la propiedad de mostrar los joinpoints afectados por advices, esto no es una propiedad de la herramienta AOP en si misma. Esto hace que en cierto punto los advices sean incluso peores que el GOTO, ya que el GOTO al menos es visible en el código, mientras que los advices son absolutamente transparentes.
- En segundo lugar, los joinpoints afectados por un pointcut no siempre pueden determinarse estáticamente, pues en el caso de los pointcuts dinámicos, los joinpoints afectados dependen de los valores que toman los joinpoint durante su ejecución. Si bien con la POO y el polimorfismo ocurre algo similar -el en sentido de que ante un método polimórfico, no es posible saber que código se ejecutará realmente, hasta el momento de la ejecución- la comunidad OO ha desarrollado un conjunto de reglas y buenas prácticas sobre como utilizar y como no, la redefinición de métodos.

En cuanto al primer punto, tal como lo ha manifestado Kiczales, a diferencia del GOTO, los entornos de desarrollo brindan información sobre los puntos donde se aplicarán aspectos y si bien esto implica una dependencia del entorno de desarrollo, hoy en día el desarrollo de cualquier aplicación real es casi impensado sin el uso de entornos de desarrollo integrados, dado el gran esfuerzo que requeriría recordar la sintaxis tanto de las bibliotecas propias como de las provistas por la plataforma.

En cuanto al segundo punto, AOP mediante el soporte de pointcuts dinámicos permite elevar el nivel de abstracción y al igual que ocurre con el polimorfismo en POO, las abstracción es inversamente proporcional al conocimiento del flujo de ejecución: cuanto mayor es el nivel de abstracción, más dificultoso es establecer el flujo de ejecución del sistema.

Es interesante también la posición de Rod Bodkin al respecto de estas objeciones. Para él la mejor herramienta para entender un sistema es un buen diseño. AOP mejora la modularización facilitando el entendimiento del sistema, incluso sin contar con herramientas integradas al entorno de

desarrollo. La práctica de mirar el código fuente para entender el sistema, no aplica en el caso de las grandes aplicaciones, donde una buena modularización resulta mucho más importante.

Al mismo tiempo en [Elrad01], Harold Ossher consultado respecto de si AOP dificulta el entendimiento de la aplicación, afirma que las incumbencias están presentes en toda aplicación, mezcladas de una u otra forma y que representan la principal fuente de complejidad. Los enfoques orientados a aspectos permiten que las incumbencias transversales implícitas sean extraídas y explicitadas, posibilitando a los programadores que vean con qué están lidiando.

Por su parte Ramnivas Laddad, en su artículo "AOP myths and realities"[Laddad06], identifica esta problemática con el mito "los aspectos oscurecen el flujo del programa". Ante este mito Laddad afirma que bien vale la pena sacrificar cierta claridad del flujo del programa a cambio de elevar el nivel de abstracción, lo cual permite un mapeo más directo entre los requisitos, el diseño y el código, evitando que las incumbencias transversales se pierdan dispersas en el código. Al mismo tiempo, esta pérdida de claridad puede ser mitigada en con el uso de herramientas integradas al entorno de desarrollo.

Vayamos a un caso concreto, analicemos el código del método Ejecutar de la clase Transferencia en el siguiente fragmento de código.

```
public abstract class CuentaBancaria {
      public abstract void debitar(double monto);
      public abstract void acreditar(double monto);
}
public class CuentaCorriente extends CuentaBancaria {
      public void debitar(double monto) { ...   }
      public void acreditar(double monto) { ... }
}
public class CajaDeAhorro extends CuentaBancaria {
      public void debitar(double monto) { ...   }
      public void acreditar(double monto) { ... }
}

public class Transferencia {
      private CuentaBancaria cuentaOrigen;
      private CuentaBancaria cuentaDestino;

      public void ejecutar(){
            origen.debitar(monto);
            destino.acreditar(monto);
       }
}
```

El código mostrado sólo hace uso de OO, pero dado que se ha utilizado herencia no es posible saber mirando el código que código se ejecutará al invocar a los métodos *debitar* y *acreditar*. Y si bien tampoco es posible saber si durante la ejecución intervendrá algún aspecto, si utilizaramos un entorno desarrollo con soporte de AOP como Eclipse, el mismo entorno nos indicaría de la intervención de los aspectos. Por la tanto, si la OO y el polimorfismo también oscurecen el flujo de ejecución y a pesar de

ello son utilizados en pos de mejorar la modularización, lo mismo deberíamos considerar para AOP.

AOP viola el encapsulamiento

Esta es otra de las objeciones planteadas comúnmente al uso de AOP y suele resultar consecuencia de ver AOP desde una óptica orientada a objetos. Para entender porqué es errónea esta afirmación es necesario en primer lugar dejar en claro que es lo que se pretende encapsular. Lo que queremos encapsular son las distintas incumbencias.

En ocasiones al trabajar con OO, una clase debe lidiar con más de una incumbencia y al mismo las incumbencias transversales están dispersas por varias clases, ante lo cual uno podría decir que las incumbencias transversales no están encapsuladas.

En cambio, para AOP las incumbencias transversales son encapsuladas por los aspectos, haciendo que las clases cuenten con una mejor cohesión, ya que no deben lidiar con las incumbencias transversales. Por supuesto que según esta concepción, la aplicación está constituida por la totalidad de las incumbencias, las modeladas con clases y también las modeladas con aspectos.

Al mismo tiempo, es cierto que algunas herramientas AOP permiten interferir en los miembros privados de las clases, lo cual violaría el encapsulamiento, pero también es cierto que hay lenguajes OO que permiten definir atributos públicos, lo cual también viola el encapsulamiento. El hecho de que las herramientas permitan hacer ciertas cosas no implica que las mismas sean parte del paradigma. Como ocurre en todo contexto, al usar cualquier herramienta siempre en necesaria una cuota de criterio.

Depurar aplicaciones con AOP es difícil

Esta afirmación es identificada como mito en [Laddad06], donde se sostiene que la depuración siempre es difícil mientras que no se cuente con herramientas apropiadas y AOP no es la excepción. Ciertamente la depuración puede ser un verdadero problema dependiendo de la herramienta AOP utilizada, llegando a ser un factor de gran influencia a la hora de elegir una herramienta AOP. En el caso de las herramientas AOP basadas en proxies, la depuración no es demasiado distinta a la depuración en tecnologías orientadas a objetos. Por su parte, en el caso de las herramientas AOP de entretejido estático la depuración ciertamente suele ser bastante más compleja y representa un claro desafió para la evolución de AOP. Si bien en el caso particular de AspectJ, el plug-in de Eclipse AJDT facilita la depuración poniéndola al nivel de la depuración de java puro, no ocurre lo mismo con todas las herramientas AOP y esto ha motivado varias lineas de investigación [Eidelman06], [Eaddy05].

AOP y Patrones de Diseño

Este tema ha sido tratado en múltiples publicaciones [Laddad03], [Miles05], [Lesiecki05], [Erdody06], pero a pesar de ello, existe el mito de que puede obviarse el uso de AOP si se utilizan patrones de diseño. La realidad es que si bien los patrones de diseño ofrecen soluciones genéricas a

problemas recurrentes y la mayoría de los programadores que utilizan tecnologías orientadas a objetos están familiarizados con su uso, los patrones de diseño presentan algunas desventajas como por ejemplo:

- En ocasiones afectan a más de una clase.
- A veces son intrusivos.
- Su reutilización (a nivel código fuente) es difícil.

Como lo demuestran las publicaciones previamente mencionadas, el uso de AOP puede enriquecer la implementación de patrones de diseño, superando las desventajas mencionadas, disminuyendo su complejidad, facilitando su reuso y aumentando el grado de flexibilidad. Incluso en algunos caso, el uso de AOP puede llegar a reemplazar el uso de ciertos patrones de diseño como se sugiere en [Isberg05].

Si bien la gran mayoría de las publicaciones sobre esta temática tratan sobre la implementación de patrones de diseño de objetos mediante el uso de AOP, algunas publicaciones también proponen el uso de patrones de diseño de aspectos como por ejemplo “implicit protocol”, “anchor protocol” [Pawlak05], Director y BorderControl [Miles05]

Finalmente, más allá de lo antes mencionado, en cada diseño el programador deberá decidir -en base a ciertos factores del contexto del problema a resolver- si usa un patrón tradicional, una variante AOP, un patrón AOP o simplemente una solución ad-hoc.

La piedra en el zapato

Posiblemente el problema más serio que enfrenta la programación orientada a aspectos en la actualidad está dado por la fragilidad de los modelos de pointcuts de las herramientas AOP. Los pointcuts identifican los puntos de la aplicación en los que intervendrán los aspectos. El problema con las herramientas actuales radica en que generalmente la definición de los pointcuts se hace utilizando (pseudo)expresiones regulares para identificar joinpoints en el código de la aplicación. Esto hace que los pointcuts dependan de cómo se estructura el código sobre el que actuarán los aspectos. De esta forma si el programador decide por ejemplo cambiar el nombre de un método, puede que ciertos poincuts “queden sin efecto”. Del mismo modo, si el programa evoluciona puede que los pointcuts afecten a nuevos joinpoints, por una simple coincidencia de nombres a pesar de que conceptualmente no debieran.

Este problema ha sido analizado en un trabajo llamado “On the existence of the AOSD-Evolution Paradox”[Tourwé03]. Dicho trabajo destaca el hecho de que las herramientas AOP han puesto gran énfasis en aumentar la modularidad de las aplicaciones, dejando de lado las capacidades de evolución de la mismas.

Si bien tanto en java como en .NET este problema puede mitigarse con el agregado de metadata en forma de anotaciones, esto por sí solo no resulta una solución óptima ni mucho menos, aunque, constituye un elemento más para la construcción de posibles soluciones. Otra posible forma de mitigación comúnmente citada, es la utilización de estándares de desarrollo para la estructuración del código y la nomenclatura.

Debido al gran impacto que tiene este problema en la evolución del paradigma, varios trabajos han planteado algunas posibles soluciones. En [Koppen04] se propone el uso de una herramienta(PCDiff) para detectar las diferencias de pointcuts a medida que un sistema evoluciona. En [Cyment04] se plantea un herramienta AOP basada en pointcuts semánticos (setpoint) que parece ser muy prometedora.

AOP y AspectJ no son sinónimos

Si bien AspectJ es claramente la herramienta AOP más importante y ha ejercido una gran influencia en el paradigma AOP, algunas de las propiedades de AspectJ exceden el paradigma. Muy comúnmente los conceptos del paradigma AOP y las propiedades de AspectJ se mimetizan, a punto tal que varios autores suelen hacer referencia a propiedades de AspectJ como si fueran conceptos de AOP. Un caso típico de este fenómeno se da cuando ciertos autores mencionan entre las usos de AOP, el forzado de políticas, lo cual es una de la utilidades especificas de AspectJ.

Esta funcionalidad de forzado de políticas tiene que ver con la posibilidad que ofrece AspectJ de declarar ciertos pointcuts como errores de compilación. Veamos un ejemplo concreto: uno podría definir como política de diseño que ninguna clase perteneciente a la capa de presentación puede invocar directamente a clases en la capa de acceso a datos. Utilizando la sentencia *declare error*, de AspectJ uno podría definir un pointcut que identifique las llamadas de clases de presentación a clases de acceso a datos y asociar a dicho pointcut una sentencia *declare error*.

```
public aspect AspectoForzadorDePoliticas {
    pointcut enPresentacion() : within(presentacion..*);
    pointcut enPersistencia() : within(persistencia..*);
    pointcut llamadaPersistencia() : call(* persistencia..*(..)) && !
            enPersistencia();
    declare error : llamadaPersistencia() && enPresentacion(): "Acceso a datos
            en la presentacion";
}
```

Figura 16: ejemplo de forzado de políticas con AspectJ

Si bien este uso de AspectJ puede resultar muy útil, no constituye bajo ningún punto de vista una propiedad de AOP.

Pros y contras según la industria

De acuerdo al estudio realizado por Allison Duck [Duck06] sobre el uso de AOP en desarrollos

industriales, la principal ventaja de AOP es que "permite hacer exactamente lo que uno quiere, de forma rápida, limpia y poderosa". Al mismo tiempo el inconveniente más destacado según dicho estudio, es "la dificultad de depuración y la imposibilidad de saber a simple vista en que partes del código intervendrá un aspecto".

Proceso de adopción

Ron Bodkin han definido un proceso de cuatro fases para lograr una exitosa adopción de AOP [Bodkin06], generando experiencias positivas y manejando el riesgo que implica el uso de una nueva tecnología:

1. ***Aprendizaje y experimentación***: preguntas típicas de esta etapa son "¿que puedo hacer con aspectos?" y "¿cómo los hago funcionar?". Se elige alguna herramienta y se crean aspectos de exploración que brindan información sobre la aplicación durante su ejecución, casos representativos son el logging y sus derivados profiling y tracing. Estos aspectos son absolutamente transparentes tanto para la aplicación como para el equipo de desarrollo. Una actividad importante para esta etapa es probar los aspectos que suelen proveer los tutoriales de las distintas herramientas AOP.
2. ***Resolución de problemas reales***: en esta fase las preguntas centrales son "¿Cómo puede AOP ayudarme?", "¿cómo puedo integrar AOP en el trabajo cotidiano?" y "¿cómo hacer para que el resto del equipo adopte AOP?". El foco son aspectos técnicos y relacionados a atributos de calidad como manejo de excepciones, transacciones, caché y sincronización, los cuales pueden o no que sean transparentes el equipo de desarrollo.
3. ***Integración en el dominio***: el comienzo de esta fase representa un punto de inflexión, las cuestiones centrales pasan a ser cómo hacer colaborar aspectos y clases y cómo integrar AOP en el proceso de desarrollo. Se utilizan aspectos para modelar incumbencias del dominio. Todo el equipo de desarrollo está al tanto de los aspectos.
4. ***Reuso***: es el máximo nivel de madurez, habiendo utilizado AOP en al menos 3 proyectos, estamos en condiciones de extraer los aspectos comunes, definiendo interfases estables y componentizandolas en una biblioteca de aspectos.

Al mismo tiempo hay ciertos principios que deben respectarse a lo largo de todas las etapas.

- La adopción debe ser incremental dentro de cada etapa.
- Primero reusar y luego crear, buscar soluciones existentes que puedan reusarse o extenderse en lugar de siempre intentar crear nuevas soluciones.
- Invertir en sorpresas placenteras, es importante que las primeras experiencias con AOP sean exitosas para poder facilitar el proceso de adopción.

- Aprender con teoría y práctica, balancear la escritura de código con la lectura de artículos de de referencia.

Figura 17: Proceso de adopción de AOP propuesto por Bodkin

En el estudio de Duck previamente mencionado la mayoría de los entrevistados resultaron estar en la fase 1 de adopción, mientras que ninguno estaba en fase 3 o 4.

Adicionalmente a las fases propuestas por Bodkin, para poder hacer pleno uso de AOP, es necesario hacer algunas modificaciones en las distintas disciplinas del proceso de desarrollo principalmente en lo que respecta a modelado de dominio y la captura de requisitos.

Arquitectura base sin AOP

En los siguientes dos capítulos se analizarán la incumbencias transversales presentes en las AE y dado que la transversalidad de una incumbencia puede variar dependiendo de como se arme la arquitectura, resulta necesario dejar en claro la arquitectura base que se asume en lo que resta del presente trabajo.

Esta arquitectura será modificada en el capítulo 7, cuando se analice el caso de estudio, y se considere el papel de AOP a nivel de arquitectura.

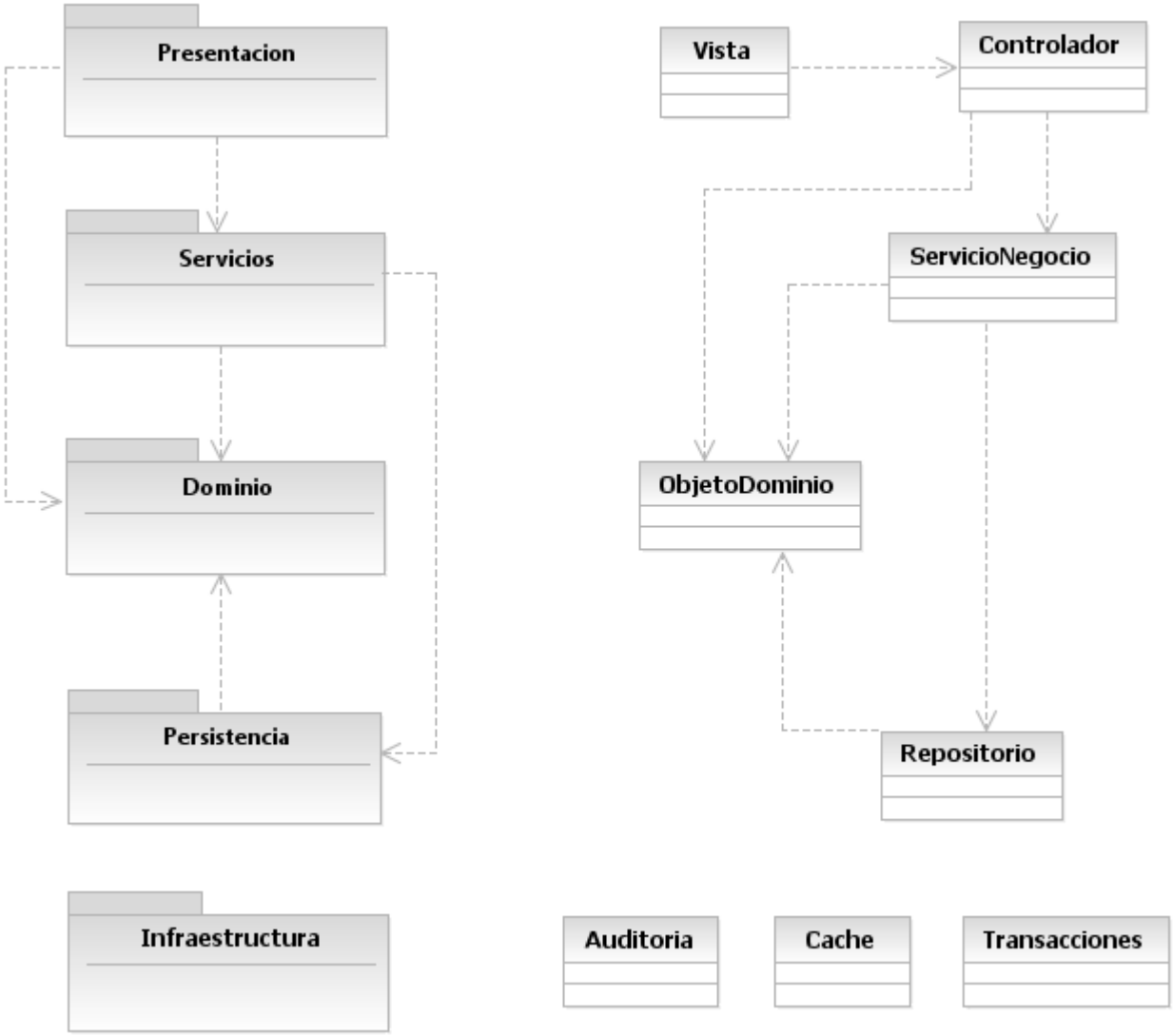

Figura 18: vista lógica de la arquitectura base

La figura 18 muestra la arquitectura asumida, la cual consta de 5 capas:

- ***Presentación***: muestra la información al usuario y capta la interacción del mismo convirtiéndola en estímulos hacia el resto de aplicación. Se encuentra estructurada en base al patrón Model-View-Controller.
- ***Servicios***: esta capa define las funcionalidades que ofrece la aplicación, por lo cual en ocasiones también es llamada capa de aplicación. Cada servicio de esta capa ofrece un conjunto de operaciones con sentido para el negocio. Capa operación coordina la interacción de un conjunto de objetos del dominio, al mismo tiempo que se encarga de la interacción con los servicios de infraestructura. Cabe destacar que los servicios de esta capa carecen de estado.
- ***Dominio***: agrupa los objetos del dominio que representan los conceptos y reglas del negocio. Los objetos de esta capa reflejan el estado del negocio y por ello constituyen el corazón de la aplicación.

- ***Persistencia***: esta capa se encarga de la persistencia de los objetos del dominio en un repositorio permanente, que salvo contadas excepciones, es una base de datos relacional. Está representada por clases que actúan como repositorios de objetos.
- ***Infraestructura***: provee funcionalidades de carácter técnico, que son utilizadas por las demás capas de la aplicación. Si bien en la figura 18 esta capa se muestra desconectada del resto de las capas, es solo por una cuestión de claridad, ya que todas las capas hacen uso de esta. Las funcionalidades brindadas por esta capa suelen denominarse servicios enterprise o de infraestructura.

Esta arquitectura está basada en la propuesta de Jimmy Nilsson[Nilsson06], con algunas salvedades, por ejemplo: la capa de infraestructura y persistencia han sido separadas, por considerar a esta última capa más amplia e importante.

Capítulo V: Incumbencias técnicas

Consideraciones preliminares

En este capitulo analizaremos las incumbencias transversales relacionadas a la implementación de atributos de calidad y requisitos técnicos que suelen estar presentes en toda AE y que comúnmente son conocidas como servicios enterprise. Antiguamente gran parte de estas incumbencias eran denominadas requisitos no funcionales, lo cual demuestra claramente que no forman parte de la dimensión dominante del problema.

En cuanto a la implementación de estas incumbencias, es conveniente en la mayoría de los casos utilizar una herramienta AOP entretejido dinámico (basada en proxies dinámicos), ya que esto posibilitaría “agregar y/o quitar aspectos” en forma declarativa modificando simplemente archivos de configuración, sin necesidad de volver a compilar toda la aplicación. Es por esto que en los ejemplo de código de aspectos mostrados en este capítulo se utilizará una notación genérica simplificada y fácilmente adaptable a la notación de cualquier herramienta AOP basada en proxies.

Al mismo tiempo hay que tener en cuenta que las herramientas AOP basadas en proxies requieren que los objetos a los que se les pretende agregar aspectos, sean instanciados utilizando alguna clase de la propia herramienta AOP, para posibilitar la creación del proxy. Si bien esto puede resultar intrusivo en muchos casos, no lo es el caso de las AE, ya que el uso de fábricas de objetos de una práctica común.

Las incumbencias que se analizarán en este capítulo generalmente son implementadas a nivel de la capa de servicios con el fin de disminuir su influencia transversal, pero en la mayoría de los casos esta estrategia no resulta lo suficientemente efectiva, ya que la transversalidad no es eliminada, sino simplemente reubicada.

Para cada una de las incumbencias identificadas, daremos una breve descripción poniendo de relevancia su naturaleza transversal, comentaremos las formas más típicas de implementación actuales y propondremos, cuando valga la pena, una implementación basada en AOP.

Monitoreo y auditoría

Problemática

El monitoreo brinda visibilidad sobre el funcionamiento de la aplicación. Existen básicamente tres técnicas de monitoreo cada una con un fin específico: logging, tracing y profiling.

El logging es una de las técnicas más utilizadas para entender el comportamiento de una aplicación. Durante la fase de desarrollo el logging suele usarse en ocasiones como herramienta de depuración. En su forma más básica el logging consiste en escribir en un archivo de texto (o simplemente en la salida estándar) mensajes describiendo las operaciones realizadas por la aplicación.

El tracing es un logging particular, donde se hace logging de la ejecución de determinadas operaciones.

El profiling tiene que ver con recolectar información sobre el tiempo que insume la ejecución de cada operación.

Por su parte la auditoría es una incumbencia relacionada a la seguridad del sistema, pero que desde el punto de vista de implementación está muy cercana al monitoreo, ya que ambas incumbencias requieren de la recolección de información durante la ejecución de la aplicación. La auditoría, a diferencia del monitoreo, recolecta información específica del negocio y suele almacenarla en una base de datos.

```
using System;
using Common.Logging;

namespace BancoX.Dominio
{
    public class CajaDeAhorro : Cuenta
    {
        protected static ILog _logger =
            LogManager.GetLogger(typeof(CajaDeAhorro));

        public override void Debitar(decimal monto)
        {
            _logger.Info("Intentando CajaDeAhorro:" +
                this.Numero + ".Debitar:" + monto.ToString());

            if (monto > this.Saldo)
                throw new SaldoInsuficienteException(this.Saldo,
            monto);

            this.Saldo -= monto;

            _logger.Info("Completado CajaDeAhorro:" +
                this.Numero + ".Debitar:" + monto.ToString());
        }
    }
}
```

Figura 19: Clase de dominio con código de logging.

En la actualidad en las aplicaciones enterprise, para la recolección de la información de monitoreo y auditoría suelen utilizarse bibliotecas de logging como las provistas por el proyecto Apache Logging Services [APS]. Las bibliotecas de logging se encargan exitosamente de la publicación de la información recolectada, pero el uso de dichas bibliotecas produce código disperso y , tal como puede verse en el fragmento de código de la figura 19.

Solución AOP

La naturaleza transversal del logging (y sus derivados) es tan clara, que ha convertido a esta incumbencia en el ejemplo introductorio a AOP y como tal, ha sido ha sido tratada en varias publicaciones [Laddad03], [Lesiecki03] y [Miles05].

Dado que la publicación de la información a loguear está bien modularizada por bibliotecas específicas, el aspecto de logging deberá encargarse de la recolección de la información a publicar y de invocar a la correspondiente biblioteca de logging eliminando así el código disperso y modularizando completamente esta incumbencia.

Dependiendo de los fines que persiga el logging (auditoria, trace, profiling, debugging, etc) la información a publicar puede ser distinta, al igual que la forma de interacción con la biblioteca de logging. Para soportar esta problemática se propone la jerarquía de advices mostrada en la figura 20.

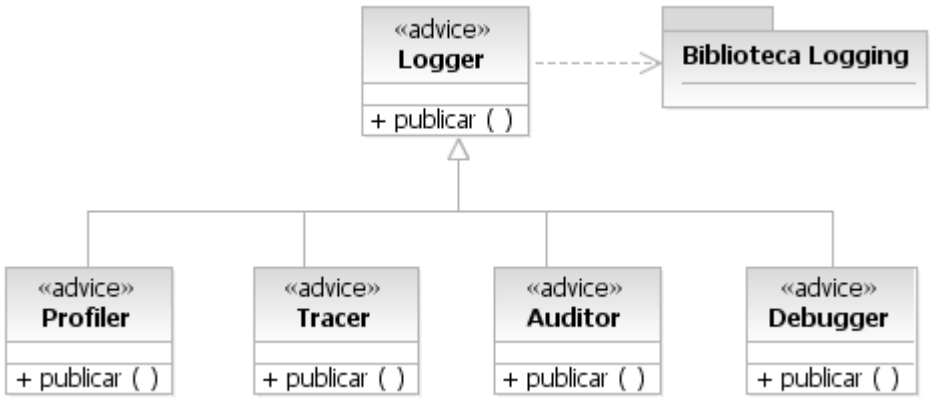

Figura 20: Advices para monitoreo y auditoría

A la hora de incluir esta incumbencia en un aplicación concreta, solo habrá que definir los pointcuts en los que deberá actuar el aspecto y establecer los parámetros de configuración de la librería de logging.

El código de la figura 21 define que se aplique el advice AuditoriaAdvice a todas las llamadas a métodos de la clase Cuenta y sus descendientes

```
<aspect name="Auditoria">
  <advice type="AspectosComunes.AuditoriaAdvice"/>
  <poincut match="call" class="Cuenta+" method="*" />
</aspect>
```

Figura 21: Definición del aspecto de auditoria para el dominio bancario

Persistencia

En más de una ocasión, diversos autores se han referido a la persistencia como un ejemplo típico de incumbencia transversal y por consiguiente se ha planteado su implementación como aspecto [Soares02], [Rashid02], [Miles05].

En [Rashid02] se plantea que la persistencia no sólo puede ser modularizada usando AOP, sino que también es posible lograrlo con un alto grado de reuso, llegando al punto extremo de poder desarrollar la aplicación ignorando la naturaleza persistente de los datos.

Dada la importancia que tiene la persistencia de la información en el caso de las AE, resulta difícil pensar que una aplicación pueda desarrollarse ignorando la existencia de la base de datos. Como ya hemos mencionado las AE suelen contar con una capa de persistencia que encapsula la interacción con la base de datos y resuelve el problema de la diferencia de impedancia. Dependiendo del diseño que se adopte en dicha capa de persistencia, la transversalidad de la persistencia puede llegar a reducirse de forma considerable. Un claro ejemplo de esto es el soporte de persistencia que ofrecen los contenedores EJB. Otras de las herramientas que ofrece una interesante solución a la persistencia de objetos es el framework Hibernate, el cual se ha convertido en un estándar de facto, principalmente en el mundo java.

Si bien la utilización de soluciones como las mencionadas pareciera no dejar lugar para el uso de AOP, resulta que ambas soluciones internamente hacen uso de técnicas muy cercanas a las utilizadas por las herramientas AOP. Hibernate por ejemplo, cuando carga un objeto desde la base de datos, agrega un interceptor para detectar los cambios posibles cambios y persistir dichos cambios al finalizar la sesión.

En el fragmento de código de la figura 22 se ve como la interacción con la base de datos y el manejo del ciclo de vida de los objetos está en manos de objetos denominados *Repositorios*. Dichos repositorios encapsulan la interacción con algún framework de persistencia como Hibernate y elevan el nivel de abstracción proveyendo métodos con la semántica del dominio. Resumiendo, existen actualmente soluciones para la persistencia en AE que ofrecen un grado aceptable de modularidad, haciendo innecesario el uso AOP, aunque hay que destacar la utilidad de AOP para la implementación de dichas soluciones.

Un tema muy relacionado a la persistencia es el manejo de transacciones, pero dada su importancia merece una análisis particular, el cual haremos en la siguiente sección.

```
using System;
using BancoX.Dominio;

namespace BancoX.Servicios
{
    public class ServicioTranferencia
    {
        public Transferencia TransferirFondos(string origen,
            string destino, decimal monto)
        {
            if (monto <= 0)
                throw OperacionNoValidaException();

            Cuenta cuentaOrigen =
                  _repositorioCuentas.ObtenerCuenta(origen);
            Cuenta cuentaDestino =
                  _repositorioCuentas.ObtenerCuenta(destino);

            if (cuentaOrigen == null)
                throw new CuentaNoValidaException();


            if (cuentaDestino == null)
                throw new CuentaNoValidaException();

            Banco banco = new Banco();
            Transferencia transferencia =
            banco.TransferirFondos(cuentaOrigen,cuentaDestino,monto);

            return transferencia;
        }

        private IRepositorioCuentas _repositorioCuentas;
    }
}
```

Figura 22: Manejo de persistencia mediante Repositorios

Transaccionalidad

Problemática

Como mencionamos en el capítulo 3, una transacción de negocio puede implicar varias transacciones del sistema. Por poner un ejemplo, una transferencia de fondos de una cuenta a otra, representa una transacción a nivel negocio, pero representa al menos dos transacciones a nivel sistema, una por cada cuenta implicada en la trasferencia. Esta diferencia de grano entre los dos tipos de transacciones, provoca que la delimitación y administración de una transacción de negocio adquiera una naturaleza transversal.

```
public Transferencia TransferirFondos(Cuenta cuentaOrigen, Cuenta
            cuentaDestino, decimal monto)
{
      Banco banco = new Banco();
      Transferencia transferencia = null;

      using (TransactionScope scope = new TransactionScope())
      {
            transferencia = banco.TransferirFondos(cuentaOrigen,
                  cuentaDestino, monto);
            _repositorioCuentas.Actualizar(cuentaOrigen);
            _repositorioCuentas.Actualizar(cuentaDestino);
            scope.Complete();
      }
      return transferencia;
}
```

Figura 23: Implementación de una transacción de negocio

Solución AOP

La recurrencia de esta problemática en las AE, es uno de los principales motivos para el uso de servidores de componentes y contenedores livianos, ya que los mismos brindan soporte declarativo de transacciones, evitando así una gran cantidad de código disperso.

Algunas herramientas como Spring Framework y Jboss han implementado el soporte declarativo de transacciones utilizando AOP, posicionando esta incumbencia junto con el monitoreo como las dos incumbencias más comúnmente implementadas con AOP en la actualidad.

```
namespace AspectosComunes
{
    public class TransaccionAdvice : IAroundAdvise
    {
        private TransactionScopeOption _scopeOption;

        public object Invoke(IMethodInvocation invocation)
        {
            object resultado = null;
            using(TransactionScope scope = new TransactionScope(_scopeOption))
            {
                resultado = invocation.Proceed();
                scope.Complete();
            }
            return resultado;
        }

    }
}
```

Figura 24: Advice para el manejo de transacciones

Una forma simple de aplicar transaccionalidad con AOP es identificando los métodos transaccionales con ciertas anotaciones y luego definir los poincuts del aspecto transaccional en base a

dichas anotaciones[8].

```
<aspect name="Transaction">
  <advice type="AspectosComunes.TransaccionAdvice">
    <property name="ScopeOption" value="REQUIERED" />
  </advice>
  <poincut methodAnnotation="Transaccional" />
</aspect>
```

Figura 25: Definición del aspecto de manejo de transacciones

El fragmento de código de la figura 25 aplica el advice TransaccionAdvice a todos las clases con métodos que contengan la anotación *Transaccional*.

```
[Transaccional]
public Transferencia TransferirFondos(Cuenta cuentaOrigen, Cuenta
            cuentaDestino, decimal monto)
{
      Banco banco = new Banco();
      Transferencia transferencia = null;
      transferencia = banco.TransferirFondos(cuentaOrigen,
            cuentaDestino, monto);
      _repositorioCuentas.Actualizar(cuentaOrigen);
      _repositorioCuentas.Actualizar(cuentaDestino);
      return transferencia;
}
```

Figura 26: Implementación de una transacción de negocio con transaccionalidad declarativa

Caching

Problemática

Con el fin de mejorar el desempeño de las aplicaciones suele buscarse minimizar la cantidad de tiempo y recursos invertidos en la resolución de consultas y realización de cálculos. Para ello se utilizan estrategias de caché. A diferencia de otras incumbencias, el caché puede encontrarse en todas las capas de la aplicación: en la capa de persistencia para minimizar los accesos a la base de datos; en la capa de negocio, para evitar la ejecución de algún algoritmo y en la capa de presentación, para evitar redibujar información que no ha variado desde la última consulta. Pero más allá de en qué capa se utilice el caché, la forma de utilizarlo es siempre la misma. Si bien los servidores de aplicaciones suelen brindar funcionalidades de caché, su uso tiene ciertas limitaciones y generalmente genera código entremezclado.

8 Si bien en técnologia .NET las anotaciones son llamadas Atributos, se ha utilizado el término anotaciones proveniente de java, por resultar más claro.

```
public class RepositorioCuentas : IRepositorioCuentas
{
    public IList<Cuenta> ObtenerCuentasPorCliente(Cliente cliente)
    {
        string consulta = "from Cuenta c where c.Cliente.Id=" + cliente.Id;

        IList<Cuenta> cuentas = Cache.Get(consulta) as IList<Cuenta>;

        if (cuentas == null)
        {
             cuentas = Mapper.Find<Cuenta>(consulta);
            Cache.Agregar(cliente, valor);
        }

        return cuentas;
    }
}
```

Figura 27: Uso típico de Caché en la capa de persistencia

Solución AOP

Dada la claridad de la naturaliza transversal de esta incumbencia y el bajo acoplamiento con las particularidades de cada dominio, resulta muy evidente la conveniencia de encapsularla en un aspecto.

Una vez más y similarmente a lo que ocurre con logging, el aspecto de Caché funcionará como integrador entre los componentes de la aplicación y la biblioteca de Caché.

```
namespace AspectosComunes
{
    public class CacheAdvice : IAroundAdvise
    {
        public object Invoke(IMethodInvocation invocation)
        {
            ClaveCache clave = new ClaveCache(invocation.StaticPart,
                  invocation.Arguments);

            object valor = Cache.Get(clave);

            if (valor == null)
            {
                valor = invocation.Proceed();
                CacheAgregar(clave, valor);
            }
            return valor;
        }
      }
}
```

Figura 28: Advice Caché

El fragmento de código de la figura 29 aplica el CacheAdvice a todas las llamadas métodos cuyo nombre comience con “Obtener” y pertenezcan a clases cuyo nombre comience con “Repositorio”. Si bien el código no lo muestra generalmente también suelen definirse por configuración los parámetros de funcionamiento de la biblioteca de caché.

```
<aspect name="Cache">
  <advice type="AspectosComunes.CacheAdvice" />
  <poincut match="call" class="Repositorio*" method="Obtener*" />
</aspect>
```

Figura 29: Aspecto Caché aplicado a los métodos Obtener de los repositorios

Otra técnica utilizada para mejorar desempeño es la ***carga tardía.*** Esta técnica es muy utilizada por los frameworks de persistencia para cargar objetos desde la base de datos, bajo demanda, evitando cargar información que no resulta necesaria. Lo interesante es que para la implementación de esta técnica algunas herramientas utilizan AOP.

Manejo de excepciones

Problemática

El manejo de excepciones es una problemática presente en toda aplicación. En las tecnologías orientadas a objetos el mismo suele hacerse mediante el uso de bloques try-catch-finally y la clausula throw.

Cuando una aplicación está corriendo en un ambiente de producción es importante estar al tanto de la situaciones excepcionales que ocurran, lo cual hace imprescindible contar un buen manejo de excepciones.

En forma general existen dos tipos de excepciones: las técnicas y las del dominio. Las excepciones técnicas, tienen que ver con desperfectos en la infraestructura de la aplicación como ser falta de conectividad, fallos en el hardware, etc. Por su parte las excepciones de dominio tienen que ver con la violación de alguna regla del dominio. A su vez, el comportamiento de la aplicación ante estos tipos de excepciones es distinto. Ante excepciones técnicas es posible que la aplicación reintente realizar la operación fallida y en caso de no lograrlo seguramente muestre al usuario un mensaje del estilo: “La operación requerida no puedo completarse, por favor contacte al servicio de ayuda.”. Al mismo tiempo, ante excepciones de dominio generalmente la aplicación no reintenta, sino que directamente muestra al usuario un mensaje indicándole la regla que ha sido violada.

Básicamente podríamos decir que el manejo de excepciones abarca 3 cuestiones: la detección de las excepciones, su publicación y finalmente su propagación.

La detección de la excepciones en el caso de las excepciones técnicas es hecha en ocasiones utilizando técnicas como el ***diseño por contratos*** [Meyer92]. Esta técnica, popularizada por Bertrand Meyer, permite especificar la comunicación entre componentes mediante la definición de contratos los cuales especifican las obligaciones mutuas y los resultados esperados esperados de la comunicación. La verificación de los contratos se hace por medio de aserciones, las cuales son expresiones lógicas que

deben ser verdaderas, en caso de ser falsas, la aserción falla y se genera una excepción. Según el diseño por contratos, existen 3 tipos de aserciones: precondiciones, postcondiciones e invariantes. Si bien esta técnica beneficia la calidad y el reuso, a nivel de implementación, el código queda minado de aserciones, o de llamadas a frameworks de diseño por contratos como Contract4J [contract4j].

En cuanto a la publicación y propagación de las excepciones, son comportamientos que se codifican dentro de los bloques *catch*, pero puede ocurrir que ante una misma excepción, dependiendo del contexto en que ocurra, las acciones de publicación y propagación a ejecutar sean distintas, de manera tal que pueda no haber dos bloques *catch* iguales.

De esta forma en los bloques try se codifica el comportamiento normal de la aplicación, mientras que en los bloques catch, se codifican los distintos comportamientos excepcionales. Como consecuencia de esto, ocurre que dentro de un mismo método nos encontremos con al menos 2 comportamientos (el normal y el excepcional).

El fragmento de código de la figura 30 muestra muestra en el manejo de una excepción y la verificación de dos precondiciones del método *TransferirFondos*.

```
public Transferencia TransferirFondos(Cuenta cuentaOrigen, Cuenta
cuentaDestino, decimal monto)
{
    if (cuentaOrigen == null)
        throw new CuentaNoValidaException();

    if (cuentaDestino == null)
        throw new CuentaNoValidaException();

    if (!cuentaOrigen.Moneda.Equals(cuentaDestino.Moneda))
        throw new OperacionNoValidaException("Monedas distintas");

    try
    {
        cuentaOrigen.Debitar(monto);
        cuentaDestino.Acreditar(monto);

        Transferencia transferencia = new Transferencia(cuentaOrigen,
            cuentaDestino, monto);

        _repositorioOperaciones.RegistrarOperacion(transferencia);
    }
    catch (Exception e)
    {
        Logger.Error("Error al transferir fondos", ex);
        throw new OperacionFallidaException(ex);
    }
    return transferencia;
}
```

Figura 30: Manejo de excepciones

Solución AOP

Existen algunos trabajos que han tratado el manejo de excepciones con AOP. En particular hay un trabajo realizado por Cristina Lopes y Martin Lipper [Lopes00], quienes tomando como caso de estudio un framework para el desarrollo de aplicaciones de negocio, lograron reducir notablemente la cantidad de bloques catch, utilizando AOP para manejar los cinco tipos de excepciones más referenciados en bloques catch. Esto resultó en una reducción de 4 veces la cantidad de código

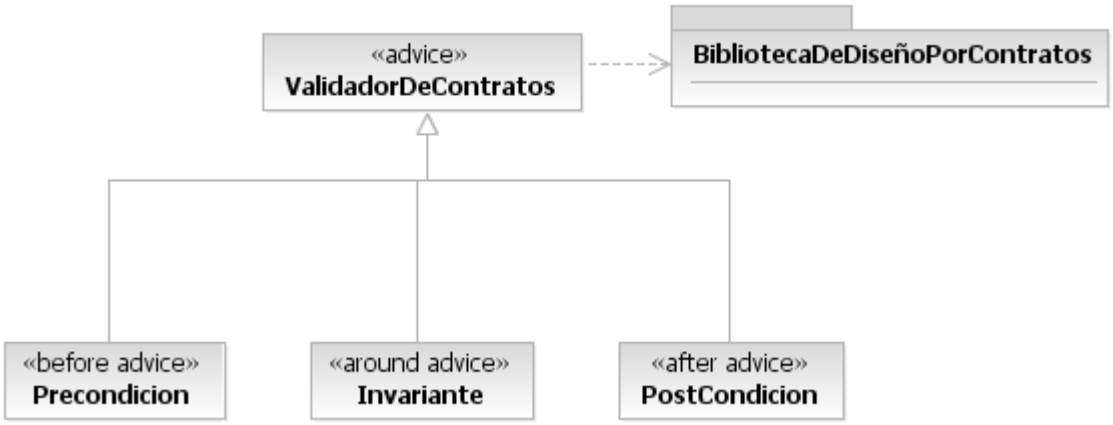

Figura 31: Advices para integración de bibliotecas de diseño por contratos

relacionado al manejo de excepciones.

En lo que hace a la detección de excepciones, resulta beneficioso utilizar la técnica de diseño por contratos, lo que ha motivado algunas propuestas para implementar dicha técnica utilizando AOP [Diotalevi04]. Incluso la biblioteca de aspectos de AspectJ trae un conjunto de aspectos que brindan soporte para esta técnica. La figura 31 muestra un posible diseño para facilitar el uso de bibliotecas de diseño por contrato mediante el uso de AOP.

```
namespace AspectosComunes
{
    public class ArgNoNuloAdvice : Precondicion
    {
         public void Before(System.Reflection.MethodInfo method, object[]
                   args, object target)
         {
             AdministradorDeContractos.AsegurarNoNulos(args);
         }
    }
}
```

Figura 32: Advice verificador de precondición de argumentos no nulos

El código de la figura 33 muestra la configuración del advice que verifica argumentos no nulos, indicando que dicho advice se aplique a todos los métodos que reciban como argumento una instancia de la clase *Cuenta* o descendientes.

```
<aspect name="ArgumentoNoNuloAdvice">
  <advice type="AspectosComunes.ArgNoNuloAdvice" />
  <poincut args="Cuenta+" />
</aspect>
```

Figura 33: Aplicación del advice verificador de argumentos no nulos a todos los joinpoint que reciben como argumento objetos Cuenta

En cuanto a la publicación y propagación de las excepciones, varias herramientas AOP brindan en sus bibliotecas aspectos para el manejo de estas problemáticas.

```
<object name="exceptionHandlingAdvice"
      type="Spring.Aspects.Exceptions.ExceptionHandlerAdvice, Spring.Aop">
  <property name="exceptionHandlers">
    <list>
      <value>on ArithmeticException wrap
                System.InvalidOperationException</value>
    </list>
  </property>
</object>
```

Figura 34: Configuración del advice de manejo de excepciones de Spring

En particular el soporte de manejo de excepciones brindado por Spring resulta muy interesante. Spring brinda un Advice para el manejo de excepciones y un lenguaje específico para especificar el comportamiento de dicho advice en forma declarativa, pero si bien esto resulta muy interesante hay que mencionar que no es posible definir los pointcuts sobre los que se pretende que actué el advice, ya que el mismo aplica automáticamente sobre todos los objetos a los que Spring les ha agregado aspectos.

Autenticación y autorización

Problemática

Estos dos problemáticas relacionadas a la seguridad de las aplicaciones, están muy relacionadas entre sí, ya que para permitir el uso (autorización) de cierta funcionalidad,primero se debe identificar quien es el usuario (autenticación) que pretende hacer uso de la misma. Al mismo tiempo estas problemáticas están presentes en todas las capas de la aplicación, pero en particular en la capa de presentación, ya que es el punto de acceso a la funcionalidad de la aplicación. Asimismo estas incumbencias determinan en gran medida las vistas a las que el usuario podrá acceder.

Generalmente, el usuario presenta su credenciales en alguna vista de la aplicación, que se encarga de autenticarlo utilizando un componente específico. Luego de ello la plataforma de ejecución se encarga de crear un contexto de ejecución conteniendo la identidad del usuario y evitando que este deba autenticarse constantemente. Esta es una solución aceptable desde el punto de vista de la modularización.

Por otro lado, la implementación de la autorización suele ser mucho más trabajosa, porque si bien los permisos que tiene el usuario suelen almacenarse en el contexto de ejecución junto con la identidad del mismo, dichos permisos deben ser constantemente consultados por cada acción que el usuario realiza en la aplicación, lo que produce los mencionados fenómenos de código disperso y entremezclado.

```
public IList<Cuenta> ObtenerCuentas(string numeroCliente)
{
    if (!Thread.CurrentPrincipal.IsInRole("USUARIO_HOMEBANKING"))
    {
        throw new UnauthorizedAccessException("Privilegios insuficientes.
            USUARIO_HOMEBANKING requerido");
    }
    return this._repositorioCuentas.ObtenerCuentasPorCliente(numeroCliente);
}
```

Figura 35: verificación de permisos para realizar una consulta de cuentas

Si bien las tecnologías actuales brindan mecanismos declarativos de autorización, ocurre que generalmente los mismos son de grano grueso y centrados en la capa de presentación o de servicios.

Solución AOP

Esta incumbencia es otra de las que se encuentra en el Top 5 de las incumbencias implementadas con AOP. El tema es abordado de forma interesante en [Laddad03] y también hay implementaciones con un importante grado de adopción como ser el framework ACEGI [acegi] basado en Spring.

```
namespace AspectosComunes
{
    public class AutZAdvice : IBeforeAdvice
    {
        public void Before(MethodInfo method, object[] args, object target)
        {
            string permiso = GetPermiso(method);
            if (!string.IsNullOrEmpty(permiso))
            {
                if (!Thread.CurrentPrincipal.IsInRole(permiso))
                {
                    Logger.Info("Intento de acceso no autorizado" +
                        method.DeclaringType.Name + method.Name);
                    throw new UnauthorizedAccessException("Privilegios
                        insuficientes." + permiso + "requerido.");
                }
            }
        }

        protected string GetPermiso(MethodInfo method){...}
}
```

Figura 36: fragmento de código de un advice de autorización

Con AOP la implementación de la autorización puede reducirse a la definición de un BeforeAdvice, que tomando ciertos parámetros de configuración, se encargue de verificar los permisos. El código de la figura 37 aplica el advice *AutZAdvice* a todas las clases del componente *BancoX.Servicios* y al mismo tiempo define que para ejecutar el método *ObtenerCuentas* de la clase *ServicioHomeBanking* es necesario contar con el permiso USUARIO_HOMEBANKING.

```
<aspect name="Autorizacion">
  <advice type="AspectosComunes.AutZAdvice">
    <property name="Permisos">
      <item name="ServicioHomeBanking.ObtenerCuentas()"
                value="USUARIO_HOMEBANKING"/>
    </property>
  </advice>
  <poincut assembly="BancoX.Servicios"/>
</aspect>
```

Figura 37: aplicación de advice de autorización a las clases de servicio del BancoX

En lo que autenticación respecta, podría aplicarse una estrategia análoga a la mostrada, pero en general las tecnologías actuales brindan soluciones aceptables desde el punto de vista de la modularización haciendo que no valga la pena el desarrollo de soluciones basadas en AOP.

Distribución, concurrencia y sincronización

Problemática

En ocasiones las AE se ejecutan distribuidas ya sea por que el negocio se encuentra distribuido o bien por razones técnicas como alta disponibilidad y escalabilidad. Existen en la actualidad varias

tecnologías de distribución que van desde Corba hasta Web servicios pasando RMI y NET Remoting.

El uso de arquitectura lógicas en capas permite que las mismas sean distribuidas físicamente en distintos nodos. Generalmente la distribución física pone a la capa de presentación en el rol de cliente y a la capa de negocio el rol de servidor. La capa servidor se encarga de manejar los pedidos de la capa cliente, creando un hilo de ejecución por cada pedido, generando situaciones de concurrencia. Es crítico para las AE garantizar que el acceso concurrente a los datos no genere inconsistencia en los mismos.

Uno de los mecanismos más comunes para el manejo de situaciones concurrentes en las AE, son las transacciones y que como ya se vio, resulta conveniente el uso de AOP para su implementación.

Al mismo tiempo resulta que no todas la situaciones concurrentes pueden manejarse con transacciones y por ello es necesario utilizar los mecanismos de sincronización para garantizar que un dato no sea modificado por más de un hilo de ejecución al mismo tiempo. Si bien en ocasiones es posible manejar estas situaciones directamente en la base de datos, en otros casos, las mismas deben manejarse en la capa de negocio, utilizando semáforos y demás artefactos típicos de programación concurrente o bien haciendo uso de las herramientas específicas de cada lenguaje. Por ejemplo en Java es posible utilizar el modificador *syncronized* en la declaración de métodos, evitando la ejecución concurrente del mismo. Por su parte .NET permite definir bloques de exclusión mutua mediante el uso de la cláusula *lock*, tal como lo muestra la figura 38.

```
public class CajaDeAhorro : Cuenta
{
    public override void Debitar(decimal monto)
    {
        if (monto > this.Saldo)
           throw new SaldoInsuficienteException(this.Saldo, monto);
        lock (this)
        {
           this.Saldo -= monto;
        }
    }
}
```

Figura 38: sincronización de la modificación de saldo

Solución AOP

En primer lugar hay que destacar que si bien la problemática de distribución ha sido planteada como un aspecto en algunos trabajos, en el caso particular de las AE dicha problemática se ha resuelto de forma aceptable (en lo que a modularización respecta) mediante el uso de los mecanismos nativos de las tecnologías enterprise y ciertos patrones de diseño.

Una situación similar ocurre con la concurrencia ya que tanto los servidores de aplicaciones como motores de base de datos se encargan se manejar pedidos concurrentes creando distintos hilos de ejecución para cada uno.

En cuando a la sincronización, la situación es distinta, ya que el uso de los artefactos característicos de programación concurrente genera mucho código disperso dando lugar al uso de AOP tal como se ha planteado en diversos trabajos. Por ejemplo en [Erdody06] se plantea un conjunto de patrones implementados con AspectJ, mientras que en [Nilsson06] se propone una interesante solución basada en Spring.AOP.

El fragmento de código de la figura 39 muestra un advice para el manejo de secciones de exclusión en tecnología .NET.

```
namespace AspectosComunes
{
    public class SyncAdvice : IAroundAdvise
    {
        public object Invoke(IMethodInvocation invocation)
        {
            object resultado;
            lock(invocation.Target)
            {
                resultado = invocation.Proceed();
            }
            return resultado;
        }
    }
}
```

Figura 39: Advice de sincronización

La figura 40 muestra como aplicar el advice de sincronización a todos los métodos de la clase *Cuenta* que tengan la anotación *Sync*.

```
<aspect name="Sincronizador">
  <advice type="AspectosComunes.SyncAdvice" />
  <poincut class="Cuenta" methodAnnotation="Sync" />
</aspect>
```

Figura 40: Aplicación del advice de sincronización a los métodos con anotación Sync

Otras incumbencias

Si uno compara las incumbencias tratadas en este capítulo con las tratadas en el capítulo 3, a simple vista encuentra que más de una incumbencia no ha sido tratada. Esto se debe a que varias de las incumbencias relacionadas a atributos de calidad no se traducen directamente a código sino que se ven reflejadas en el diseño de la aplicación. Por ejemplo, el hecho de haber planteado una arquitectura en capas facilita la mantenibilidad y escalabilidad de la aplicación.

Al mismo tiempo ocurre que algunas incumbencias calificadas comúnmente como transversales, no lo son en el caso de las aplicaciones enterprise (la persistencia por ejemplo), o no vale la pena modelarlas con AOP, ya sea por que dichas incumbencias tienen un impacto menor o bien porque

modelarlas con AOP podría resultar muy dificultoso. Dos incumbencias con las que se da esta situación son la encriptación y la lógica de reintentos.

Si bien la lógica de reintentos puede considerarse una incumbencia transversal resulta que en las AE es la infraestructura la que se encarga de la misma, de forma transparente para el programador.

En cuanto a la encriptación, la misma se suele utilizar principalmente a la hora de transmitir información confidencial y resulta que una vez más es la infraestructura la que se encarga de esta incumbencia a partir de seteos de configuración.

Resumiendo...

A lo largo de este capítulo hemos analizado distintas incumbencias técnicas de las AE, la mayoría de las cuales resulta conveniente implementarlas con aspectos. Los aspectos resultantes actúan en la mayoría de los casos como integradores de bibliotecas.

Al mismo tiempo el hecho de haber planteado la implementación de estos aspectos con una herramienta AOP de entretejido dinámico hace que los mismos puedan ser reutilizados en distintas aplicaciones con el mínimo esfuerzo de definir los pointcuts en el archivo de configuración y sin necesidad de tocar el código fuente.

Capítulo VI: Incumbencias del negocio

Consideraciones preliminares

El dominio y la lógica de negocio

Habiendo analizado las incumbencias técnicas, que en cierta forma podríamos llamar secundarias, es hora de analizar las incumbencias del negocio, que si bien son la parte distintiva y más compleja de las aplicaciones enterprise, no es común que se las modele como aspectos.

Generalmente en las incumbencias de negocio de las aplicaciones enterprise puede distinguirse:

- Lógica del dominio, la cual es común a todas las aplicaciones centradas en el dominio en cuestión. Esta lógica está constituida por los conceptos centrales del dominio. Si tomamos el dominio bancario, nos encontraremos con conceptos como cuenta, cliente y transacción.
- Lógica del negocio, la cual es propia de cada aplicación y está constituida por reglas del negocio particulares de cada organización. Siguiendo dentro del dominio bancario, podremos encontrarnos con reglas particulares del Banco X como: "todas las transacciones que superen cierto monto deben ser autorizadas por la casa central".

Mientras que la lógica del dominio suele ser (relativamente) estable, las reglas del negocio suelen ser más volátiles. Al mismo tiempo estas reglas suelen afectar a varias entidades del dominio, lo cual dificulta su modelado. Las incumbencias del dominio se modelan en la capa de dominio logrando un buen nivel de modularización con el uso de OO. Por su parte las reglas del negocio suelen ser más difíciles de modelar, lo cual ha dado lugar a diferentes estrategias.

Si las reglas son simples, se las suele manejar en la capa de dominio, modelándolas con objetos del dominio. En el caso de reglas más complejas[9], es común moverlas a la capa de servicios, o bien utilizar motores de reglas.

Todas estas estrategias para modelar las reglas del negocio tienen en mayor o menor medida cierta naturaleza transversal, lo cual genera una oportunidad para el uso de AOP.

Implementación de aspectos del negocio

A diferencia de los aspectos tratados en el capitulo anterior, a los aspectos del negocio resulta conveniente implementarlos utilizando herramientas AOP de entretejido estático que actúen a nivel de código fuente (o código de máquina virtual) como compiladores, pre procesadores y pos compiladores,

9 Entiéndase reglas complejas a aquellas que involucran a varios objetos del dominio e interacciones con aplicaciones remotas

ya que las modificaciones en el negocio no se “apagan y prenden” durante la ejecución de la aplicación, sino que deben estar siempre presentes. Al mismo tiempo si se utilizara una herramienta AOP basada en proxies dinámicos, se podría llegar a correr el riesgo de que por un descuido de configuración, se modificara lógica de la aplicación. Adicionalmente, desde un punto de vista de componentes, tanto los aspectos del negocio como las clases, podrían convivir en una misma unidad.

Dado que las incumbencias de negocio dependen de cada aplicación, para el desarrollo de este capítulo tomaremos una aplicación de ejemplo, sobre la que plantearemos ciertas problemáticas transversales, que una vez analizadas, intentaremos modelar como aspectos.

A lo largo de este capítulo, pasaremos por alto todas las incumbencias que no formen parte de la lógica de negocio, ya que las hemos tratado en el capítulo anterior y al mismo tiempo nos distraerían del objetivo de este capítulo: analizar las incumbencias del negocio.

El ejemplo que utilizaremos a lo largo de este capítulo es el de una aplicación de banca electrónica personal que ya ha sido mencionada en capítulos anteriores.

Caso de estudio: Banco X

Esta hipotética aplicación permite a los clientes de un ficticio Banco X, consultar el saldo de sus cuentas y realizar movimientos de fondos entre cuentas. Como es de esperar, para poder acceder a estas funcionalidades es necesario que el usuario inicie una sesión identificándose con su DNI y su clave personal.

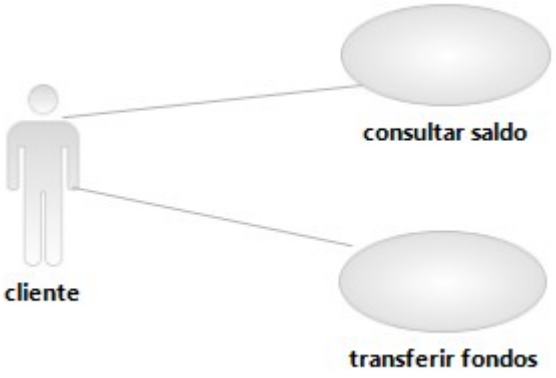

Figura 41: Funcionalidades ofrecidas vía web por el Banco X a sus clientes

El diagrama de la figura 42 muestra las clases de la capa de dominio del Banco X. A simple vista parece que todas las incumbencias han sido felizmente modularizadas, lo cual es bastante predecible dado que todos los conceptos del dominio pertenecen a la dimensión dominante del problema.

La figura 43 muestra un escenario simplificado de transferencia de fondos, partiendo desde la capa de servicios y considerando la arquitectura base descripta en el capítulo 4. Decimos que este

escenario es simplificado por no tener en cuenta la interacción con los servicios de infraestructura como ser transacciones y auditoria, dos incumbencias infaltables en toda aplicación bancaria.

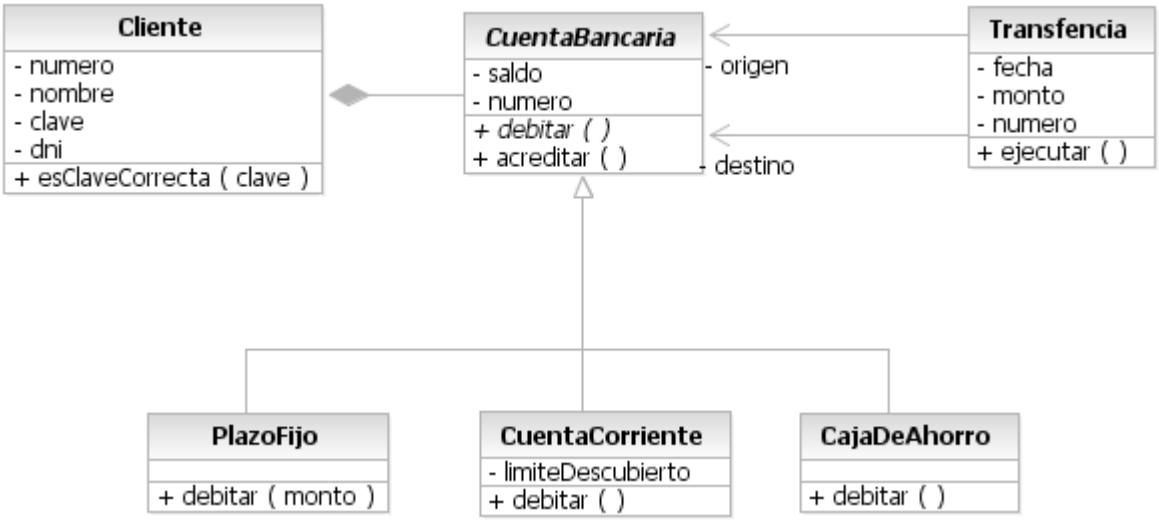

Figura 42: clases del dominio del Banco X

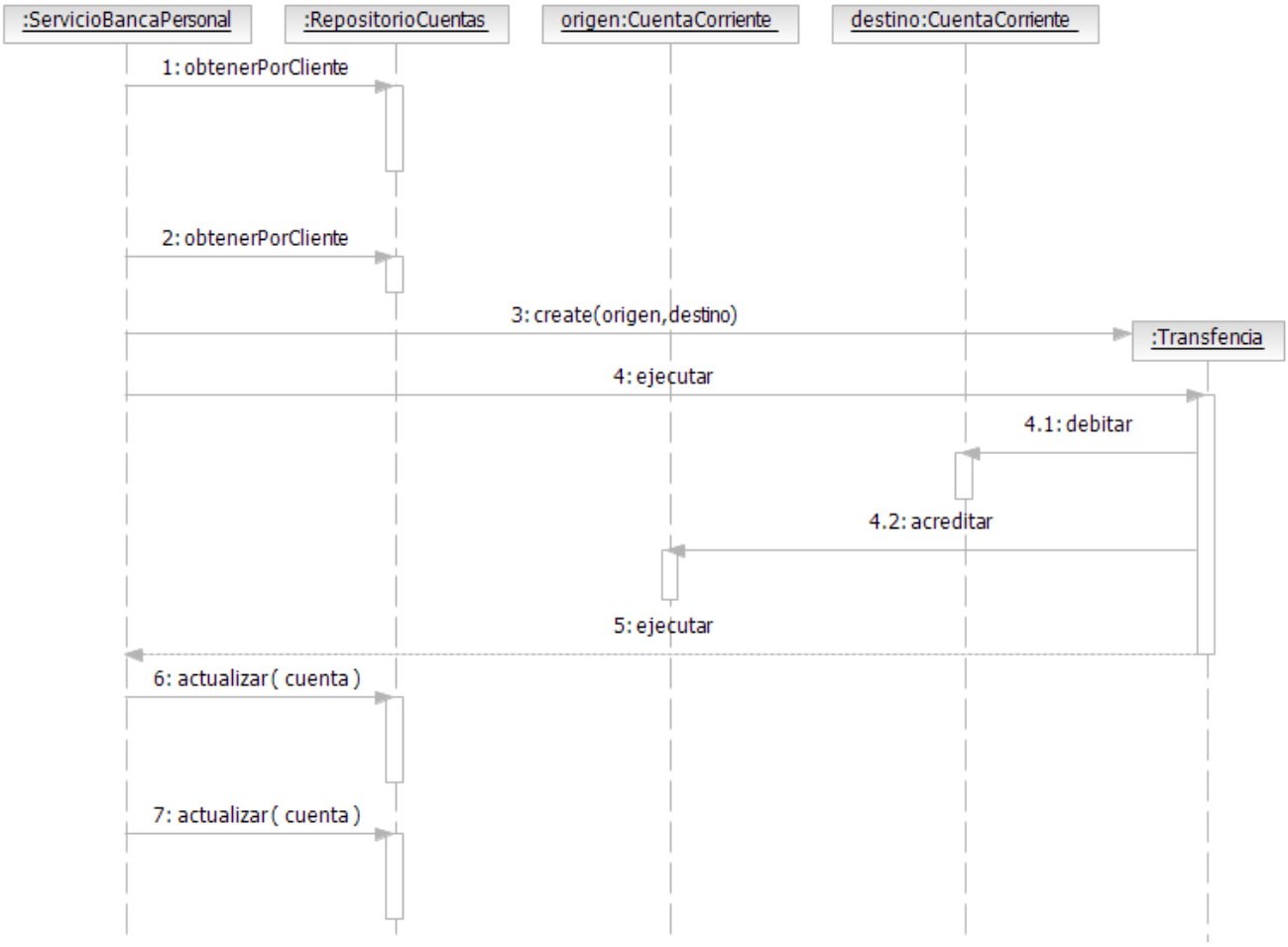

Figura 43: Escenario de transferencia de fondos

Incumbencias transversales en el negocio del BancoX

Habiendo expuesto la funcionalidad básica del dominio, queda a la vista que el dominio podría perfectamente ser reutilizado por cualquier aplicación de banca electrónica. Ahora es momento de centrarnos en las particularidades del negocio del Banco X. Para ello intentaremos modelar las siguientes dos reglas de negocio específicas de dicho banco.

1. No debe ser posible realizar débitos de cuentas cuyo titular tenga deudas con la AFIP.
2. Todas las operaciones que superen cierto monto, deben ser informadas al Banco Central de la República Argentina.

Al intentar modelar estas reglas se plantean las siguientes cuestiones.

- No resulta evidente qué objeto del dominio debería tomar estas responsabilidades.
- Estas reglas son específicas del Banco X y no del dominio bancario.
- Estas reglas son mucho más volátiles que las reglas del dominio, y como tales, es mucho más probable que sufran modificaciones.

De intentar modelar estas reglas en la capa de dominio, deberíamos decidir que objeto debería tomar su responsabilidad. Una opción sería asignar la responsabilidad a alguno de los objetos existentes, pero ello significaría violar el principio de responsabilidad única.

Una alternativa distinta podría ser agregar nuevos objetos que tomaran exclusivamente estas nuevas responsabilidades, pero en ese caso las llamadas a estos objetos se esparcirían por el todo el código de la aplicación, dificultando la evolución de la aplicación y restando claridad al código.

```
public class CajaDeAhorro : Cuenta
{
    public override void Debitar(decimal monto)
    {
        if (monto > Saldo)
            throw new SaldoInsuficienteException(this.Saldo, monto);

        if (Banco.GetProxyAfip().TieneDeuda(this))
            throw new OperacionFallidaException("Cuenta con deudas de Afip");

        Saldo -= monto;
    }
}
```

Figura 44: regla 1 implementada en la clase CajaDeAhorro. El mismo fragmento de código debería agregarse a los otros tipos de cuenta

Otra alternativa sería modelar estas reglas en la capa de servicios, lo cual es conceptualmente muy discutible. Al mismo tiempo, asegurar que estas reglas sean verificadas en cada interacción de objetos del dominio, podría generar código disperso y alto acoplamiento, dos fenómenos no deseados. Esta última situación también ocurriría en caso de modelar estas reglas con un motor de reglas.

Para el caso particular de la regla 2, una solución habitual es el uso del patrón *Observer*, pero el

uso de dicho patrón tiene las siguientes consecuencias.

- El objeto observado (en este caso la transferencia) debe recibir una referencia al objeto observador (en este caso el BancoCentral).
- Cada vez que hay un cambio de estado, el objeto observado debe notificar a todos los observadores, esto obliga a que el código de notificación sea invocado desde múltiples lugares del objeto observado.
- El objeto observador, se suscribe al objeto observable y no a eventos específicos, lo cual resulta una complicación cuando un objeto observable tiene más de un evento interesante sobre el cual notificar.

```
public class Transferencia : Operacion
{
    public void Ejecutar()
    {
        this.FechaHora = DateTime.Now;
        Origen.Debitar(Monto);
        Destino.Acreditar(Monto);

        ProxyBancoCentral proxyBancoCentral = Banco.GetProxyBancoCentral();
        if (Monto > proxyBancoCentral.MontoLimite)
            proxyBancoCentral.NotificarTransferencia(this);
    }
}
```

Figura 45: Implementación de la regla 2 en la clase Transferencia

Habiendo analizado estas situaciones, encontramos que ninguna de las alternativas ofrece una buena modularización. Veamos entonces si con AOP podemos mejorar la situación.

Respecto de la regla 1, podríamos incluirla en una incumbencia “Regulaciones de Afip”, la cual claramente cuenta con una naturaleza transversal. Para atacar esta incumbencia modelaremos una aspecto llamado *RegulacionesAfip* y que en un principio contará con un solo advice, de tipo before, encargado de la validación de la regla 1.

```
aspect RegulacionesAFIP
{
    pointcut debitos(CuentaBancaria c): target(c) && call (Debitar);

    before debitos(CuentaBancaria c)
    {
        if (Banco.GetProxyAfip().TieneDeuda(c))
            throw new OperacionFallidaException("Cuenta con deudas de Afip");
    }
}
```

Figura 46: Implementación AOP de la regla 1

Por su parte, respecto de la regla 2, aplicando una razonamiento análogo, podríamos plantear un

aspecto encargado de concentrar las reglas impuestas por el Banco Central y agregar al mismo un afteradvice que tome a su cargo la notificación al Banco Central.

El diagrama de la figura 48 muestra una solución AOP para la implementación de las reglas 1 y 2. Como puede apreciarse, la existencia de los aspectos para las clases es totalmente transparente, pero no al revés, cosa que ha sido discutida por algunos. Hay autores que argumentan que para lograr una real separación de incumbencias, los aspectos también deben ser independientes de las clases sobre las que actúan.

```
aspect RegulacionesBCRA
{
    pointcut transferencias(Transferencia t): target(t) && call (Ejecutar);

    after transferencias(Transferencia t)
    {
        if (t.Monto > Banco.GetProxyBancoCentral().MontoLimite)
            proxyBancoCentral.NotificarTransferencia(t);
    }
}
```

Figura 47: Implementación AOP de la regla 2

Un hecho que se desprende del diagrama de la figura 48 y que puede resultar llamativo es que el aspecto *RegulacionesBCRA* aplica sólo sobre la clase *Transferencia.* Ante esto alguien podría tentarse de eliminar el aspecto y mover la lógica directamente a la clase *Transferencia*. Esta estrategia no sería conveniente por varios motivos.

- La regla del Banco Central aplica sobre todas las operaciones, no solo las transferencia y si bien en el dominio planteado solo hay transferencias, nada quita que puedan agregarse otros tipos de operaciones, lo cual nos llevaría a replicar el código de la regla en la nueva clase de operación. En el caso de la solución AOP, ante esta misma situación de evolución planteada, a lo sumo habría que modificar un pointcut del aspecto.
- La decisión de delegar la responsabilidad de la regla en un aspecto está motivada principalmente porque se trata de una regla específica del negocio del Banco X y no del dominio bancario.

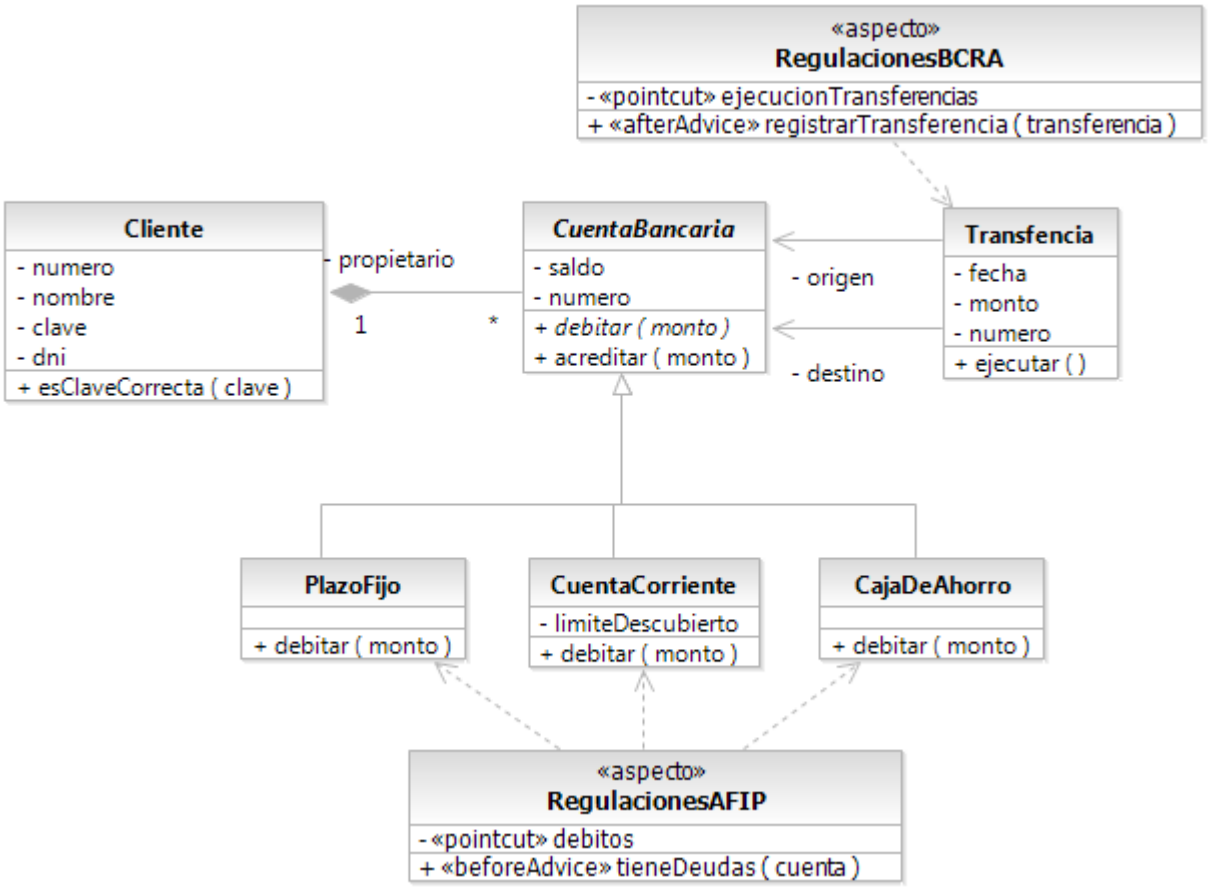

Figura 48: solución con aspectos

Generalizando las reglas de negocio

Elevando un poco el nivel de abstracción, podemos clasificar las reglas de negocio vistas en dos grupos muy genéricos.

- Validaciones, que actúan como pre condiciones, modificando el flujo de ejecución. A nivel código fuente rememoran la problemática del diseño por contratos tratada en el capítulo anterior y al igual que ese caso, AOP resulta una excelente alternativa de implementación. La regla 1 pertenece claramente a este grupo.
- Notificaciones, que actúan como disparadores de otras reglas del negocio, pudiendo incluso coordinar la interacción entre ciertos objetos de negocio y aplicaciones externas. Si bien en ocasiones se implementan utilizando patrones como *Observador*, generalmente AOP constituye una mejor alternativa. Dentro de este grupo se encuentra la regla 2.

Obviamente, esta clasificación es extremadamente genérica, pero justamente nos da una idea del tipo de reglas de negocio que pueden implementarse con AOP.

El rol de los aspectos

En las soluciones previamente planteadas, los aspectos son los encargados de modelar cierta

lógica del negocio. Un enfoque diferente sería plantear a los aspectos como simples coordinadores, abstrayendolos del negocio y dejando en manos de otros objetos la aplicación de la lógica de negocio correspondiente.

Siguiendo este último criterio, otra posible solución para la regla 2, sería utilizar la versión AOP del patrón *Observador*, delegando en un aspecto, la administración de los observadores y la notificación de cambios de estado. De esta forma, la clase *Transacción* permanecería inalterada, el aspecto debería notificar al objeto *ProxyBancoCentral* de cualquier transacción y finalmente dicho objeto debería distinguir las transacciones de su interés y notificar al Banco Central cuando correspondiera.

Esta estrategia de uso de aspectos, es similar a la planteada en el capítulo anterior, donde los aspectos actúan como simples coordinadores que se encargan de detectar puntos interesantes en la ejecución de la aplicación y delegan las decisiones en otros objetos.

Pasando en limpio, podemos decir que a la hora de diseñar un aplicación con AOP, los aspectos pueden jugar dos roles.

- ***Aspectos coordinadores o integradores***: los aspectos son utilizados como una herramienta para integrar incumbencias transversales que son modeladas por clases. Estos aspectos en la mayoría de los casos sólo tienen pointcuts y advices. Los advices sólo tienen una lógica limitada para delegar el trabajo en las clases que modelan las incumbencias de la aplicación.
- ***Aspectos inteligentes***: los aspectos son utilizados para modelar incumbencias transversales, ya sean de negocio o técnicas. Además de los pointcuts y advices, los aspectos tienen métodos y atributos relacionados a las incumbencias que modelan. Cada advice del aspecto tiene lógica de la incumbencia que modela el aspecto.

La decisión de que estrategia utilizar depende principalmente del tipo de la incumbencia a modelar.

Generalmente al modelar incumbencias técnicas se utilizan los aspectos como integradores de infraestructura.

Por us parte, al modelar incumbencias del negocio, el criterio no es tan claro. Por ejemplo, al modelar reglas de negocio utilizando un motor de reglas, los aspectos tendrán un rol de integradores. Pero de no utilizar un motor de reglas, tendremos que analizar de que tipo de regla se trate.

Llevando esta discusión a otro plano, podríamos decir que el uso de herramientas AOP de entretejido dinámico predispone al uso de aspectos coordinadores, mientras que las herramientas de entretejido estático son más afines a los aspectos inteligentes.

Resumiendo...

Toda AE tiene reglas particulares que muchas veces suelen implementarse directamente en la capa de dominio, lo cual en ocasiones puede funcionar bien, pero a cambio le quitan claridad al dominio y restan flexibilidad a dichas reglas, ya que las reglas suelen ser mucho más volátiles que la lógica del dominio. El modelar dichas reglas con aspectos puede ayudar a lograr una mejor separación de incumbencias brindando una mayor flexibilidad a dichas reglas y evitando la duplicación de código.

Las reglas candidatas a modelarse como aspectos, son aquellas que implican la ejecución de alguna lógica adicional o secundaria desde el punto de vista del dominio.

Como ya ha sido mencionado por algunos autores, la utilización de aspectos para modelar incumbencias del negocio, requiere ciertas modificaciones en las actividades del proceso de desarrollo.

Capitulo VII: La foto completa

En los dos capítulos anteriores vimos como implementar las incumbencias transversales de las AE utilizando AOP. En el presente capítulo, veremos la foto completa del uso de AOP en las EA, exponiendo un caso de estudio sobre la plataforma .NET. Para cumplir con este cometido comenzaremos presentado una arquitectura enterprise AOP-compatible para la plataforma .NET.

Luego, tomando una vez más el ejemplo del Banco X, plantearemos una solución tradicional y otra basada en AOP y analizaremos sus diferencias.

Arquitectura Enterprise AOP-Compatible

En este punto no hemos de inventar la rueda nuevamente, ya que la arquitectura base descripta en el capítulo 4 representa un buen punto de partida. Por ello, simplemente realizaremos algunas modificaciones de cara a facilitar la inclusión de aspectos.

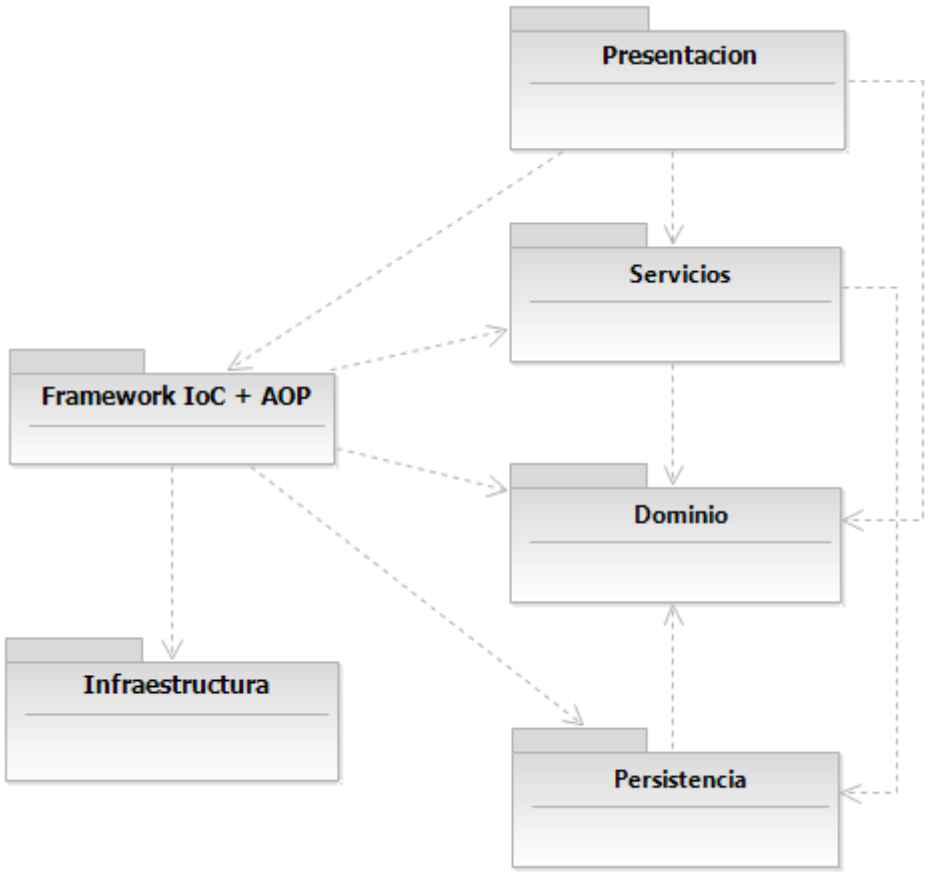

Figura 49: arquitectura enteprise AOP-Compatible

La figura 49 muestra la arquitectura AOP propuesta, que a diferencia de la arquitectura base del

capítulo 4, sólo tiene un componente adicional, el Frameowork IoC[10] + AOP. Dicho componente, ademas de facilitar la resolución de dependencias a partir de la técnica de inversión de control, permite integrar transparentemente las bibliotecas y funcionalidades de la infraestructura a partir de sus posibilidades AOP.

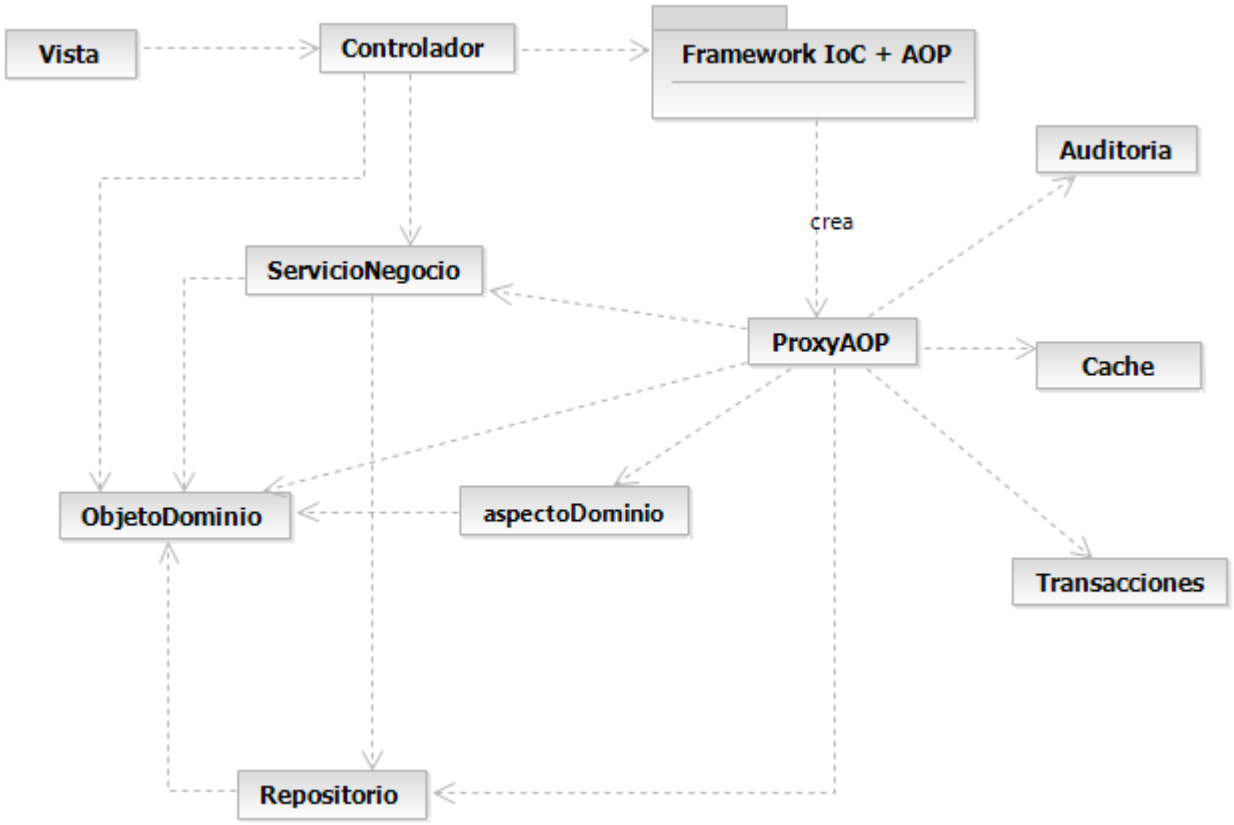

Figura 50: Relaciones entre clases y aspectos de la arquitectura base

Algunos puntos a tener en cuenta sobre esta arquitectura son lo siguientes.

- Tal como está planteada, esta arquitectura asume que la todas las capas corren en un mismo proceso. De ser necesario distribuir la capas en distintos procesos sería necesario hacer algunos ajustes.
- Sólo la capa de presentación tiene conocimiento explícito de Framework IoC.
- Si bien se ha tomado como base la plataforma .NET esta arquitectura aplica perfectamente a Java.
- Si bien la aplicación a desarrollar tiene una presentación web, la arquitectura también aplica para el caso de una presentación de escritorio.

Herramientas AOP seleccionadas

Como lo hemos planteado a lo largo de esta tesis, para implementar una aplicación basada en esta

10 IoC: inversion of control

arquitectura de referencia son necesarias 2 herramientas AOP, una de entretejido dinámico, basada en proxies dinámicos (para implementar las incumbencias técnicas) y otra de entretejido estático.

Antes de entrar en detalle sobre las herramientas elegidas, es necesario destacar que el objetivo de este trabajo está centrado la utilización del paradigma AOP y no en las herramientas AOP. Esta aclaración es muy necesaria ya que en la actualidad existe cierta distancia entre lo propuesto por el paradigma y lo que ofrecen algunas herramientas. Es en parte por ello que todo lo expuesto hasta el momento, fue presentado utilizando una notación genérica de una herramienta AOP ficticia.

En cuanto a la herramienta de entretejido dinámico, se ha optado por Spring AOP, debido principalmente a su amplio soporte al paradigma AOP y a su biblioteca de aspectos, aunque hay algunos otros hechos interesantes para destacar como por ejemplo su estabilidad y su orientación a las aplicaciones enterprise.

En cuanto a la herramienta de entretejido estático, la cuestión es más compleja, ya que este tipo de herramientas son más escasas. En el caso de .NET ocurre que varios de los proyectos relacionados a herramientas de este tipo, han sido abandonados o aún no cuentan con versiones estables. Esta situación ha acotado el espectro de evaluación a unas pocas herramientas, de las que finalmente resulto seleccionada Aspect.NET.

Si bien Aspect.NET no implementa la totalidad del paradigma AOP (carece de declaraciones de intertipos entre otras cosas), ofrece una serie de características muy interesantes que la ubican por encima de las demás herramientas de su tipo. Entre estas características cabe mencionar su estabilidad, su plan de evolución y su integración con el principal entorno integrado de desarrollo .NET (Visual Studio). Este último punto resulta de una gran importancia, considerando que se planea utilizar la herramienta para implementar aspectos del negocio.

Caso de estudio: Banco X Bis

Al caso del Banco X presentado parcialmente a lo largo de este trabajo, lo presentaremos ahora de forma completa para poder así aplicar todo lo desarrollado en los capítulos anteriores.

Casos de uso y reglas de negocio

Tal como lo muestra la figura 51, la aplicación ofrece las siguientes 4 funcionalidades de cara al cliente.

- Transferir fondos: permite al cliente realizar un movimiento de fondos entre dos cuentas bancarias de su propiedad. El monto no puede ser negativo como tampoco pueden transferirse fondos de una cuenta a si misma, ni tampoco se deben admitir transferencias entre cuentas de distinta moneda.
- Pagar prestación: permite al cliente pagar las prestaciones brindadas por un conjunto de

empresas con las que el Banco X tiene acuerdos.

- Consultar movimientos: el cliente puede consultar en la aplicación los movimientos de sus cuentas.
- Consultar saldo: el cliente puede consultar los saldos de sus cuentas bancarias.

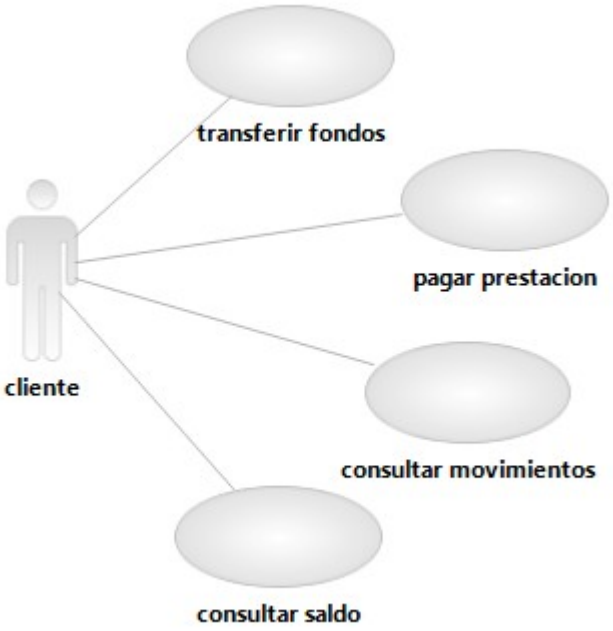

Figura 51: Casos de uso del Banco X

Como es de esperar en aplicaciones de este tipo, para realizar cualquier operación es necesario que el usuario acredite su identidad ante la aplicación ingresando su DNI y clave de acceso.

Más allá de estas funcionalidades perceptibles directamente por el usuario, hay ciertas reglas de negocio que la aplicación debe implementar, las cuales se detallan a continuación.

1. Toda operación que supere cierto monto deberá ser notificada al Banco Central.
2. No debe ser posible realizar operaciones sobre cuentas cuyos titulares tengan deudas con las Afip.
3. Ante el tercer intento consecutivo de ingreso fallido al sistema por parte de un cliente, dicho cliente deberá quedar bloqueado de forma permanente sin poder acceder al sistema.
4. Existe un límite diario del monto total por el cual un cliente puede realizar operaciones.
5. La aplicación debe dejar constancia de todas las acciones realizadas por el cliente.

El siguiente diagrama muestra las clases del dominio de este problema, que constituyen la base de cualquier aplicación que ataque la problemática planteada. Dicho diagrama no contempla las reglas particulares del Banco X.

Más allá de las funcionalidades y reglas mencionadas, es de esperar que cualquier solución

propuesta contemple las incumbencia técnicas: manejo de excepciones, transacciones autorización y auditoría.

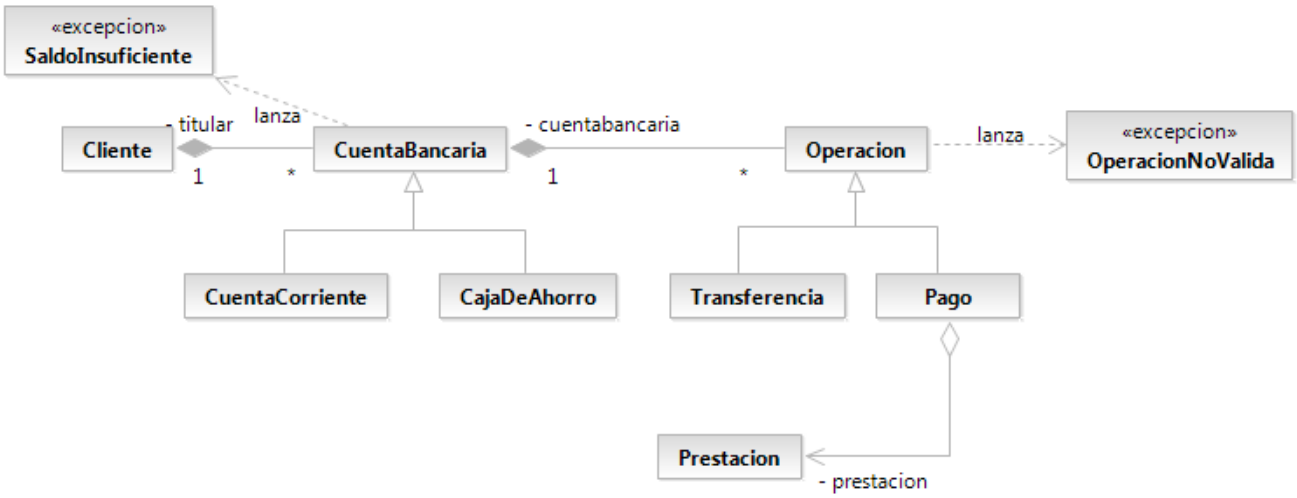

Figura 52: Dominio base del Banco X

Solución tradicional (sin AOP)

Partiendo del dominio base representado en el diagrama de la figura 52, trabajaremos ahora sobre las reglas 1 a 5 (particulares del Banco X) para encontrar una primera solución al Banco X, sin utilizar AOP.

Las reglas 1 y 2, ya las hemos analizado en el capítulo anterior. En este caso se ha optado por implementar ambas reglas en la capa de dominio, ya que ponerlas en la capa de servicios sería conceptualmente incorrecto porque la capa de servicios sólo debe coordinar objetos de dominio. Es la clase *Operación* la encargada de interactuar con las interfaces de Afip y del Banco Central para llevar a cabo la reglas 1 y 2. La desventaja de esta solución radica en que el código relacionado a estas reglas quedará esparcido por todas las clases de la jerarquía operación.

La regla 3, es consecuencia de políticas de seguridad y resulta muy común en aplicaciones de este tipo. Si bien esta regla tiene cierta relación con la clase *Cliente*, la misma excede las responsabilidades de dicha clase. Es por ello que hemos creado una nueva clase *ControladorAccesos* encargada de esta nueva responsabilidad. Al mismo tiempo agregamos a la clase *Cliente* un atributo para indicar si el cliente está bloqueado. Por último decidimos dejar en manos del servicio de autenticación la coordinación entre las clases *Cliente* y *ControladorAccesos*.

La regla 4 pide limitar el monto diario de operaciones del cliente. No resulta evidente como modelar esta regla. La estrategia adoptada en este caso, fue asignar dicha responsabilidad a la clase operación, de manera que a la hora de ejecutar cualquier operación, sea la misma instancia de la clase operación la encargada de hacer cumplir esta regla. Esta solución tiene la misma contra que la solución adoptada para las reglas 1 y 2.

La regla 5 nos pide dejar constancia de todas las acciones realizadas por el cliente, esto claramente es auditoria. Dado que toda acción que puede realizar un cliente está expresada por un método de la capa de servicios, implementaremos la auditoria a nivel de los servicios. La misma estrategia adoptaremos para las incumbencias técnicas y de calidad particularmente manejo de transacciones, caché y seguridad, ya que implementarlas en la capa de dominio podría resultar bastante más complejo y al mismo tiempo impondría una dependencia del dominio con la infraestructura. Hay que destacar que el agregado de nuevos atributos técnicos y de calidad puede requerir de la modificación del código de los servicios.

```
public override void Ejecutar()
{
    if (!Origen.Moneda.Equals(Destino.Moneda))
        throw new OperacionNoValidaException("Monedas distintas");

    if (_afip.TieneDeuda(Destino.Titular) || _afip.TieneDeuda(Origen.Titular))
        throw new ClienteSuspendidoPorDeudasException();

    VerificarLimiteDiario(_origen.Titular, Monto);

    Origen.Debitar(Monto);
    Destino.Acreditar(Monto);

    _bancoCentral.NotificarOperación(this);

    FechaHora = DateTime.Now;
    Numero = Secuencia;
}
```

Figura 53: Fragmento de la clase Transferencia con la implementación de las reglas 1,2 y 4.

Respecto al diagrama de la figura 55, cabe aclarar que si bien se muestra la dependencia entre la clase operación y las interfaces de Afip y BancoCentral, en realidad la dependencia se extiende a cada una de las clases descendientes de operación, ya que cada subclase de operación deberá interactuar con las mencionadas interfaces al redefinir el método abstracto Ejecutar. Esta situacion puede apreciarse en el fragmento de código de la clase *Transferencia* en la figura 53, donde las líneas resaltadas estarán presentes en la clase *Pago* como también en cualquier otra clase de operación que se pretenda agregar.

```
public Cliente ValidarCliente(string dni, string clave)
{
    Cliente cliente = null;
    try
    {
        cliente = RepositorioClientes.ObtenerPorDni(dni);
        if (cliente != null)
        {
            if (cliente.EstaBloqueado)
                throw new ClienteBloqueadoException();

            if (cliente.EsClaveCorrecta(clave))
            {
                _logger.Info("Autenticación cuenta:" + dni);
                _controladorAccesos.RegistrarAccesoExitoso(cliente);
                return cliente;
            }
            else
            {
                _logger.Info("Intento fallido de autenticación cuenta:" +dni);
                _controladorAccesos.RegistrarIntentoAccesoFallido(cliente);
            }
        }
        _logger.Info("Intento fallido de autenticación cuenta:" + dni);
    }
    catch (ClienteBloqueadoException ex)
    {
        _logger.Info("Cliente bloqueado");
        throw;
    }
    catch (Exception ex)
    {
        _logger.Error("Operacion fallida", ex);
        throw new OperacionFallidaException(ex);
    }
    return null;
}
```

Figura 54: Fragmento del servicio de autenticación con la implementación de las reglas 3 y 5

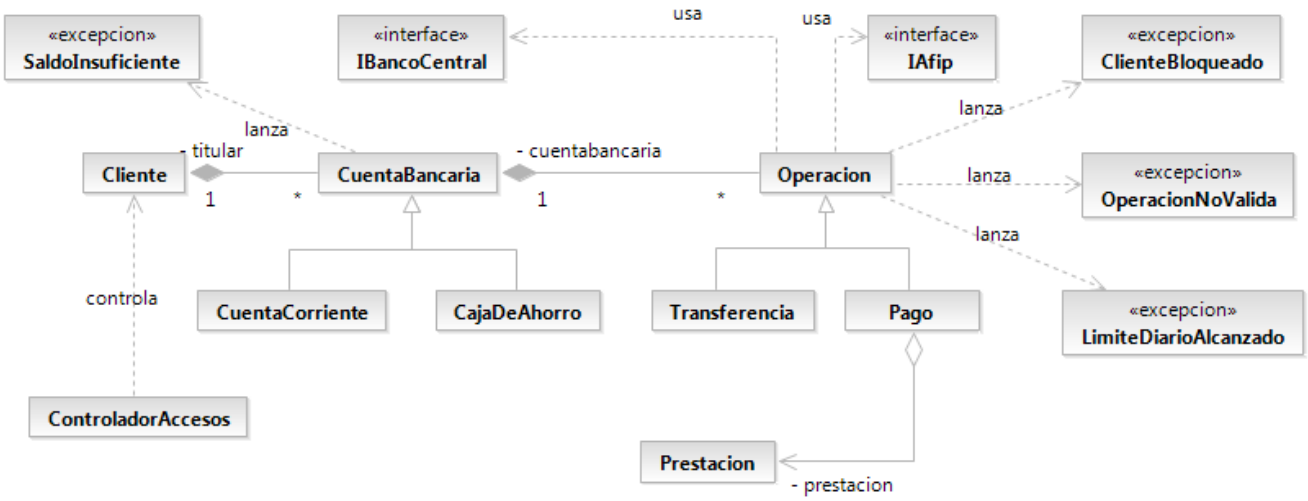

Figura 55: Dominio extendido con reglas del Banco X

Los siguientes diagramas muestran las relaciones de las tres clases de la capa de servicios. Claramente puede observarse que además de la coordinación de los objetos de dominio, los servicios también se encargan de la interacción con la infraestructura, lo cual no sólo establece una dependencia entre los servicios y las incumbencias técnicas sino que además genera código duplicado.

Figura 56: Dependencias del servicio de autenticación

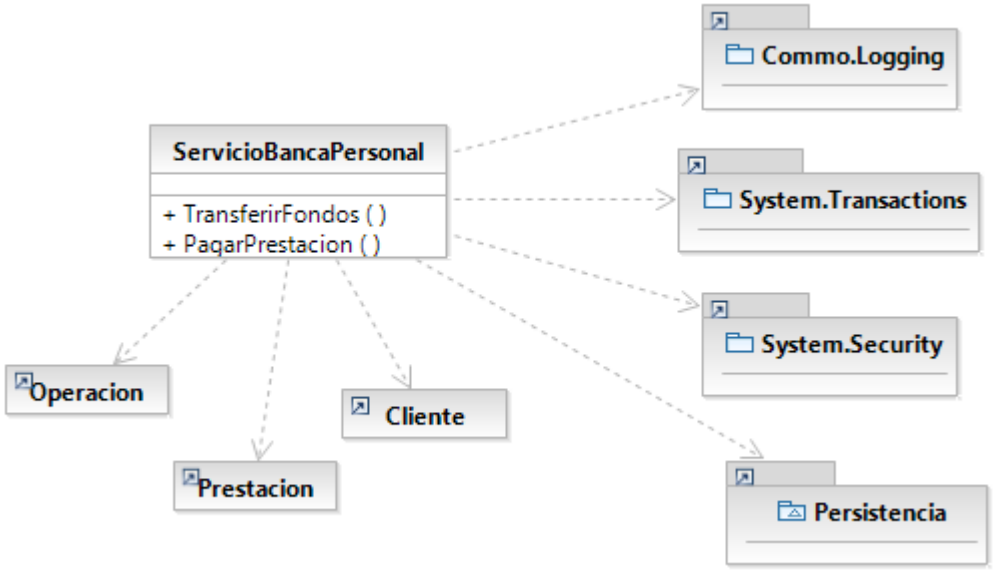

Figura 57: Dependencias del servicio de banca personal

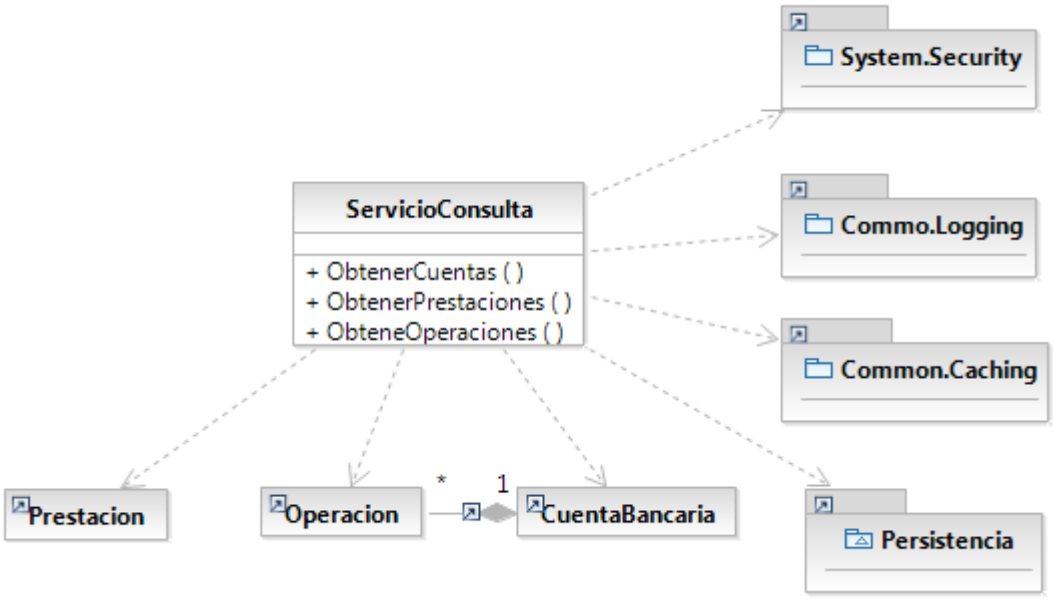

Figura 58: Dependencias del Servicio de Consulta

Solución AOP

La solución AOP, al igual que la solución tradicional, parte del dominio base, sobre este modela las reglas 1 a 5.

En cuanto a las reglas 1 y 2, la estrategia a utilizar es la misma que en el capítulo anterior: crear un aspecto por cada una de estas reglas. Dado que estas dos reglas son reglas del negocio, los aspectos encargados de modelarlas los implementaremos con Aspect.NET, la herramienta AOP de entretejido

estático definida por nuestra arquitectura. A continuación se puede apreciar un fragmento de código de estos aspectos, en este caso escritos con la sintaxis de Aspect.NET.

```
[AspectDescription("Regulaciones bancarias decretadas por la Administracion
Federal de Ingresos Publicos")]
public class RegulacionesAfip : Aspect
{
    [AspectAction("%before %call *Operacion.Ejecutar(..)")]
    static public void VerificarDeudas()
    {
        Operacion operacion = TargetObject as Operacion;

        if (TieneDeuda(operacion.Actor))
            throw new CuentaSuspendidaException();
    }

    static public bool TieneDeuda(Cliente cliente) {....}
}
```

Figura 59: Implementación AOP de la regla 2

```
[AspectDescription("Regulaciones bancarias decretadas por el Banco Central de
la Republica Argentina")]
public class RegulacionesBancoCentral : Aspect
{
    [AspectAction("%after %call *Operacion.Ejecutar(..)")]
    static public void NotificarOperación()
    {
        Operacion operacion = TargetObject as Operacion;

        if (operacion.Monto > _pisoNotificacion)
        {
            RegistrarOperacion(operacion);
        }
    }

    private static void RegistrarOperacion(Operacion operacion){....}
}
```

Figura 60: Implementación AOP de la regla 1

La regla 3, si bien tiene cierta relación con la seguridad del sistema, no tiene una naturaleza transversal como suelen tener las incumbencias relacionadas a la seguridad. La estrategia utilizada en la solución tradicional, puede parecer conveniente a algunos lectores, pero también podría argumentarse que conceptual no es del todo correcta, ya que deja en manos del servicio la aplicación de regla. Al mismo tiempo, como ya hemos mencionado, esta regla es particular del negocio del Banco X y por ende no sería tampoco correcto modelarla en el dominio. Es por esto que hechando mano de AOP, proponemos modelar esta regla con un aspecto *ControladorDeAccesos* cuyo código se expone en la figura 61.

```
[AspectDescription("Verifica los accesos del cliente al sistema")]
public class ControladorDeAccesos : Aspect
{
    private static void Registar(Cliente cliente, int numeroDeIntento) {...}

    [AspectAction("%instead %call *Cliente.EsClaveCorrecta(string) &&
                        %args(arg[0])")]
    static public bool ValidarIntentoDeAcceso(string clave)
    {
        Cliente cliente = TargetObject as Cliente;

        if (cliente.EstaBloqueado)
            throw new ClienteBloqueadoException();

        bool intentoCorrecto = cliente.EsClaveCorrecta(clave);

        if (intentoCorrecto)
             Registar(cliente, 0);

        else
         {
            int intentos = _registro[cliente];
             intentos++;
             Registar(cliente, intentos);
            if (intentos.Equals(3))
                 cliente.Bloquear();
         }
        return intentoCorrecto;
    }
}
```

Figura 61: Implementación AOP de la regla 3

La regla 4, nos pide limitar el monto diario de operaciones del cliente. La estretegia a seguir en este caso es la misma que la utilizada para las reglas 1 y 2, crear un aspecto encargado de realizar esta verificación.

Dado que tanto las reglas 1 y 4 hacen referencia a verificaciones previas a la ejecución de una operación, se podría haber optado por una estrategia distinta, agrupando todas las verificaciones previas (precondiciones) en un solo aspecto que defina un advice por cada precondición. De forma análoga podríamos plantear otro aspecto para concentrar las notificaciones o pos condiciones. Si bien ambas estrategias permiten una separación entre las incumbencias del dominio y las del negocio, esta última estrategia mezcla en un mismo aspecto distintas reglas del negocio.

Finalmente la regla 5 hace referencia a una de las incumbencias transversales más reconocidas (auditoría), pero a pesar de que el uso de AOP es claramente conveniente, se plantea el dilema de sobre qué objeto aplicar el aspecto. Dado que se busca dejar constancia de todas las acciones realizadas por el cliente en la aplicación y considerando que todas las acciones posibles están disponibles mediante la capa de servicios, resulta suficiente aplicar el aspecto de auditoria sobre los objetos de la capa de servicios. Al mismo tiempo como esta regla no hace al negocio en sí, sino que resulta como un

requisito no funcional, la implementaremos con la herramienta AOP de entretejido dinámico.

```
[AspectDescription("Verifica que las operaciones realizadas por un cliente en
un dia no excedan un cierto límite prestablecido")]
public class ControladorDeLimiteDiario : Aspect
{
    [AspectAction("%before %call *Operacion.Ejecutar(..)")]
    static public void VerificarLimiteDiario()
    {
        Operacion operacion = TargetObject as Operacion;

        decimal contador = 0;

        foreach (CuentaBancaria cuenta in operacion.Actor.Cuentas)
        {
            IList<Operacion> operaciones =
                  repositorioOperaciones.ObtenerOperaciones(cuenta.Numero);

            foreach (Operacion operacionRealizada in operaciones)
            {
                if
                  (operacionRealizada.FechaHora.Date.Equals(DateTime.Now.Date))
                     contador += operacionRealizada.Monto;
            }
        }
        if (operacion.Monto + contador >= _limiteDiario)
            throw new LimiteDiarioAlcanzadoException();
    }
}
```

Figura 62: Implementación AOPde la regla 4

```
public class AuditoriaAdvice : IMethodInterceptor
{
    private ILog _logger = LogManager.GetLogger("Auditoria");

    public object Invoke(IMethodInvocation invocation)
    {
        _logger.Info(GetMensajePreInvocacion(invocation));
        object resultado = invocation.Proceed();
        _logger.Info(GetMensajePosInvocacion(invocation));
        return resultado;
    }

    protected string GetMensajePreInvocacion(IMethodInvocation invocation){..}
    protected string GetMensajePosInvocacion(IMethodInvocation invocation){..}
}
```

Figura 63: Implementación AOP de la regla 5

El diagrama de la figura 64 muestra los objetos y aspectos que constituyen la capa de negocio la solución AOP para la problemática planteada por el Banco X. El diagrama de la figura 65 muestra los objetos de la capa de servicios y su relación con los aspectos de infraestructura.

En cuanto a las incumbencias técnicas, las mismas se implementaron a nivel de la capa de

servicios y se utilizó para su implementación los aspectos mostrados en el capítulo 5. Hay que destacar que el agregado de nuevas incumbencias técnicas no requiere de la modificación del código existente de la aplicación, sino que simplemente bastaría con desarrollar los aspectos correspondientes y agregarlos en la configuración de la aplicación.

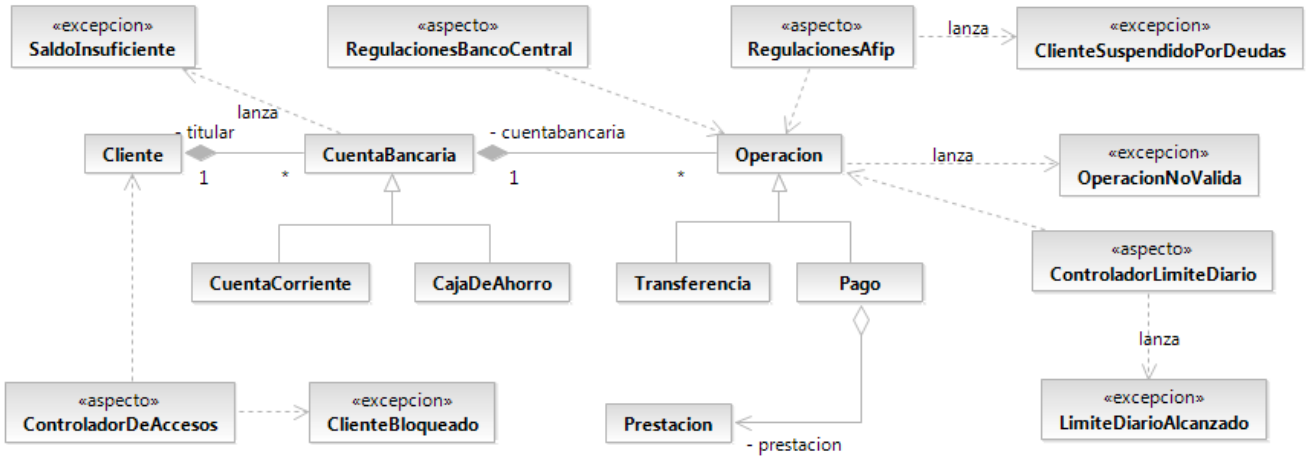

Figura 64: clases del domino extendidas con aspectos del negocio del Banco X

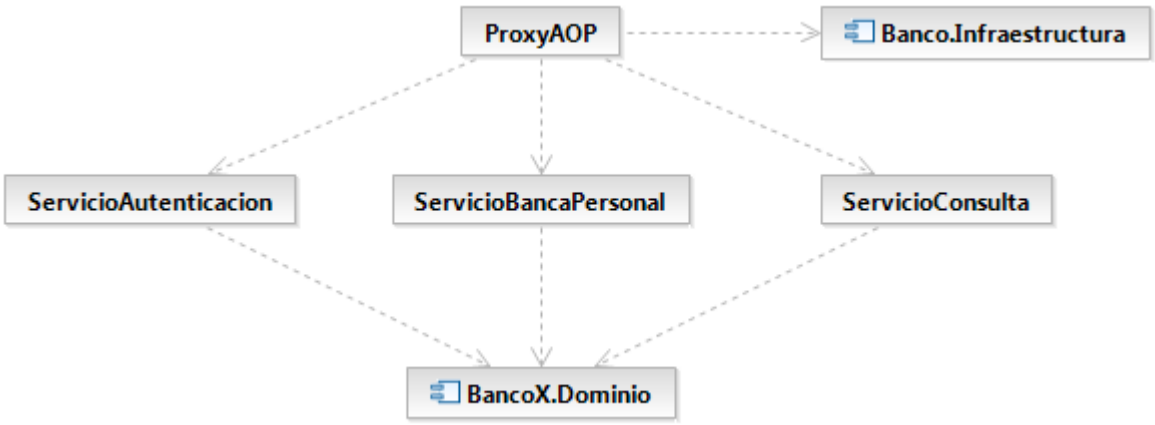

Figura 65: Clases de la capa de servicios con aspectos de infraestructura

Hay algunas cuestiones relacionadas a la implementación de la solución AOP que vale la pena mencionar.

La solución AOP implementada para el Banco X consta de 6 assemblies[11] representados en la figura 66 junto con sus dependencias.

11 El assembly es la unidad de despliegue de código de máquina virtual de la plataforma .NET

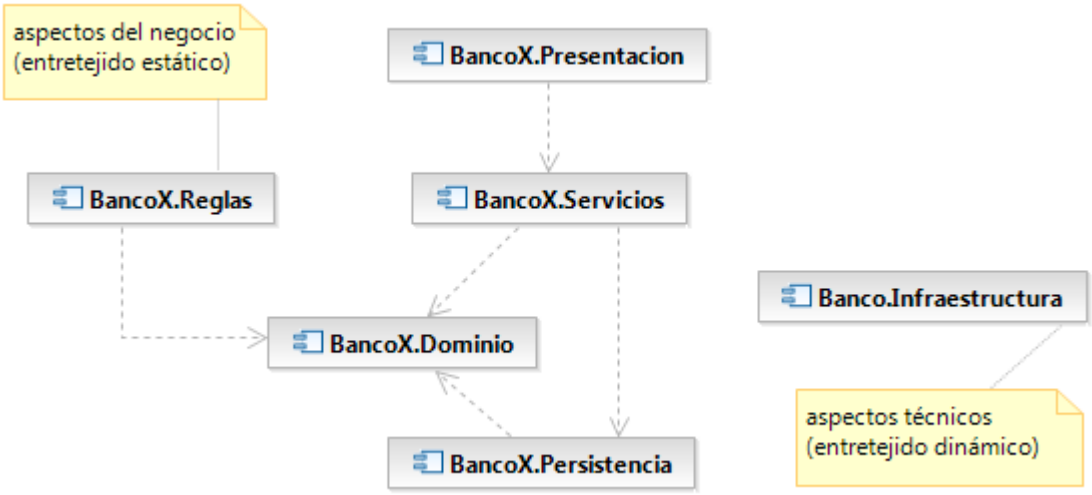

Figura 66: Assemblies de la solución AOP previo al entretejido

Aspect.NET funciona como un pos compilador (o procesador de assemblies), en el sentido que una vez compilados los assemblies con las clases y los aspectos, Aspect.NET se encarga de procesar el assembly que contiene las clases entretejiendo los aspectos en los puntos definidos por los pointcuts. En el caso de nuestra solución, los aspectos implementados con Aspect.NET se encuentran en el assembly *BancoX.Reglas*. Al analizar sobre que assembly se hará el entretejido de los aspectos, a primera vista una podría tentarse de decir *BancoX.Dominio*, pero este no es nuestro caso. Esto se debe a que la versión utilizada de Aspect.NET sólo soporta poincuts de tipo llamada (call) y si miramos la definición de pointcuts de nuestros aspectos, veremos que sus pointcuts hacen referencia a las llamadas a objetos del dominio. Si bien las llamadas a objetos del dominio podrían estar en varios assemblies, en nuestro caso particular, las llamadas a objetos de dominio que nos interesan se encuentran en el assembly *BancoX.Servicios* y es justamente sobre ese assembly que se realizará el entretejido. Como resultado del entretejido realizado se obtiene un nueva versión del assembly original con los aspectos incluidos, lo cual cambia la relación de los assemblies de nuestra solución.

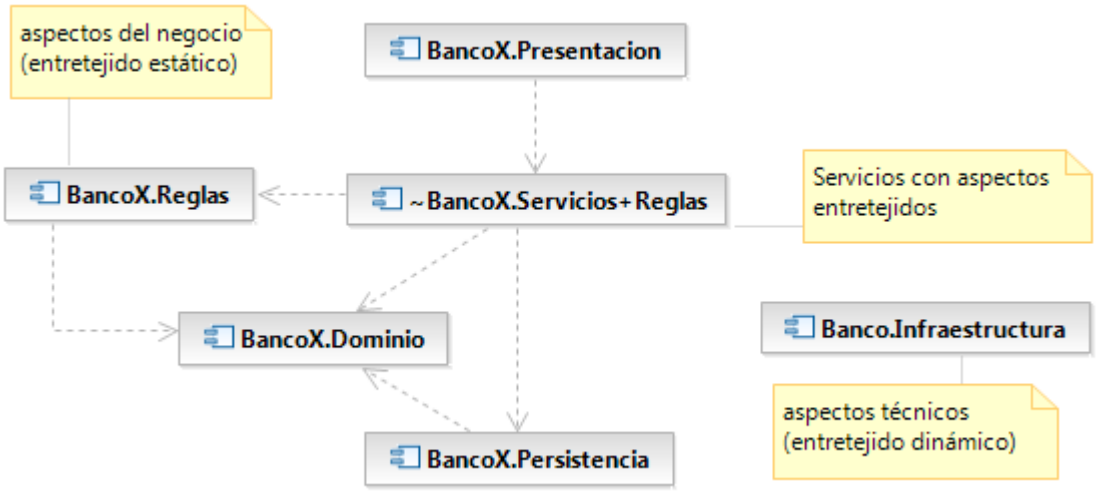

Figura 67: ssemblies de la solución AOP luego del entretejido

Un punto para destacar de este último diagrama respecto al anterior es que existe una nueva dependencia entre el assembly de servicios y el de las reglas, lo cual es consecuencia del proceso de entretejido entre servicios y aspectos.

Otro punto destacable de la solución AOP es que no existe ninguna dependencia con el assembly de infraestructura que contiene la implementación de los aspectos técnicos. Esto se debe a que los aspectos técnicos suelen ser lo suficientemente genéricos como para aplicar a diferentes soluciones y al mismo tiempo resulta que se encuentran implementados con una herramienta AOP de entretejido dinámico que permite la definición de los pointcuts a partir del archivo de configuración de la aplicación. Cabe mencionar que no siempre que se utilice AOP para la implementación de los aspectos técnicos se logrará este nivel de desacoplamiento ya que en ocasiones en necesario que los aspectos técnicos tengan cierto conocimiento de los componentes sobre los que se espera que actúen.

Comparación de soluciones

Habiendo expuesto ambas soluciones resulta interesante comparar algunos fragmentos de código de cada una de ellas. La figuras 68 y 69 muestran el código del método *Ejecutar* de la clase *Pago* de la solución tradicional y de la solución AOP respectivamente.

```
public override void Ejecutar()
{
    if (Monto <= 0)
        throw new OperacionNoValidaException("El monto no puede ser
negativo");

    if (_afip.TieneDeuda(_cuenta.Titular))
        throw new CuentaSuspendidaException();

    VerificarLimiteDiario(_cuenta.Titular, Monto);

    _cuenta.Debitar(Monto);
    numero = _prestacion.Pagar(_codigo, Monto);
    FechaHora = DateTime.Now;

    _bancoCentral.NotificarOperación(this);
}
```

Figura 68: Método Ejecutar de la clase Pago de la solución tradicional

La líneas resaltadas en el código de la solución tradicional se encuentran duplicadas en todas las clases de la jerarquía *Operación*, lo cual expresa cierta transversalidad.

Por su parte el código de la solución AOP remueve dichas líneas encapsulando las incumbencias expresadas por las mismas en los correspondientes aspectos, cuyo código se vió en las figuras 59,60 y 62.

```
public override void Ejecutar()
{
    if (Monto <= 0)
        throw new OperacionNoValidaException("El monto no puede ser
negativo");

    _cuenta.Debitar(Monto);
    _prestacion.Pagar(_codigo, Monto);

    FechaHora = DateTime.Now;
}
```

Figura 69: Método Ejecutar de la clase Pago de la solución AOP

Una situación similar ocurre si se compara el código de las clases de la capa de servicios de las figuras 70 y 71. Las líneas resaltadas de la figura 70 son consecuencia de las incumbencias transversales: manejo de excepciones, auditoría y transacciones. Por su parte la solución AOP modela las mencionadas incumbencias transversales con aspectos, lo cual simplifica notablemente el código de la figura 70 removiendo las líneas resaltadas.

```
public Pago PagarPrestacion(string numeroCuenta, string idPrestacion, string
codigo, decimal monto)
{
    if (Thread.CurrentPrincipal == null)
        throw new SecurityException("Usuario no autenticado");

    Pago pago = null;

    CuentaBancaria cuenta = _repositorioCuentas.ObtenerCuenta(numeroCuenta);

    if (cuenta == null)
        throw new CuentaNoValidaException();

    Prestacion prestacion =
_repositorioPrestaciones.ObtenerPrestacion(idPrestacion);

    if (prestacion == null)
        throw new OperacionNoValidaException("Prestacion inexistente.");

    try
    {
        using (TransactionScope scope = new TransactionScope())
        {
            pago = new Pago(cuenta, prestacion, codigo, monto);
            pago.RepositorioOperaciones = _repositorioOperaciones;
            pago.BancoCentral = _bancoCentral;
            pago.Afip = _afip;
            pago.Ejecutar();

            _repositorioOperaciones.RegistrarOperacion(pago);
            scope.Complete();
            _logger.Info("Pago completado." + pago.ToString());
        }
    }
    catch (LimiteDiarioAlcanzadoException ex)
    {
        _logger.Info("Intento de operacion no valida", ex);
        throw;
    }
    catch (CuentaNoValidaException ex)
    {
        _logger.Info("Intento de operacion no valida", ex);
        throw;
    }
    catch (OperacionNoValidaException ex)
    {
        _logger.Info("Intento de operacion no valida", ex);
        throw;
    }
    catch (Exception ex)
    {
        _logger.Error("Intento de operacion fallido", ex);
        throw new OperacionFallidaException;
    }
    return pago;
}
```

Figura 70: Método PagarPrestacion de la clase ServicioBancaPersonal de la solución tradicional

```
public Pago PagarPrestacion(string numeroCuenta, string idPrestacion, string
codigo, decimal monto)
{
    Pago pago = null;

    CuentaBancaria cuenta = _repositorioCuentas.ObtenerCuenta(numeroCuenta);

    if (cuenta == null)
        throw new CuentaNoValidaException();

    Prestacion prestacion =
_repositorioPrestaciones.ObtenerPrestacion(idPrestacion);

    if (prestacion == null)
        throw new OperacionNoValidaException("Prestacion inexistente.");

    pago = new Pago(cuenta, prestacion, codigo, monto);
    pago.Ejecutar();

    _repositorioOperaciones.RegistrarOperacion(pago);

    return pago;
}
```

Figura 71: Método PagarPrestacion de la clase ServicioBancaPersonal de la solución AOP

Cabe destacar que en el caso de la solución AOP, podría agregarse la verificación de ciertos invariantes de forma transparente para la aplicación, a partir de combinar el uso de AOP con una herramienta de diseño por contractos.

Capitulo VIII: Conclusiones

El objetivo principal de este trabajo era analizar la conveniencia de la utilización de AOP en el desarrollo de aplicaciones enterprise, proponiendo una solución AOP a aquellas problemáticas en las que el uso de AOP resulte beneficioso. De cara a este objetivo se identificaron dos tipos de incumbencias claramente diferenciadas en las AE; las del negocio y las técnicas- y se analizó el uso de AOP en relación a cada uno de estos tipos de incumbencias, a partir de lo cual surgieron los siguientes cuestiones para destacar.

- Gran parte de las incumbencias técnicas cuentan con una naturaleza transversal que las convierte en excelentes candidatos a ser implementadas con AOP.
- El uso de AOP permite una clara separación de incumbencias, permitiendo que las incumbencias del negocio puedan implementarse independientemente de las incumbencia técnicas, facilitando así el foco en el negocio.
- La implementación de las incumbencias técnicas utilizando una herramientas de entretejido dinámico, permite el agregado de incumbencias técnicas de forma prácticamente trasparente para la aplicación. Al mismo tiempo el uso de una herramienta de entretejido estático para la implementación de los aspectos del negocio asegura que el cumplimiento de las reglas del negocio, evitando que la aplicación de las mismas dependa de la configuración de la aplicación como ocurre en el caso de las herramientas de entretejido dinámico.
- Las incumbencias técnicas implementadas con aspectos resultan altamente reutilizables en distintas aplicaciones. Asimismo queda claro que la implementación de incumbencias técnicas utilizando AOP requiere de un esfuerzo mucho menor que una implementación tradicional.
- Tal como en la programación orientada a objetos el programador puede decidir que un método o una clase no puedan ser extendidos o redefinidos (en .NET mediante la clausula *sealed* y en Java mediante el la clausula *final*), resulta necesario en la programación orientada a aspectos poder especificar que no se le agreguen aspectos a una clase dada. Si bien esta problemática no se vio en el ejemplo expuesto, la misma fue detectada durante la implementación de la aplicación de referencia. Esta necesidad se ve acentuada por el hecho de que a partir de la clara separación de incumbencias es posible que sean distintos programadores los que se encarguen de las incumbencias técnicas y de las incumbencias del negocio.

Como objetivo secundario, se propuso el desarrollo de una aplicación de referencia sobre la plataforma .NET. Este objetivo se cumplió satisfactoriamente con la implementación del caso de estudio y permitiendo durante su desarrollo detectar algunas cuestiones interesantes.

- Las herramientas AOP para la plataforma .NET están un paso atrás respecto de las herramientas para la plataforma Java.
- Es necesario que las herramientas puedan integrarse con los entornos de desarrollo para brindar soporte, principalmente, a dos cuestiones: depuración y trazabilidad del código fuente. Respecto de este último punto, resulta indispensable que al visualizar el código, el entorno de desarrollo indique aquellas porciones de código que se encuentran afectadas por un aspecto (esta funcionalidad existe en Eclipse para el caso de AspectJ, pero en el caso de .NET no hay nada similar).
- Tal como se esperaba, la solución AOP permitió eliminar gran cantidad de código transversal, mejorando la modularización de las incumbencias, aumentado la cohesión y disminuyendo notablemente el acoplamiento.
- En cuanto a la lógica de negocio, el uso de aspectos permitió separar claramente las incumbencias del dominio de las incumbencias específicas de la organización.

A continuación se enumeran algunas cuestiones abiertas que no han sido tratadas en este trabajo y que bien podrían ser objeto de futuras investigaciones.

- El uso de AOP en la capa de presentación de toda aplicación que requiera de interfases de usuario interactivas, es una problemática que aún no ha sido plenamente desarrollada.
- La explotación de las posibilidades ofrecidas para AOP por la plataforma .NET 3.5 a partir de la incorporación del soporte para lenguajes dinámicos, es una linea de trabajo aún inexplorada.

Finalmente cabe mencionar que las técnicas avanzadas de separación de incumbencias son una temática relativamente nueva y ausente aún hoy en la mayoría de los planes de estudio, lo cual convierte a dicha temática en un interesante objeto de investigación para tesis de grado y posgrado.

Apéndice I: Herramientas AOP

Aspect.NET

Esta herramienta de entretejido estático, desarrollada en la Universidad de San Pertersburgo, por el equipo liderado por Vladimir Safonov, consta de tres componentes: un convertidor de metalenguaje (o pre procesador), un weaver y un framework.

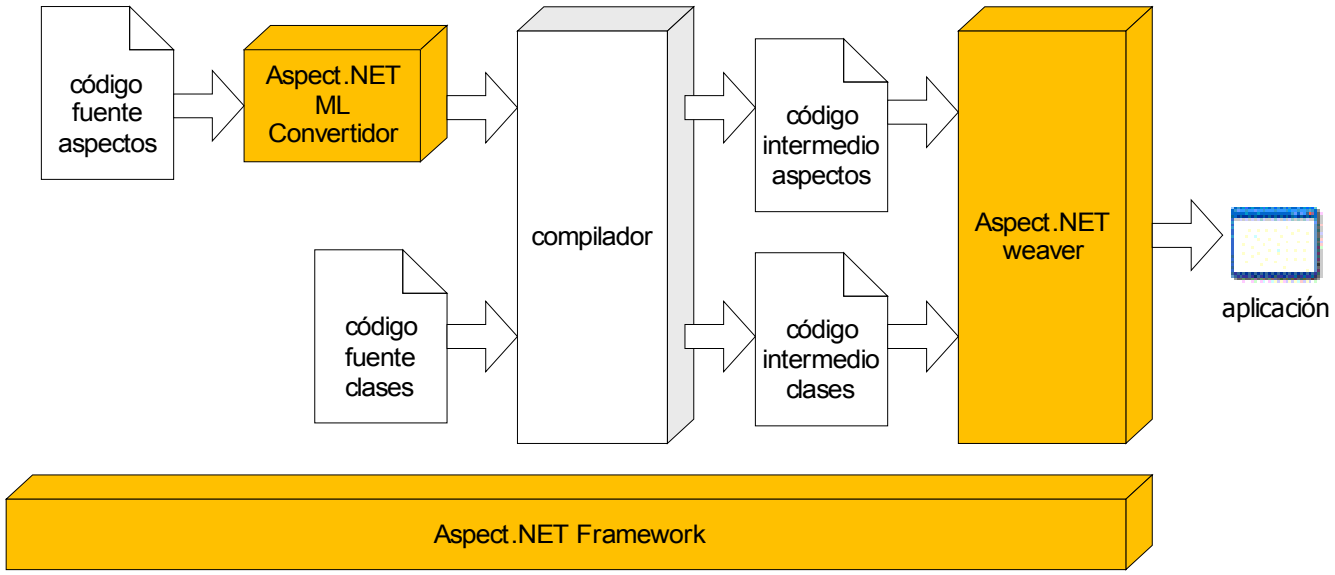

Figura 72: Componentes de Aspect.NET

La definición de un aspecto en Aspect.NET se hace utilizando un metalenguaje. El convertidor de metalenguaje, se encarga de la traducción a código C#, generando una clase por cada aspecto y agregando a dicha clase determinadas anotaciones que serán utilizadas posteriormente por el weaver para realizar el entretejido. También existe la posibilidad de definir aspectos, a partir de clases, heredando de la clase *Aspect* y agregándo ciertas anotaciones provistas por Aspect.NET.

La definición de un aspecto en el metalenguaje de Aspect.NET consta de 4 partes:

- Un encabezado opcional, que le indica al convertidor el nombre de la clase que representará el aspecto.
- Datos, que son simplemente atributos del aspecto y serán traducidos como tales a C#.
- Módulos, que son simplemente métodos del aspecto.y serán traducidos como tales a C#.
- Reglas, que especifican como debe realizarse el entretejido. Cada regla consiste en una condición y una acción. La condición es utilizada por el weaver para determinar los joinpoints donde aplicará la acción.

Haciendo un mapeo de los términos de Aspect.NET a los conceptos de AOP, podemos decir que las acciones juegan el rol de advices, mientras que las condiciones resultan equivalentes a poincuts. La versión actual de Aspect.NET (2.1) solo brinda soporte para definir condiciones (poincuts) identificando llamadas a métodos, aunque esto será ampliado en futuras versiones.

El framework de Aspect.NET brinda integración con el Visual Studio, permitiendo al programador intervenir en el proceso de entretejido eligiendo exactamente en que joinpoints de los que cumplen con la condiciones definidas por los aspectos, se aplicarán las acciones.

A diferencia de lo que ocurre con AspectJ, todos los miembros de un aspecto en Aspect.NET deben definirse como miembros de clase.

Un hecho interesante sobre Aspect.NET es que está basado en Phoenix [Phoenix], framework que constituye la plataforma base para la futura generación de compiladores Microsoft y que como tal brinda soporte para construcción de compiladores y herramientas de análisis y optimización de código.

Al trabajar con Aspect.NET uno desarrolla la aplicación tradicionalmente utilizando el compilador de C#. Luego escribe los aspectos según define Aspect.NET. Finalmente se realizar el entretejido tomando como entrada el código binario de la aplicación y de los aspectos. Esto dará como resultado una nueva aplicación con aspectos incluidos.

Spring AOP

Spring AOP es el módulo de SpringFramework que brida capacidades de programación orientada a aspectos complementando las funcionalidades ofrecidas por el resto de los módulos del framework. Existe dos implementaciones del framework, una para Java y otra para .NET que como es de esperar tienen algunas diferencias. En lo que respecta al módulo de AOP, la diferencia más notable está dada por el hecho de que la versión Java brinda cierta integración con AspectJ.

Todo el framework Spring es un desarrollo de código abierto, llevado adelante por la empresa Interface 21.

Spring AOP.NET está completamente implementado C#, sin hacer uso alguno de APIs no manejadas ni de modificaciones en los assemblies. Todo el entretejido es dinámico y como tal se hace en tiempo de ejecución, lo cual lo hace compatible con cualquier ambiente CLR.

Al igual que varias herramientas AOP basadas en proxies dinámicos, la definición de un aspecto se hace en dos partes: por un lado se deben definir los advices, a partir de implementar una determinada interface que depende del tipo de advice que se pretenda implementar y por otro lado se definen mediante configuración (generalmente en XML) los pointcuts sobre los que cada advice aplicará.

Spring AOP soporta todos los conceptos definidos por el paradigma AOP y debemos destacar las posibilidad de extensión que brinda, las cuales llegan incluso a permitir la definición de nuevos tipos de

advice y pointcuts.

Apéndice II: Patrones de diseño

A continuación se describe brevemente algunos de los patrones de diseño más utilizados en las aplicaciones enterprise.

Model-View-Controller

Este es uno de los patrones más difundido. Pertenece a la categoría estilo/patrón de arquitectura. Surgió en los años 70 implementado para Smalltalk y hoy en día tiene una gran influencia en lo que respecta al diseño de la capa de presentación de las aplicaciones enterprise. Básicamente el patrón ayuda a estructurar aplicaciones que requieran de interfases de usuario interactivas.

Transaction Script

Este patrón propone organizar toda la lógica asociada a una operación del usuario en un único procedimiento, interactuando directamente con la base de datos. Suele utililizarse cuando la lógica del negocio de la aplicación es relativamente simple.

Domain model

Propone estructurar la lógica de la aplicación en torno a clases que representan conceptos del negocio. En el diseño de dichas clases se utilizan fuertemente los conceptos de orientación a objetos para así obtener un modelo cercano a la realidad. Las clases no sólo encapsulan datos sino también comportamiento que expresa las reglas del dominio.

Service Layer

Permite definir la frontera de la aplicación a través de un conjunto de operación que pueden ser invocadas tanto desde una capa de presentación como desde otra aplicación. Encapsula toda la aplicación tomando a su cargo ciertas cuestiones de infraestructura, como la administración de transacciones y delega a las capa inferiores la resolución de la lógica de negocio.

Datamapper

Este patrón pertenece a la capa de persistencia. Capa mapper actua como nexo entre objetos en memoria y su representación en la base de datos, siendo respensable de la transferencia de datos entre ambos lugares.

Data Transfer Object

Ofrece una solución para distribución de objetos. Un Data Transfer Object (DTO) es un objeto serializable que puede viajar fácilmente a través de la red y que contiene generalmente información perteneciente a varios objetos de dominio.

Dependency Injection

Este patrón facilita la resolución de depencias entre objetos a partir de la introducción de una clase *Ensamblador*, que funcionando de forma similar a una fábrica de objetos, se encarga a partir de información de configuración, de resolver las dependencias, permitiéndonos adicionalmente separar interfaz de implementación.

Para más información sobre patrones en general ver [Gamma95] y en particular para el caso de patrones para aplicaciones enterprise ver [Fowler03].

Bibliografía

- [Acegi] Acegi Security, http://www.acegisecurity.org/, verificado el 02/11/2007.
- [Alexander77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel, A Pattern Language:Towns, Buildings, Construction , Oxford University Press, Nueva York, 1977, ISBN: ISBN: 0195019199.
- [Ambler00] S. Ambler, The Design of a Robust Persistence Layer for Relational Databases, www.ambysoft.com/downloads/**persistenceLayer**.pdf , verificado 02/11/2007.
- [Ambler04] S. Ambler, The Object-Relational Impedance Mismatch, http://www.agiledata.org/essays/impedanceMismatch.html, verificado 02/11/2007.
- [AopAlliance] AOP Alliance, http://aopalliance.sourceforge.net/, verificado 02/11/2007.
- [AORE] Aspect-Oriented Requirements Engineering, http://www.cs.toronto.edu/cser/aore.html, verificado el 02/11/2007.
- [AOSD] Aspect-Oriented Software Development, http://www.aosd.net/, verificado 02/11/2007.
- [APS] Apache Logging Services, http://logging.apache.org/, verificado 02/11/2007.
- [Ashmore04] D. Ashmore, The J2EE architect's handbook: how to be a successful technical architect for J2EE applications, DVT Press 2004, ISBN: 0-972-95489-9.
- [Aspect#] Aspect#, http://www.castleproject.org/aspectsharp/index.html, verificado 02/11/2007.
- [Aspectc++] AspectC++, http://www.aspectc.org, verificado 02/11/2007.
- [AspectDNG] AspectDNG, http://aspectdng.tigris.org/, verificado 02/11/2007.
- [Aspectj] AspectJ, http://www.eclipse.org/aspectj, verificado 02/11/2007.
- [AspectNet] AspectDNG, http://aspectdng.tigris.org , verificado 02/11/2007.
- [Bass03] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Second Edition, Addison Wesley 2003, ISBN: 0-321-15495-9.
- [Bodkin06] R. Bodkin, AOP@Work: Next steps with aspects, http://www.ibm.com/developerworks/java/library/j-aopwork16/index.html?S_TACT=105AGX02&S_CMP=EDU#resources, verificado 02/11/2007.
- [Brown] K. Brown, B. Whitenack, Crossing Chasms: A pattern language for object-

RDBMS integration, http://www.ksc.com/article5.htm, verificado 02/11/2007.

- [Buschmann95] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A system of patterns, John Wiley & Sons Ltd, ISBN: 0-471-95869-7.
- [Contract4j] Contract4J: Desing by contract for Java, http://www.contract4j.org/contract4j, verificado 02/11/2007.
- [Cyment04] A. Cyment, R. Altman, SetPoint: Un enfoque semántico para la resolución de pointcuts en AOP, Tesis de Licenciatura en Ciencias de la Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Noviembre 2004.
- [Dijkstra68] E. Dijkstra, Go To Considered Harmful, Letter to Communications of the ACM (CACM), vol. 11 no. 3, March 1968, pp. 147-148.
- [Diotalevi04] F. Diotalevi, Contract enforcement with AOP: Apply Design by contract to java software development with AspectJ, http://www.ibm.com/developerworks/library/j-ceaop/, verificado el 02/11/2007.
- [Duck06], A. Duck, Implementation of AOP in non-academic projects, 5th International Conference on Aspect-Oriented Software Development, Industry Track Proceedings, Bonn Germany March 2006.
- [Eaddy05] M. Eaddy, A. Aho, W. Hu, P. McDonald y J. Burguer, Debugging Woven Code, Columbia University, New York; Microsoft Corporation, Redmond, 2005.
- [EarlyAspects] Early Aspects, http://www.early-aspects.net/, verificado 02/11/2007.
- [Eclipse] Eclipse, http://www.eclipse.org/, verificado el 02/11/2007.
- [Eidelman06] A. Eidelman, Hacia una mejor experiencia de debugging en desarrollos AOP, Tesis de grado en Ingeniería Informática, Facultad de Ingeniería, Universidad de Buenos Aires, Noviembre 2006.
- [Elrad01] T. Elrad, M. Aksit, G. Kiczales,K. Lieberherr,H. Osser, Discusing aspects of AOP, Communications of the ACM, Volume 44 , Issue 10 (October 2001) Páginas: 33 a 38, 2001.
- [Endrei04] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, Patterns: Service-Oriented Architecture and Web Services, IBM RedBooks 2004.
- [Erdody06] D. Erdody, Un Framework para la visualización de patrones de diseño distribuidos y concurrentes implementados con programación orientada a aspectos: ACVF (Aspectual Component Visualization Framework), Tesis de grado en Ingeniería Informática, Facultad de Ingeniería, Universidad de Buenos Aires, Dicimebre 2006.

- [Erl05] T. Erl, Service-Oriented Architecture: Concepts, Technology and Design, Prentice Hall PTR 2005, ISBN: 0-13-185858-0.
- [Evans03] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison Wesley 2003, ISBN: 0-321-12521-5.
- [Filman04] R. Filman, T. Elrad, S. Clarke, M. Aksit, Aspect-Oriented Software Development, Addison-Wesley 2004, ISBN: 0321219767
- [Fowler03] M. Fowler, D. Rice, M Foemmel, E. Hieatt, R Mee, R Stafford, Patterns of Enterprise Application Architecture, Addison Wesley 2002, ISBN: 0-321-12742-0.
- [Gamma95] E. Gamma, R Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software , Addison-Wesley, 1995, ISBN: 0201633612 .
- [Harrison02] W. Harrison, H. Ossher, P. Tarr, Asymmetrically vs. Symmetrically Organized Paradigms For Software Composition, IBM Research Report, Dicember 2002.
- [Hohmann03] L. Hohmann, Beyond Software Architecture: Creating and Sustaining Winning Solutions, Addison Wesley 2003, ISBN: 0-201-77594-8.
- [Isberg05] W. Isberg, AOP@work: Design with pointcuts to avoid pattern density, http://www.ibm.com/developerworks/java/library/j-aopwork7/index.html, verificado 02/11/2007.
- [JbossAop] JbossAOP, http://labs.jboss.com/portal/jbossaop/index.html,verificado 02/11/2007.
- [Jezierski03] E. Jezierski, Application Architecture for .NET: Desinging Applications and Services, Microsoft Press 2003, ISBN: 0735618372.
- [Johnson02] R. Johnson, J. Hoeller. Expert One-on-One J2EE Desing and Development, Wrow Press 2002, ISBN
- [Johnson04] R. Johnson, J. Hoeller. Expert One-on-One J2EE Development without EJB, Wiley Publishing Inc. 2004, ISBN 0-7645-5831-5.
- [Kiczales97] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Videira Lopes, C. Maeda and A. Mendhekar. Aspect-Oriented Programming, ECOOP 1997 .
- [Laddad03] R. Laddad, AspectJ in action, Practical Aspect-Oriented Programming, Manning Publications 2003, ISBN 1-930110-93-6.
- [Laddad06] R. Laddad, AOP@Work: AOP myths and realities, Beyond hype and misunderstandings, http://www.ibm.com/developerworks/java/library/j-aopwork15/, verificado 02/11/2007.
- [Lesiecki03] N. Lesiecki, J. Gradecki, Mastering AspectJ, Aspect-Oriented Programming

in Java, Wiley Publishing Inc. 2003, ISBN 0-471-43104-4.

- [Lesiecki05] N. Lesiecki, Enhance design patterns with AspectJ, http://www.ibm.com/developerworks/library/j-aopwork5/, verificado el 02/11/2007.
- [Loom] LOOM.NET, http://www.dcl.hpi.uni-potsdam.de/research/loom/, verificado 02/11/2007.
- [Lopes00] C. Videira Lopes, M. Lippert, A study on Exception Detection and Handling using Aspect-Oriented Programming, ICSE 2000.
- [Meyer92] B. Meyer, Applying Design by Contract, Computer, Volume 25, Octubre 1992, Páginas 40-51, 1992, ISSN: 0018-9162
- [Miles05] R. Miles, AspectJ Cookbook, O'Reilly 2005, ISBN: 0-596-00654-3.
- [Naspect] Naspect, http://www.puzzleframework.com/WikiEngine/WikiPageViewer.aspx?ID=83, verificado 02/11/2007.
- [Nilsson06] J. Nilsson, Applying Domain-Driven Design and Patterns, Addison-Wesley 2006, ISBN: 0-321-26820-2.
- [Pawlak03] R. Pawlak, The AOP Alliance: Why did we get in?, Julio 2003, http://aopalliance.sourceforge.net/white_paper/ , verificado el 02/11/2007.
- [Pawlak05] R. Pawlak, L Seinturier, J.P. Retaillé, Foundations of AOP for J2EE Development, APress 2005, ISBN: 1-59059-507-6.
- [Phoenix] Phoenix, https://connect.microsoft.com/Phoenix/ verificado el 02/11/2007.
- [Rashid02] A. Rashid, R. Chitchyan, Persistence as an Aspect, Proceedings of the 2nd international conference on Aspect-oriented software development, Boston, Massachusetts, Pages: 120 – 129 2003, ISBN:1-58113-660-9
- [Soares02] S. Soares, E. Laureano, P. Borba, Implementing distribution and persistence with AspectJ, OOPSLA 2002, ACM Press páginas 174-190.
- [Sousa04] G. Sousa, S. Soares, P. Borba, J. Castro, Separation of crosscutting concers from Requirements to Design: Adapting an Use case driven approach, In Proc. Early Aspects Workshop at AOSD 2004.
- [Spring] Spring Framework, http://www.springframework.com, verificado 02/11/2007.
- [Stoerzer04] M. Stoerzer, T. Skotiniotis, C. Constantinides, AOP considered harmful, 1st European Interactive Workshop on Aspect Systems (EIWAS), 2004.
- [Koppen04] C. Koppen, M. Stoerzer, PCDiff: Attacking the Fragile Pointcut Problem, 1st European Interactive Workshop on Aspect Systems (EIWAS), 2004.

- [Tarr99] P. Tarr, H. Ossher, S. Sutton, W. Harrison, N Degrees of separation: Multi-Dimensional Separation of Concerns, Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, California, USA, 1999.
- [Tourwé03] T. Tourwé, J. Brichau, K. Gybels, On the Existence of the AOSD-Evolution Paradox, AOSD Workshop on Software-engineering, Boston, USA, 2003
- [Trowbridge03] D. Trowbridge, D. Mancini, D Quick, G. Hohpe, J Newkirk, D Lavigne, Enterprise Solution Patterns Using Microsoft.NET Versión 2.0, Microsoft Press
- [WCF] Windows Communication Foundations, http://wcf.netfx3.com/, verificado 02/11/2007.