CAPÍTULO II

TECNOLOGÍAS ORIENTADAS A ASPECTOS

Contenidos del capítulo:

TECNOLOGÍAS ORIENTADAS A ASPECTOS

En este capítulo se describirá el estado del arte de las tecnologías orientadas a aspectos, en particular las diseñadas para la plataforma .NET. La Programación Orientada a Aspectos, conocida por el término inglés *Aspect Oriented Programming* (AOP) fue originariamente empleada en Java y es por ello que en este lenguaje es donde dicha tecnología ha ido evolucionando principalmente. Java es el lenguaje de programación que más herramientas de desarrollo dispone para dar soporte a la orientación a aspectos. El lenguaje por excelencia es *AspectJ*, y además de ser el que más avanzado se encuentra, es en el que se han basado la mayoría de aproximaciones existentes para otros lenguajes.

Actualmente, la AOP está teniendo un gran empuje. Por esta razón, a pesar de que .NET es una plataforma relativamente nueva, existen bastantes proyectos basados en ella que ofrecen soporte a AOP. Algunos de los más importantes son *AspectC#, JAsCo.NET, Rapier-Loom.NET* y *SetPoint!*, aunque aún se encuentran en fase de desarrollo.

En las siguientes secciones se definen las principales características de estas tecnologías, así como sus ventajas e inconvenientes.

2.1 Precedentes en JAVA

2.1.1 AspectJ

Desarrollado originariamente por el PARC[1] a finales de 1997, y financiado conjuntamente por el NIST[2] y el DARPA[3], a finales de 2002 fue liberado como proyecto de desarrollo abierto en *eclipse.org*. En sus 6 años de vida, AspectJ [AsJ04] ha pasado de ser un prototipo de investigación a ser el lenguaje más utilizado por la comunidad de desarrolladores en AOP, y utilizado en sistemas en producción.

[1] Palo Alto Research Center, USA
[2] National Institute of Standards and Technology, USA
[3] Defense Advanced Research Projects Agency, USA

AspectJ utiliza un compilador propio mediante el cual se realizan los enlaces entre aspectos y aplicaciones Java. AspectJ es una tecnología que extiende el lenguaje origen, es decir, el programador que necesite utilizar AOP necesariamente deberá aprender una sintaxis adicional para utilizar aspectos en su aplicación. Además, AspectJ es un enlazador de aspectos *estático*, porque el proceso de enlazado (*weaving*) se realiza en tiempo de compilación, y por tanto, no podrá modificarse en tiempo de ejecución.

Los lenguajes basados en AspectJ incorporan las siguientes construcciones: *Join Points, Pointcuts, Advices* y *Aspects*. Los *joinpoints* están ligados al código base de la aplicación, mientras que los *aspects*, *pointcuts* y *advices* se definen en el código aspecto. Cada una de estas construcciones se explica en detalle a continuación:

2.1.1.1 Aspects

Un aspecto es una construcción del lenguaje que encapsula un *crosscutting concern*. Su definición es muy similar a una clase de Java, con la salvedad que además añade *advices y pointcuts*, estructuras sintácticas que definen cómo y a qué código se enlazarán los aspectos, respectivamente.

Un aspecto se enlaza con uno o varios métodos del código base, a través de los *pointcuts* (puntos de corte), y mediante los *advices* se especifica si la ejecución del código del aspecto será antes, después o en lugar de la ejecución del método indicado por el *pointcut*. En el siguiente ejemplo, cuya finalidad es auditar el acceso a una cuenta, se muestra cómo es un aspecto sintácticamente en Java, y se señalan las construcciones sintácticas que lo forman (*pointcuts* y *advices*):

```
Aspect AccountTracking {
  pointcut Track(): call(public decimal Account.Balance( ));
  before(): Track() {
  // Cuerpo del aspecto a ejecutar
  System.out.println("The account is going to be consulted");
  }
  after(): Track() {
  // Cuerpo del aspecto a ejecutar
  System.out.println("The account has been consulted");
  }
}
```


La forma en que AspectJ combina (*aspect weaving*) los *concerns* con el código origen, es mediante el enlazado en tiempo de compilación. El código original es transformado por el compilador de AspectJ de forma que los *advices* son transformados en métodos estándar, cuya llamada es insertada en aquellos puntos del código origen indicada por los *pointcuts*. Si existen zonas del código que son resueltas dinámicamente, el código es dirigido a las zonas controladas por los *advices*.

2.1.1.2 Join Points

Un *Join Point* (punto de unión) es un concepto semántico introducido en *AspectJ* que se define como un punto bien definido en el flujo de ejecución de un programa. Una llamada a un método o un acceso a un atributo serían ejemplos de *joinpoints*. Existen diferentes tipos de *joinpoints* definidos en AspectJ. Los más utilizados son los que se identifican con:

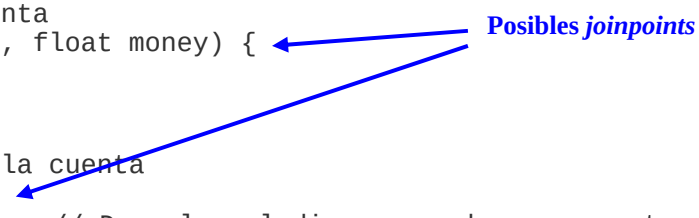
- la llamada o invocación a un método o constructor,
- la ejecución del cuerpo de un método o constructor,
- los métodos *get* y *set* de propiedades,
- el tratamiento de una determinada excepción.

Un *joinpoint* debe considerarse como un punto en la ejecución de la aplicación susceptible de ser interceptado por los aspectos, es decir, como un punto de entrada donde podrá introducirse el código de los aspectos para alterar el flujo de ejecución normal del código base. Es importante tener en cuenta que cada *joinpoint* es distinto en tiempo de ejecución, tiene un contexto de ejecución diferente. Por ejemplo, un *joinpoint* correspondiente a la invocación de un método, será diferente según sea el objeto que lo llame.

Los *joinpoints* forman parte del código base y no necesitan de ninguna construcción explícita del lenguaje para especificarlos. A continuación se muestra la definición de una clase *Account* en Java con la definición de sus correspondientes atributos y métodos públicos, en la que se han señalado los diferentes *joinpoints* que podrían ser interceptados.

```
class Account {
   // Definición de variables
   private string ID;          // ID de la cuenta
   private float money;        // Dinero depositado en la cuenta

   // Definición de métodos
   // Constructor de la cuenta
   public Account(string ID, float money) {
      this.ID = ID;
      this.money = money;
   }
   // Devuelve el saldo de la cuenta
   public float Balance() {
      return money;            // Devuelve el dinero que hay en cuenta
   }
   // ... otros métodos
}
```



Como se puede observar, en la definición de una clase sencilla existen numerosos puntos de su ejecución que pueden ser interceptados (*joinpoints*): por ejemplo, la ejecución del cuerpo del constructor o del método *Balance* correspondería a dos *joinpoints* diferentes.

2.1.1.3 Pointcuts

Por tanto, una aplicación tendrá un conjunto de *joinpoints*, de los cuales tan sólo un pequeño subconjunto interesará interceptar. Para ello, se definen los *pointcuts* (puntos de corte) mediante los cuales se podrá seleccionar, del conjunto de todos los *Join Points* posibles, aquellos que se desean utilizar para asociarlos con un aspecto concreto. Es por esto que los *pointcuts* son los elementos de AspectJ que permiten definir el *weaving* (enlazado) de un aspecto con el código base.

Existe una gran variedad de *Pointcuts*, algunos de ellos son:

- ***call(**<patron_método>**)***: captura todos los *joinpoints* de llamadas a métodos o constructores cuya signatura coincida con el patrón especificado.

  Por ejemplo, el *pointcut* definido como **call**(public decimal Account. Balance(*)), capturará todas las llamadas al método *Balance* de los objetos de Account, con cualquier número de argumentos y que retornen un valor *decimal*.

- ***args(**<lista_tipos>**)***: captura todos aquellos *joinpoints* cuyos argumentos concuerden, en orden y tipo, con la lista de tipos especificada.

  Por ejemplo, el siguiente *pointcut* **args(**object, int, string) capturará todos aquellos métodos cuyos parámetros sean un objeto, un entero y una cadena, en ese orden.

- **target**(<tipo_clase>): selecciona aquellos *joinpoints* que pertenezcan a una instancia cuyo tipo es la clase especificada.

  Por ejemplo, **target**(Account) capturará el conjunto de *joinpoints* que hagan referencia a instancias de la clase Account.

Además, también pueden utilizarse los operadores && (and), || (or) y ! (not) para permitir la combinación de varios *Pointcuts*.

Por tanto, mediante la siguiente declaración:

```
pointcut Track(): call(public decimal Account.Balance());
```

se define un *pointcut* cuyo nombre identificativo es *Track()*, que no toma ningún argumento, y selecciona los *joinpoints* correspondientes a la invocación del método Balance de la clase Account (habrá tantos como instancias diferentes de la clase Account).

2.1.1.4 Advices

Un advice define el código que deberá ser ejecutado en los *joinpoints* seleccionados por los *pointcuts*. La ejecución de dicho código estará

condicionada por el tipo de *advice* que lo contenga. AspectJ dispone de diferentes tipos:

- *before:* el código enlazado se ejecuta antes que cada *joinpoint*
- *after:* el código enlazado se ejecuta después de la ejecución de cada *joinpoint*. Existen dos subtipos: *after throwing* y *after returning*. El primero es utilizado cuando se lanza una excepción, y el segundo cuando se finaliza con normalidad la ejecución del *joinpoint*
- *around:* el código del aspecto reemplaza el código que se iba a ejecutar en el *joinpoint*.

Volviendo al ejemplo mostrado en el apartado 2.1.1.1, se puede observar cómo se han definido dos *advices*, uno de tipo *before* y otro de tipo *after*:

```
before(): Track() {
   // Cuerpo del aspecto a ejecutar
   System.out.println("The account is going to be consulted");
}
after(): Track() {
   // Cuerpo del aspecto a ejecutar
   System.out.println("The account has been consulted");
}
```

Cada uno de los *advices* define un código que deberá ejecutarse antes y después, respectivamente, de los *joinpoints* capturados por el *pointcut Track()*, definido en el apartado anterior.

Por tanto, y viendo el proceso de forma global, cuando en el flujo de ejecución del programa base se active uno o más de los *joinpoints* capturados por el *pointcut Track()* (siguiendo el ejemplo, la invocación de los servicios cuyo nombre sea *Balance*), se ejecutará en primer lugar el código del *advice before*. Después se ejecutará el código del método interceptado en la aplicación base, y cuando finalice su ejecución se ejecutará el código asociado al *advice after*. Una vez finalice, el flujo del programa volverá al punto en el que se había quedado la aplicación.

Las principales ventajas de esta aproximación son las sencillas extensiones del lenguaje Java realizadas con la potencia del modelo de *joinpoints*. Además, también ha asentado los diferentes tipos de enlace de aspectos, que como se verá a lo largo de este capítulo, ha influido en las demás aproximaciones. Para más información sobre AspectJ, puede consultarse la documentación completa y la descripción completa de la sintaxis en [AsJ03].

2.1.2 Hyper/J

Una alternativa a AspectJ es Hyper/J [Ossh00], ésta propone una forma diferente de agrupar los *crosscutting concerns*. La mayoría de las tecnologías orientadas a aspectos son paradigmas asimétricos: por una parte se consideran los objetos, y por otra parte los aspectos, como unidades a las que

se redirige el flujo de información desde los objetos. Por contra, Hyper/J es un paradigma simétrico, ya que considera a los aspectos como entidades de primer orden, al igual que los objetos. Propone un modelo de composición mediante el cual los *concerns* pueden ser integrados en un programa o en un componente, además de poder ser utilizados en cualquier etapa del ciclo de desarrollo de software.

Hyper/J define el término *hiperespacio (Hyperspaces)* como el conjunto de aquellos elementos del lenguaje que van a ser utilizados para la composición de *concerns:* pueden ser paquetes Java, interfaces, clases, operaciones o atributos. La composición de estos elementos en estructuras declarativamente completas[4] forman los denominados *Hyperslices (*o *hipersecciones*), que permiten la encapsulación de código con una funcionalidad determinada en una estructura independiente, favoreciendo la reutilización y la independencia y limitando el impacto de los cambios. Cada *hyperslice* puede ser un aspecto o un objeto perteneciente a la lógica de negocio. En la Figura 2 puede observarse cómo los diferentes elementos del hiperespacio se agrupan en *hyperslices*, algunos de los cuales, como *Logging* son aspectos, mientras que otros son de la lógica de la aplicación (como *Employee*).

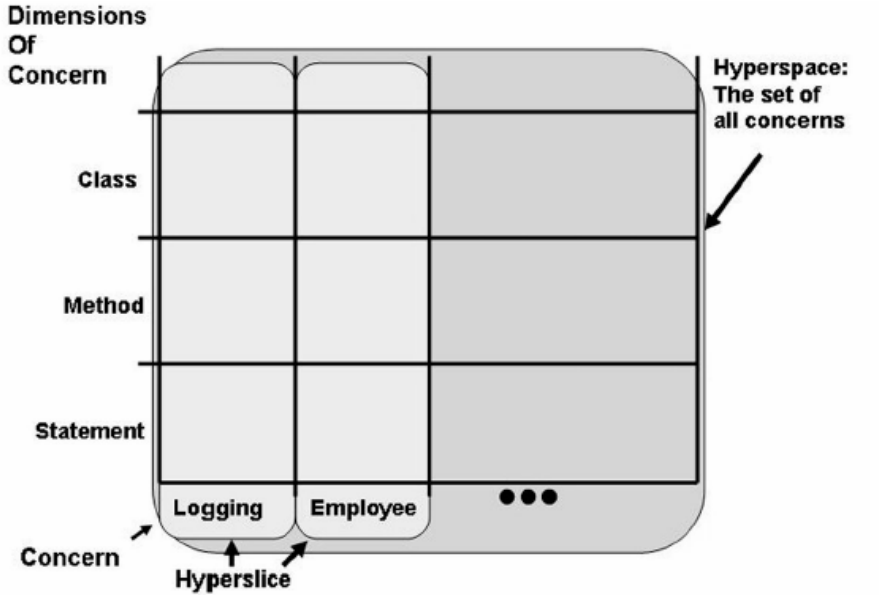


Figura 2 - Hyper/J

Finalmente, los *hypermodules* o *hipermódulos* incluyen un conjunto de *hyperslices* que se pueden componer entre sí, junto con un conjunto de reglas de composición que especifican cómo han de hacerlo. Pueden anidarse entre sí permitiendo la composición de estructuras más complejas.

[4] Una estructura declarativamente completa es un elemento funcional que declara todo aquello que referencia. Por ejemplo, en Java podría corresponderse con un paquete de código que declara como abstracto aquellos métodos externos que referencia.

A continuación se muestra un ejemplo muy sencillo, cuyo objetivo es mostrar cómo se pueden combinar dos clases independientes entre sí mediante Hyper/J:

```
public class A {
   public void diHola() {
      System.out.println("Hola");
   }
}
public class B {
   public void diAdios() {
      System.out.println("Adios");
   }
}
```

La composición de ambas en Hyper/J:

```
hyperspace
   hyperspace demo
   composable class A;
   composable class B;

-concerns
   class A : Feature.hola
   class B : Feature.adios

-hypermodules
   hypermodule DemoHM
      hyperslices: Feature.hola, Feature.adios;
      relationships:
         mergeByName;
         equate operation Feature.hola.diHola, Feature.adios.diAdios
            into saluda;
         merge class Feature.hola.A, Feature.adios.B;
   end hypermodule;
```

Para realizar la composición entre ambas clases, en primer lugar se definen los paquetes donde se encuentran los elementos del hiperespacio. En este caso son las clases A y B del código original que desea combinarse. En la sección *concerns*, se define la asociación entre los elementos del hiperespacio (que pueden ser paquetes, clases o métodos) y los *concerns* que definen. En este caso, ambas clases pertenecen a la misma dimensión: *Feature*, pero definen distintos *concerns*. Todos los concerns de la misma dimensión y mismo nombre pertenecen a un mismo *hyperslice*. En este ejemplo, se tiene un *hyperslice* distinto para cada clase. Por último, la sección *hypermodules* define el módulo que se va a construir, que está formado por las *hyperslices* definidas anteriormente (*Feature.hola* y *Feature.adios*). El apartado de *relationships* define las reglas para combinar dichos *hyperslices*: *mergeByName* indica que los elementos con el mismo nombre en distintas *hyperslices* se fusionarán en una nueva unidad, mientras que *equate operation ... into* indica que las operaciones, aunque tienen distinto nombre, se fusionarán en una nueva unidad compuesta de nombre *saluda*, en este caso. El resultado de esta composición sería un paquete con un método *Saluda* que provocaría la ejecución del código asociado a *A.diHola()* y *B.diAdios()*.

Mediante este sencillo ejemplo, se pretende mostrar una pequeña parte de la potencia expresiva que Hyper/J posee. Existe también una versión prototipo que, de forma gráfica y a través de sencillos modelos, permite componer o modificar los diferentes *hiperslices*, que se traducen por las reglas de composición de Hyper/J. Por otra parte, a diferencia de AspectJ, Hyper/J trabaja con *bytecodes*, es decir, con módulos ya compilados. Por lo tanto, es posible trabajar con COTS (*commercial off-the-self*) de las cuales en la mayoría de los casos no se dispone del código fuente.

La principal aportación de esta tecnología es la consideración de los aspectos como entidades de primer orden y la facilidad para la integración de módulos que encapsulan diferentes *concerns* independientes entre sí. Todo ello sin necesitar extender el lenguaje. El lenguaje específico de Hyper/J sólo se utiliza para la composición de módulos. Dicha composición puede hacerse mediante el prototipo gráfico que proporciona Hyper/J, tal y como se ha mencionado anteriormente.

2.2 Primeras aproximaciones en .NET

Una vez vistas algunas de las más importantes aproximaciones AOP de Java, a continuación se presentan las diferentes aproximaciones existentes en .NET. Mientras que en Java las herramientas existentes son de gran calidad y su estado es bastante avanzado, en .NET aún se encuentran en fases tempranas de desarrollo. En esta sección se esbozan las características de las primeras aproximaciones, algunas ya abandonadas, que surgieron para .NET.

2.2.1 CLAW (Cross Language Aspect Weaving)

CLAW, desarrollado por John Lam[5] en 2002 y cuya arquitectura se define en [Lam02], comparte muchas similitudes con AspectJ, como los *advices* y algunos *Join points*. CLAW se diferencia de AspectJ por su arquitectura, concretamente por la forma de enlazar los aspectos con código base. Los objetivos perseguidos por el proyecto eran:

- Evitar extender el lenguaje utilizado por el programador para definir los aspectos.
- Separar el código base (o código de aplicación) del código que define los aspectos.
- Utilizar un mecanismo sencillo para unir código base y aspectos.
- Permitir que cualquier lenguaje pueda formar parte del código base o de los aspectos.

[5] Naleco Research Inc., Toronto, Canada – http://www.iunknown.com

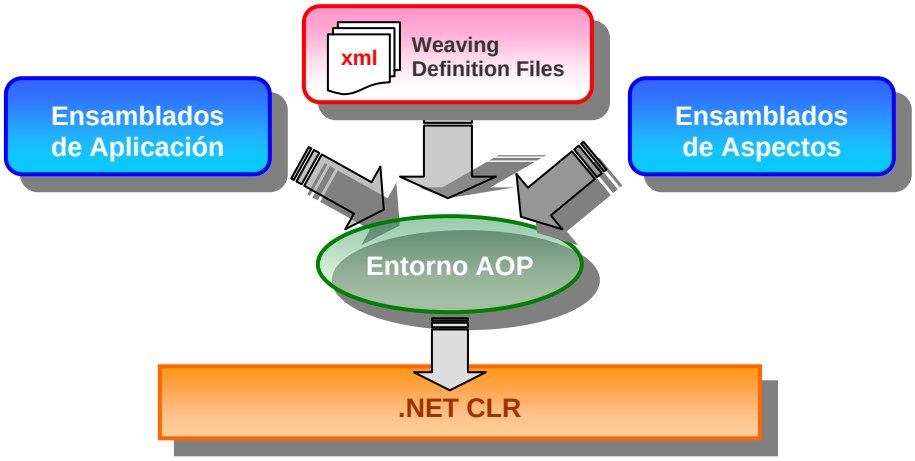


Figura 3 - Arquitectura CLAW y AOP#

La arquitectura se puede observar en la Figura 3. Los ensamblados de aplicación contienen todo el código de la aplicación y la lógica de negocio, mientras que los ensamblados de Aspectos contienen todo el código asociado a los aspectos a utilizar. El weaving se realiza en tiempo de ejecución de forma dinámica, uniendo dichos ensamblados. Esto posibilita que el código origen sea totalmente independiente de los aspectos, hasta el punto que el código fuente no necesita estar escrito en el mismo lenguaje que el código de los aspectos, o proceder del mismo proveedor de software. El weaving se define mediante unos ficheros XML (*weaving definition files*) que definen el mapeado entre los ensamblados base y los aspectos. En tiempo de ejecución, previamente a la ejecución del código por el compilador JIT (*Just In Time*) de .NET, CLAW modifica el código IL (*Intermediate Language*) del ensamblado base para que el código de los aspectos sea ejecutado de acuerdo con el weaving especificado. No obstante, dicha modificación crea una fuerte dependencia con el núcleo de .NET, lo que conlleva problemas de compatibilidad importantes con futuras versiones de la plataforma. Esto es debido a que la generación de código IL por el compilador de .NET, o la ejecución de dicho código en el CLR (*Common Language Runtime,* la capa más interna de la plataforma .NET) puede ser susceptible de cambios en futuras versiones. Como consecuencia, la herramienta debería actualizarse cada vez que saliese una nueva versión del *framework* .NET, lo cual puede ser una desventaja importante frente a otras herramientas.

Por otra parte, el código IL es generado por .NET exclusivamente para código manejado, es decir, para ensamblados que son interpretados por el CLR. Por ello, otra limitación es que CLAW no podrá unir entre sí ensamblados de código no manejado (o código nativo). Además, este modelo de ejecución también acarrea problemas de seguridad importantes, pues supone que cualquier desarrollador podría modificar el comportamiento de un ensamblado del cual el usuario confía, con tan sólo añadirle aspectos con código malicioso.

Es por estas razones que el proyecto fue abandonado en 2002, sin llegar a disponer de un prototipo estable. No obstante, al ser uno de los primeros

prototipos para .NET, ha influido en otras aproximaciones posteriores, como AspectC#, que han utilizado algunos conceptos del modelo de ejecución introducidos en CLAW. Por ejemplo, la interoperabilidad de los ensamblados y la no extensión del lenguaje utilizado por el desarrollador mediante el uso de la definición de los weavings en ficheros separados del código.

2.2.2 AOP#

El proyecto AOP#[Schü02], diseñado para Siemens Corporation, es casi idéntico al anteriormente expuesto en cuanto a sus objetivos y arquitectura. La diferencia entre ambos es que AOP# puede enlazar/desenlazar dinámicamente los aspectos al código base ("*Aspectual Polymorphism*"). El entorno AOP intercepta el código en tiempo de ejecución e inserta el código de los aspectos (a nivel de código IL de .NET) mediante la *Profiling API*[6] de la plataforma .NET. En cuanto a la definición de los aspectos, y a diferencia del caso anterior, los aspectos se definen como clases que heredan de una clase base llamada *Aspect*. No obstante, al funcionar casi de la misma manera que el anterior, presenta los mismos inconvenientes: sólo permite tratar con código manejado, depende del núcleo de .NET y puede originar problemas de seguridad.

2.3 AspectC#

AspectC# [Kim02] es una aproximación orientada a aspectos que tiene puntos en común con CLAW y AOP# como: separar el código base y los aspectos, no extender el lenguaje y definir los *weavings* mediante un fichero XML. A diferencia de AOP#, no soporta *Polimorfismo Aspectual*, los aspectos son enlazados en tiempo de compilación y por tanto el código generado es estático. Actúa sobre el código fuente, en concreto en C#: no actúa sobre el CLR ni modifica código intermedio de .NET, por lo que no adolece de los problemas comentados anteriormente.

Está formado por un compilador cuyas entradas son el código base, el código de los aspectos, y una especificación XML (*Aspect Deployment Descriptor*) del *weaving* a realizar entre ambos. La estructura interna del compilador se puede observar en la figura siguiente:

[6] La *Profiling API* de .NET permite, aparte de analizar el funcionamiento de aplicaciones sobre el CLR de .NET (código manejado), interceptar entradas/salidas a métodos y llamar a funciones personalizadas antes o después de llamar a dichos métodos. Puede encontrarse más información en: http://msdn.microsoft.com/msdnmag/issues/01/12/hood/

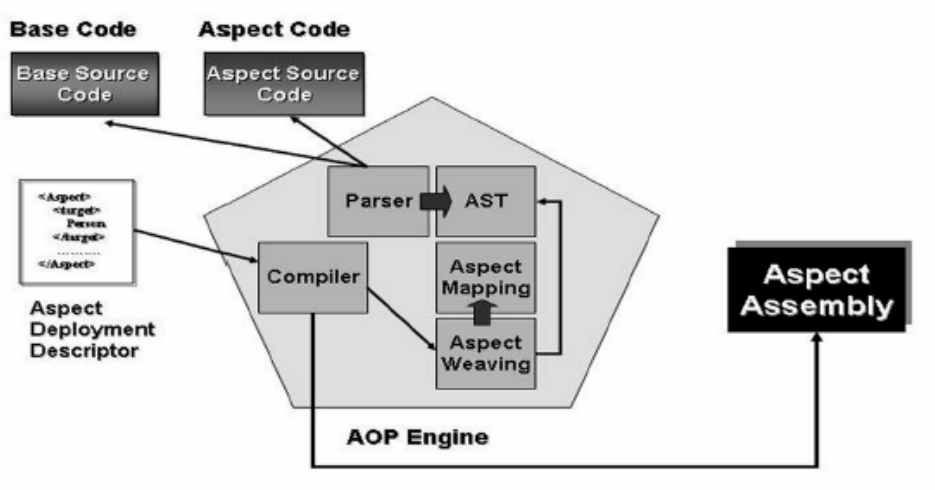


Figura 4 - Compilador de AspectC#

El código base es analizado por el *Parser*, que tras comprobar que la sintaxis es válida, genera un *Abstract Syntax Tree* (AST) en la que se plasma la estructura del código fuente. Con el código correspondiente a los aspectos se genera otro árbol AST de la misma manera. Estos dos árboles son fusionados teniendo en cuenta los *weavings* definidos en el *Aspect Deployment Descriptor*, generando un único árbol AST. Finalmente, mediante la tecnología *CodeDOM*[7] de .NET, éste árbol es transformado en código ejecutable a través de cualquier compilador estándar de .NET, generando el ensamblado con la aplicación final.

A continuación se muestra un ejemplo de definición de *weavings* en AspectC# mediante XML (Figura 5), y el resultado tras generar el ensamblado y enlazar los aspectos (Figura 6):

[7] CodeDOM (Code Document Object Model) es una tecnología ofrecida por .NET que permite el desarrollo automático de generadores de código fuente, mediante la representación de código fuente en forma de árboles abstractos de datos: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconCodeDOMQuickReference.asp

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Aspect>
    <TargetBase>C:\AspectCSDemo\CodigoBase</TargetBase>
    <AspectBase>C:\AspectCSDemo\CodigoAspectos</AspectBase>
    <Aspect-Method>
        <Name>AspectoFuncional1</Name>
        <Namespace>OtroNameSpace</Namespace>
        <Class>AspectoFuncional</Class>
        <Method>EmitirMensaje()</Method>
    </Aspect-Method>
    <Aspect-Method>
        <Name>AspectoFuncional2</Name>
        <Namespace>OtroNameSpace</Namespace>
        <Class>AspectoFuncional</Class>
        <Method>Suma(int a, int b)</Method>
    </Aspect-Method>
    <Target>
        <Namespace>Test</Namespace>
        <Class>Base</Class>
        <Method>
            <Name>Saluda()</Name>
            <Type>after</Type>
            <Aspect-Name>AspectoFuncional1</Aspect-Name>
        </Method>
        <Method>
            <Name>Saluda()</Name>
            <Type>before</Type>
            <Aspect-Name>AspectoFuncional1</Aspect-Name>
        </Method>
```

Figura 5 - Ejemplo de descriptor de aspectos en AspectC#

```
namespace Test {
    using Ie.Tcd.AspectCSharp;
    using System;

    public class Base {
        private string test;

        public Base() {
            JoinPoint ThisJoinPoint = new JoinPoint("Base ","Base",
                                      @"C:\AspectCSDemo\CodigoBase\Base.cs");
            // TODO: Add constructor logic here
        }

        public virtual void Saluda() {
            JoinPoint ThisJoinPoint = new JoinPoint("Base ","Saluda",
                                      @"C:\AspectCSDemo\CodigoBase\Base.cs");
            Console.WriteLine("Soy el metodo EmitirMensaje de la clase AspectoFuncional");
            Console.WriteLine("Estoy introducido en el metodo: " + ThisJoinPoint.GS_MethodName);
            Console.WriteLine("");

            Console.WriteLine("Hola Mundo");

            Console.WriteLine("Soy el metodo EmitirMensaje de la clase AspectoFuncional");
            Console.WriteLine("Estoy introducido en el metodo: " + ThisJoinPoint.GS_MethodName);
            Console.WriteLine("");
        }

        // Method renamed to: funcion001a
        public virtual int funcion(int a, int b) {
            if(a+b > 10) {
                return -1;
            }
```

Definición de JoinPoint

Aspecto Funcional: Weaving BEFORE

Aspecto Funcional: Weaving AFTER

Figura 6 - Ejemplo de weaving en AspectC#

En la Figura 5 se puede observar la estructura del descriptor de aspectos, en el que se define en primer lugar la ubicación del ensamblado base y el de los aspectos (*TargetBase* y *AspectBase*). Cada método de los aspectos que se

quiera enlazar (el equivalente en *AspectJ* a los *advices*) se define mediante un nodo *AspectMethod*, y la definición de dónde se enlaza el aspecto con el método base (los *pointcuts*) se especifica mediante un nodo <*Target*><*Method*>. En este nodo se define para cada método del código base el aspecto que se va a aplicar y el tipo (*after, before* o *around*).

En la Figura 6 se detalla el código generado por el compilador de AspectC# antes de generarse el ensamblado correspondiente. Únicamente resaltar cómo se inserta el código correspondiente a los aspectos antes/después del código base, y cómo se introducen los *joinPoints* mediante los cuales el aspecto puede obtener información de dónde ha sido ubicado su código.

No obstante, el proyecto, al igual que los anteriores, ha sido abandonado en 2002, y también presenta algunos inconvenientes importantes. La principal desventaja es que utilizar un *parser* propio de código fuente, a no ser que incorpore toda la semántica de .NET existente para C#, limita la expresividad del programador, pues si utiliza estructuras no definidas en el *parser* el ensamblado no podrá generarse produciendo diversos errores de compilación. Además, si dicho *parser* incorporase toda la funcionalidad ofrecida por .NET, tarea inabordable pues equivaldría a generar un compilador casi idéntico al de la plataforma y estaría condenado a quedarse desfasado tan pronto apareciesen nuevas versiones de la plataforma con nuevas estructuras de código. Aparte de este inconveniente, AspectC# no es una plataforma apta para desarrollar aplicaciones AOP, debido a que se abandonó en un estado temprano de desarrollo. La documentación es escasa y el compilador no dispone de la funcionalidad mínima que cabría esperar, como permitir añadir librerías externas, tampoco soporta estructuras básicas como: anidaciones de clases, estructuras, delegados, eventos y atributos.

2.4 JAsCo.NET

JAsCo[8] [Suv03] es una tecnología que, siguiendo la tendencia actual, trata de unir el Diseño Basado en Componentes (DSBC) y el Desarrollo de Software Orientado a Aspectos (DSOA). Diseñado originariamente para Java, cuya versión se encuentra en mayor estado de desarrollo, se está portando a dispositivos Java embebidos, se han creado una serie de plugins para *eclipse*, y dispone de un prototipo para .NET: JAsCo.NET [Vers03]. Este prototipo es el analizado en esta sección, y es importante resaltar que únicamente tiene implementadas las funcionalidades básicas.

[8] En http://ssel.vub.ac.be/jasco/files/JAsCo.ppt se puede encontrar una presentación en diapositivas bastante detallada de gran parte de la funcionalidad ofrecida por JAsCo, aunque en su versión para Java.

2.4.1 Funcionalidad

A diferencia de las tecnologías AOP anteriores, introduce tres nuevas conceptos semánticos: los aspectos, los *hooks* y los conectores, que serán descritos en detalle a continuación.

2.4.1.1 Aspectos y Hooks

Un aspecto en JAsCo es un concepto que, como define la filosofía AOP, está concebido para ser reutilizable en diferentes contextos de ejecución. Además, JAsCo permite el *polimorfismo aspectual*: el *weaving* de los aspectos con el código base es completamente dinámico, es decir, pueden agregarse/eliminarse aspectos en tiempo de ejecución sin que para ello deba detenerse la aplicación. Esto es un requerimiento fundamental para una tecnología que ha sido diseñada para el campo de los Servicios Web, en el que la disponibilidad del servicio es prioritaria. Gracias a esto, si un aspecto que implementa la conexión con la Base de Datos no funciona adecuadamente, puede compilarse por separado y sustituir dinámicamente al que estaba en ejecución sin llegar a detener la ejecución de la aplicación servidora.

Un aspecto está formado por uno o varios *hooks*, o garfios, que definen la forma de enlazarse al contexto (los *pointcuts* de AspectJ), y el comportamiento frente a los distintos tipos de *advices*. Cada *hook* consta de dos partes:

- cuándo se activa (*pointcut*): Puede indicarse mediante el constructor del *hook* o mediante un método condicional *IsApplicable* que permite especificar condiciones más específicas y detalladas.

  El constructor especifica la signatura del método o métodos que van a provocar su activación. Los nombres de dichos métodos son abstractos, es decir, no se corresponden con los nombres de los métodos con los cuales se enlazarán en el código base, aunque sí que tienen la misma signatura. Dichos métodos abstractos están definidos en los conectores, que establecen el mapeado entre el código base y los aspectos, con el objetivo de aumentar la reutilización de los aspectos y su independencia del código base al que se van a enlazar.
- que hará (*advice*): Permite indicar el comportamiento del aspecto para los distintos casos *after, before* o *replace*, cuyo significado es el mismo que en las otras aproximaciones AOP. Será en los conectores donde se especifique si el enlace con el método del código base es de un tipo o del otro.

JAsCo presenta la ventaja de tener un lenguaje de definición de aspectos muy rico, a diferencia de las otras aproximaciones. Los aspectos pueden combinarse entre sí mediante relaciones de agregación o herencia, pues son ciudadanos de primer orden, y por ello pueden formar estructuras más complejas. También puedan acceder al contexto en el que se están

ejecutando, definir métodos abstractos que sean posteriormente definidos en los conectores, utilizar la reflexión o incluso definir transiciones de estados.

A modo de ejemplo, en la siguiente figura se muestra la definición de un aspecto sencillo en JAsCo.Net:

```
namespace Aspectos {

    class AspectoTest {

        hook ReemplazarHook {
                                    Definición de Hook
            ReemplazarHook(string metodo()) {
                execute(metodo);
            }

            Replace() {                                    Comportamiento de un Aspecto
                MessageBox.Show("Metodo reemplazado!","AspectoTest");
                Console.WriteLine("Metodo reemplazado por AspectoTest.ReemplazarHook");
                return null;
            }
        }

        hook InformacionHook {

            InformacionHook (string metodo()) {
                execute(metodo);
            }

            Replace() {
                string original = (string)calledmethod.Invoke(calledobject,null);
                original = original + " - Interceptado por un aspecto";
                return original;
            }
        }
    }
}
```

Figura 7 - Ejemplo de Aspecto JAsCo.NET

Como se puede observar, un aspecto está formado por uno o varios *hooks*, que definen en su constructor qué método se va a interceptar, así como qué hacer frente a cada posible *advice*, donde se define el comportamiento del aspecto.

2.4.1.2 Conectores

Una vez vista la parte reutilizable del modelo, a continuación queda por describir la parte que se encarga de unir dichos elementos con el código base: los conectores. Éstos definen la unión entre un hook y uno o varios elementos del código base, que pueden ser: métodos, eventos, objetos o clases. La unión se realiza mediante el uso de estructuras sintácticas como los símbolos comodín '*' o '?'.

Asimismo, pueden definirse relaciones de precedencia entre *hooks* asociados a un mismo conector, encadenamientos o combinaciones más complejas (mediante el uso de clases específicas que definen cómo los diferentes *hooks* pueden combinarse entre sí). También se permite el uso de la reflexión para conocer dinámicamente el contexto en el cual se ubica el conector: obtener la lista de *hooks* del conector, añadir dinámicamente un objeto para realizar

cualquier acción, comprobar si el conector está habilitado o deshabilitado, decidir qué *hooks* aplicar o no, etc.

A modo de ejemplo, en la siguiente figura se muestra un conector, y cómo define un *hook* asociándolo con un método específico del código base. La línea `hook.Replace()`especifica qué *advice* debe aplicarse.

```
using System.IO;
using System.Collections;
using Aspectos;

namespace Conectores {

    static connector Reemplazar {

        AspectoTest.ReemplazarHook hook = AspectoTest.ReemplazarHook(
             sring ProyectoJASCO.frmBase.EmitirMensaje() );

        hook.Replace();
    }
}
```

Figura 8 - Ejemplo de conector JAsCo.NET

2.4.2 Modelo de Ejecución de JAsCo

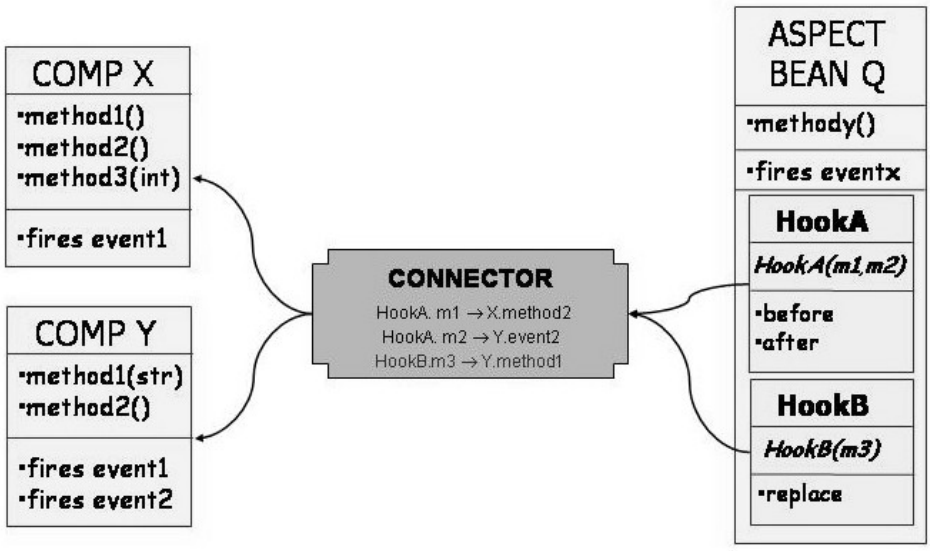


Figura 9 - Modelo de componentes de JAsCo

Una vez vistos los diferentes elementos que forman parte de JAsCo, en la Figura 9 se resume el modelo de componentes de JAsCo y la conexión entre ellos, desde los aspectos hasta el código base. Puede observarse cómo un aspecto Q, que representa una determinada funcionalidad, está formado por dos *hooks* que contienen los diferentes tipos de *advices* (*before, after* y *replace*). Cada uno de los *hooks* intercepta unos métodos determinados: *m1, m2* para el *hookA* y *m3* para el *hookB*. No obstante, el conector es el que

realmente establece el mapeado entre los métodos del código base con dichos *hooks* y permite separar la implementación de los aspectos del código base.

Para enlazar los conectores de forma dinámica con los componentes del código base en ejecución se utiliza el Registro de Conectores de JAsCo. Todos los componentes de la aplicación base son adaptados (se insertan llamadas a funciones externas (*traps*) a cada método público) para que las ejecuciones de sus métodos sean reenviadas a este Registro, el cual contiene una base de datos con todos los conectores cargados en el sistema y detecta qué *hooks* se pueden aplicar o no, en función del contexto de ejecución.

Por tanto, la plataforma de ejecución de JAsCo queda formada por los siguientes componentes:

- El compilador de aspectos (ficheros *.asp*) y el compilador de conectores (ficheros *.con*): se encargan de compilar el código con las definiciones de aspectos y conectores, respectivamente.
- *DotNetTransformer*: es la aplicación que se encarga de adaptar los ensamblados en .NET para que sean soportados por JAsCo, que se reduce a insertar *traps* en cada una de las ejecuciones de métodos publicos.
- *Introspector*: es la plataforma de ejecución de JAsCo, que permite insertar, eliminar, habilitar, o deshabilitar conectores en tiempo de ejecución a una aplicación determinada.

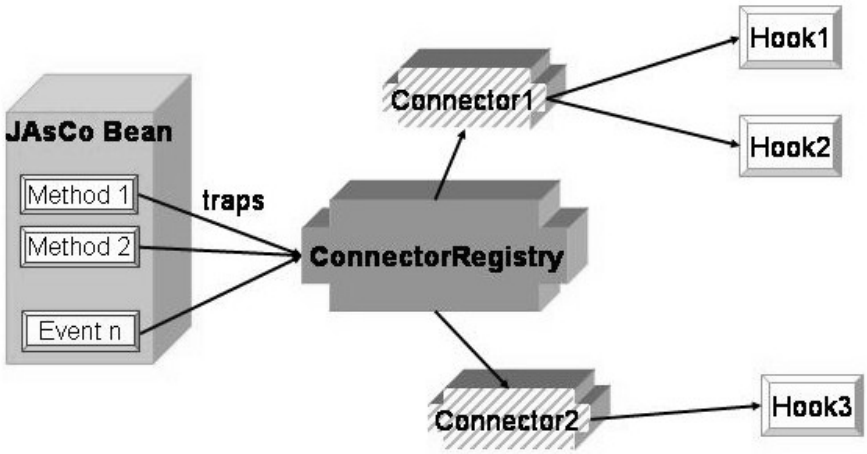


Figura 10 - Modelo de ejecución de JAsCo

2.4.3 Ventajas e Inconvenientes

JAsCo es una aproximación que intenta unificar las corrientes orientadas a Aspectos y basadas en Componentes, añadiendo el concepto de Conector. Además, dispone de un lenguaje muy expresivo: permite la combinación de aspectos, la utilización de mecanismos de herencia y una alta reutilización. Permite la incorporación de aspectos en tiempo de ejecución sin necesidad de

detener la aplicación objetivo y no necesita el código fuente del código base (por lo que no afecta a los COTS).

No obstante, también tiene inconvenientes a tener en cuenta:

- Extiende el lenguaje utilizado con nuevas estructuras sintácticas
- El registro de conectores se puede realizar por código mediante llamadas a la API del *Introspector*, pero esta funcionalidad no está aún implementada en .NET.
- La versión disponible para .NET es aún muy prematura y sólo incluye la funcionalidad básica. Además, los compiladores de componentes y conectores son incompletos en cuanto a flexibilidad sintáctica (por ejemplo, para definir métodos no se permite situar las llaves en la línea siguiente a la que define el método, y esto no está documentado). Tampoco se especifica qué funcionalidad del modelo completo (la versión de Java) se encuentra ya implementada, lo que dificulta su aprendizaje y uso.

2.5 Loom.NET

Loom.NET [Sch00], desarrollado por Wolfang Schult y Andreas Polze en la Universidad de Potsdam, Alemania, se diferencia de las demás tecnologías en la forma de definir los aspectos. Mientras que las analizadas hasta ahora utilizaban un lenguaje extendido o el mismo lenguaje en el que se programaba la aplicación base, Loom.Net utiliza plantillas de código reutilizables.

Estas plantillas o patrones en realidad son código C# con etiquetas que indican dónde se situarán las definiciones de clases, métodos y propiedades. Estas etiquetas serán sustituidas por los elementos correspondientes del código base cuando el usuario enlace las plantillas con el código fuente. Como ejemplo, a continuación se muestra una plantilla para definir el constructor de una clase, que se limita a mostrar un mensaje por pantalla:

```
/*[PROTECTION]*/ /*[CLASSNAME]*/( /*[PARAMDECLARATION]*/ ) :
                          base (/*[PARAMLIST]*/) {
     System.Windows.Forms.MessageBox.Show("Se ha llamado al constructor
          de ClaseAux");
}
```

El significado de las etiquetas es bastante intuitivo:

- [Protection] indica el nivel de protección de la clase
- [Classname] indica el nombre de la clase
- [ParamDeclaration] indica la definición de los parámetros del constructor
- [ParamList] es una lista de tipos de parámetros

Otro posible ejemplo sería la definición de un método, cuya apariencia es la siguiente:

```
public new virtual /*[RESULTTYPE]*/ /*[METHODNAME]*/ (
/*[PARAMDECLARATION]*/ ) {
        /*[RETVALINIT]*/
        System.Windows.Forms.MessageBox.Show("Ejecucion del aspecto en
el contexto de /*[METHODNAME]*/" );
        /*[RETVALASSIGN]*/base./*[METHODNAME]*/(/*[PARAMLIST]*/) + " "
+ /*[CADENA]*/;
        System.Windows.Forms.MessageBox.Show("El mensaje ya se ha
enviado hacia el cliente" );
        /*[RETVALRETURN]*/
}
```

Este método se limita a mostrar un mensaje que indica qué método se está ejecutando y a componer una cadena que será devuelta por el método. Las etiquetas utilizadas tienen el siguiente significado:

- [ResultType] indica el tipo de retorno del método
- [MethodName] indica el nombre del método definido
- [RetValInit] se sustituye por la inicialización del valor de retorno
- [RetValAssign] se reemplaza por una asignación a la variable de retorno
- [RetValReturn] equivale a devolver el contenido de la variable de retorno.

La sustitución de las etiquetas por el código correspondiente se realiza mediante las funcionalidades ofrecidas por .NET para expresiones regulares, ubicadas en el espacio de nombres *System.Text.RegularExpression*. El compilador sustituye todos los patrones encapsulados por los comentarios de C# '/*' y '*/' por el código necesario del código base. De esta manera, los aspectos se definen mediante patrones de código reutilizables.

Loom.NET no trabaja sobre código fuente base, sino que trabaja a nivel de ensamblados. Mediante una aplicación gráfica (ver Figura 11 - Interfaz de Loom.NET) el usuario elige el ensamblado base, y por reflexión se obtienen todos los métodos y propiedades públicas de dicho ensamblado. Los aspectos son cargados por la aplicación desde una librería de plantillas. Para añadir un nuevo aspecto a dicha librería, el desarrollador ha de añadir manualmente la definición XML del aspecto (ver Figura 12 - Especificación XML de un aspecto en Loom.NET) en la que se especifican parámetros adicionales que la plantilla pueda requerir. Después el usuario enlaza los aspectos con los métodos o clases del ensamblado, por reflexión se rellenan las plantillas de los aspectos con el código necesario y finalmente, se genera el ensamblado.

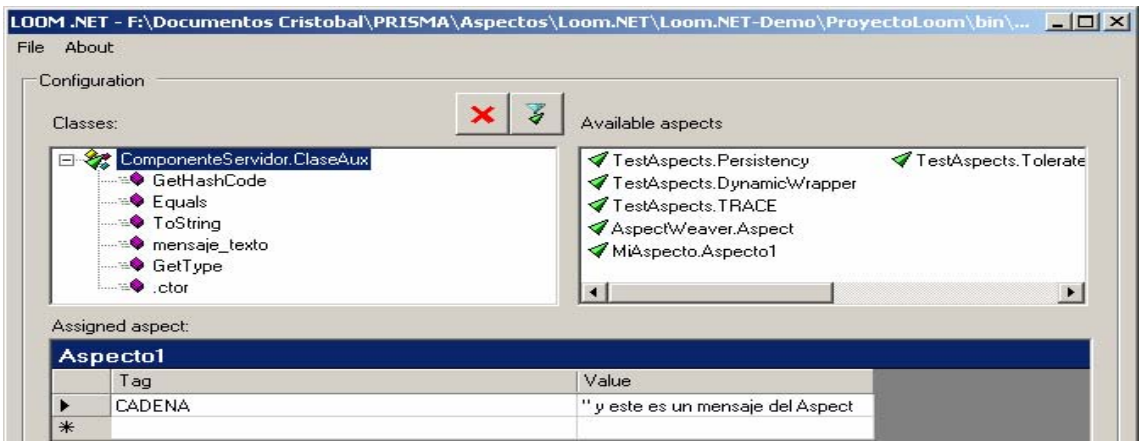


Figura 11 - Interfaz de Loom.NET

```
  <Aspect Name="MiAspecto.Aspecto1" Description="Simplemente engancha una cadena a otro metodo">
    <Tag Type="file" Name="CTOR">ctor_MiAspecto-Cadena.tpl</Tag>
    <Tag Type="file" Name="METHOD">method_MiAspecto-Cadena.tpl</Tag>
    <Tag Type="text" Name="BASECLASS">/*[BASENAMESPACE]*/./*[CLASSNAME]*/</Tag>
    <Tag Type="parameter" Name="CADENA">" y este es un mensaje del Aspecto ;-)"</Tag>
  </Aspect>
</Templates>
```

Figura 12 - Especificación XML de un aspecto en Loom.NET

Finalmente, el código generado tras enlazar los aspectos con el ensamblado base(etapa previa a la generación del ensamblado), es el siguiente:

```
///_____________________________________________________________________
/// <autogenerated>
///     This class is generated by LOOM .NET
///     (C) 2001-2003 by Wolfgang Schult
///     http://www.dcl.hpi.uni-potsdam.de
///
///     Changes of this file may cause incorrect
///     behavior and will be lost on next generation.
/// </autogenerated>
///_____________________________________________________________________

namespace Aspecto1Proxy {
  public class ClaseAux:ComponenteServidor.ClaseAux {
    public ClaseAux(): base() {
      System.Windows.Forms.MessageBox.Show("Se ha llamado al
          constructor de ClaseAux");
    }

    public new virtual System.String mensaje_texto() {
      System.String _RetVal=null;
      System.Windows.Forms.MessageBox.Show("Ejecucion del aspecto en
          el contexto de mensaje_texto" );
      _RetVal=base.mensaje_texto() + " " + " y este es un mensaje del
          Aspecto =====.;-)";
      System.Windows.Forms.MessageBox.Show("El mensaje ya se ha
          enviado hacia el cliente" );
      return _RetVal;
    }
  }
}
```

Como se puede observar, mediante mecanismos de herencia se genera una clase derivada (*Aspecto1Proxy.ClaseAux*) de la clase base (*Componente Servidor.ClaseAux)* que sobrecarga los métodos públicos de la clase base.

No obstante, esta aproximación está más orientada a la definición de *proxies* que a los aspectos concebidos como *concerns*. Como se puede observar en el código anterior, el compilador no enlaza los aspectos con el código base, sino que genera un nuevo componente que encapsula a la clase correspondiente del código base e implementa el código del aspecto. El cliente de la aplicación llamará a esta clase *proxy* generada, y en función del comportamiento definido para el aspecto, decidirá si llama o no al código base. Además, presenta limitaciones importantes como que sólo se puede enlazar un aspecto por clase (o método), y además no permite la conexión / desconexión dinámica de aspectos (*polimorfismo aspectual*).

Por ello, se puede afirmar que esta aproximación no es una AOP estricta como se ha visto hasta ahora, sino que únicamente se limita a encapsular código para integrar los aspectos.

2.6 Rapier-Loom.NET

Por otra parte, Rapier-Loom.NET [Sch02] sí que está diseñado para soportar el polimorfismo aspectual, y además no extiende el lenguaje.

En primer lugar, los aspectos se definen en cualquier lenguaje de .NET, mientras que el código base puede ser código fuente o un ensamblado. Cada aspecto se corresponde con una clase que hereda de *Loom.Aspect*, y mediante el uso de atributos se declaran las siguientes propiedades:

- Atributos de Enlace (*Interweaving*):

  Permiten señalar la inclusión/exclusión de métodos de la clase objetivo con la que se va enlazar el aspecto (corresponderían a los *pointcuts* de AspectJ). El filtrado de métodos puede realizarse por el nombre, por el tipo de la clase a la que pertenece, por la interfaz que implementa o por atributos que tenga definidos.

- Connection Points:

  Indican la forma de enlazarse al método (*advices* de AspectJ). Permite los típicos operadores *before, after, instead*, además de *after returning* y *after throwing*. Con *After returning* sólo es ejecutado el aspecto si el método devuelve un valor y en el caso de *After throwing* el aspecto es ejecutado si es lanzada una excepción.

- Acceso al contexto:

  En el caso de *weavings instead*, se permite al aspecto acceder al contexto en el que se está ejecutando, con lo que es posible acceder a la

instancia del objeto en el que se iba a ejecutar el método, cambiar los parámetros que se pasarán al método e invocarlo mediante la sentencia *invoke*, etc.

A modo de ejemplo, se muestra a continuación un fragmento de código con la definición de un aspecto. En él se puede comprobar cómo se define el *ConnectionPoint* para indicar que el método que lo sigue se va a enlazar con cualquier método que se llame "IniciarFuncionalidad", así como el tipo de *weaving* que se va a realizar, mediante el atributo `[Call(Invoke.Instead)]`. También puede observarse cómo se accede al contexto del objeto en el que se ejecute el aspecto para cambiar o consultar propiedades, `Context.Instance` o `Context.Invoke` para ejecutar el método que se iba a ejecutar y modificar el valor que va a retornar.

```
// Hereda de Loom.Aspect
public class AspectoFuncional : Loom.Aspect {
    const string version = "AspectoFuncional v.01";

    // Estos atributos indican el nombre del método con el que debe enlazarse
    //   (la clase se especificará en el Weaving)
    [Loom.ConnectionPoint.Include("IniciarFuncionalidad")]
    [Loom.Call(Invoke.Instead)]
    // El código del aspecto propiamente dicho
    public string Identificarme() {
        string cadena;

        MessageBox.Show("Estoy en el metodo AspectoFuncional.Identificarme" +
            " y voy a llamar al CodigoBase", "AspectoFuncional");

        // Podemos acceder al contexto del objeto al que estamos enlazados
        // Accedemos a la clase base para modificar un atributo
        ((ClaseBase) Context.Instance).nombre = "¡¡¡Me han cambiado el nombre!!!";
        // Llamamos otra vez al código base para observar el resultado
        cadena=(Context.Invoke()).ToString() ;

        MessageBox.Show("Tras llamar a ClaseBase me ha devuelto: " + cadena,
            "AspectoFuncional");

        return "Y ahora me presento. Soy: " + version ;
    }

    // Ejemplo de función accesible desde fuera (sin weaving)
    //   De esta forma, el aspecto puede proporcionar funcionalidad adicional que pueda
    //   ser requerida por otras clases.
    public string Informacion() {
```

Figura 13 - Ejemplo de aspecto en Rapier-Loom.Net

En segundo lugar, permite instanciar dinámicamente los aspectos (y eliminarlos) en tiempo de ejecución gracias a que el *weaving* con el código base se realiza por código, a diferencia de las aproximaciones anteriores, que en su mayoría requerían de un compilador adicional para enlazar los aspectos con el código base. Para ello, define un nuevo operador *Weaver.CreateInstance* que sustituye al operador *new* en la creación de objetos, y que permite asociar una clase, método o propiedad a un aspecto. La sintaxis de este método es la siguiente:

```
public static object CreateInstance(
   Type classtype,
```

```
    object[] args,
    Aspect[] aspectarr
);
```

El primer parámetro es el tipo del objeto del código base que se quiere crear, el segundo parámetro contiene la lista de argumentos que requiere el constructor de dicho objeto y el tercer parámetro define la lista de aspectos que se quiere enlazar a dicho objeto. Como resultado, devuelve una instancia del objeto interceptado por los aspectos.

Para ilustrar esto, en la siguiente figura se muestra el código correspondiente en el que se realizaría el *Weaving* entre los aspectos y el código base.

```
// Aquí realizamos el Weaving con los Aspectos
public string LlamarComponenteConAspectos() {
    Aspect[] listaAspectos;
    string cadena;

    // Primero creamos los dos aspectos
    Aspectos.AspectoDistribucion dis = new AspectoDistribucion();
    Aspectos.AspectoFuncional asp = new AspectoFuncional();
    // Los metemos en un vector
    listaAspectos = new Aspect[2];  listaAspectos[0]=dis; listaAspectos[1]=asp;

    // Ahora enlazamos los aspectos con la clase Base
    // El orden de llamadas sigue el orden del vector, de forma que sólo se
    //   inicia el segundo si el primero le pasa el control al método que sustituye.
    ComponenteEstructural.ClaseBase obj =
        (ComponenteEstructural.ClaseBase) Loom.Weaver.CreateInstance(
            typeof(ComponenteEstructural.ClaseBase),null,listaAspectos);

    // Ejecutamos el método de la clase Base que tiene código enlazado
    return obj.IniciarFuncionalidad();
```

Figura 14 - Ejemplo de Weaving en Rapier-Loom.Net

Además, también permite enlazar varios aspectos entre sí, a diferencia de Loom.Net, de forma que los aspectos puedan comunicarse entre sí, saber si existen determinados aspectos enlazados al método actual, ejecutarse unos antes de otro, etc. Por ejemplo, en el siguiente código se puede observar cómo un aspecto obtiene por reflexión los aspectos que están enlazados al mismo método al que él está enlazado y acceder a propiedades o métodos:

```
public class AspectoDistribucion : Loom.Aspect {
  // ...
  Aspect[] listaAspectos = Weaver.GetAspects(Context.Instance,
       typeof(Aspectos.AspectoFuncional));
  ((Aspectos.AspectoFuncional) listaAspectos[0]).Informacion2("hola");
  // ...
}
```

No obstante, todo método del código base susceptible de tener aspectos asociados debe definirse como *virtual* o ser llamado a través de una interfaz. Esto es debido a la forma en que Rapier-Loom.Net realiza el entretejido entre código base y aspectos. En el CLR de .NET, por cada clase existe una tabla que almacena los punteros a los métodos que define. Con el objetivo de

permitir la sobrecarga de métodos en la herencia, los métodos definidos como virtuales se almacenan en dicha tabla con un indicador que permite su modificación, para que las clases derivadas puedan sustituir la referencia de dicho método con el puntero al método sobrecargado. Rapier-Loom.Net aprovecha este mecanismo para que dichas entradas apunten a los métodos de los aspectos. Por ello, otra limitación derivada de esto, es que tampoco pueden enlazarse aspectos con métodos estáticos. Los autores recomiendan que, para superar estas limitaciones, utilizar la idea de *proxy* implementada en Loom.Net.

Además de este pequeño inconveniente, los aspectos no pueden desenlazarse dinámicamente sin detener la ejecución del objeto que los tiene asociados. Para ello, el objeto ha de ser vuelto a crear. Por otra parte, no hay una separación clara de los *weavings* y los aspectos, sino que los *pointcuts* (o connection points) son definidos en el mismo código del aspecto, degradando la reutilización de los aspectos.

2.7 SetPoint!

SetPoint! [SetP04] nace como Proyecto de Tesis de Licenciatura en la Universidad de Buenos Aires, y pese a encontrarse en fase muy prematura, ofrece un enfoque totalmente distinto a los vistos anteriormente. Basándose en la idea de la Web Semántica, introduce el concepto de *Semantic Pointcuts*, en un intento de unir la programación orientada a aspectos y la semántica para realizar el enlazado entre aspectos y código base.

El objetivo perseguido es etiquetar de alguna forma el código base para que los aspectos se enlacen (estática o dinámicamente) mediante predicados lógicos. Para ello, se define el término *Descriptor Semántico* (*setPoint*) como aquella marca en el código que permite asignarle una característica particular, de forma equivalente a los *Join Points* de AspectJ. Este Descriptor Semántico equivaldría a un atributo de .NET.

De esta forma, los *pointcuts* pasan a ser predicados lógicos sobre descriptores, en lugar de utilizar mecanismos de filtrado basados en la sintaxis del código como los vistos hasta ahora, que creaban una dependencia sintáctica del weaving con el código base. Por ejemplo, en las otras aproximaciones el *weaving* se crea en la mayoría de los casos en base al nombre del método a enlazar, mediante el uso o no de comodines. Si quisiéramos enlazar todos los métodos “Get” del código base (aquellos que devuelven los valores de las variables privadas de una clase) para auditar su acceso mediante un aspecto, en , Rapier-Loom.Net se haría de esta forma:

```
// CODIGO DEL ASPECTO
[Loom.ConnectionPoint.Include("Get*")]
public void AuditGetterMethods() { /* ... */ }
```

Mientras que en SetPoint se traduciría en el predicado lógico: "Asociar el aspecto *AuditGetterMethods* a todos los métodos etiquetados como *[GetterMethod]*", cuya traducción a .NET[9] podría ser la siguiente: en primer lugar se insertarían los *SetPoints* en los métodos "Get" del código base,

```
// CODIGO BASE
// ...
[GetterMethod]
public int GetStatus() { /* ... */ }
// ...
```

y después se definirían los *pointcuts* en el aspecto:

```
// CODIGO DEL ASPECTO
[SetPoint.SelectAll("GetterMethod")]
public void AuditGetterMethods() { /* ... */ }
```

Con esto, se consigue una mayor independencia del código, pues si se cambia el código base sin eliminar los *SetPoints*, el enlace de aspectos no se vería afectado.

Un inconveniente que se puede observar es que los *SetPoints* se agregan al código fuente del código base. No obstante esto se puede solucionar si los descriptores semánticos se implementan como atributos .Net y, gracias a la reflexión y a la emisión de código que brinda la plataforma, no habría problemas en volver a generar el ensamblado con los *SetPoints* ya aplicados.

Por otra parte, este proyecto se encuentra en una fase muy temprana y hay escasa documentación al respecto. La sintaxis del lenguaje de definición de *SetPoints* no está definido, como tampoco la forma de realizar los *pointcuts*, por lo que no se han podido efectuar pruebas sobre el prototipo para evaluar características de reutilización, usabilidad y posibles carencias.

[9] Los siguientes ejemplos únicamente son aproximaciones sobre la sintaxis que podría tener en .NET, puesto que SetPoint! aún no tiene publicada ninguna sintaxis específica, únicamente los conceptos.

2.8 Otras aproximaciones

Las tecnologías mostradas en las secciones anteriores no son las únicas existentes, pues la programación orientada a aspectos aún está dando sus primeros pasos y constantemente aparecen nuevos proyectos de investigación. A continuación se citan brevemente otras aproximaciones existentes pero que no han sido analizadas en profundidad por distintas cuestiones: no disponer de prototipos o documentación suficiente, por no aportar nuevos conceptos, etc.

2.8.1 AOP.NET

AOP.NET [Schm02] fue un proyecto de Siemens del año 2002 en el que se implementaron conceptos de AOP usando .Net.

Realiza el *weaving* de clases en ejecución a nivel del CLR, trabajando con la interfaz *.NET Profiling API* no manejada y modificando en tiempo de ejecución las tablas donde se ubican las definiciones de los métodos (de forma similar a *Rapier-Loom.Net*) para interceptar el código y redirigirlo a los aspectos.

Esta aproximación trabaja a muy bajo nivel con el CLR de .Net, no dispone de un prototipo accesible ni documentación al respecto y además, el proyecto no parece haber seguido adelante.

2.8.2 Weave.NET y SourceWeave.NET

Estas dos aproximaciones son proyectos del mismo grupo de investigación que desarrolló *AspectC#*, del cual *SourceWeave* es una evolución. *SourceWeave.NET* [Jack04] es una aproximación que siguiendo con AspectC# intenta realizar la unión de aspectos con código base a nivel del código fuente utilizando para ello la plataforma .NET. La idea es definir el código base, al igual que los aspectos, en cualquier lenguaje soportado por .NET y a través de un descriptor XML que define el *weaving*, generar el código fuente resultante de unir los aspectos al código base. Esto plantea los mismos problemas que aparecían en AspectC#, entre ellos el no soportar el polimorfismo aspectual.

Por otra parte, de forma muy similar a CLAW y AOP#, Weave.Net [Laff03] genera el ensamblado destino a partir de un ensamblado con el código base, otro con el código de los aspectos y una especificación XML con los *weavings* a realizar. Por ello, también presenta los mismos problemas que dichas aproximaciones.

2.8.3 EOS

EOS [Raj03] es otra aproximación cuya aportación reside en el concepto de *Instance-Level Aspects*. Con este término definen a los aspectos como entidades que se enlazan a instancias de clases, diferenciando entre ellas si aplicar el *weaving* o no mediante diferentes construcciones sintácticas. Esto permite decidir en tiempo de ejecución si un aspecto se va a activar para un determinado objeto o no. Sin embargo, la asignación de aspectos a clases se realiza en tiempo de compilación. Para llevar a cabo estos conceptos, proporciona una extensión al lenguaje C# para .NET y por ello requiere de un compilador que está aún en fase beta.

2.8.4 AspectDNG

AspectDNG [Dng04] es una aproximación que ha sido desarrollada recientemente por la comunidad *OpenSource* y presenta otra alternativa basada en la unión de ensamblados de código base y aspectos, mediante la definición de los *weavings* en XML. La diferencia radica en que se desensamblan los diferentes *assemblies* a un lenguaje intermedio llamado *ILML,* al que posteriormente se le aplica el *weaving*. Después se genera el ensamblado final con todo el código enlazado.

2.9 Conclusiones

Una vez mostradas las principales tecnologías orientadas a Aspectos disponibles en .NET, a continuación se muestra una tabla resumen en la que se pueden comparar cómo las diferentes tecnologías aplican el *weaving*, cómo definen los aspectos, o qué ideas nuevas aportan al AOP.

AspectC# y SourceWeave.NET	***Nivel de Unión***	Código Fuente (Parser propio + CodeDom)
	Weaving	Parsea el código fuente uniendo código Base con Aspectos, basándose en un fichero XML para los *pointcuts*, generando código fuente ya enlazado.
	Definición Aspectos	No extiende el lenguaje, lo utiliza para definir los aspectos.
	Ideas aportadas	Definición de *pointcuts* de forma externa al código. Separación clara de Código Base y Aspectos.
	Dinámico	☒ La compilación de aspectos es estática.

Loom.Net	***Nivel de Unión***	A través del CLR (usando Introspección y Reflexión)
	Weaving	Los Aspectos encapsulan al código Base, y el cliente llama al nuevo componente, a modo de Proxy.
	Definición Aspectos	Se definen mediante plantillas reutilizables y XML para realizar el *weaving*. Extiende el lenguaje.
	Ideas aportadas	Utilización de expresiones regulares en el código para las sustituciones de aspectos y uso de plantillas.
	Dinámico	☒ La generación de aspectos se realiza en compilación
Rapier-Loom.Net	***Nivel de Unión***	La intercepción de código base se realiza modificando la tabla de métodos virtuales.
	Weaving	Mediante atributos se configuran los *pointcuts*. El *weaving* se realiza en tiempo de ejecución al crear una nueva instancia de una clase del código base.
	Definición Aspectos	Añade estructuras sintácticas adicionales para realizar los *weavings* en tiempo de ejecución, como los atributos.
	Ideas aportadas	Definición de un constructor para unir instancias de Aspectos y clase Base. Utilización de atributos para indicar los *pointcuts.*
	Dinámico	☑
JAsCo.Net	***Nivel de Unión***	A través del CLR (Parser propio + modificación posterior del ensamblado generado)
	Weaving	Los Aspectos son reutilizables, el weaving se realiza a través de los Conectores. Al componente base se le insertan *traps* capturadas por una aplicación intermedia.
	Definición Aspectos	Extiende el lenguaje, pero añade mayor expresividad para el uso de los aspectos.
	Ideas aportadas	Definición de *pointcuts* mediante otra entidad: los conectores. Los Aspectos son abstractos, heredables y separables en estados.
	Dinámico	☑
SetPoint!	***Nivel de Unión***	A través del CLR para etiquetar ensamblados ya compilados. Si se dispone del código fuente, a través de atributos.
	Weaving	Se establecen Meta-Etiquetas en el código Base, que permitirán definir los *pointcuts* por su contenido semántico
	Definición Aspectos	No extiende el lenguaje. Se definen en el mismo lenguaje objetivo.
	Ideas aportadas	El *weaving* está basado en funciones que enlazan el código base a los aspectos según sea su contenido semántico.
	Dinámico	☑

Tabla 1 - Comparación entre las diferentes tecnologías AOP en .NET