Información Tecnológica
Vol. 28(5), 37-44 (2017)
doi: 10.4067/S0718-07642017000500006

# Comparación de Uso del Patrón de Diseño Decorator y la Programación Orientada a Aspectos en .NET para Modularizar Incumbencias Cruzadas

**Cristian A. Pereira-Vásquez[(1)]*; Cristian L. Vidal-Silva[(2)] y Marco A. Morris[(1)]**
(1) Servicios Profesionales de Informática Morris y Opazo, Antonio Varas 920, Oficina 35, Temuco-Chile
(2) Facultad de Ingeniería, Ingeniería Informática, Universidad Autónoma de Chile, 5 Norte 417, Talca-Chile
(e-mail: cpereira@morrisopazo.com, cristian.vidal@uautonoma.cl; marco@morrisopazo.com)

* Autor a quien debe ser dirigida la correspondencia

*Recibido Dic. 28, 2016; Aceptado Mar. 3, 2017; Versión final Mar. 22, 2017, Publicado Oct. 2017*

**Resumen**

En la búsqueda de desarrollo de soluciones software .NET modulares, este trabajo describe e ilustra como producir soluciones .NET modulares mediante el uso del patrón de diseño de software orientado a objetos Decorator y la herramienta de Programación Orientada a Aspectos (POA) PostSharp para la modularización de incumbencias cruzadas de una aplicación. La aplicación es un ejemplo base y tradicional de incumbencias cruzadas (ejemplo de logging). Este trabajo presenta soluciones de logging con el uso de Decorator y PostSharp para modularizar asuntos cruzados asociados. También detalla las ventajas y desventajas de ambas soluciones. Así mismo, este trabajo puntualiza detalles de POA presentes en PostSharp junto con plantear el uso de PostSharp y Decorator para lograr soluciones con un mayor nivel de modularidad.

*Palabras clave: POA; .NET; Decorator; PostSharp; programación orientada a aspectos*

# Comparing of the Use of Design Pattern Decorator and Aspect-Oriented Programming in .NET to Modularize Crosscutting Concerns

**Abstract**

Looking for modular .NET software solutions, this article describes and illustrates how to produce modular .NET solutions through the use of the object-oriented software design pattern Decorator and the aspect-oriented programming (AOP) tool PostSharp for the crosscutting concerns modularization of an application. The application is a base and traditional crosscutting concerns example (logging example). This article presents solutions of logging by the use of Decorator and PostSharp for the modularization of its crosscutting concerns. It also gives details of the advantages and disadvantages of both solutions. Furthermore, this article specifies AOP information that is part of PostSharp and proposes applying PostSharp and Decorator to achieve higher modularity level solutions.

*Keywords: AOP; .NET; Decorator; PostSharp; aspects oriented programing*

## INTRODUCCIÓN

En programación, las incumbencias cruzadas representan características de software cuya modularización y aislamiento es compleja o en algunos casos no es posible ya que son funcionalidades que son parte de manera implícita de otras funcionalidades (Seemann, 2011). Así, la implementación de incumbencias cruzadas se propaga entre diferentes módulos de una solución software (Kiczales et al., 1997; Gradecki y Lesiecki, 2003). Tal y como indican (Groves, 2013; Seemann, 2011), ejemplos típicos de incumbencias cruzadas lo son la persistencia, la auditoría, la seguridad, el manejo de errores, transacciones y el logging o registro definido de acciones. Una referencia que aporta elementos interesantes no considerados en este artículo es (Seemann, 2011).

La figura 1 muestra un ejemplo simple de logging en .NET, con clases Ejemplo y Procesador, donde la clase principal Ejemplo crea una instancia de Procesador la que hace uso de una objeto Logger, instancia de la clase ILog para un registro de acciones o logging. Este ejemplo representa la aplicación base de este estudio.

```
using System;
using log4net;

namespace Logging{
 class Ejemplo{
  static void Main(string[] args){
    log4net.Config.XmlConfigurator.Configure();

    var procesador = new Procesador();
    procesador.Execute();
    Console.ReadLine();
  }
 }
```

```
class Procesador {
 private static readonly ILog Logger =
 LogManager.GetLogger(typeof(Procesador));

 public void Execute() {
  Logger.Info("Inicio de Logging");
  Console.WriteLine("Inicio");

  Console.WriteLine("Aplicación Ejemplo!");

  Logger.Info("Fin de Logging");
  Console.WriteLine("Fin");
 }
}
}
```

Fig. 1: Ejemplo de Logging .NET.

Tal y como se aprecia en la figura 1, el método Main() de la clase Ejemplo crea una instancia de la clase Procesador, del cual se ejecuta el método Execute(), dicho método presenta dos responsabilidades: 1) Imprimir un mensaje por pantalla; y 2) Tener un registro en archivos de Log de la librería Log4Net (Log4net, 2016). Entonces, la clase Procesador presenta una violación del principio de única responsabilidad (Single Responsability Principle SRP) (Martin, 2002), lo cual es un ejemplo de incumbencia cruzada (crosscutting concern) (Groves, 2013). De acuerdo a (Gamma et al., 1994), Decorator es un patrón de diseño de software orientado a objetos mediante el cual se definen componentes los cuales se encierran en otro objeto llamado Decorator, donde el decorador se ajusta a la interfaz de dichos componente para que su presencia sea transparente para los clientes del componente (Gamma et al., 1994; Martin, 2002; Millett, 2010). Gracias a esto, el patrón de diseño Decorator puede manejar solicitudes de acción de sus componentes, así como también realizar acciones adicionales. Este trabajo revisa como modularizar incumbencias cruzadas mediante Decorator.

Según (Vidal et al., 2012; Vidal et al., 2014; Vidal y Villarroel, 2014; Bodden et al., 2014), la Programación Orientada a Aspectos (POA), gracias a la separación de incumbencias cruzadas, soporta un conjunto de principios de buen diseño orientado a objetos; entre ellos SRP y el principio de abierto-cerrado. Además, (Wampler, 2007) indica que el bajo acoplamiento y la alta cohesión son principios fundamentales y necesarios para lograr un diseño de software modular. Justamente, gracias a la separación de incumbencias, POA permite un alto nivel de cohesión y bajo acoplamiento entre módulos aconsejados y aspectos. Según (Seemann, 2011) un aspecto es sólo otra definición de incumbencia cruzada, razón por la cual cuando se habla de incumbencia cruzada se habla de POA y vice-versa.

Así, el principal objetivo de este trabajo es revisar la aplicabilidad del patrón de diseño Decorator y el uso de PostSharp (SharpCrafters, 2016) para la modularización del código de la figura 1, así como analizar el uso de ambos enfoques de solución para modularizar incumbencias cruzadas, en la búsqueda de soluciones .NET completamente modulares.

**DECORATOR**

Decorator (decorador) es un patrón de diseño permite añadir un nuevo comportamiento a un objeto dinámicamente a través de la composición; que corresponde al proceso de envolver una clase existente con una clase que extiende el comportamiento o estado (Groves, 2013; Log4net, 2016). La figura 2 (Gamma et al., 1994) muestra la estructura general de una solución con el patrón de diseño Decorator, la cual presenta diferentes componentes y relaciones. La tabla 1 describen los componentes de la figura 2.

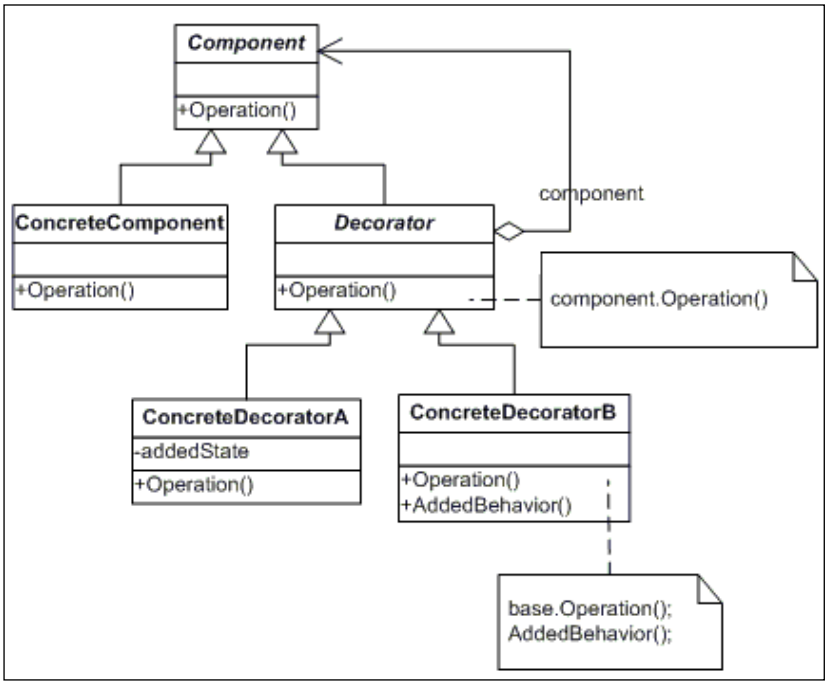


Fig. 2: Estructura de una Solución Decorator.

Tabla 1: Hipótesis del estudio

| *Componente* | *Definición* |
|---|---|
| Interfaz Component | Interfaz o clase abstracta que puede tener comportamiento dinámico. |
| Clase ConcreteComponent | Clase que implementa la interfaz Component para implementar responsabilidades propias de la clase. |
| Clase Decorator | Clase que implementa la interfaz Component y a su vez contiene una referencia a una instancia de dicha interfaz. Estructuralmente, esta clase a su vez es una interfaz para clases Decorator más específicas. |
| Clase Concrete Decorator (Clases ConcreteDecoratorA y ConcreteDecoratorB) | Clase que herada o implementa el Decorator para ampliar y especializar su funcionalidad. |

De acuerdo a (Millett, 2010), el patrón Decorator es necesario cuando hay necesidades de añadir comportamiento dinámicamente, así como eliminar responsabilidades a una clase. Así, la figura 3 muestran parte de la aplicación del patrón Decorator del ejemplo de la figura 1: la interfaz Component y la clase ConcreteComponent, respectivamente (interfaz IProcesador y clase Procesador). Nótese que el método Execute(..) esta vez recibe un parámetro para identificar el objeto que lo invoca.

Al analizar la solución en la figura 3, la clase Procesador ahora presenta sólo una responsabilidad, sin un registro de acciones o logging, para lo cual se requiere una clase Decorator y una clase ConcreteDecorator, ambas con una única responsabilidad tal y como se presenta en las figuras 4 y 5. La figura 6 muestra el programa principal del ejemplo junto con la salida del mismo. Entonces, se comprueba que mediante el uso del patrón de diseño Decorator se permite la inclusión de comportamiento adicional sobre métodos existentes de instancias de clases aconsejadas. Además, gracias a la propiedad de Herencia, una clase ConcreteDecorator puede incluir atributos y métodos adicionales, los cuales no se agregan directamente al objeto aconsejado.

```
namespace LoggingDecorator
{
    interface IProcesador{
        void Execute(string name);
    }
}
```

```
using System;

namespace LoggingDecorator
{
  class Procesador:IProcesador{
     public void Execute(string name) {
        Console.WriteLine("Hola! - {0}",name);
     }
  }
}
```

Fig. 3: Interfaz Component y Clase ConcreteComponent para Solución Decorator de Logging

```
using System;

namespace LoggingDecorator
{
 abstract class ProcesadorDecorator : IProcesador{
    private Procesador Logger;

    public ProcesadorDecorator(Procesador Logger) {
        this.Logger = Logger;
    }

    public virtual void Execute(string name) {
        this.Logger.Execute(name);
    }
  }
}
```

Fig. 4: Clase Decorator para Solución Decorator de Logging

```
using System;
using log4net;

namespace LoggingDecorator{
 class ProcesadorLoggingDecorator:
              ProcesadorDecorator{

   private readonly ILog _Logger =
    LogManager.GetLogger(typeof(ProcesadorLoggingDecorator));

   public override void Execute(string name){
     _Logger.Info("Inicio de Logging");
     Console.WriteLine("Inicio Decorator");

     base.Execute(name);

     _Logger.Info("Fin de Logging");
     Console.WriteLine("Fin Decorator");
   }
 }
}
```

Fig. 5: Clase ConcreteDecorator para Solución Decorator de Logging

## PROGRAMACIÓN ORIENTADA A ASPECTOS

Programación Orientada a Aspectos (POA) busca el comportamiento modular de clases, para lo cual define clases ingenuas aconsejables y modula incumbencias cruzadas mediante aspectos (Kiczales et al., 1997). De esta forma, soluciones POA permiten que las clases respeten el principio de única responsabilidad, además de definir una interacción de aspectos y clases aconsejadas. POA nace junto a AspectJ, una versión POA de Java.

```
using System;
using log4net;

namespace LoggingDecorator
{
  class Program{
   static void Main(string[] args){
     log4net.Config.XmlConfigurator.Configure();

     IProcesador p = new Procesador();
     Console.WriteLine("Procesador Básico");
     Console.WriteLine("------------");
     p.Execute("Cristian-1");

     Console.WriteLine();
     IProcesador log =
         new ProcesadorLoggingDecorator(p);
     Console.WriteLine("Decortator Procesador");
     Console.WriteLine("------------");
     log.Execute("Cristian-2");
     Console.Read();
   }
  }
}
```

```
Procesador Básico
------------
Hola! -  Cristian-1

Decortator Procesador
------------
Inicio Decorator
Hola! -  Cristian-2
Fin Decorator
```

Fig. 6: Clase Principal y su Ejecución para Solución Decorator de Logging

Así, tal como afirma (Groves, 2013), los principales componentes de POA son el consejo (advice), el punto de unión (joinpoint), y el punto de corte (pointcut). Entonces, los consejos son el "Qué" de POA ya que la real misión de los aspectos es la encapsulación y modularización de incumbencias cruzadas; mientras que los puntos de corte representan el "Dónde y Cuándo" de POA, esto es, que momento y en qué lugar una clase es aconsejada (Groves, 2013). Además, un punto de corte corresponde a la definición de puntos de unión (usualmente, mediante pasos lógicos de la ejecución de un programa aconsejable).

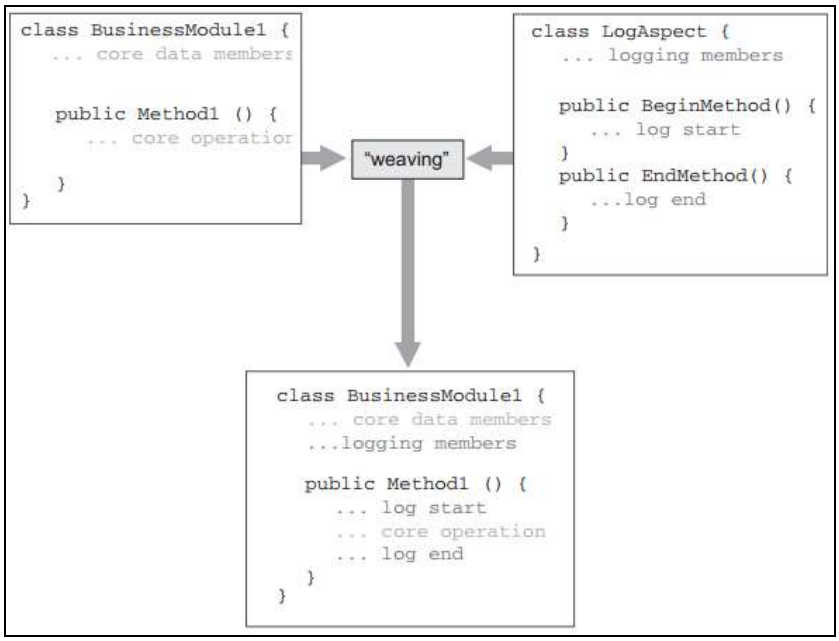


Fig. 7: Clase Principal y Aspecto de Logging en Proceso de Tejido

Así, mediante POA es posible separar clases y las denominadas incumbencias cruzadas, logrando soluciones modulares, y dada la definición de reglas de puntos de corte, un proceso de tejido (weaving) genera código objeto de la solución originalmente deseada tal como muestra la figura 7 (Groves, 2013), figura que justo se adapta al caso de estudio de este trabajo. Además, según (Wampler, 2007; Groves, 2013), dentro de los beneficios del uso de POA se destaca la producción de código limpio que es fácil de leer, menos propenso a fallas o bugs y más fácil de mantener.

Como se mencionó antes, la solución de logging de figura 1 no respeta el SRP, ya que la clase Procesador, además de su funcionalidad principal, presenta una instancia de clase para registrar actividades o log, y

realiza un registro de acciones antes y después de realizar la actividad principal. Entonces, en una solución POA tradicional, dicha clase requiere ser ingenua respecto a la inyección de estructura y comportamiento, ejemplo de dependencia implícita de clases a aspectos (Vidal et al., 2013; Vidal y Villarroel, 2014). En este contexto, PostSharp permite que una clase se exhiba de manera gradual o completa, ya que una clase explícitamente indica métodos que exhibe.

```
using System;

namespace LoggingPostSharp{
 class Program{
    static void Main(string[] args){
        log4net.Config.XmlConfigurator.Configure();

        Console.WriteLine("Procesador Aconsejado");
        Console.WriteLine("------------");
        Procesador p = new Procesador();

        p.Execute("Cristian-1");
        Console.WriteLine();

        Console.Read();
    }
 }
}
```

```
using System;

namespace LoggingPostSharp{
    [LoggingAspectIT]
    class Procesador
    {
      [LoggingAspect]
      public void Execute(string name){
        Console.WriteLine("Hola! -  {0}",name);
      }
  }
}
```

Fig. 8: Clase Principal y Clase Procesador de Solución Logging PostSharp

Una diferencia estructural respecto a soluciones Java de POA tales como AspectJ (Bodden et al., 2014; Griswold et al., 2016) y JPI es que con PostSharp los aspectos son clases que heredan de clases de aspectos (clase-aspecto). Así es posible clasificar tipos de aspectos, y establecer relaciones de clases tales como herencia y composición entre aspectos, además de relaciones adicionales entre clases y aspectos cuya revisión de viabilidad y utilidad es parte de trabajo futuro.

```
using System;
using PostSharp.Aspects;
using PostSharp.Aspects.Advices;
using log4net;

namespace LoggingPostSharp{
  [Serializable]
  public class LoggingAspectIT : InstanceLevelAspect{
    [IntroduceMember(OverrideAction =
    MemberOverrideAction.Ignore)]
    public ILog _Logger{
       get{
           return LogManager.GetLogger(typeof(Procesador));
       }
    }
  }
}
```

Fig. 9: Clase-Aspecto LoggingAspectoIT de Solución Logging PostSharp ceso de Tejido.

Las figuras 8, 9 y 10 muestran una solución PostSharp a ejemplo de figura 1. Como se aprecia en las figura 8 y 9, la clase *Procesador* es aconsejada por la clase-aspecto LoggingAspectIT para la introducción del atributo _Logger en la clase aconsejada. La figura 9 presenta el código de la clase-aspecto LoggingAspectIT. Además, la clase Procesador exhibe la ejecución del método Execute(..) a la clase-aspecto LoggingAspect. La figura 10 exhibe el código de clase-aspecto LoggingAspect. Así, la clase Procesador no es ingenua y tampoco tiene una doble responsabilidad para la realización directa de acciones, ya que exhibe clases-aspectos para la inclusión de elementos estructurales y comportamiento dinámico.

Este trabajo presentó como producir soluciones modulares .NET con la aplicación del patrón de diseño orientado a objetos Decorator y mediante el framework de POA PostSharp. A continuación, se indican ideas finales de cada uno de ellos:

```
using System;
using PostSharp.Aspects;
using log4net;
using System.Reflection;

namespace LoggingPostSharp{
  [Serializable]
  public class LoggingAspect : OnMethodBoundaryAspect{
     private string methodName;
     private Type className;

     public override void CompileTimeInitialize(
      MethodBase method, AspectInfo aspectInfo){
        methodName = method.Name;
        className = method.DeclaringType;
     }

     public override void OnEntry(MethodExecutionArgs args){
        Console.WriteLine(
          "Entrada de Método Aconsejado - Aspect Logging");

        var _Logger =
          className.GetProperty("_Logger", typeof(ILog));
        ILog logNene = (ILog)_Logger.GetValue(args.Instance);
        logNene.Info("Inicio de Logging");

        Console.WriteLine("-----------");
     }
     public override void OnExit(MethodExecutionArgs args){
         Console.WriteLine("----------");
         Console.WriteLine(
             "Salida de Método Aconsejado - Aspect Logging");

         var _Logger = className.GetProperty("_Logger", typeof(ILog));
         ILog logNene = (ILog)_Logger.GetValue(args.Instance);
         logNene.Info("Salida de Logging");
     }
  }
}
```

Fig. 10: Clase-Aspecto LoggingAspecto de Solución Logging PostSharp ceso de Tejido

Decorator, como patrón de diseño de soluciones orientadas a objetos, posibilita el agregar funcionalidades a objetos en tiempo de ejecución sin modificar la estructura de la clase de dichos objetos. Además, una de las propiedades de Decorator es que puede ser implementado en cualquier lenguaje de programación orientado a objeto ya que no requiere de ningún plugin o framework adicional para su implementación, funcionamiento y ejecución. Sin embargo, una de las desventajas prácticas de Decorator es que su implementación requiere un trabajo extra para el desarrollador ya que se debe establecer la estructura jerárquica del envoltorio propio de este patrón de diseño, así como codificar llamadas correctas a métodos y en tiempos correctos, esto es, no hay transparencia ni acciones implícitas en el desarrollo de este patrón de diseño. Se ha constatado además, que añadir funcionalidad a los objetos en tiempo de ejecución complica el proceso de depuración.
PostSharp, como framework de POA .NET, permite la encapsulación de incumbencias cruzadas mediante clases-aspectos, y permite transparencia en el proceso de inyección de código, ya sea en la estructura de la clase como en la ejecución de métodos aconsejados. Una de las grandes propiedades de PostSharp es el trabajo mediante clases-aspectos para la implementación de consejos o advices, y así no se requiere trabajar con nuevos módulos tales como aspectos como en lenguajes POA estilo AspectJ. Además, se plantea la posibilidad de establecer relaciones o asociaciones orientadas a objetos tanto entre clases-aspectos y clases-aspectos, como entre clases-aspectos y clases, junto con analizar pros y cons de una simbiosis práctica entre PostSharp y Decorator. De esta forma, las clases no son completamente ingenuas. Así, ya no hay una dependencia implícita de clase a aspecto, y de aspecto a clase tal como en la metodología POA JPI (Bodden et al., 2014).

Una potencial desventaja de POA es el uso de nuevos framework tales como PostSharp en .NET cuya versión profesional es libre por un tiempo limitado, aun cuando hay proyecto de una versión open-source del mismo. Además, no siempre es posible la homogenización de incumbencias cruzadas de manera, que pueden ocurrir situaciones con múltiples clases similares pero con algún comportamiento aconsejable diferente, lo que requiere así mismo de múltiples aspectos. Cabe señalar que siempre en los aspectos es posible el acceso al estado actual de los objetos aconsejados.

## CONCLUSIONES

Este trabajo demuestra que la encapsulación de incumbencias cruzadas en soluciones .NET es relevante en la búsqueda de soluciones .NET modulares. Por un lado, Decorator, como patrón de diseño de soluciones orientadas a objetos, posibilita el agregar funcionalidades a objetos en tiempo de ejecución sin modificar la estructura de la clase de dichos objetos. Por otro lado, PostSharp, como framework de POA .NET, permite la encapsulación de incumbencias cruzadas mediante clases-aspectos, y permite transparencia en el proceso de inyección de código, ya sea en la estructura de la clase como en la ejecución de métodos aconsejados. Además, es relevante destacar, tal y como se ejemplifica en este trabajo, que se puede lograr una relación ente clases y aspectos sin dependencia explícitas. Entonces, mediante PostSharp es posible el desarrollo de soluciones .NET modulares.

## REFERENCIAS

Bodden, E., E. Tanter y M. Inostroza, Join Point Interfaces for Safe and Flexible Decoupling of Aspects, in ACM Trans. Softw. Eng. Methodol., 23(1), 1-41 (2014)

Gamma, E., R. Helm, R. Johnson, J. Vlissides y G. Booch, Design Patterns: Elements of Reusable Object-Oriented Software, 1ª Ed., 175-185, Addison-Wesley Professional, U.S.A. (1994)

Gradecki, J. D. y N. Lesiecki, Mastering AspectJ: Aspect-Oriented Programming in Java, 1ª Ed. John Wiley & Sons, Inc., New York, NY, U.S.A. (2003)

Griswold, B., E. Hilsdale, J. Hugunin, W. Isberg y G. Kiczales, Aspect-Oriented Programming with AspectJ™, AspectJ.org, Xerox PARC (en línea), 2001

Groves, M. D., AOP in .NET: Practical Aspect-Oriented Programming, 1ª Ed., 85-107. Manning Publications, U.S.A. (2013)

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier y J. Irwin J., Aspect oriented programming, en Actas de European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 124, Finlandia (1997)

Log4net, Logging Services TM - The Apache log4net project (en línea), 2014. https://goo.gl/DBd54i. Acceso: 28 de Octubre (2016)

Martin, R., Agile Software Development: Principles, Patterns and Practices, 1ª Ed., 82-155. Prentice Hall, U.S.A. (2002)

Millett, S., Professional ASP.NET Design Patterns, 1ª Ed., 36-101, Wrox, U.S.A. (2010)

Seemann, M., Aspect-Oriented Programnming with Dependency Injection, Internation Software Development Conference GOTO, Copenhague, Dinamarca, Mayo (2011)

SharpCrafters, SharpCrafters s.r.o. - PostSharp Principles, (en línea), 2007. https://goo.gl/AwRmJZ. Acceso: 28 de Octubre (2016)

Vidal, C., D. Hernández, C. Pereria y C. Del Rio, Aplicación de la Modelación Orientada a Aspectos, Información Tecnológica, 23(1), 3-12 (2012)

Vidal, C., S. Rivero, L. López, C. Pereira, Propuesta y Aplicación de Diagramas de Clases JPI, Información Tecnológica, 25(5), 113-120 (2014)

Vidal, C., R. Saens, C. Del Rio y R. Villarroel, OOAspectZ and Aspect-Oriented UML Class Diagram, Ingeniería e Investigación, 3(3), 66-71 (2013)

Vidal, C. y R. Villarroel, JPI UML: JPI class and sequence diagrams for aspect-oriented JPI applications, en Actas de XXXIII International Conference of the Chilean Computer Society, Talca, Chile, Noviembre (2014)

Wampler, D., Aspect-Oriented Design Principles: Lessons from Object-Oriented Design, Sixth International Conference on Aspect-Oriented Software Development (AOSD'07), Vancouver, British Columbia, Canada, 12-16, Marzo (2007)