Modulating Crosscutting Concerns By The Decorator Design Pattern Vs. Aspect Oriented Programming In .NET

Cristian Pereira-Vásquez, Cristian Vidal-Silva, Erika Madariaga, Claudia Jiménez, Luis Urzúa

Abstract: This article describes and illustrates how to produce modular .NET software solutions by the use of Decorator design pattern and Aspect-Oriented Programming (AOP) tool PostSharp. We applied both techniques for modularizing crosscutting concerns of a traditional modularization example: logging function. This work presents logging solutions with the use of Decorator and PostSharp to modularize associated cross issues, along with detailing the advantages and disadvantages of both solutions. Likewise, this work points out details of in PostSharp along with proposing the use of PostSharp and Decorator to achieve solutions with a higher level of modularity.

Index Terms: AOP, .NET, Decorator, PostSharp, Crosscutting Concerns, Aspects.

1 INTRODUCTION

I n programming, cross issues represent software features whose modularization and isolation are complex or impossible in some cases. Its implementation spreads between different modules of the solution [1] [2]. As [3] indicate, typical examples of cross-reference are persistence, error handling, and defined logging or recording of actions. Figure 1 shows a simple example of logging in .NET that represents the base example of this study. As can be seen in Figure 1, the Main (..) method of the class example creates an instance of the Processor class to proceed with the Execute () method on it, a method that has two responsibilities:

1. Print a message on the screen.
2. Have a record in Log files of the Log4Net library [4].

The Processor class then presents a violation of the principle of single responsibility (Single Responsibility Principle SRP) [5], which is an example of cross-cutting concern [3].

According to [6], Decorator is an object-oriented software design pattern to define components of objects called Decorators, where each Decorator conforms to the interface of each component so that its presence is transparent to the clients of such component [5] [6] [7]. Thanks to this, the Decorator design pattern can handle action requests of its components, as well as perform additional actions. This paper reviews how to modularize cross issues using the Decorator design pattern.

- *Cristian Pereira-Vásquez, Professional Computer Services Morris and Opazo, Antonio Varas No. 920, Office No. 35, Temuco-Chile. E-mail: cpereira@morrisopazo.com*
- *Cristian Vidal-Silva, professor at Departamento de Administración, Facultad de Economía y Negocios, Universidad Católica del Norte, Antofagasta, Chile. E-mail: cristian.vidal@ucn.cl*
- *Erika Madariaga, director of Ingeniería Informática, Facultad de Ingeniería, Ciencia y Tecnología, Universidad Bernardo O'Higgins, Santiago, Chile. E-mail: erika.madariaga@ubo.cl*
- *Claudia Jiménez, director of Ingeniería Civil Informática, Facultad de Ingeniería y Negocios, Universidad Viña del Mar, Viña del Mar, Chile. E-mail: cjimenez@uvm.cl*
- *Luis Urzúa, professor at Escuela de Kinesiología, Facultad de Salud, Universidad Santo Tomás, Talca, Chile. E-mail: lurzua@santotomas.cl*


```
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("Ejemplo de Aplicación!");

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

Fig. 1. Example of Logging Function in .NET.

As they point out [8] [9] [10] [11], Aspect-Oriented Programming (POA), thanks to the cross-sectional separation, supports a set of principles of good object-oriented design; among them SRP and the open-close principle. Besides, [12] indicates that low coupling and high cohesion are fundamental and necessary principles to achieve a modular software design. Precisely, thanks to the separation of incumbencies, POA allows a high level of cohesion and low coupling between recommended modules and aspects.

Thus, the main objective of this work is to review the applicability of the Decorator design pattern and the use of PostSharp [13] for the modularization of code in Figure 1. Hence, this article analyzes the use of

both approaches to modularize crosscutting concerns to look for modular .NET solutions.

2 DECORATOR

Decorator is a design pattern that allows new behavior to be added dynamically throughout the composition, which corresponds to the process of wrapping an existing class with a class that extends the behavior or state [3] [4].

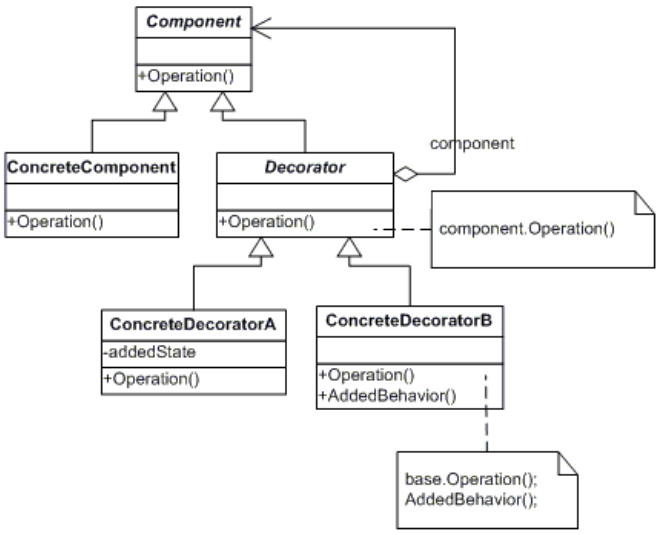

***Fig. 2**. Structure of a Decorator Solution.*

Figure 2 [6] shows the general structure of a solution with the Decorator design pattern, which presents different components and relationships. Table I describes the components of Figure 2.

According to [7], the Decorator pattern is necessary when there is a need to dynamically add behavior, as well as eliminate responsibilities to a class. Thus, Figure 3 shows part of the application of the decorator pattern of the example of Figure 1: the Component interface and the ConcreteComponent class, respectively (IProcessor interface and Processor class). Note that the Execute (..) method this time receives a parameter to identify the object that invokes it. When analyzing the solution in Figure 3, the Processor class now presents only one responsibility, without a record of actions or logging that requires a Decorator class and a ConcreteDecorator class, both with a single responsibility as Figures 4 and 5 illustrate.

TABLA I
SURVEY FOR TEACHERS IN THE AREA OF TECHNOLOGY AND INFORMATION TECHNOLOGY OF HIGHER EDUCATION ENTITIES IN CHILE.

Component	*Definition*
Interface	Abstract interface or class that can have dynamic behavior.
ConcreteComponent class	A class that implements the Component interface to implement class responsibilities.
Decorator class	A class that implements the Component interface and contains a reference to an instance of that interface. Structurally, this class is an interface for more specific Decorator classes.
Concrete Decorator Class	A Class that freezes or implements the Decorator to expand and specialize its functionality.

Figure 6 shows the main program of the example, together with its output. Then, it verifies that the use of the Decorator design pattern allows the inclusion of additional behavior on existing methods of instances of recommended classes. Besides, thanks to the Inheritance property, a ConcreteDecorator class can include additional attributes and methods, which are not added directly to the recommended object.

```
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**. Decorator class for Logging Decorator Solution.*

```
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**. Class Concrete Decorator for Logging Decorator Solution.*

3 ASPECT-ORIENTED PROGRAMMING

Aspect-Oriented Programming (AOP) seeks the modular behavior of classes, for which it defines advisable naive classes and modulates cross-sectional issues through aspects [1]. In this way, POA solutions allow classes to respect the principle of sole responsibility, in addition to defining an interaction of aspects and classes advised. POA is born with AspectJ, a POA version of Java. Thus, such as [3] affirms, the main components of AOP are the advice (advice), the point of union (joinpoint), and the point of cut (pointcut). Advice instances are the "What" of AOP since the real mission of the aspects is the encapsulation and modularization of crosscutting concerns. Pointcut instances represent the "Where and When" of AOP solutions (what time and where a class is advised [3]). Also, a pointcut corresponds to the definition of joinpoints (usually, by logical steps of the execution of an advisable program).

```
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**. Main Class and its Execution for Logging Decorator Solution.*

4 SOLUTION & DISCUSSION

Thus, using AOP is possible to separate classes and the so-called cross-incidents, achieving modular solutions. Given the definition of cut-point rules, a weaving process generates object code of the originally desired solution, as shown in Figure 7 [13], that can adapt to the case study of this work. Besides, according to [3] [12], the benefits of using POA include the production of clean code that is easy to read, less prone to failures or bugs and easier to maintain. As mentioned before, the logging solution in Figure 1 does not respect the SRP, since the Processor class, in addition to its main functionality, presents a class instance to record activities or log, and performs a record of actions before and after Perform the main activity. So, in a traditional POA solution, this class needs to be naive concerning the injection of structure and behavior, an example of implicit dependence of classes on aspects [10] [14]. In this context, PostSharp allows a class to be displayed gradually or completely since a class explicitly indicates the methods it exhibits.

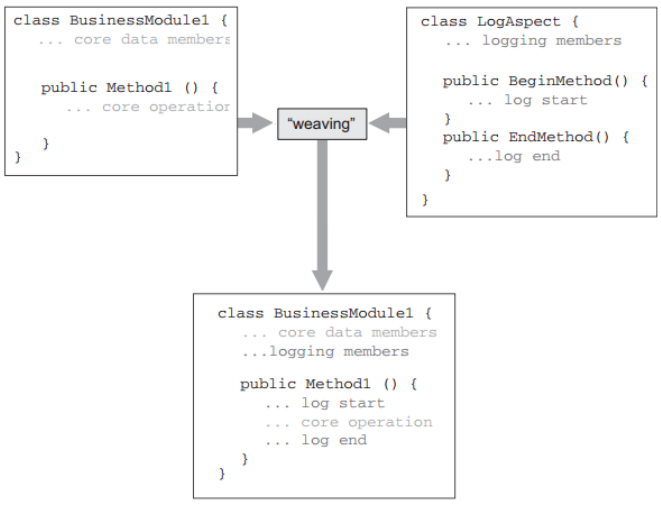

Fig. 7. Main Class and Aspect of Logging in the Weaving Process.

A structural difference concerning Java solutions from POA such as AspectJ [11]; Griswold et al., 2016) and JPI is that with PostSharp, the aspects are classes that inherit from classes of aspects (class-aspect). Thus it is possible to classify types of aspects, and establish class relationships such as inheritance and composition between aspects, in addition to additional relationships between classes and aspects whose review of feasibility and usefulness is part of future work. Figures 8, 9, and 10 show a PostSharp solution as an example of Figure 1. As can be seen in Figures 8 and 9, the Processor class is advised by the LoggingAspectIT class-aspect for the introduction of the _Logger attribute in the recommended class. Figure 9 presents the code of the LoggingAspectIT class-aspect. Also, the Processor class exhibits the execution of the Execute(..) method to the LoggingAspect class-aspect. Figure 10 shows the LoggingAspect class-aspect code. Thus, the Processor class is not naive and does not have a double responsibility for the direct realization of actions, since it exhibits classes-aspects for the inclusion of structural elements and dynamic behavior.

5 CONCLUSIONS

This work presented how to produce modular .NET solutions with the application of the Decorator object-oriented design pattern and through the PostSharp POA framework. Below are the final ideas of each of them:- Decorator, as a design pattern for object-oriented solutions, allows you to add functionalities to objects at runtime without modifying the class structure of those objects. Besides, one of the Decorator properties is that it can be implemented in any object-oriented programming language since it does not require any additional plugin or framework for its implementation, operation, and execution. However, one of the practical disadvantages of Decorator is its implementation that requires extra work since it is necessary to define the hierarchical structure of the packaging of this design pattern, as well as correct coding calls of methods. Thus, no transparency or implicit actions exist in the development of this design pattern. Adding functionality to objects at runtime complicates the debugging process.

- PostSharp, as a POA .NET framework, allows cross-case encapsulation through classes-aspects, and allows transparency in the process of code injection, either in the class structure or in the execution of recommended methods. One of the great properties of PostSharp is the work through classes-aspects for the implementation of tips or advice instances, and thus it is not necessary to work with new modules such as AspectJ aspects. Besides, the possibility of establishing object-oriented relationships or associations between classes and aspects, and aspects and classes. That allows analyzing the pros and cons of a practical symbiosis between PostSharp and Decorator. A potential disadvantage of POA is the use of new frameworks such as PostSharp in .NET, whose professional version is free for a limited time, even when there is a project of an open-source version of it.

```
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();
    }
 }
}
```

Fig. 8. Main Class and PostSharp Logging Solution Processor Class.

```
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.** Logging Aspect-Class AspectIT of Logging Solution PostSharp Tissue stop.*

```
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**. Class-Aspect Logging Aspect of Logging Solution PostSharp.*

This work demonstrates that cross-encapsulation encapsulation in .NET solutions is relevant in the search for modular .NET solutions.

REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, Proceeding of the European Conference on Object-Oriented Programming (ECOOP), 1997, Springer-Verlag LNCS 124, Finland.

[2] J. D. Gradecki, and N. Lesiecki, “Mastering AspectJ: Aspect-Oriented Programming in Java”, John Wiley & Sons, Inc., 2003, New York, NY, USA.

[3] M. D. Groves, “AOP in .NET: Practical Aspect-Oriented Programming”, 2013, Manning Publications, USA.

[4] Log4net, “Logging Services TM”, The Apache log4net project, https://logging.apache.org/log4net/. 2019.

[5] R. Martin, “Agile Software Development: Principles, Patterns and Practices”, 2002, Prentice Hall, USA.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional, 1994, USA.

[7] S. Millett, “Professional ASP.NET Design Patterns”, 2010, Wrox, USA.

[8] C. Vidal, D. Hernández, C. Pereira, and C. Del Rio, “Aplicación de la Modelación Orientada a Aspectos”, Información Tecnológica, 2012, vol. 23, no. 1, pp. 3 – 12, doi: 10.4067/S0718-07642012000100002.

[9] C. Vidal, S. Rivero, L. López, and C. Pereira, “Propuesta y Aplicación de Diagramas de Clases JPI”, Información Tecnológica, 2014, vol. 25, no. 5, pp. 113 – 120, doi: 10.4067/S0718-07642014000500016.

[10] C. Vidal, and R. Villarroel, “JPI UML: JPI class and sequence diagrams for Aspect-Oriented JPI applications”, in Proceedings of XXXIII International Conference of the Chilean Computer Society, 2014, Talca, Chile, November.

[11] E. Bodden, E. Tanter, and M. Inostroza, “Join Point Interfaces for Safe and Flexible Decoupling of Aspects”, in ACM Transaction on Software Engineering and Methodology, 2014, vol. 23, no. 1, pp. 1 – 41, doi: 10.1145/2559933.

[12] D. Wampler, “Aspect-Oriented Design Principles: Lessons from Object-Oriented Design”, Sixth International Conference on Aspect-Oriented Software Development (AOSD'07), 2007, Vancouver, British Columbia, Canada, pp. 12 – 16, March.

[13] SharpCrafters, “SharpCrafters s.r.o.”, PostSharp Principles, http://www.postsharp.net/blog/post/Day-1-e28093-OnExceptionAspect. 2019.

[14] C. Vidal, R. Saens, C. Del Rio, and R. Villarroel, “OOAspectZ and Aspect-Oriented UML Class Diagram”, Ingeniería e Investigación Journal, 2013, Medellín, Colombia, vol. 33., no. 3, pp. 66 – 71.