# Aspect Oriented Programming in C# using DispatchProxy

## Introduction

[Aspect Oriented Programming (AOP)](https://en.wikipedia.org/wiki/Aspect-oriented_programming) is very powerful approach to avoid boilerplate code and archive better modularity. The main idea is to add behavior (advice) to the existing code without making any changes in the code itself. AOP provides a way of weaving an aspect into the code. An aspect is supposed to be generic so it can be applied to any object and object should not have to know anything about advice. AOP allows to separate [cross-cutting concerns](https://en.wikipedia.org/wiki/Cross-cutting_concern) and makes easier to follow [Single Responsibility Principle](https://en.wikipedia.org/wiki/Single_responsibility_principle) (one of the [SOLID](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)) principles). Logging, security, transactions and exceptions handling are the most common examples of using AOP. If you are not familiar with this programming technic you can read [this](https://en.wikipedia.org/wiki/Aspect-oriented_programming) or [this](http://study.com/academy/lesson/aspect-oriented-programming-definition-concepts.html). It could be very useful because this article mostly about how to use AOP in C# rather than what AOP is. Don’t be scared if you still do not understand what is all about. After locking at several examples, it becomes much easier to understand.

In Java AOP implemented in [AspectJ](https://eclipse.org/aspectj/) and [Spring](https://spring.io/) frameworks. There are [PostSharp](https://www.postsharp.net/) (not free), [NConcern](https://github.com/Virtuoze/NConcern) and some other frameworks (not very popular and easy to use) to do almost the same in .Net.

It is also possible to use **RealProxy** class to implement AOP. You can find some examples how to do it:

Example1: [Aspect-Oriented Programming: Aspect-Oriented Programming with the RealProxy Class](https://msdn.microsoft.com/en-us/magazine/dn574804.aspx).

This article also contains a lot of explanation what is AOP, how [Decorator Design Pattern](https://en.wikipedia.org/wiki/Decorator_pattern) works and examples of implementing logging and authentication using AOP.

Example2: [MSDN](https://msdn.microsoft.com/en-us/library/system.runtime.remoting.messaging.returnmessage(v=vs.110).aspx).

Unfortunately, these examples have some significant drawbacks. Example1 does not support out parameters. Example2 has limitation. Decorated class should be inherited from **MarshalByRefObject** (it could be a problem if it is not your class). Also, both examples do not support asynchronous functions as expected. Several months ago, I changed the first example support **Task** results and output parameters and wrote article about it.

Example3: [Aspect Oriented Programming in C# with RealProxy](https://www.codeproject.com/Articles/1204871/Aspect-Oriented-Programming-in-Csharp-with-RealPro).

Unfortunately, .Net Core does not have **RealProxy** class. There is **DispatchProxy** class instead. Using of **DispatchProxy** class is a bit different than using **RealProxy** class.

Let’s implement logging using **DispatchProxy** class.

Code of this articles and example of using **RealProxy** with unit tests for both can be found [here](https://github.com/ValeraT1982/AOP).

## Solution

### Extension to log exception (Extensions.cs)

using System;

using System.Text;

namespace AOP

{

public static class Extensions

{

public static string GetDescription(this Exception e)

{

var builder = new StringBuilder();

AddException(builder, e);

return builder.ToString();

}

private static void AddException(StringBuilder builder, Exception e)

{

builder.AppendLine($"Message: {e.Message}");

builder.AppendLine($"Stack Trace: {e.StackTrace}");

if (e.InnerException != null)

{

builder.AppendLine("Inner Exception");

AddException(builder, e.InnerException);

}

}

}

}

### Logging Advice (LoggingAdvice.cs)

public class LoggingAdvice<T> : DispatchProxy

{

private T \_decorated;

private Action<string> \_logInfo;

private Action<string> \_logError;

private Func<object, string> \_serializeFunction;

private TaskScheduler \_loggingScheduler;

protected override object Invoke(MethodInfo targetMethod, object[] args)

{

if (targetMethod != null)

{

try

{

try

{

LogBefore(targetMethod, args);

}

catch (Exception ex)

{

//Do not stop method execution if exception

LogException(ex);

}

var result = targetMethod.Invoke(\_decorated, args);

var resultTask = result as Task;

if (resultTask != null)

{

resultTask.ContinueWith(task =>

{

if (task.Exception != null)

{

LogException(task.Exception.InnerException ?? task.Exception, targetMethod);

}

else

{

object **taskResult** = null;

if (task.GetType().GetTypeInfo().IsGenericType &&

task.GetType().GetGenericTypeDefinition() == typeof(Task<>))

{

var property = task.GetType().GetTypeInfo().GetProperties()

.FirstOrDefault(p => p.Name == "Result");

if (property != null)

{

**taskResult** = property.GetValue(task);

}

}

LogAfter(targetMethod, args, **taskResult**);

}

},

\_loggingScheduler);

}

else

{

try

{

LogAfter(targetMethod, args, result);

}

catch (Exception ex)

{

//Do not stop method execution if exception

LogException(ex);

}

}

return result;

}

catch (Exception ex)

{

if (ex is TargetInvocationException)

{

LogException(ex.InnerException ?? ex, targetMethod);

throw ex.InnerException ?? ex;

}

}

}

throw new ArgumentException(nameof(targetMethod));

}

public static T Create(T decorated, Action<string> logInfo, Action<string> logError,

Func<object, string> serializeFunction, TaskScheduler loggingScheduler = null)

{

object proxy = Create<T, LoggingAdvice<T>>();

((LoggingAdvice<T>)proxy).SetParameters(decorated, logInfo, logError, serializeFunction, loggingScheduler);

return (T)proxy;

}

private void SetParameters(T decorated, Action<string> logInfo, Action<string> logError,

Func<object, string> serializeFunction, TaskScheduler loggingScheduler)

{

if (decorated == null)

{

throw new ArgumentNullException(nameof(decorated));

}

\_decorated = decorated;

\_logInfo = logInfo;

\_logError = logError;

\_serializeFunction = serializeFunction;

\_loggingScheduler = loggingScheduler ?? TaskScheduler.FromCurrentSynchronizationContext();

}

private string GetStringValue(object obj)

{

if (obj == null)

{

return "null";

}

if (obj.GetType().GetTypeInfo().IsPrimitive || obj.GetType().GetTypeInfo().IsEnum || obj is string)

{

return obj.ToString();

}

try

{

return \_serializeFunction?.Invoke(obj) ?? obj.ToString();

}

catch

{

return obj.ToString();

}

}

private void LogException(Exception exception, MethodInfo methodInfo = null)

{

try

{

var errorMessage = new StringBuilder();

errorMessage.AppendLine($"Class {\_decorated.GetType().FullName}");

errorMessage.AppendLine($"Method {methodInfo?.Name} threw exception");

errorMessage.AppendLine(exception.GetDescription());

\_logError?.Invoke(errorMessage.ToString());

}

catch (Exception)

{

// ignored

//Method should return original exception

}

}

private void LogAfter(MethodInfo methodInfo, object[] args, object result)

{

var afterMessage = new StringBuilder();

afterMessage.AppendLine($"Class {\_decorated.GetType().FullName}");

afterMessage.AppendLine($"Method {methodInfo.Name} executed");

afterMessage.AppendLine("Output:");

afterMessage.AppendLine(GetStringValue(result));

var parameters = methodInfo.GetParameters();

if (parameters.Any())

{

afterMessage.AppendLine("Parameters:");

for (var **i** = 0; **i** < parameters.Length; **i**++)

{

var parameter = parameters[**i**];

var arg = args[**i**];

afterMessage.AppendLine($"{parameter.Name}:{GetStringValue(arg)}");

}

}

\_logInfo?.Invoke(afterMessage.ToString());

}

private void LogBefore(MethodInfo methodInfo, object[] args)

{

var beforeMessage = new StringBuilder();

beforeMessage.AppendLine($"Class {\_decorated.GetType().FullName}");

beforeMessage.AppendLine($"Method {methodInfo.Name} executing");

var parameters = methodInfo.GetParameters();

if (parameters.Any())

{

beforeMessage.AppendLine("Parameters:");

for (var **i** = 0; **i** < parameters.Length; **i**++)

{

var parameter = parameters[**i**];

var arg = args[**i**];

beforeMessage.AppendLine($"{parameter.Name}:{GetStringValue(arg)}");

}

}

\_logInfo?.Invoke(beforeMessage.ToString());

}

}

Let’s assume that we have an interface and a class.

public interface IMyClass

{

int MyMethod(string param);

}

public class MyClass

{

public int MyMethod(string param)

{

return param.Length;

}

}

To decorate **MyClass** by **LoggingAdvice** we should do the following.

var decorated = LoggingAdvice<IMyClass>.Create(

new MyClass(),

s => Debug.WriteLine("Info:" + s),

s => Debug.WriteLine("Error:" + s),

o => o?.ToString());

To understand how it works we call **MyMesthod** of **decorated** instance.

var length = decorated.MyMethod("Hello world!");

This line of code does:

1. **decorated.MyMethod("Hello world!")** calls **Invoke** method of **LoggingAdvice** with **targetMethod** equal to **MyMethod** and **args** equal toarraywith one element equal to *"Hello world!"*(**LogBefore**)*.*
2. **Invoke** method logs **MyMethod** method name and input parameters (**LogBefore**).
3. **Invoke** method calls **MyMethod** method of **MyClass** (instance of this object was created internally).
4. If method call succeed output parameters and result are logged (**LogAfter**) and **Invoke** method returns result.
5. If method call throws an exception, the exception is logged (**LogException**) and **Invoke** throwsthis exception.
6. Result of the **Invoke** method execution (result or exception) returns as a result of calling **MyMethod** of **decorated** object**.**

## Example

Let assume that we are going to implement calculator which adds and subtracts integer numbers.

public interface ICalculator

{

int Add(int a, int b);

int Subtract(int a, int b);

}

public class Calculator : ICalculator

{

public int Add(int a, int b)

{

return a + b;

}

public int Subtract(int a, int b)

{

return a - b;

}

}

It is easy. Each method has only one responsibility.

One day some users start complaining that sometimes **Add(2, 2)** returns 5. You don’t understand what's going on and decide to add logging.

public class CalculatorWithoutAop: ICalculator

{

private readonly ILogger \_logger;

public CalculatorWithoutAop(ILogger logger)

{

\_logger = logger;

}

public int Add(int a, int b)

{

\_logger.Log($"Adding {a} + {b}");

var result = a + b;

\_logger.Log($"Result is {result}");

return result;

}

public int Subtract(int a, int b)

{

\_logger.Log($"Subtracting {a} - {b}");

var result = a - b;

\_logger.Log($"Result is {result}");

return result;

}

}

There are 3 problems with this solution.

1. **Calculator** class coupled with logging. Loosely coupled (because **ILoger** it is an interface), but coupled. Every time you make changes in this interface it affects Calculator.
2. Code become more complex.
3. It breaks Single Responsibility principle. **Add** function don't just add numbers. It logs input values, add values and logs result. The same for Subtract.

Code in this article allows you don't touch **Calculator** class at all.

You just need to change creation of the class.

public class CalculatorFactory

{

private readonly ILogger \_logger;

public CalculatorFactory(ILogger logger)

{

\_logger = logger;

}

public ICalculator CreateCalculator()

{

return LoggingAdvice<ICalculator >.Create(

new Calculator(),

s => \_logger.Log("Info:" + s),

s => \_logger.Log("Error:" + s),

o => o?.ToString());

}

}

## Conclusion

This code works for my cases. If you have any examples when this code does not work or how this code could be improved – fill free to contact me in any way.

That's it — enjoy!