# Aspect Oriented Programming in C# with RealProxy

## 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. In Java, this idea 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)

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. Ok, technically it is supported. But I’m expecting “after” code execution AFTER task completion, and if “after” code tries to get value of *Result* property from *Task* object (for example during result serialization) it makes asynchronous code synchronous (not cool ☹).

I tried to fix first example.

## Before continue reading

This article is how to fix some problems in solution, provided by **Bruno Sonnino** (Example1). That article has great explanation how code is supposed to work and what kind of problems it solves. Please read [Aspect-Oriented Programming : Aspect-Oriented Programming with the RealProxy Class](https://msdn.microsoft.com/en-us/magazine/dn574804.aspx) first.

## Solution

This solution is an example of logging implementation. Code could be found [here](https://github.com/ValeraT1982/AOP).

Differences with original code:

1. Extension method *GetDescription* was added to log Exception data (*Extensions.cs*).
2. *DynamicProxy* class was renamed to *LoggingAdvice.*
3. Constructor was made private. Static function *Create* creates class instance and returns *TransparentProxy* (*LoggingAdvice.cs* - lines 35-41). It makes impossible to create instance of *LoggingAdvice* class, because only proxy, created by this class, is going to be used.
4. *LoggingAdvice* receives actions to log function calls and errors, and function to serialize complex type values as parameters (*LoggingAdvice.cs* - lines 19-20).
5. *TaskScheduler* was added as an optional parameter to support task results logging using different task scheduler. *TaskScheduler.FromCurrentSynchronizationContext()* will be used by default (*LoggingAdvice.cs* - line 36).
6. Functions *LogException*, *LogBefore* and *LogAfrer* were added to log corresponding data (*LoggingAdvice.cs* - lines 150-209).
7. Try/catch blocks were added to handle situation when *logInfo* function throws an exception (*LoggingAdvice.cs* - lines 53-61, 99-107).
8. If result value of the function is *Task*, execution result will be logged after task completion (*LoggingAdvice.cs* - lines 69-96).
9. Added output parameters support (*LoggingAdvice.cs* - lines 63, 110-111).

### 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)

using System;

using System.Linq;

using System.Reflection;

using System.Runtime.Remoting.Messaging;

using System.Runtime.Remoting.Proxies;

using System.Text;

using System.Threading.Tasks;

namespace AOP

{

public class LoggingAdvice<T> : RealProxy

{

private readonly T \_decorated;

private readonly Action<string> \_logInfo;

private readonly Action<string> \_logError;

private readonly Func<object, string> \_serializeFunction;

private readonly TaskScheduler \_loggingScheduler;

private LoggingAdvice(T decorated, Action<string> logInfo, Action<string> logError,

Func<object, string> serializeFunction, TaskScheduler loggingScheduler)

: base(typeof(T))

{

if (decorated == null)

{

throw new ArgumentNullException(nameof(decorated));

}

\_decorated = decorated;

\_logInfo = logInfo;

\_logError = logError;

\_serializeFunction = serializeFunction;

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

}

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

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

{

var advice = new LoggingAdvice<T>(decorated, logInfo, logError, serializeFunction, loggingScheduler);

return (T)advice.GetTransparentProxy();

}

public override IMessage Invoke(IMessage msg)

{

var methodCall = msg as IMethodCallMessage;

if (methodCall != null)

{

var methodInfo = methodCall.MethodBase as MethodInfo;

if (methodInfo != null)

{

try

{

try

{

LogBefore(methodCall, methodInfo);

}

catch (Exception ex)

{

//Do not stop method execution if exception

LogException(ex);

}

var args = methodCall.Args;

var result = typeof(T).InvokeMember(

methodCall.MethodName,

BindingFlags.InvokeMethod | BindingFlags.Public | BindingFlags.Instance, null, \_decorated, args);

if (result is Task)

{

((Task)result).ContinueWith(task =>

{

if (task.Exception != null)

{

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

}

else

{

object taskResult = null;

if (task.GetType().IsGenericType && task.GetType().GetGenericTypeDefinition() == typeof(Task<>))

{

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

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

if (property != null)

{

taskResult = property.GetValue(task);

}

}

LogAfter(methodCall, methodCall.Args, methodInfo, taskResult);

}

},

\_loggingScheduler);

}

else

{

try

{

LogAfter(methodCall, args, methodInfo, result);

}

catch (Exception ex)

{

//Do not stop method execution if exception

LogException(ex);

}

}

return new ReturnMessage(result, args, args.Length,

methodCall.LogicalCallContext, methodCall);

}

catch (Exception ex)

{

if (ex is TargetInvocationException)

{

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

return new ReturnMessage(ex.InnerException ?? ex, methodCall);

}

}

}

}

throw new ArgumentException(nameof(msg));

}

private string GetStringValue(object obj)

{

if (obj == null)

{

return "null";

}

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

{

return obj.ToString();

}

try

{

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

}

catch

{

return obj.ToString();

}

}

private void LogException(Exception exception, IMethodCallMessage methodCall = null)

{

try

{

var errorMessage = new StringBuilder();

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

errorMessage.AppendLine($"Method {methodCall?.MethodName} threw exception");

errorMessage.AppendLine(exception.GetDescription());

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

}

catch (Exception)

{

// ignored

//Method should return original exception

}

}

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

{

var afterMessage = new StringBuilder();

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

afterMessage.AppendLine($"Method {methodCall.MethodName} 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(IMethodCallMessage methodCall, MethodInfo methodInfo)

{

var beforeMessage = new StringBuilder();

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

beforeMessage.AppendLine($"Method {methodCall.MethodName} 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 = methodCall.Args[i];

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

}

}

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

}

}

}

## How to use

var decoratedInstance = LoggingAdvice<IInstanceInteface>.Create(

instance,

s => Console.WriteLine(“Info:” + s),

s => Console.WriteLine(“Error:” + s),

o => o?.ToString());

## Example

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

namespace AOP.Example

{

public interface ICalculator

{

int Add(int a, int b);

int Subtract(int a, int b);

}

}

namespace AOP.Example

{

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.

namespace AOP.Example

{

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.

namespace AOP.Example

{

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());

}

}

}

## Tests

NUnit and NSubstitute is used to write Unit Tests.

*CurrentThreadTaskScheduler* allows to schedule tasks in current thread in test project (code is not listed here).

using System;

using System.Collections.Generic;

using System.Threading;

using System.Threading.Tasks;

using NSubstitute;

using NUnit.Framework;

namespace AOP.Tests

{

[TestFixture]

public class LoggingAdviceTests

{

private TaskScheduler \_taskScheduler;

public class Data

{

public string Prop { get; set; }

}

public interface ITestClass

{

string Property { get; set; }

void MethodWithVoidResult();

string MethodWithStringResultAndIntParameter(int intParam);

Data MethodWithClassResultAndClassParameter(Data dataParam);

int MethodWithOutParameter(out string stringParam);

int MethodWithRefParameter(ref string stringParam);

int MethodWithOutParameter(out int intParam);

int MethodWithRefParameter(ref int intParam);

int MethodWithMixedParameter(ref int refParam, out string outParam, bool param);

Task MethodWithTaskResult();

Task<string> MethodWithTaskStringResult();

}

[SetUp]

public void SetUp()

{

SynchronizationContext.SetSynchronizationContext(new SynchronizationContext());

\_taskScheduler = new CurrentThreadTaskScheduler();

}

[Test]

public void ThrowException\_WhenDecoratedClassNull()

{

Assert.Throws<ArgumentNullException>(() =>

{

var decorated = LoggingAdvice<ITestClass>.Create(

null,

s => { },

s => { },

o => o?.ToString());

});

}

[Test]

public void LogInfo()

{

var testClass = Substitute.For<ITestClass>();

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

decorated.MethodWithVoidResult();

testClass.Received().MethodWithVoidResult();

Assert.AreEqual(2, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("MethodWithVoidResult"));

Assert.IsTrue(infoMassages[1].Contains("MethodWithVoidResult"));

Assert.AreEqual(0, errorMassages.Count);

}

[Test]

public void DoNotStop\_WhenLogInfoThrowsException()

{

var testClass = Substitute.For<ITestClass>();

var errorMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => { throw new Exception(); },

s => errorMassages.Add(s),

o => o?.ToString());

decorated.MethodWithVoidResult();

testClass.Received().MethodWithVoidResult();

Assert.AreEqual(2, errorMassages.Count);

}

[Test]

public void LogInfo\_WhenClassResultAndClassParameter()

{

var testClass = Substitute.For<ITestClass>();

testClass.MethodWithClassResultAndClassParameter(Arg.Any<Data>()).Returns(new Data { Prop = "Result12345" });

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => ((Data)o).Prop);

var result = decorated.MethodWithClassResultAndClassParameter(new Data { Prop = "Parameter12345"});

testClass.Received().MethodWithClassResultAndClassParameter(Arg.Any<Data>());

Assert.AreEqual("Result12345", result.Prop);

Assert.AreEqual(2, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("dataParam:Parameter12345"));

Assert.IsTrue(infoMassages[1].Contains("dataParam:Parameter12345"));

Assert.IsTrue(infoMassages[1].Contains("Result12345"));

Assert.AreEqual(0, errorMassages.Count);

}

[Test]

public void LogInfo\_WhenParameters()

{

var testClass = Substitute.For<ITestClass>();

testClass.MethodWithStringResultAndIntParameter(Arg.Any<int>()).Returns("Result12345");

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

var result = decorated.MethodWithStringResultAndIntParameter(12345);

testClass.Received().MethodWithStringResultAndIntParameter(Arg.Any<int>());

Assert.AreEqual("Result12345", result);

Assert.AreEqual(2, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("intParam:12345"));

Assert.IsTrue(infoMassages[1].Contains("Result12345"));

Assert.IsTrue(infoMassages[1].Contains("intParam:12345"));

Assert.AreEqual(0, errorMassages.Count);

}

[Test]

public void DoNotStop\_WhenSerializerThrowsException()

{

var testClass = Substitute.For<ITestClass>();

testClass.MethodWithClassResultAndClassParameter(Arg.Any<Data>()).Returns(new Data { Prop = "Result12345" });

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => { throw new Exception();});

Data result = null;

Assert.DoesNotThrow(() => result = decorated.MethodWithClassResultAndClassParameter(new Data { Prop = "Parameter12345"}));

testClass.Received().MethodWithClassResultAndClassParameter(Arg.Any<Data>());

Assert.AreEqual("Result12345", result?.Prop);

Assert.AreEqual(2, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("Data"));

Assert.IsTrue(infoMassages[1].Contains("Data"));

Assert.AreEqual(0, errorMassages.Count);

}

[Test]

public void LogError()

{

var testClass = Substitute.For<ITestClass>();

testClass.When(t => t.MethodWithVoidResult()).Do(info => { throw new Exception(); });

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

Assert.Throws<Exception>(() => decorated.MethodWithVoidResult());

testClass.Received().MethodWithVoidResult();

Assert.AreEqual(1, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("MethodWithVoidResult"));

Assert.AreEqual(1, errorMassages.Count);

Assert.IsTrue(errorMassages[0].Contains("MethodWithVoidResult"));

}

[Test]

public void DoNotStop\_WhenLogErrorThrowsException()

{

var testClass = Substitute.For<ITestClass>();

testClass.When(t => t.MethodWithVoidResult()).Do(info => { throw new Exception(); });

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => { throw new ArgumentNullException(); },

o => o?.ToString());

Assert.Throws<Exception>(() => decorated.MethodWithVoidResult());

testClass.Received().MethodWithVoidResult();

Assert.AreEqual(1, infoMassages.Count);

}

[Test]

public void LogInfo\_WhenTaskMethodCall()

{

var testClass = Substitute.For<ITestClass>();

testClass.MethodWithTaskResult().Returns(Task.CompletedTask);

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString(),

\_taskScheduler);

decorated.MethodWithTaskResult().Wait();

ReleaseContext();

testClass.Received().MethodWithTaskResult();

Assert.AreEqual(2, infoMassages.Count);

Assert.AreEqual(0, errorMassages.Count);

}

[Test]

public void LogInfo\_WhenTaskMethodCallWithException()

{

var testClass = Substitute.For<ITestClass>();

testClass.MethodWithTaskResult().Returns(Task.FromException(new Exception("ERROR!!!!")));

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString(),

\_taskScheduler);

Assert.Throws<AggregateException>(() => decorated.MethodWithTaskResult().Wait());

ReleaseContext();

testClass.Received().MethodWithTaskResult();

Assert.AreEqual(1, infoMassages.Count);

Assert.AreEqual(1, errorMassages.Count);

Assert.IsTrue(errorMassages[0].Contains("ERROR!!!!"));

}

[Test]

public void LogInfo\_WhenTaskHasResult()

{

var testClass = Substitute.For<ITestClass>();

testClass.MethodWithTaskStringResult().Returns("String result");

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString(),

\_taskScheduler);

var result = decorated.MethodWithTaskStringResult().Result;

ReleaseContext();

Assert.AreEqual("String result", result);

testClass.Received().MethodWithTaskStringResult();

Assert.AreEqual(2, infoMassages.Count);

Assert.IsTrue(infoMassages[1].Contains("String result"));

Assert.AreEqual(0, errorMassages.Count);

}

[Test]

public void LogInfo\_WhenTaskWithResultThrowException()

{

var testClass = Substitute.For<ITestClass>();

testClass.MethodWithTaskStringResult().Returns(Task.FromException<string>(new Exception("ERROR!!!!")));

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString(),

\_taskScheduler);

Assert.Throws<AggregateException>(() => decorated.MethodWithTaskStringResult().Wait());

ReleaseContext();

testClass.Received().MethodWithTaskStringResult();

Assert.AreEqual(1, infoMassages.Count);

Assert.AreEqual(1, errorMassages.Count);

Assert.IsTrue(errorMassages[0].Contains("ERROR!!!!"));

}

[Test]

public void LogInfo\_WhenOutParameter()

{

var testClass = Substitute.For<ITestClass>();

var val = "s2";

testClass.MethodWithOutParameter(out val)

.Returns(x =>

{

x[0] = "s5";

return 25;

});

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

var result = decorated.MethodWithOutParameter(out val);

var val2 = "s2";

testClass.Received().MethodWithOutParameter(out val2);

Assert.AreEqual(0, errorMassages.Count);

Assert.AreEqual(2, infoMassages.Count);

Assert.AreEqual(25, result);

Assert.AreEqual("s5", val);

Assert.IsTrue(infoMassages[1].Contains("25"));

Assert.IsTrue(infoMassages[1].Contains("stringParam:s5"));

}

[Test]

public void LogInfo\_WhenRefParameter()

{

var testClass = Substitute.For<ITestClass>();

var val = "s2";

testClass.MethodWithRefParameter(ref val)

.Returns(x =>

{

x[0] = "s5";

return 25;

});

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

var result = decorated.MethodWithRefParameter(ref val);

var val2 = "s2";

testClass.Received().MethodWithRefParameter(ref val2);

Assert.AreEqual(0, errorMassages.Count);

Assert.AreEqual(2, infoMassages.Count);

Assert.AreEqual(25, result);

Assert.AreEqual("s5", val);

Assert.IsTrue(infoMassages[0].Contains("stringParam:s2"));

Assert.IsTrue(infoMassages[1].Contains("25"));

Assert.IsTrue(infoMassages[1].Contains("stringParam:s5"));

}

[Test]

public void LogInfo\_WhenOutParameterOfValueType()

{

var testClass = Substitute.For<ITestClass>();

var val = 2;

testClass.MethodWithOutParameter(out val)

.Returns(x =>

{

x[0] = 5;

return 25;

});

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

var result = decorated.MethodWithOutParameter(out val);

var val2 = 2;

testClass.Received().MethodWithOutParameter(out val2);

Assert.AreEqual(0, errorMassages.Count);

Assert.AreEqual(2, infoMassages.Count);

Assert.AreEqual(25, result);

Assert.AreEqual(5, val);

Assert.IsTrue(infoMassages[1].Contains("25"));

Assert.IsTrue(infoMassages[1].Contains("intParam:5"));

}

[Test]

public void LogInfo\_WhenRefParameterOfValueType()

{

var testClass = Substitute.For<ITestClass>();

var val = 2;

testClass.MethodWithRefParameter(ref val)

.Returns(x =>

{

x[0] = 5;

return 25;

});

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

var result = decorated.MethodWithRefParameter(ref val);

var val2 = 2;

testClass.Received().MethodWithRefParameter(ref val2);

Assert.AreEqual(0, errorMassages.Count);

Assert.AreEqual(2, infoMassages.Count);

Assert.AreEqual(25, result);

Assert.AreEqual(5, val);

Assert.IsTrue(infoMassages[0].Contains("intParam:2"));

Assert.IsTrue(infoMassages[1].Contains("25"));

Assert.IsTrue(infoMassages[1].Contains("intParam:5"));

}

[Test]

public void LogInfo\_WhenMixedParameters()

{

var testClass = Substitute.For<ITestClass>();

var refParam = 0;

var outParam = "s2";

testClass.MethodWithMixedParameter(ref refParam, out outParam, true)

.Returns(x =>

{

x[0] = 5;

x[1] = "s5";

return 25;

});

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

var result = decorated.MethodWithMixedParameter(ref refParam, out outParam, true);

var refParam2 = 0;

var outParam2 = "s2";

testClass.Received().MethodWithMixedParameter(ref refParam2, out outParam2, true);

Assert.AreEqual(0, errorMassages.Count);

Assert.AreEqual(2, infoMassages.Count);

Assert.AreEqual(25, result);

Assert.AreEqual(5, refParam);

Assert.AreEqual("s5", outParam);

Assert.IsTrue(infoMassages[0].Contains("refParam:0"));

Assert.IsTrue(infoMassages[0].Contains("outParam:s2"));

Assert.IsTrue(infoMassages[0].Contains("param:True"));

Assert.IsTrue(infoMassages[1].Contains("25"));

Assert.IsTrue(infoMassages[1].Contains("refParam:5"));

Assert.IsTrue(infoMassages[1].Contains("outParam:s5"));

Assert.IsTrue(infoMassages[1].Contains("param:True"));

}

[Test]

public void LogInfoWithProperClassName()

{

var testClass = new TestClass2();

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass2>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

decorated.Method();

Assert.AreEqual(2, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("Class AOP.Tests.TestClass2"));

Assert.IsTrue(infoMassages[1].Contains("Class AOP.Tests.TestClass2"));

Assert.AreEqual(0, errorMassages.Count);

}

[Test]

public void LogInfoWithProperStackTrace()

{

var testClass = new TestClass2();

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass2>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

Assert.Throws<Exception>(() => decorated.MethodWithException());

Assert.AreEqual(1, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("Class AOP.Tests.TestClass2"));

Assert.AreEqual(1, errorMassages.Count);

Assert.IsTrue(errorMassages[0].Contains("Class AOP.Tests.TestClass2"));

//Only original Exception should be logged

Assert.IsFalse(errorMassages[0].Contains("LoggingAdvice.cs"));

Assert.IsTrue(errorMassages[0].Contains("LoggingAdviceTests.cs"));

}

[Test]

public void LogInfoWithProperStackTrace\_WhenTask()

{

var testClass = new TestClass2();

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass2>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString(),

\_taskScheduler);

Assert.Throws<AggregateException>(() => decorated.AsyncMethodWithException().Wait());

Assert.AreEqual(1, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("Class AOP.Tests.TestClass2"));

Assert.AreEqual(1, errorMassages.Count);

Assert.IsTrue(errorMassages[0].Contains("Class AOP.Tests.TestClass2"));

//Only original Exception should be logged

Assert.IsFalse(errorMassages[0].Contains("LoggingAdvice.cs"));

Assert.IsTrue(errorMassages[0].Contains("LoggingAdviceTests.cs"));

}

[Test]

public void LogInfo\_WhenPropertySet()

{

var testClass = Substitute.For<ITestClass>();

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

decorated.Property = "Str1";

testClass.Received().Property = "Str1";

Assert.AreEqual(2, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("set\_Property"));

Assert.IsTrue(infoMassages[1].Contains("set\_Property"));

Assert.AreEqual(0, errorMassages.Count);

}

[Test]

public void LogInfo\_WhenPropertyGet()

{

var testClass = Substitute.For<ITestClass>();

testClass.Property.Returns("Str1");

var errorMassages = new List<string>();

var infoMassages = new List<string>();

var decorated = LoggingAdvice<ITestClass>.Create(

testClass,

s => infoMassages.Add(s),

s => errorMassages.Add(s),

o => o?.ToString());

var value = decorated.Property;

var tmp = testClass.Received().Property;

Assert.AreEqual("Str1", value);

Assert.AreEqual(2, infoMassages.Count);

Assert.IsTrue(infoMassages[0].Contains("get\_Property"));

Assert.IsTrue(infoMassages[1].Contains("get\_Property"));

Assert.AreEqual(0, errorMassages.Count);

}

/// <summary>

/// Call to process scheduled tasks

/// </summary>

protected void ReleaseContext()

{

Task.Delay(1).Wait();

}

}

public interface ITestClass2

{

void Method();

void MethodWithException();

Task AsyncMethodWithException();

}

public class TestClass2 : ITestClass2

{

public void Method()

{

}

public void MethodWithException()

{

throw new Exception();

}

public Task AsyncMethodWithException()

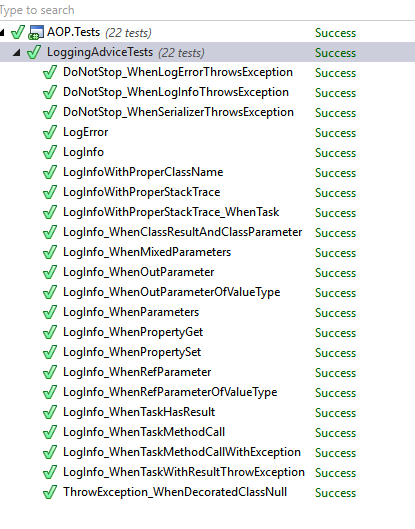
{

return Task.Factory.StartNew(() => { throw new Exception(); });

}

}

}



## 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!