Exceptions as an Aspect: Feasibility of Using AOP to Enforce Exception Handling in .NET

Bruno Cabral
CISUC, University of Coimbra
Dep. Eng. Informática – Pólo II
3030-290 Coimbra, Portugal
+351 239 79 00 00
bcabral@dei.uc.pt

ABSTRACT
With the advent of Aspect Oriented Programming a new approach to Exception Handling was introduced, suggesting that Exception Handling code should be separately developed from business logic code and later merged into it as a crosscutting concern.

This article discusses the feasibility of using Aspect Oriented Programming to enforce Exception Handling in .NET. The discussion is based on the data collected from several .NET programs regarding the Exception Handling constructs they use. The desired information is obtained by means of reflection over the applications.

This paper also demonstrates two different techniques, using IL code instrumentation, that allow the implementation of exception handlers as Aspects.

Keywords
Exception Handling, Aspect Oriented Programming, .NET Platform, Code Instrumentation.

1. INTRODUCTION
Exception Handling (EH) has been a research topic for many years. During this time much work was done by the object-oriented community to improve exception detection and handling in applications [1][2][3].

With the advent of Aspect Oriented Programming (AOP) [4][5] a new approach was introduced. Some authors suggested a clear separation between EH code and business logic code [6], proposing the treatment of exceptional behaviors as a crosscutting concern to the application. This allows developers to define aspects such as: “All the methods where the error X may occur must be prepared to handle it and react the following way...”

In this article is discussed the feasibility of using AOP to enforce EH in .NET applications. The work described starts with an analysis of the EH mechanisms implemented in several .NET applications. This analysis focuses in the data collected from the programs metadata and Intermediate Language (IL) code using reflection [7][8].

After this initial analysis, two code instrumentation techniques were developed with the intent of improving the exception handlers’ implementation process. They allow the implementation of EH as an Aspect. Meaning that, EH code can be developed separately from business logic code and latter merged following an AOP approach.

This article is organized in the following way: the following section discusses some of the related work; Section 3 presents the three commercial applications that will be used in this work to collect data about common EH practices in the industry; Section 4 describes the mechanism used to collect the just mentioned information about EH and discusses this data; Section 5 describes two approaches that use code instrumentation to separate the EH code from the rest of the application code and latter merge them to obtain a final product; In section 6 we draw some conclusions.

2. RELATED WORK
Since its origin in the early 1990’s at the Xerox Park [4], AOP has clearly identified EH as an application scenario. In the work reported at [6], Martin Lippert and Cristina Lopes show how an AOP tool for Java can be used to modify an application source code to apply exception handling and detection as a crosscutting concern.

The work described in this article is clearly an extension to the .NET platform of Lippert’s work. It is important to explain in what it consisted and the main differences with our work.

In [6] the JWAM [9], an existing framework written in Java, was partially reengineered in its exception detection and handling aspects using AspectJ [10], an AOP extension to Java. For doing so it was required access to the original source code in order to remove and re-write the EH code. Several data was collected from the source code, pre and post reengineering, to analyze the advantages and disadvantages of the process.

This article targets the .NET platform. In .NET, programs are called assemblies; an assembly is a collection of modules (files) that implement a set of classes [11].

Our work does not require the source code of the applications being analyzed since it uses reflection mechanisms to collect information from the assemblies’ metadata [11] and IL code. Instead of reengineering an application, this work focuses in the analysis of three .NET applications EH mechanisms and in the demonstration of two IL code instrumentation techniques for EH enforcing.

It must also be noted that EH policies in .NET are different from Java. For instance, .NET does not have a “throws” statement

equivalent [11]. Java methods use the "throws" statement to declare the exceptions types that can be throw and force the code that uses these methods to "catch" the declared exceptions [12].

.NET prefers to trust that the developers will use the documentation to declare which exceptions a method may throw and does not force the use of protected scopes for methods that actually declare that information [11].

3. TARGET APPLICATIONS

The first part of this study is the analysis of the EH mechanisms implemented in several .NET applications. This analysis was based in the information collected using a reflection mechanism. Three .NET applications were chosen for being very representative of several characteristics of the .NET applications universe.

3.1 Fotovision

Fotovision [13] is a sample photo manipulation and sharing application for the .NET platform, developed by Vertigo Software, Inc in association with Microsoft.

Its main objective is to demonstrate the combined features of Windows Forms, ASP.NET, XML Web services, and .NET Compact Framework but in reality is a very functional .NET product. It was completely developed in VB.NET and consists of a desktop, web and Pocket PC applications. In our work only the desktop application is analyzed.

Because this application uses a lot of .NET features it is a good example of commercial development using the .NET framework.

3.2 FxCop

FxCop [14] was developed inside Microsoft and it is a rule based engine which helps the developer to enforce industry coding best practices in the design and implementation of .NET software libraries.

FxCop is a code analysis tool that checks .NET assemblies for conformance to the Microsoft .NET Framework Design Guidelines. An assembly is a set of modules (file) that declare and implement different sets of classes. FxCop uses reflection, MSIL parsing, and callgraph analysis to inspect assemblies for defects in the following areas:

- Library design
- Localization
- Naming conventions
- Performance
- Security

FxCop includes both GUI and command line versions of the tool, as well as an SDK to create custom rules.

3.3 System.Web.dll

The System.Web.dll is a .NET framework assembly; it includes important namespaces for web applications development such as:

- System.Web
- System.Web.Hosting
- System.Web.Mail
- System.Web.Security
- System.Web.UI
- And others.

The EH practices observed in this assembly should be very representative of the kind of concerns with EH present in the framework development.

4. EXCEPTION HANDLING ANALISYS

A reflection based technique was used to collect data from the target applications described in the previous section. This data was collected with the intent of better understanding the way EH mechanisms are implemented in these programs. This information is useful to discuss the advantages and shortcomings of the separated writing of the EH code and the business logic code.

4.1 Data Collecting Mechanism

For collecting information about the EH mechanisms inside assemblies was used the Runtime Assembly Instrumentation Library (RAIL) [15][16].

This library creates an object-oriented structure representing the assembly members. The objects representing the Intermediate Language (IL) code instructions are placed in tables.

There are several classes in the library dedicated to the representation of the EH structures like "try-catch" blocks, "catch" handlers and "finally" handlers. Each method that uses these structures has a table associated with it that holds the objects representing those constructs.

```
<ExceptionReport>
  <ExceptionsTable>
    <Type>
        Microsoft.Tools.FxCop.Common.
        ProjectVersionMismatchException
    </Type>
    <CodeSize>12</CodeSize>
    <Code>
        pop ldarg.0 ldstr ldstr call ldstr
        ldc.i4.0 ldc.i4.s ldc.i4.0 call pop
        leave.s
    </Code>
  </ExceptionsTable>
  <ExceptionsTable>
    <Type>System.ArgumentException</Type>
    <CodeSize>5</CodeSize>
    <Code>
        stloc.s ldloc.s ldloc.0 call leave.s
    </Code>
  </ExceptionsTable>

…
</ExceptionReport>
```

Figure 1. Extract from the XML document generated for FxCop application

By the analysis of these tables is possible to find out which instructions are inside the protected scopes, how many handlers a protected scope has, which type of exceptions are being handled,

where do handlers start and finish and the IL instructions of each handler.

A small application was written that uses this information to create an XML file which contains the results of the analysis.

In Figure 1 is illustrated the generated XML documents, it is possible to observe the following tags: `<ExceptionReport>` is the parent tag for all the report; `<ExceptionsTable>` represents each "catch" statement in the code of the application; `<Type>` describes the type of the exception being handled; `<CodeSize>` records the number of IL instruction in that handler; `<Code>` the opcodes (IL instructions name) of the instructions inside the handler.

It was conducted another study related to EH. It consisted in finding out which possible exceptional behaviors are not handled by application code. For doing so was used an application named Analyser [17], also developed in the University of Coimbra, which uses RAIL to generate a XML document with this information.

When analyzing this information, it is important to have in mind that there are differences between EH in Java and in .NET. One of those differences is that Java forces the programmer to declare what kind of exceptions a method throws by the use of the "throws" statement. .NET, on the other hand, does not provide a "throws" statement or any other equivalent. It prefers to trust that programmers will use its code documentation standards to describe this information and there state which exceptions a method may raise [11]. Any further analysis of these differences is out of the scope of this article.

```
<exception_report
  assembly_path="FotoVision.exe"
  assembly_name="FotoVision,
  Version=1.0.1034.32473,
  Culture=neutral,
  PublicKeyToken=6250413531e7a2c0" type="">
...
<summary ...>
...
<exception
  name="System.MissingMethodException"
  amount="7149" percentage="22,62986%">
</exception>
<exception
  name="System.NullReferenceException"
  amount="9426" percentage="29,83761%">
</exception>
...
<exception
  name="System.OverflowException"
  amount="287" percentage="0,9084866%">
</exception>
<exception
  name="System.Security.SecurityException"
  amount="7898" percentage="25,00079%">
</exception>
...
</summary>
</exception_report>
```

Figure 2. Extract from the XML document generated by the Analyser for the Fotovision application

Using this information about exceptions included in code documentation and in the .NET framework documentation, it was then possible to find which exceptions are not handled in the code. Figure 2 shows a sample of the XML document produced with Analyser for the Fotovision application.

In Figure 2 only a very small extract from the generated document is shown. Although there is much more information available, only the summary section holds information relevant for the current study. The tags used in the file are: `<exception_report>` that identifies the target assembly; `<summary>` initiates the summary section of the file; the `<exception>` tag that identifies the type of the exception, the number of times it was found not handled inside the code and its relative height (percentage) in the total number of the not handled exceptions found.

4.2 Replicated EH Code

In these section is discussed the information withdrawn from the test applications regarding the number of exception handlers, the type of instructions handled (the "finally" statement was ignored because it does not identify the type of exception treated) and the instructions compounding the handler.

Table 1 presents the data collected from the Fotovision application. The first column identifies the type of exception found, the second the number of "catch" statements for that particular type of exception and the last column the number of different handlers in all the "catch" statements. It is the same as saying the number of different sequences of instructions found inside each handler.

From this data is possible to immediately draw some conclusions. First there are 50 "catch" statements for the `System.Exception` type and only 12 different handlers. This means that there is code duplicated several times across the assembly. Writing repeatedly the same code is a dull task for the programmer and error-prone.

Exception type	Number of "catch" statements	Different handlers
System.Exception	50	12
System.Net.WebException	1	1
System.Threading.ThreadAbortException	2	1

Table 1. Types of exceptions, number of "catch" statements and different handlers in Fotovision

The best possible case would be to write each different handler one time. That would provoke a significant reduction in the number of handlers effectively written in the development of this application. From a total of 53 handlers only 14 would be written, less 73.5% then originally.

Considering that the average number of instructions of all the handlers inside the assembly is 8.23. The decrease in the number of instruction effectively used would be of approximately 321 instructions. It is no possible to infer what would be the effective decrease in lines of code (source code), but all the "try-catch" statements would disappear from the method code and all the EH code would be concentrated in the same place. This would make

the code more readable avoiding the tangling between EH code and the business logic code [18].

Exception type	Number of "catch" statements	Different handlers
System.Exception	46	17
System.FormatException	5	1
System.IO.FileNotFoundException	2	2
System.ArgumentException	2	2
System.Runtime.InteropServices.COMException	2	2
System.IO.PathTooLongException	1	1
Microsoft.Tools.FxCop.Common.ProjectVersionMismatchException	1	1
System.NotSupportedException	1	1
System.Object	1	1
System.OverflowException	1	1
System.Xml.XmlException	1	1

Table 2. Types of exceptions, number of "catch" statements and different handlers in FxCop

In Table 2 we have the same kind of results found in Table 1, but this time relative to the FxCop application.

Inside the FxCop application, more precisely at the FxCop.exe assembly there are, for instance, 46 "catch" statements for the `System.Exception` type and only 17 different handlers. For `System.FormatException` there are 5 "catch" statements and only 1 kind of handler. On the other hand, for the rest of the 9 effectively handled exceptions all the handlers are different from each other.

Based on this data the improvement would be of 52.3% less handlers being effectively written. That would represent approximately 363 less instructions in handlers knowing that the average size of handlers is 11 here.

The results obtained from the analysis of the System.Web.dll assembly are shown in Table 3. The most significative examples of the use of replicated handlers are associated with the "catch" statements for the `System.Exception` and `System.Object` types. `System.Object` appears in this context as an exception type because in .NET it is possible to write "catch" statements without any arguments, when this happens any object can be thrown and the compiler associates `System.Object` as the class of the object to catch.

If each different handler was written only one time, there would be an improvement of 53.2% less "catch" statements. This improvement approximately represents 2094 less instructions.

But, only similar handlers were compared for the same type of exception. Another question can be formulated regarding the use of identical handlers for different types of exceptions. It would be possible to write much less handlers if they were re-used across different types of exceptions.

From the analysis of the collected data from this new point of view, it was concluded that some handlers are common to different types of exceptions. Nevertheless, this scenario does not occur in the Fotovision application, where the number of different handlers remains 14 even doing inter-exception-type comparative. The only significate results are for the FxCop application, where the effective number of different handlers would decrease from 30 to 24. Representing an improvement of 61.9% less handler wrote regarding the original assembly.

Exception type	Number of "catch" statements	Different handlers
System.Exception	125	72
System.Object	106	26
System.ArgumentException	5	3
System.Configuration.ConfigurationException	4	2
System.Threading.ThreadAbortException	4	3
System.FormatException	4	1
System.Xml.XmlException	2	2
System.Threading.ThreadInterruptedException	2	2
System.IO.IOException	2	2
System.Data.ConstraintException	1	1
System.IndexOutOfRangeException	1	1
System.InvalidCastException	1	1
System.IO.DirectoryNotFoundException	1	1
System.IO.FileNotFoundException	1	1
System.IO.PathTooLongException	1	1
System.Reflection.TargetInvocationException	1	1
System.Runtime.Serialization.SerializationException	1	1
System.Security.SecurityException	1	1
System.Threading.ThreadInterruptedException	1	1
System.Web.HttpException	1	1

Table 3. Types of exceptions, number of "catch" statements and different handlers in System.Web.dll

In the System.Web.dll there are globally only 104 different types of handlers. That is a reduction of 16.13% from the originally accounted 124 different handlers. This would decrease the number of effectively written handlers from 47.7% to 40% which, considering the size of this assembly, represents a lot less lines in the original source code.

4.3 Size of Handlers

Other information that is useful to understand how the EH mechanisms are implemented is looking at the size of the handlers. For doing this analysis the number of handlers with the

same number of instructions was counted for each application. The results are shown in Figure 3.

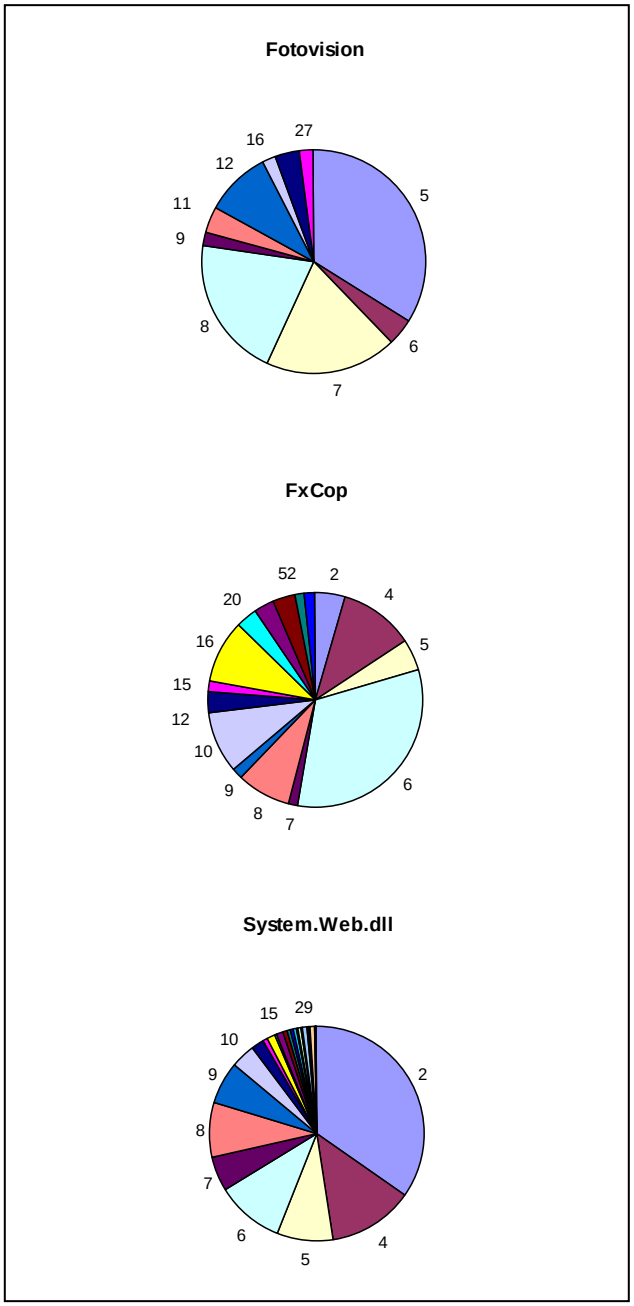

Figure 3. Distribution of handlers characterize by the number of instructions that compound them

An analysis of this data shows a common pattern in all the applications. The majority of the handlers, approximately 75%, have less then 9 instructions.

This is an interesting detail and it leads to the conclusion that the code of the handlers is usually very simple and sometimes even not existent. This is supported by the presence of a lot of handlers with only 2 IL instructions, `pop` and `leave`, generated by the .NET compilers when they encounter an empty "catch" block. This type of handler is very frequent in System.Web.dll assembly as can be inferred analyzing Figure 3.

A deeper analysis of the code of the handlers revealed that the great majority of handlers are used for logging the exception and/or warning the user of the error occurred.

4.4 Not Handled Exceptions

Other information related with EH that was possible to extract was the total number of instructions that could throw some kind of exception and were not inside any protected block. This information is shown in Table 4 for each one of the applications.

Interestingly all the top three less handled exceptions types are common to all the applications. They are:

- `System.NullReferenceException`;
- `System.Security.SecurityException`;
- `System.MissingMethodException`.

But, the main reason of this last analysis of the applications is to find out if any of the handled exceptions types may still be raised by a instruction out of the existing protection scopes. This can be verified by comparing the Exception Type column in Tables 1, 2 and 3 with Table 4.

In Fotovision this scenario never happens, in FxCop it happens with `System.OverflowException` where it may occur in 110 instructions. Inside System.Web.dll assembly, there are 4 exceptions types in this situation, `System.Security.SecurityException`, `System.InvalidCastException`, `System.IndexOutOfRangeException` and `System.ArgumentException`.

As said before, it is not possible to assure that there are not other kinds of mechanisms preventing these exceptions from being thrown.

To conclude this section are resumed some of the quantitative and qualitative analysis made.

4.4.1 Quantitative Analysis

Handlers were classified accordingly to its instructions. Handlers of the same kind execute the same operations (do the same thing).

The most interesting result withdrawn from this section is the possible decrease in the amount of handlers effectively written. The average of handler saving for the three applications is of approximately 59.67% when, each different handler is written only one time per instruction type.

An even major improvement of approximately 65.13% less handlers, is achieved when one kind of handler is written only once independently of the exception types.

4.4.2 Qualitative analysis

Separating the EH code from the business logic code avoids some tangling between the two types of code. Application code becomes more readable and the debug tasks become simpler.

With less tangling in the code it becomes more tolerant to design modifications, the re-implementation of business logic may, in extreme situations, not affect any of the EH code.

If the developer is fully concentrated in implementing the application main functionality, it is probable that he will make fewer errors. Obviously, this separation will also oblige him to

fully concentrate in the EH tasks latter, and this can contribute to improve the quality of EH code and diminish the number of empty catch statements that many times populate source code.

Exception type	Number of error-prone instructions	%	
System.NullReferenceException	9426	29,8	Fotovision
System.Security.SecurityException	7898	25	
System.MissingMethodException	7149	22,6	
System.MissingFieldException	2724	8,62	
System.Security.VerificationException	1950	6,17	
System.OutOfMemoryException	968	3,06	
System.ArrayTypeMismatchException	335	1,06	
System.IndexOutOfRangeException	335	1,06	
System.OverflowException	287	0,91	
System.InvalidCastException	147	0,47	
System.InvalidOperationException	81	0,26	
System.ArithmeticException	12	0,04	
System.DivideByZeroException	12	0,04	
System.NullReferenceException	14725	29,6	FxCop
System.Security.SecurityException	10828	21,8	
System.MissingMethodException	10002	20,2	
System.MissingFieldException	6047	12,2	
System.Security.VerificationException	3815	7,68	
System.OutOfMemoryException	2053	4,13	
System.ArrayTypeMismatchException	510	1,03	
System.IndexOutOfRangeException	510	1,03	
System.InvalidCastException	300	0,6	
System.OverflowException	110	0,22	
System.InvalidOperationException	53	0,11	
System.DivideByZeroException	7	0,01	
System.ArithmeticException	3	0,006	
System.NullReferenceException	25479	26	System.Web.dll
System.Security.SecurityException	22529	23	
System.MissingMethodException	15664	16	
System.Security.VerificationException	15601	15,9	
System.MissingFieldException	9449	9,66	
System.OutOfMemoryException	2887	2,95	
System.InvalidCastException	1118	1,14	
System.ArrayTypeMismatchException	1098	1,12	
System.IndexOutOfRangeException	1098	1,12	
System.OverflowException	510	0,52	
System.InvalidOperationException	324	0,33	
System.DivideByZeroException	73	0,07	
System.ArithmeticException	48	0,05	
System.ArgumentException	19	0,02	
System.ArgumentNullException	19	0,02	
System.NotSupportedException	13	0,01	
System.ArgumentOutOfRangeException	1	0,001	

Table 4. Types of exceptions and number of instructions prone to raise them

Other import result is the re-utilization of code; it will avoid many of the "copy-and-paste" operations and decrease the amount of error usually produced by these actions.

But, new problems can be introduced. If the EH code uses many of the business logic constructions, the merging and even the separate writing of the code can become a complex task. This shortcoming will become more understandable in the next section.

5. EH AS AN ASPECT

.NET already provides some ways of implementing EH differently from the traditional approach. For instance, there is the Exception Management Application Block [19] that provides a simple yet extensible framework for handling exceptions. With a single line of application code it is possible to log exception information to the Event Log or extend it by creating components that log exception details to other data sources or notify operators, without affecting application business logic code. The Exception Management Application Block can be used as a building block in a .NET application.

A shortcoming is the need to write at least one line of code per handler. This line is repeated many times across the program's code which is not very elegant.

It is possible to identify this line of code as a crosscutting concern to the application and use AOP techniques.

The .NET platform has AOP support, in the shape of Custom Attributes and `ContextBoundObject` [20]. A Custom Attribute extends the Common Language Runtime (CLR) [11] and can be associated with any one of the assembly members. When associated with methods, Custom Attributes allow the execution of methods before and after the target method. They can even propagate an exception through all the methods in the pre and post execution list.

Unfortunately is a .NET requirement that, to use Custom Attributes, classes must derive from the `ContextBoundObject` class in order to supply a context for the execution of the methods associated with the Attribute. `ContextBoundObjects` are very slow when compared with normal objects [21].

Ideally the result for the merge operation between the EH code and the application business logic should not modify the OO hierarchy of the application; using Custom Attributes this requirement would be compromised.

The approach described in this paper is compliant with this requirement. First the code of the application business logic is written, compiled using the platform compiler (or any other) producing a new assembly. Using a code instrumentation library,

this newly created assembly is manipulated and the exception handlers are introduced into the code. Finally a new assembly is generated and is ready to be executed.

In this work exceptions are treated as aspects, a good abstraction illustrating this is "all methods that raise an exception of type `<exception type>` should react doing `<sequence of operations>`"

To accomplish this we propose two different techniques, the first one involves minor tangling with the business logic code. The second is completely autonomous.

5.1 Use of Custom Attributes

The first technique uses Custom Attributes to identify the types of exceptions that a method can throw. Because they are only used for this reason and not for executing code before or after the method, the classes that declare the methods "marked" no longer have to inherit from `ContextBoundObject` since no context information is required by the Custom Attribute objects. Custom Attributes are used only as markers for methods, special markers that are kept after the source code compilation a let us know which types of exceptions should be handled in that method.

The code for the Custom Attribute used is shown in Figure 4, it corresponds to the first block of code and is really very simple. In the b) block of code of the same figure is illustrated the way this attribute is used. The two attributes mark that method to be instrumented with the correct handling code for the `System.IO.IOException` and `System.OverflowException`.

The last block of code in Figure 4 is the actual instrumentation code using the RAIL library. As we said before this library builds an Object-Oriented (OO) representation of an assembly and uses tables to hold objects representing methods' IL instructions. It also uses tables to save EH information for each method.

The piece of code in question runs through the OO structure of the assembly, using a Visitor pattern, to look for method marked with the Exception attribute holding the value System.IO.IOException. When it founds a method matching this requirement, it creates a new table to hold the EH data for that method (let us call it Exception table). In this table it inserts a new definition of a protected scope, starting at the first instruction of the method and ending in the last instruction before the method return.

For this example we only developed one kind of handler, its functionality is to catch and log the exception. Other handlers could be added and classified using the `HanlderCode` enumeration. This enumeration is used as a parameter to the method that adds handlers to the application.

The writing of the code to produce the handlers is probably the most complex task here, especially if the handling code includes references to method arguments and local variables. But it is not an impossible task; it requires only some creativity from the programmer.

The logging code is appended to the original method body and the handler for the desired exception type is registered in the Exception table of the method. A new return point is set in the method.

Finally the Custom Attribute is removed since there is no other use to it and the new assembly is generated.

a)

```
[AttributeUsage(AttributeTargets.Method,
  AllowMultiple=true,
  Inherited=false)]
public class ExceptionAttribute :
System.Attribute
{
  private Type exceptionType;
  public ExceptionAttribute(Type
    exceptionType)
  {
    this.exceptionType = exceptionType;
  }

  public Type ExceptionType
  {
    get
    {
      return this.exceptionType;
    }
  }
}
```

b)

```
[Exception(typeof(System.IO.IOException))]
[Exception(typeof(System.OverflowException))]
public static void Main(string [] args)
{
 //Method code
}
```

c)

```
RAssemblyDef rAssembly =
RAssemblyDef.LoadAssembly("example.exe");
RType rt = rAssembly.GetRTypeRef(
  "System.IO.IOException");
AddExceptionHandling aeh1 =
  new AddExceptionHandling(
        rt,HandlerCode.CatchAndLog,true);
rAssembly.Accept(aeh1);
rAssembly.SaveAssembly("label.exe");
```

a) The Custom Attribute class we created to indicate which exceptions should handle inside the marked method.

b) Examples of method marking with the Exception attribute.

c) The instrumentation code to apply a "try-catch" block to methods marked for `System.IO.IOException` type with logging functionalities.

Figure 4. Examples of code

It is possible to notice that the granularity of the protected scopes is at method level. It means that we are not able to automatically set a protected scope for only a small set of instructions inside a method. To do so would be a very complex task, and it would involve the fulfillment of some rules. For instance, two of those rules are:

- The stack must be empty before entering a "try-catch" block.
- The stack must be clean before exiting a "try-catch" block.

The enforcement of these rules would lead to complex stack simulation mechanism with the only objective of validating the position where to insert begin and end of the protected block.

As a workaround to accomplish this two static methods were created to be used as labels for the start and end of protected scopes in the source code. For methods with more attributes them markers, these markers are assumed to belong to the attributes define in the same order as they appear. If there are less attributes then markers, the last attribute is used for all remaining markers. Also the calls to the marking methods are removed from the final code, only its position is used to define the protected scopes in the Exception table.

5.2 Automatic EH

The next technique described completely avoids any EH code inside the application.

This technique consists in creating an XML report using the Analyser application described in section 4.1. Each register in this document describes the type of exception, the index of the instruction that may raise the exception, the method and the class to which that instruction belongs. It is possible to see an example in Figure 5.

```
<exception_report assembly_path="FxCop.exe"
  assembly_name="FxCop, Version=1.30.0.0,
  Culture=neutral,
  PublicKeyToken=31bf3856ad364e35"
  type="">
<method name=
"Microsoft.Tools.FxCop.UI.DictionaryView.
OnVisibleChanged(System.EventArgs)">
  <exception type="code" id="Exception1">
    <name>System.ArithmeticException</name>
    <code_line>8</code_line>
  </exception>
  <exception type="code" id="Exception2">
    <name>System.DivideByZeroException</name>
    <code_line>8</code_line>
  </exception>
<exception type="code" id="Exception3">
...
```

Figure 5. Extract of the XML report for FxCop application

As it was said before, the analyzer looks for the exceptions that a method may raise by parsing the documentation of the framework or of the libraries used by instructions inside the method.

First is built a table that holds the identification of the methods that raise the exception type being targeted. This knowledge is achieved by parsing the XML report of the Analyser.

The second step is similar to the one of the previously described technique. It consists in creating one Exception table per method identified, marking the complete method body as protected scope (because of the complexity discussed earlier we are not able to mark only a subset of instructions as the protected scope), add the handler code to the method, update the exception table info and set a new return point in the method.

Figure 6 shows the code needed for doing the described instrumentation for the `System.IO.IOException`.

And this sets protected scopes, exception handlers with logging functionalities across the assembly.

```
RAssemblyDef rAssembly =
RAssemblyDef.LoadAssembly("Example.exe");
RType rt = rAssembly.GetRTypeRef(
  "System.IO.IOException");
AddExceptionHandling aeh1 =
  new AddExceptionHandling(
    rt,
    HandlerCode.CatchAndLog,
    "exception_report.xml");
rAssembly.Accept(aeh1);
rAssembly.SaveAssembly("Example.exe");
```

Figure 6. Instrumentation code example

5.3 Performance Analysis

There is no visible variation in execution time between an application developed the traditional way, writing “try-catch” blocks and handler code directly into the applications code or by any of the presented instrumentation techniques. And the reason is that the produced assembly uses the same constructs for EH as the traditional one and the IL code produced is very similar if not identical. No complexity, extra-objects or context swap operations, common in AOP, are added to the application by means of this instrumentation.

6. CONCLUSIONS

The work reported in this article shows that it is possible to separate EH code from the business logic code in .NET. Plus, it proves that there is much to gain in code re-usability, simplicity and readability. It reduces significantly the amount of code effectively written and diminishes the error probability in the code of the handlers and even in the business logic code, since it lets the developer fully concentrate in its writing.

The techniques shown to accomplish this are still very rudimentary but with the present interest in AOP, some more advanced tools will eventually appear. These AOP tools will simplify, for instance, the handler code generation feature.

Some difficulties, like diminishing the granularity of the protected scopes from the method to the block of code, will need further research especially when looking for a completely automatic process.

In .NET the EH policy seams to support the use of AOP for EH implementation. .NET compilers allow applications to be compiled without a single “try-catch” statement, allowing the merging of EH mechanism directly into the application. This would be quite impossible in real world applications written in Java and avoids some complex source code manipulations.

The introduction of EH mechanisms into a working assembly as another advantage that is the possible use of this type of EH in assemblies written in languages that do not support it.

7. ACKNOWLEDGMENTS

This project was partially supported by FCT, thought CISUC (R&D unit 326/97), scholarship SFRH/BD/12549/2003.

8. REFERENCES

[1] Bordiga, A. Language Features for Flexible Handling of Exceptions in Information Systems. *In ACM Transactions on Database Systems*, Vol. 10, No. 4, pp. 565-603, Dec. 1985.

[2] Dony, C. Exception Handling and Object-Oriented Programming: towards a synthesis. *In Proceedings of OOPSLA/ECOOP '90*, SIGPLAN Notices, Vol. 25, No. 10. ACM Press, October 1990.

[3] Meyer, B. *Object-Oriented Software Construction*, Prentice Hall, New Jersey, 1997.

[4] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. Aspect-Oriented Programming. *In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, Springer-Verlag, Finland, 1997, pp.220-242.

[5] Elrad, T., Filman, R.E., and Bader, A. Aspect-Oriented Programming. *Communications of the ACM*, ACM Press, New York, New York, USA, October 2001, Vol.44, N.10, pp. 29-32. ISSN 0001-0782.

[6] Lippert, M., Lopes, C. V. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. *In Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, ACM Press, New York, NY, USA.

[7] Ferber, J. Computational reflection in class based object-oriented languages. In *Proceedings of the 4th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, ACM SIGPLAN, New Orleans, Louisiana. October 1989.

[8] J. Malenfant, C. Dony, and P. Cointe. Behavioral Reflection in a Prototype-Based Language. In *Workshop on Reflection and Meta-Level Architectures (IMSA'92)*, Tokyo, 1992.

[9] JWAM web site, 2004, available at: http://www.jwam.de.

[10] Lopes, C., Kiczales, G. Recent Developments in AspectJ. *In ECOOP '98 Workshop Reader*, Springer-Verlag, 1998.

[11] ECMA International. Standard ECMA-335 Common Language Infrastructure (CLI), ECMA Standard, 2003, available at: http://www.ecma-international.org/publications/standards/ecma-335.htm.

[12] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification*. Sun Microsystems, Inc, Mountain View, California, U.S.A., 2000. ISBN 0-201-31008-21.

[13] Fotovision web site, Vertigo Software, Inc, 2004, available at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetcomp/html/FotoVisionDesktop.asp.

[14] Microsoft Corporation FxCop web site, Microsoft Corporation, 2004, available at: http://www.gotdotnet.com/team/fxcop/.

[15] Cabral, B., Marques, P., Silva, L. IL Code Instrumentation with RAIL. *In .NET Developers Journal*, Vol. 2, Number 1, pp. 34, January 2004.

[16] CISUC RAIL web site. University of Coimbra, July 2004, available at: http://rail.dei.uc.pt/stats

[17] Sacramento, P. *Estudo da Robustez dos Mecanismos de Tratamento de Erros por Excepções na Platforma .NET - Relatório de Estágio*. DEI, Universidade de Coimbra, Julho 2004.

[18] Dunn, M. F., Knight, J. C. Software Reuse In An Industrial Setting: A Case Study, IEEE, 1991.

[19] Jone, K., Malcolm, G., Mackman, A., Jezierski E. Exception Management Application Block for .NET. *In Microsoft Patterns and Practices for Application Architecture and Design*. Microsoft Corporation, April 2002, available at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp.

[20] Microsoft Corporation ContextBoundObject Class. In .NET Framework Class Library. Microsoft Corporation, 2004, available at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcontextboundobjectclasstopic.asp.

[21] Microsoft Corporation Remotable Objects. In .NET Framework Class Library. Microsoft Corporation, 2004, available at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconRemotableObjects.asp.