Aspect Oriented Programming Through C#.NET

Harsha Bopuri *bopuri@gmail.com*
Director Business Applications/ IT Developments
IMATRIX Corp
North Brunswick, NJ 08902 USA

Prof. Dr. Raied Salman *rsalman.faculty@unva.edu*
University of Northern Virginia
Adjunct faculty, Computer Science Department
7601 Little River Turnpike, Annandale, VA 22003, USA

Abstract

.NET architecture was introduced by Microsoft as a new software development environment based on components. This architecture permits for effortless integration of classical distributed programming paradigms with Web computing. .NET describes a type structure and introduces ideas such as component, objects and interface which form the vital foundation for distributed component-based software development. Just as other component frameworks, .NET largely puts more emphasis on functional aspects of components. Non-functional interfaces including CPU usage, memory usage, fault tolerance and security issues are however not presently implemented in .NET's constituent interfaces. These attributes are vital for developing dependable distributed applications capable of exhibiting consistent behavior and withstanding faults.

Keywords: Aspect Oriented Programming, Cross Cutting Concerns.

1. INTRODUCTION

Aspect Oriented Programming (AOP) is a new development technology that permits separation of crosscutting concerns that have in the past proved difficult to implement using object oriented programming (OOP). According to [4], AOP is an elegant and simple construct with the ability of really altering the manner in which we develop software. It is a way of performing arbitrary code orthogonal to the primary purpose of a module, with the purpose bettering the encapsulation and reuse of the arbitrarily invoked code and the target module. Crosscutting concerns exists in most large systems; however, in others, the system may be redesigned to convert the crosscutting into an object. For aspect oriented programming though, the assumption is that crosscutting concerns exists in systems by default and cannot be transformed out of the system design.

1.1. Crosscutting Concerns

AOP divides crosscutting concerns into single parts referred to as aspects. An aspect represents a modular part of crosscutting implementation. Under AOP, we initially implement a project using an object oriented language such as Java or C# then independently handle crosscutting concerns by implementing aspects. In the end, an aspect weaver is used to integrate the both the code and the aspect into an executable file.

Component based programming is a simplistic method to compose systems out of units having contractually precise boundaries and unequivocal context dependencies. Since software components are developed by third parties, they can be deployed autonomously. Several distributed component frameworks exits including; Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), .NET framework among others. Despite the fact

that the implementation of intricate distributed systems is considerably simplified by these frameworks, there is limited support for techniques such as fault tolerance, reliability and security. Fault tolerance expansion for components needs to substitute abstraction and encapsulation with the execution explicit knowledge concerning a component’s internal timing performance, memory usage, CPU usage, and communication and access models
AOP is best illustrated by example, the best one being event logging [20]. Let us say you have a class Foo and you want to write to a log file every time a particular system is called for auditing purposes or rudimentary performance statistics. You may normally write code like the following to meet this requirement:

```
public class Foo
        {
            protected EventLog eventLog;
            public Foo()
            {
                eventLog = new EventLog(); // create an event log
                eventLog.Source = "Foo Application"; // Name a Source
            }
            public void bar()
            {
                eventLog.WriteEntry("Bar method begin");
                // do bar()
                eventLog.WriteEntry("Bar method end");
            }
        }
```

Is there anything incorrect with this code? Historically, nothing is really incorrect. However, this is just because we are accustomed to writing codes like that. It is considered okay to incorporate EventLog code in the Foo class because before AOP there was no method of logging events without clearly calling event logging code from inside the class itself [12]. However, with the arrival of AOP, the code above would in fact be interpreted as very wrong, virtually prohibitive to incorporate in a Foo class. This is because everyone appreciates what should be integrated in a Foo class i.e. bar methods and not logging. Therefore, if the above was to be accomplished using AOP, it would look as follows:

```
       [EventLoggingAttribute]
        public class Foo : ContextBoundObject
        {
            public void bar()
            {
                // do bar()
            }
        }
```

The above shows that the code tangential to the bar() method is transferred to another place, particularly, the logging aspect. An aspect is executed without any more knowledge on the client’s part and is functionality factored out of a client’s module in an AOP - like approach. In the above example, the bar() technique does its job, regardless of other aspects. This is beneficial because it increases maintainability, improves reuse and encapsulation of both aspect and module code because of the introduction of decoupling.

2. METADATA AND REFLECTION IN .NET

Reflection refers to a programming language tool which permits access to type information during execution. This mechanism has been affected for various object oriented languages including java, C#.net and C++. .NET does not confine reflection to a single coding language but rather allows inspection of any .NET assembly using the reflection technology. Runtime type information in .NET can be accessed in two different ways namely; language runtime library and the unmanaged metadata interfaces.

2.1 Reflection Through Runtime Library

Under this, the reflection classes are declared in System. Reflection namespace. The GetType method, which is a public method, has a return value object of the typeTypecontained in the namespace System. The following definitions are represented in each type-instance.

- Class definition
- Interface definition
- Value-class
-

Through reflection we are able to query about any type characteristic including the access modifiers. The structure of metadata is one of hierarchical nature in which the class System.Reflection.Assembly is at the highest level of the hierarchy. An assembly object relates to at least one dynamic libraries (DLLs) which forms the building block of the .NET unit in question. As indicated in the figure below, System.Reflection.Module is located on the second level of the hierarchy. Drilling down further the metadata tree represents type information for any of the foundations for the .NET virtual object system member.

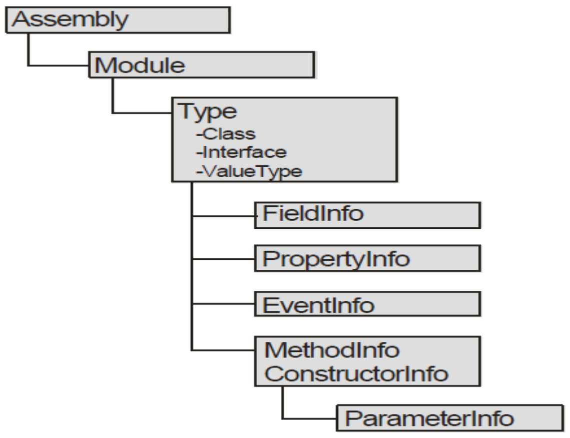

FIGURE 1: C#.NET Metadata Hierarchy.

In every circumstance, a class instance System.Reflection.MemberInfo stands for a single data element describing each of the below basic units constituting an object.

- Method (System.Reflection.MethodInfo)
- Constructor (System.Reflection.ConstructorInfo)
- Property (System.Reflection.PropertyInfo)
- Field (System.Reflection.FieldInfo)
- Event(System.Reflection.EventInfo)

2.2 Unmanaged Metadata Interface

These are an assortment of COM interfaces whose accessibility is external to the .NET environment. The interface definition is located in the COR.H, found in the Software development kit. The IMetaDataImport.IMetaDataAssemblyImportinterface aides in metadata accessibility on the .NET assembly level.

ImetadataDispenserinterface provides access to the metadata COM-interface IMetaDataImport.IMetaDataAssemblyImportinterface. The ImetadataDispenserinterface as the name suggests dispenses every types of additional metadata interfaces, permitting read and write access to the .NET metadata. The dispenser is hence accessed through calls to the COM interface.

3. FAULT TOLERANCE REQUIREMENTS EXPRESSED BY C#

We shall demonstrate a simple calculator program in C# to explain how functional C# and non-functional C# (aspect) codes can be integrated together.

3.1 The Calculator Program

As shown by the below code snippet, the C# calculator has been accomplished within a class Calculator found in the namespace Calculate. Operands are stored as data-members Ope1 and Ope2. A public member method Add is implemented by the class.

```
namespace Calculate {
    public class Calculator {
        public Calculator() { Ope1=0; Ope2=0; }
        public double Ope1;
        public double Ope2;
        public double Add() { return Ope1+Ope2; }
    }
}
```

3.1.1 The Unmanaged Metadata Interfaces

The unmanaged metadata interfaces are a collection of COM interfaces that are accessible from “outside” of the .NET environment. You can access them from any Windows program. The interface definition can be found in the COR.H, which is contained in the platform software development kit (platform SDK).

IMetaDataImport.IMetaDataAssemblyImport interface is used for accessing metadata on the .NET assembly level. Access to this interface is obtained via a second interface, called IMetadataDispenser. As the name indicates, this interface “dispenses” all kinds of additional metadata interfaces, which allow read and write access to .NET metadata. Access to the metadata dispenser is obtained via calls to the COM system.

```
hr = CoCreateInstance(
    CLSID_CorMetaDataDispenser, 0,
    CLSCTX_INPROC_SERVER,
    IID_IMetaDataDispenser,
    (LPVOID*)&m_pIMetaDataDispenser );
    hr = m_pIMetaDataDispenser->OpenScope(
    wszFileName,
    ofRead,
    IID_IMetaDataImport,
        (LPUNKNOWN *)&m_pIMetaDataImport );
```

3.1.2 Tolerating Crash-Faults in the Calculator

The C# characteristic we are going to implement will entrench fault-tolerance to the calculator class we earlier wrote. The new modified class permits the independent creation and management of objects by clients. Due to the fact that we are using a simpler application, we assume that only crash faults occur at the object level thus we propose a proxy object for management of copies which makes up a single point of failure. Consequently, we assume that consistency of replicas can be maintained without the need of interaction with other replicas. We shall use C#.NET removing so as to spread the object copies across machine interfaces. This would create a distributed environment that tolerates both object and process faults. To maintain replica consistency, consensus rules such as voting scheme and master-slave replication scheme should be implemented. We outline C# attribute to define fault-tolerance requirements

```
[TolerateCrashFault (n)]
```

The parameter n denotes the number of objects crash faults that are likely to occur before the interruption of the component services. N+1 object replicas are needed so as to tolerate n crash-faults of objects. For our application an attribute has been used to expand the definition of the Calculator class.

```
[TolerateCrashFault (4)]
public class Calculator {
/* ... */
}
```

For our calculator application, five replicas would be created and the services continue running as long as one or more object persists.

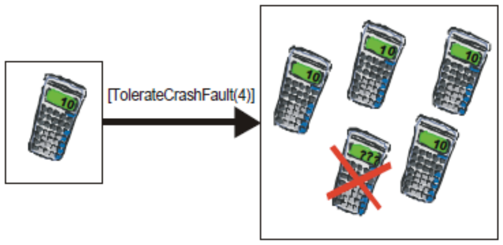

FIGURE 2: Replication in Space.

4. THE ASPECT WEAVER

This tool combines functional and aspect codes. For our case, we design a WrapperAssistant, which operates as our aspect weaver and generates snippets for replica administration. The Wrapper Assistant utilizes introspection and reflection techniques centered across the C#.NET CLR (Common Language Runtime) metadata to identify task signatures sent by a component and to create proxy classes for the exported classes. The TolerateCrashFault (n) attributecontrols the behavior of replica management scheme. The WrapperAssistantdialog provides the user with a list of classes that have been applied in a certain .NET assembly. Code will then be generated for the particular proxy class by the WrapperAssistantdepending on the class selected by the user in the list. The client programmer needs to make very few enhancements to the generated code; the programmer ought to modify just a line of code to utilize the added fault-tolerance enhancements.

```
using proxy;
// proxy namespace is imported by client
usingcalc;
//activates replica administration & fault-tolerance functionalities
void Calculate() {
    Calculator p = new Calculator (); // this comes from the proxy
    //namespace
    p.Ope1=4;
    p.Ope2=8;
    Console.WriteLine (c.Add ()); // writes to the console
}
```

4.1 Proxy Class Generation

Classes for replica management are generated by the WrapperAssistant inside the proxy namespace. The classes are instrumental in expanding the public classes employed in a particular component. For out calculator application, the below code is generated:

```
namespace proxy {
public sealed class Calculator:Calc.Calculator
    {
```

Every member role of the initial class is then overwritten with a version having an indistinguishable signature and routes the function calls to object copies instead of implementing them itself. The public variables of the initial class are declared as attributes in the tool-created proxy class. This would be as below for the

```
new public double Ope1 {
    get { /* ... */ }
    set { /* ... */ }
    }
```

The suitable count of base class interfaces has to be generated inside the constructor of the proxy class. The number is provided by the TolerateCrashFault attribute as shown below.

```
public sealed class
TolerateCrashFaults:System.Attribute {
    private int f_i;
    public TolerateCrashFaults(int i) {f_i=i; }
    public int Count
    { get { return f_i+1; } }
        }
```

The count of intolerable errors is internally recorded by the constructor. The count variable stores the number of copies that have to be created. Every overwritten member function in the class proxy routes its function-call to every occurrencereferenced in the collection. This would be represented as follows for the Add function.

```
public new double Add()
    { int i;
    double _RetVal=new double();
    for(i=0;i<_bc.Length;i++) {
    if(_bc[i]==null) continue;
    try { _RetVal=_bc[i].Add(); }
    catch(System.Exception) { _bc[i]=null; }
    }
        return _RetVal;}
```

4.2 Programmatic Tipping

Programmatic tipping is a technique used by high-level code weavers to assemble aspects from low-level devices. This technique allows addition of methods, types and fields programmatically. It is usually done using a compiled language. Below is an example of programmatic tipping.

```
public override void ProvideAspects(object targetElement,
    LaosReflectionAspectCollection collection)
{
  // Get the target type.  Type targetType = (Type) targetElement;
  // On the type, add a Composition aspect to implement
  // the IBindable interface.

  collection.AddAspect(targetType, new
AddBindableInterfaceSubAspect());

  // Add a OnMethodBoundaryAspect on each writable non-static property.
  foreach (PropertyInfo property in targetType.GetProperties())
  {
     if (property.DeclaringType == targetType &amp;&amp;
          property.CanWrite )
      {
          MethodInfo method = property.GetSetMethod();
          if (!method.IsStatic)
            collection.AddAspect(method,
                    new OnPropertySetSubAspect(property.Name, this));
       }
  }
   }
```

4.2.1 Custom Attributes

Custom attributes is an approach in which aspects are programmed as custom attributes and normally applied to fields, classes and methods. This example below implements transaction boundaries in .NET.

```
Imports PostSharp.Laos
Imports System.Transactions
 <Serializable>
Public NotInheritable Class TransactionScopeAttribute
    Inherits OnMethodBoundaryAspect
    Public Overrides Sub OnEntry(
         ByVal eventArgs As PostSharp.Laos.MethodExecutionEventArgs)
        eventArgs.State = New TransactionScope()
    End Sub
    Public Overrides Sub OnExit(
        ByVal eventArgs As PostSharp.Laos.MethodExecutionEventArgs)
        Dim transactionScope As TransactionScope = eventArgs.State
        If eventArgs.Exception Is Nothing Then
transactionScope.Complete()
        End If
       transactionScope.Dispose()
    End Sub
End Class
```

Transactional methods can be created using a new custom attribute as shown below.

```
<TransactionScope>
Sub Transfer(ByVal fromAccount As Account,
     ByVal toAccount As Account, ByVal amount As Decimal)
    fromAccount.Balance -= amount
    toAccount.Balance += amount
End Sub
```

It is required that custom attributes should be applied to each target explicitly but this is usually a challenge. This is because .NET languages do not offer the possibility to apply custom attributes to a set of code elements. A ‘multicast’ mechanism can be used to solve this problem. This is illustrated in the following code.

```
<assembly: TransactionScope(TargetTypes="MyNamespace.*")>
```

5. CROSS CUTTING CONCERN AND TANGLED CODE

Aspect Oriented programming is a technique used in programming to separate cross cut code across different modules in an application. Software applications are mostly developed to meet business concerns. For example, a customer sales management software can have the requirements such as add, update, delete customer information, track sales and customers, ability to generate and print reports and a facility to send email. We will now use this requirement to elaborate cross cutting concern and tangled code.

The above business requirements are illustrated in the class diagram below.

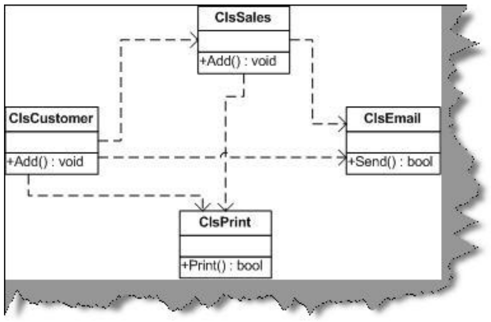

FIGURE 3: Business Concerns Class Diagram.

The diagram shows how the four concerns for application are met. According to object oriented programming (OOP), every object should only be concerned about its functionality. For example, "ClsSales" should only be concerned with maintaining sales information. From the above example, the core concerns are maintaining customer and sales records. The cross cut concerns are printing, sending email, logging and these spans across all the modules. This causes tangling of codes since there are several objects used across the modules. The codes are also called tangled in AOP methodology.

The cross cut code can be separated from the core modules by creating modules for cross cut and those for core functions separated. An AOP compiler can then generate a single executable even if the modules are separate. This process is called weaving and it is illustrated in the figure below.

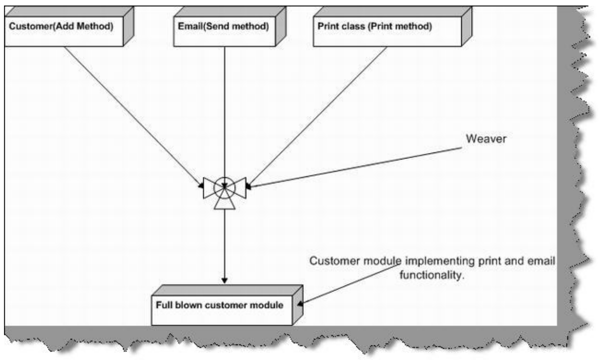
Customer(Add Method)
Email(Send method)
Print class (Print method)
Weaver
Customer module implementing print and email functionality.
Full blown customer module

FIGURE 4: Weaving Modules.

AOP compilers are helpful in addressing the cross cut challenges. Types of AOP compliers are compile time weaving, link time weaving and run time weaving.

6. CONCLUSIONS AND FUTURE WORK

There is no good method of encapsulating without violating the integrity of the code. Aspect-Oriented Programming provides a solution to this challenge and enables better isolation of responsibility, a more succinct code and encapsulation, all of which add to faster development times, increased comprehensibility and eased maintenance.

In this paper, it is pointed out, the pros and cons of the crosscutting concerns and the necessity of bringing the aspect oriented programming in to the limelight. This concept had a short term life in previous decade, but could not be extended, due to various reasons.

Though major software providers have chosen different approaches to achieve the above concept, this is the time to educate the IT world about efficiency of AOP using major .NET framework, and this paper does it to the best of my research.

As the support and implementation of AOP increases, the security concerns will also grow, which will increase the scale of fault tolerance. This will lead to further research to bring down the FT. Looking on the other side, there are few vendors who made attempts to integrate AOP with .NET framework and had also been successful to an extent. I believe this is the right time to make a smart move i.e. incorporate AOP in to corporate major programming concepts.

7. REFERENCES

[1] D. Box, “Essential COM”, 1998 Addison-Wesley, ISBN 0-201-63446-5

[2] K.Lieberherr, D. Orleans and J. Ovlinger. (2001). “Aspect-Oriented Programming with Adaptive Methods”, Communications of the ACM, Vol. 44, Issue 10.

[3] Groves, M. (2013). Aspect-Oriented Programming in .NET. Available: http://www.manning.com/groves/AOP.NETSampleCh01.pdf

[4] Clarke, S. & Jackson, A. (2004). SourceWeave.NET: Cross-Language Aspect-Oriented Programming. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.158.8736&rep=rep1&type=pdf

[5] G. Kiczaleset al. “Aspect Oriented Programming”, 1997. In proceedings of the European Conference on Object –Oriented Programming (ECOOP), Finland: Springer Verlag LNCS 1241.

[6] Kim, H. (2002). AspectC#: An AOSD Implementation for C#. Available: https://www.cs.tcd.ie/publications/tech-reports/reports.02/TCD-CS-2002-55.pdf

[7] Schult, W. & Polze, A. (2008). Design by Contract in .NET Using Aspect Oriented Programming. Available: http://www.tuplespaces.net/research/loom/Slides/DBC.pdf

[8] SUN Microsystems, “JavaBeans: The Only Component Architecture for Java Technology”, http://java.sun.com/products/javabeans/.

[9] Ferguson, D. (2004). Aspect. Net. Source Code… Available: http://www2.sys-con.com/itsg/virtualcd/dotnet/archives/0104/safonov/index.html

[10] Gnanasekaran, V. (2008). Rating of Open Source AOP Frameworks in .NET. Available: http://www.codeproject.com/Articles/28387/Rating-of-Open-Source-AOP-Frameworks-in-NET

[11] Miller, J. (2011). AOP with StructureMap Container. Available: http://weblogs.asp.net/thangchung/archive/2011/01/25/aop-with-structuremap-container.aspx

[12] Safonov, D. (2011). Aspect-oriented programming (AOP). Available: http://www.cs.helsinki.fi/en/event/58498

[13] Safonov, D. (2004). Aspect.NET: Concepts and Architecture. Available: http://www.aspectdotnet.org/articles/AspectDotNet2004_Article.pdf

[14] S. Hanenberg, R. Unland, “Concerning AOP and Inheritance”, Dept. of Mathematics and Computer Science University of Essen.

[15] Lee Breslau et al. (1999). Web caching and zipf-like distributions: Evidence and implications. In INFOCOM 1.

[16] Pei Cao and Sandy Irani.(1997). Cost-aware WWW proxy caching algorithms. In Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97), Monterey,CA.

[17] Constantinos A. Constantinides and Tzilla Elrad.(2000). On the requirements for concurrent soft-ware architectures to support advanced separation of concerns. The Workshop on AdvancedSeparation of Concerns in Object-Oriented Systems, OOPSLA.

[18] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson.(1999). Web prefetching between low-bandwidth clients and proxies: Potential and performance. In Measurement and Modeling of ComputerSystems.

[19] C. Fraleigh et al.(2001). Design and deployment of a passive monitoring infrastructure. Lecture Notes in Computer Science.

[20] Gustavo, A. & Grawehr, P. (2010). A Dynamic AOP-Engine for .NET. Available: ftp://ftp.inf.ethz.ch/doc/tech-reports/4xx/445.pdf

[21] Jangid, D. & Dave. R. (2012). Investigating the Web Application of AOP Using Aspect. Net Framework. Available: http://www.ijarcsse.com/docs/papers/8_August2012/Volume_2_issue_8/V2I800142.pdf

[22] Pérez, J. et.al (2010). Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. Available: http://www.sparxsystems.com/downloads/whitepapers/Aspect-Oriented_PRISMANET.pdf