Aspect-Oriented Programming in C#/.NET

Edward Garson, Dunstan Thomas Consulting
http://consulting.dthomas.co.uk

Abstract

This paper gives a broad overview of Aspect-Oriented Programming and its manifestation in the .NET platform.

Introduction

Aspect-Oriented Programming (AOP) will revolutionize how software is developed in the coming years. If you haven't heard about AOP or don't know what it is about, then prepare for inspiration: AOP is a simple and elegant construct with the capability of really changing the way you develop software. As a programming construct like inheritance or recursion, the language in which you develop applications must support it for you to be able to harness its capabilities. As you have probably gathered from the title of this article, C# supports AOP, as does Java. Few other languages have built-in support for AOP, so if you aren't coding in either of these languages then you might just find yourself wanting to learn one after you read this!

What is AOP?

AOP is a way of executing arbitrary code orthogonal to a module's primary purpose, with the intention of improving the encapsulation and reuse of the target module and the arbitrarily invoked code. AOP is best demonstrated by example, the classic one being event logging. The following example has been synthesized from Juval Lowy's article cited in the Bibliography below.

Suppose you have a class Foo and you wish to write to a log file each time a particular method is called for rudimentary performance statistics or auditing purposes. You might ordinarily write code such as the following to satisfy this requirement:

```
public class Foo {

  protected EventLog eventLog;

  public Foo() {
    // create an event log
    eventLog = new EventLog();

    // Name a Source
    eventLog.Source = "Foo Application";
  }

  public void bar() {
    eventLog.WriteEntry("Bar method begin");
    // do bar()
    eventLog.WriteEntry("Bar method end");
```

```
  }
}
```

What is wrong with this code? Historically, nothing really - but only because we're used to writing code like that. It is deemed acceptable to include EventLog code within our Foo class simply because prior to AOP there was no way to log events without explicitly calling event logging code from within the class itself (how else would this be accomplished?). With the advent of AOP, however, the above code would actually be construed as totally inappropriate, virtually prohibitive to include in a Foo class. This is because everybody knows what should go on in a Foo class: bar methods! Not logging! So, were we to accomplish the above using AOP, what you would see is the following:

```
[EventLoggingAttribute]
public class Foo : ContextBoundObject {

  public void bar() {
    // do bar()
  }
}
```

Well, how revealing: that nasty code tangential to the bar() method is relegated to another location - specifically, the *logging aspect*. An *aspect* is functionality that is factored out of a client module in an AOP-like manner, and is executed with no further knowledge on the client's part. In this example, the bar() method just does its job, ignorant of any other aspects. The benefit of this, apart from the obvious increase in maintainability, is the improved encapsulation and reuse of both module and aspect code due to the decoupling that is introduced.

Client-side Requirements to Implement Aspects

There are only a few requirements to satisfy an implementation of AOP in C#. The custom context attribute [EventLoggingAttribute] that adorns the Foo class above automatically enrolls it in the AOP scheme once the plumbing is in place. All calls to all instances of Foo will invoke Aspect code, which only concerns itself with event logging. The only other requirement to hook into AOP services from the client's perspective is that Foo derives from ContextBoundObject (or another class that does), which guarantees that each instance of Foo has a private Context object. From this point henceforth, the rest of the AOP mechanism is totally isolated from the client, which represents a very high level of decoupling. In short, client-side requirements are few and easy to satisfy.

Implementing the Aspect side is less straightforward. The details reveal interesting aspects of the .NET framework, but are essentially boilerplate.

A Brief Overview of Aspect-side Requirements

Interestingly, the .NET SDK declares the requisite classes to implement AOP as "not to be implemented in your applications" and are effectively undocumented. Research into this topic has proven otherwise, as may be discovered by further study of the technical citations in the Bibliography below. Herewith, a brief summary of the technical elements of an AOP implementation in .NET.

AOP is accomplished in .NET by having Aspect code insert itself and participate in the message-invocation mechanism that takes place between a client and an object. The private Context set up by the .NET runtime for an instance of a ContextBoundObject provides the means for externally defined Aspects to hook into the call chain, because the creation of the private Context forces the creation of .NET proxies. These proxies provide the means for Aspects to hook into the call-chain using message sinks. The reason that

ContextBoundObject is required is for clients and objects that are in the same AppDomain, that would otherwise have no proxies set up between them. Aspects are thus implemented as event sinks that get called on the message sink chain without any further participation or knowledge on the client's part. These event sinks are first-class citizens of the call chain and have full access to the call stack including method arguments and results and even exception objects that may have been created as a result of the method call. It is interesting to note that the .NET framework uses this call-interception mechanism in order to format messages on the call stack immediately prior to finally calling the method on the client object (the stack builder sink).

Implementing the Aspects themselves is accomplished using custom attributes, specifically ContextAttributes as defined by the `System.Runtime.Remoting.Contexts` namespace. ContextAttributes are married to the ContextBoundObject to which they are associated at runtime. ContextAttributes provide the means to install the message sinks and thus participate in the call-chain.

Custom attributes are essentially a way to dynamically extend the metadata for a given programming entity. Note that custom attributes apply not only to what one would normally think of as attributes as in the UML sense of the term, but apply equally to classes and assemblies as well. Clients are able to retrieve metadata adorned upon these entities via reflection. These decorators mark up programming entities in simple yet potentially very sophisticated ways. The architects of the .NET framework had every intention of surfacing this capability in order to support AOP and other interception-based services. This was good foresight on their part!

The Joy of Unmanaged Code

Visual C++ provides an even more sophisticated mechanism than custom attributes: that of *attributed programming*. Visual C++ is able to all that C# custom attributes do, but in addition enable customization of how the compiler generates code and metadata, and define attributes capable of inserting runtime code into applications. Ah... the joy and excitement of unmanaged code. It is there for those of you who need (or dare I say, want?) to delve into these intricacies. It is interesting to note that .NET itself is built upon a foundation of this very type of programming.

Other Manifestations of AOP

Examples of AOP-like constructs in other languages abound. One familiar to many readers will be Java Servlet filters as introduced in the Servlet 2.3 specification. Servlet filters enable the pre and post-processing execution of arbitrary code in a declarative fashion when requests are sent to a given servlet. Servlet filters may be chained together and may also abort processing along the way very much akin to message sink chains. Both of these are fundamentally an implementation of Chain of Responsibility[3]. Servlet filters, like their .NET AOP counterparts, provide the foundation for a powerful request-handling framework, and facilitate reuse of the logic segregated by each component along the request path. This method of servicing requests is almost identical to how the Common Language Runtime sets up message sinks. Coincidence?

Programmers Love Metadata

The fact that class member data (for one) may automatically participate in rich introspective capabilities is an extremely powerful paradigm. Previously, building such a framework required a lot of talent and forethought in order to provide an intuitive and performant object layer. I once had exposure to such a framework written for a very large provider of merchandise-management software. The framework had at its core a document paradigm whereby a large graph of objects was built as the result of various operations and built up as a Composite[3]. All the member data of the classes in the framework were objects

themselves, derived from a pure abstract class that defined two methods, getData() and setData(). Thus, altering the state of the classes was done purely by messages sending messages to these "class-member objects". The end result was that all messages to all classes could be hooked in an AOP-like manner. The framework used this to implement various services such as object-relational mapping, persistence and error logging. The framework worked and was really quite elegant, but required an extravagant effort to implement in the first place. It derived much of its sophistication through request interception, much like the described AOP implementation.

You may have been witness to similar efforts on other projects. Metadata and request interception are powerful programming paradigms that once exploited to great effect, programmers are loathe to live without. AOP-like constructs have existed in the hearts and minds of framework developers for some time, but building them without support from the platform is extremely difficult[5]. Having such a beast built into a language natively is very exciting and significantly expands opportunities for creativity.

AOP and the Modern Developer

AOP has the capability to elegantly untangle a number of aspects of software development. One interesting application of AOP is how it may be applied to the design of frameworks or base classes with services that generically return subclass client objects to a given state, such as undo/redo algorithms or Rollback() type calls. Traditionally, base-class methods would have to written that would save state and manage order of operations for the impelementation of these methods. Two problems arise from this, firstly that subclasses have to remember to call these service methods within the body of the methods that wish to leverage these services. The other problem is that it is harder to factor this behaviour out of the base classes for reuse in other applications because of class visibility semantics. AOP neatly solves both these problems.

Approaching Aspect-based Systems (Very Carefully!)

Refactoring or designing systems in terms of AOP is known as *aspectual decomposition*[1] and is the foundation of AOP analysis. Academic (read: fully implemented) Aspect-based systems strive to completely isolate all aspects of program code from one another. They have shown a great deal of promise as far as reducing complexity and increasing maintainability, but formal techniques to optimally implement these systems are in their infancy. Although these systems demonstrate many creative and interesting ways that AOP may be applied to development projects, developers would be well advised to resist the temptation to adopt fully Aspect-based architectures. AOP is very new and methodology has not followed pace with technology. No application of a new software technology may be applied without methodology and process to back it up that has been arrived at from real-world experience with it (witness first efforts at designing object-oriented systems!). The software industry is unique in this regard, in that software technology always outpaces its sensible application (remember the GOTO statement?). Those who do not study history are doomed to repeat it.

So given all these dire warnings, where does AOP fit into your development schedule? The answer is in very well defined ways. Discover the areas of your application domain that are obvious candidates for aspectual decomposition and judiciously apply the techniques described. Proceed with caution! Be safe and have fun.

Conclusion

The nature of method calls (or more formally, messages) in regular object-oriented programming emphasizes encapsulation and isolation of responsibility. This is a double-edged sword in that if there is a requirement to execute code orthogonal to the fundamental nature of a class, there is no good way to encapsulate this without violating the integrity of the code to be called. Aspect-Oriented Programming

provides a very elegant solution to this conundrum and enables better encapsulation, isolation of responsibility and more succinct code, all of which contribute to faster development times, eased maintenance and increased comprehensibility.

Bibliography

[1] - Kiczales et al, Aspect Oriented Programming, http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/for-web.pdf. The seminal paper on Aspect Oriented Programming as presented to the European Conference on Object-Oriented Programming 1997 by its inventor and champion Gregor Kiczales and colleagues. Fairly heavy going.

[2] - Although this interview with visionary software development pundit Grady Booch has little to do with AOP, he does cite it as being of significant importance to the future of software development (if you don't believe me!):
http://msdn.microsoft.com/msdnmag/issues/02/02/TalkingTo/

[3] Gang of Four - Design Patterns - Elements of Reusable Object-Oriented Software.

[4] - Lowy, Juval - Decouple Components by Injecting Custom Services into Your Object's Interception Chain, http://msdn.microsoft.com/msdnmag/issues/03/03/ContextsinNET/ - Juval's article provides an excellent overview of the mechanism underlying call interception in .NET and provides a broad example of AOP using the classic Logging service.

[5] - Brown, Keith - Building a Lightweight COM Interception Framework, Part 1: The Universal Delegator, http://www.microsoft.com/msj/0199/intercept/intercept.aspx