Using AOP with PostSharp in C#

Aspect Oriented Programming, or AOP, is a method of development complementary to Object Oriented Programming. Like OOP, AOP stresses the idea of code reuse to reduce clutter and confusion. However, AOP is different in the sense that it encourages the reuse implementations to be built in a manner that allows the functionality to be inserted anywhere into existing code.

In .NET, the most elegant manner of allowing code to be added to existing code is through the use of Attributes. PostSharp, a leading AOP tool for .NET, provides a developer the ability of encapsulating repetitive code into custom Attribute classes. In PostSharp lingo, these classes are typically referred to as various types of Aspect classes.

The best way to demonstrate the effectiveness of a particular technology is to use it or to observe its usage in the wild. AOP is best understood when seen in context – improving and clarifying application code. For that purpose, this article will break an existing codebase apart. Upon reconstruction the code will be improved through the creation of Aspects whenever code exists that could be reused in other areas. Not just typical reuse in the manner of componentized objects, but in the way those objects will be applied to their target.

The database in which the data will be stored is relatively simple. A user can have multiple email addresses, and there will be a unique constraint placed on the email address value for the email table. Some architectural expectations are also in place; the code will need to provide detailed logging output at debug time and each method will need to be timed and the timing output to the log as well, so that bottlenecks in performance can be observed and fixed.

The architectural requirements introduce the probability that utilitarian code – logging, timing, transaction management, threading checks – will be intermixed with actual application code. This sample application actually has more technical and architectural requirements than it does business requirements. With that in mind, the probability for copy-and-paste coding techniques is quite high. Consider the basic architectural requirement of timing; each method needs to be timed. This means that every method will have code in it that creates, starts, and stops a Stopwatch instance, and reads the elapsed time so that it can be written to the debug window.

Every application requires a little logging to understand performance and behavior. This methodology – requiring the wrapping of code in a clock and logging the results of performance each and every time – dictates that extremely repetitive code be written. This repetitive code also has little to do with the intent of the entire code base, so why not isolate it in an independent module that can be applied when it is needed – that's the provision AOP allows.

Logging, the other basic architectural requirement, introduces a single method to streamline logging, but doesn't really make it not only reusable, but automatic. As it stands now, explicit calls must be made to the method, rather than execution of the method just happen automatically. So it provides a decent example of reuse, but is still tied to the code using it because that code must make calls to it.

Opportunities for refactoring aren't limited to simple topics such as logging and timing. Exception management, external thread GUI update headache resolution, and transaction management are other areas. For each of these, some level of complexity exists, but some level of repetitiveness exists, too. Transaction usage code needs to be weaved in any time multiple tables will be updated in any database, especially when constraints exist on the data as they do in the case of this example application, which intentionally prohibits email address re-use.

The first choice for most AOP discussions is to use it to perform the basic logging requirements of an application. This quintessential example does a few very simple pieces of work:

- The Aspect stops the execution of a method
- It runs its own functionality.
- The method executes (or errors) and completes.
- Once that happens the Aspect class runs even more code.

This code eventually whittles all of the logging code out of the application and into the Aspects which are linked to the methods with Attributes as we'll see later. This keeps the logging separate from the code application business logic.

How PostSharp actually works seems like a huge mystery but really isn't all that complex from a conceptual level. At compilation time, PostSharp changes the way things work. Don't panic, but yes, that's basically it. PostSharp weaves in the code from Aspects created where they need to be applied. A lot of confusion exists around the topic of weaving, so this section will attempt to clarify things somewhat by showing the variation in the compiled MSIL code produced both before and after PostSharp Aspects are applied.

PostSharp is available via an easy-to-install NuGet package. You can view more information regarding the use of this package on the SharpCrafters web site. The NuGet command line executable can install the PostSharp package prior to compilation in a build server situation, and all of the options PostSharp offers are available at build time.

The installation of PostSharp (or the NuGet reference/inclusion) will add a PostSharp.targets step to the MSBuild code that compiles any project referencing PostSharp. Since PostSharp does the majority of its work as a post-compilation step, the build process – your compilation from within the Visual Studio IDE, usually – will be automatically modified by this addition.

PostSharp is one of the more popular means of implementing AOP within the .NET Framework. When applied appropriately, PostSharp Aspects can reduce code clutter and help to maintain architectural standards and practices without overcomplicating the code with intermingled responsibilities.

Costiug Mihai, group 922
2.06.2014