Dynamic objects in the DLR provide a foundation that has many applications. In previous chapters, we’ve seen some practical examples that leverage this foundation and do interesting things that are either awkward or impossible in static languages. In this chapter, we are going to travel further down the path and see the fantastic application of dynamic objects in aspect-oriented programming (AOP). AOP is a programming paradigm that is very good at solving the problem of cross-cutting concerns. Common cross-cutting concerns in a software system are issues like transaction management, security, auditing, performance monitoring, logging and tracing, and so on. By virtue of addressing the problem of cross-cutting concerns in an elegant manner, AOP provides tremendous value in the design and architecture of software systems. I’ll begin with an introduction of the basic AOP concepts accompanied by some simple examples, then show you how to implement an AOP framework based on dynamic objects. By the end of the chapter, you will have an AOP framework that (a) works across both static and dynamic objects and (b) is integrated with the widely adopted Spring.NET’s AOP framework.

Aspect-Oriented Programming

Let’s go over the important concepts of AOP now to set the stage for the rest of the chapter. After reading this section, you will know what AOP is and the problem it solves. You will also learn the meaning of terms such as pointcut, join point, and advice. If you are already familiar with these topics, you can skip this section and jump ahead.

Cross-Cutting Concerns

AOP solves the issue of cross-cutting concerns very well. This is best illustrated with an example. Listing 7-1 shows some code that logs one message at the beginning and one at the end of the Age property’s get method. The real business logic of the property’s get method is represented by the code comment //some business logic here. The two lines that write to the console belong to the logging concern. The code by itself might not seem to be a problem, but imagine how the code would look if we were to do this kind of logging for all property access in 20 other classes. You would quickly notice that the same code that writes messages to the console output is duplicated and scattered all over the place. That naïve approach violates the DRY (Don’t Repeat Yourself) principle and creates the problem of code scattering. Furthermore, the code in listing 7-1 also illustrates the problem of code tangling because the logging code and the real business logic are enmeshed.

The code scattering and tangling are the kinds of problems AOP solves. Logging is just an example of a cross-cutting concern we commonly encounter in a software system, and it’s the easiest and simplest to demonstrate. From this simple example, you can extrapolate and see that if this were an example of
transaction management, the two lines of logging code would be replaced by a line of code that starts a transaction and another line of code that commits or rolls back the transaction.

Listing 7-1. Logging Messages Before and After a Property Access

```csharp
public class Employee : IEmployee
{
    private int age;
    private String name;

    public int Age
    {
        get
        {
            Console.WriteLine("Employee Age getter is called.");
            //some business logic here.
            Console.WriteLine("End of Employee Age getter.");
            return age;
        }

        set { age = value; }
    }

    public String Name
    {
        get
        {
            Console.WriteLine("Employee Name getter is called.");
            return name;
        }

        set { name = value; }
    }
}

public interface IEmployee
{
    int Age { get; set; }
    String Name { get; set; }
}
```

Advice, Join Points, and Pointcuts

In AOP terms, the two lines of logging code we saw in the Age property’s get method should be modularized into something called advice. Advice is the action you’d like to take to address a cross-cutting concern. The first logging statement is at the beginning of the property getter. The second logging statement is at the end of the property getter before the employee’s age is returned. The beginning and the end of the property getter in this case are called join points. Join points are the places in code where advice can be applied. A collection of join points is called a pointcut. When you