In this chapter I’ll cover a significant, though often underused, feature that’s offered by the .NET Framework classes—the ability to generate either source code or IL code dynamically. In other words, you can have your code actually write or manipulate code instead of data. Normally, you’d conceptually imagine that the process of producing a software application involves you writing the source code, compiling it, and shipping it—and that’s it. The code and resources your organization wrote constitutes the totality of the shipped product. With dynamic code generation, however, your shipped code can itself actually generate new code to perform additional tasks—this can be useful for performance reasons, among other factors. Alternatively, your code may modify the code in other assemblies (which may be done, for example, to insert calls to create debugging or profiling information). And obviously, if your product is a developer tool that’s intended to assist developers in writing code, then it may be called on to generate some source code itself.

This chapter will cover the following:

- **Applications of dynamic code generation**: I’ll review the main reasons why you may find it useful to implement dynamic code generation.

- **Architecture**: I’ll cover the design of the code generation classes and in particular the different philosophies behind the System.Reflection.Emit classes (which generate straight assemblies containing IL code) and the System.CodeDom classes (which generate source code or assemblies).

- **Examples**: The bulk of the chapter is devoted to a couple of examples that illustrate how to use the dynamic code generation classes. For this part of the chapter, I treat the Reflection.Emit and the CodeDom classes separately.

Dynamic code generation isn’t something that has any substantial intrinsic support in the CLR—it’s a feature that’s supported almost entirely by the associated .NET class libraries supplied by Microsoft. Hence, this chapter focuses almost exclusively on the use of the relevant classes. Note, however, that I won’t make any attempt to be comprehensive (for example, to give lists of all the methods implemented by particular classes). You can find out that stuff easily enough in the MSDN documentation. Rather, my aim is to give you a feel for how the classes are used and how they’ve been designed.
Reasons to Use Dynamic Code Generation

Traditionally, code generation has been associated with compilers. However, the .NET Framework libraries make code generation sufficiently easy that it becomes feasible and potentially useful in a number of different scenarios, which I'll quickly review in the following sections.

Developer Tools

You’re probably used to developer tools that can autogenerate code for you; the most obvious examples are Visual Studio .NET’s Design View and its Properties window. Other examples from Microsoft include the xsd.exe tool, which can generate a source code file from an XML schema, and the wsdl.exe tool, which can generate client source code for XML services. And of course as I write this chapter Windows Longhorn is on the distant horizon with its Extensible Application Markup Language (XAML) files used to define client user interfaces. I don’t personally know how the underlying technology is going to work, but it’s a reasonable guess that dynamic code generation may be lurking around there somewhere.

Other situations in which dynamic code generation is important include the following:

**Templates:** Version 1.x of the .NET Framework has often been criticized for not including much support for generics, which offer similar, though more restrictive, features to unmanaged C++ template classes. (This will of course be fixed in .NET 2.x.) Since in practice a template isn’t really more than a definition that allows the compiler to generate and compile multiple classes (or methods) from the same definition, it should be obvious that a developer tool could use dynamic code generation to implement the same kind of feature.

**UML-based coding:** Dynamic code generation can be used to implement tools in which developers use some kind of diagram to indicate the code they want to write—and the tool generates the code for them. An obvious example of this is generation of code from Unified Modeling Language (UML) diagrams.

**Language conversion:** The multilanguage support in the .NET Framework should theoretically reduce the need for source code to be converted between languages, because the source language that an assembly was originally compiled from is to a large extent irrelevant to clients of that assembly. Nevertheless, developers or organizations have an occasional need for applications that can convert source code between languages. Dynamic code generation can assist in the implementation of this kind of application.

**Assembly modification:** Sometimes you may need to take the instruction stream in an assembly and modify it prior to executing it—for example, to provide notifications of when certain IL instructions are executed. Obfuscators also need to permanently modify the contents of assemblies, and in a similar vein it’s conceivable that you may want to write some software that optimizes the IL code in other assemblies (since compilers such as the C# and VB .NET ones perform little optimization of the emitted IL code). Although given how efficient the JIT compiler is at optimizing, I have some doubts whether an application such as this is particularly useful.