CHAPTER 19

Packaging, Debugging and Testing F# Code

Successful programming must involve a healthy marriage of good code with good software engineering techniques and practice. Sometimes these overlap: functional programming is a good software-engineering technique: among other benefits, anecdotal evidence indicates that functional programming frequently leads to a substantially reduced bug rate for good programmers. This is primarily because programs built using functional techniques tend to be highly compositional, building correct programs out of correct building blocks. The functional-programming style avoids or substantially reduces the use of side effects in the program, one property that makes programs more compositional. Debugging and testing are still essential activities to ensure that a program is as close as possible to its specifications, however. Bugs and misbehaviors are facts of life, and F# programmers must learn techniques to find and remove them. Often, these techniques are not inherently “functional” or even particularly “code” related, but they are still critical to the process of writing robust, maintainable, and successful software components.

You also need to learn many pragmatics of building and packaging F# code. As a result, this chapter turns to the pragmatics of packaging, debugging, and testing F# code.

Packaging Your Code

To begin your exploration of ways to package F# code, let’s first talk about the sorts of things you may be building with F#.

Mixing Scripting and Compiled Code

Small programs are often used both as interactive scripts and as small compiled applications. Here are some useful facts to know about scripting with F# and F# Interactive:

- F# scripts use the extension .fsx.
- A script file can contain #r directives. These reference a library or a type provider.
- A script file can contain #load directives. This is as if the files had been compiled using the command-line compiler and included in the same assembly as the referencing script.
A script that is referenced via a #load can itself contain further #load and #r references. This means that a script can act like a “little library.” If the same root file transitively references the same script more than once via #load, the file acts as if it is logically only referenced once.

You can access command-line arguments from within scripts by using the expression fsi.CommandLineArgs. Within compiled code, use System.Environment.GetCommandLineArgs. Within code used in both modes, use conditional compilation to switch between these, as shown in the next coding example.

You can run a script on startup by using the --exec command-line option for fsi.exe or by giving a single file name on the command line. You can find other command-line options by using fsi.exe --help.

Conditional compilation is a particularly useful feature for scripts—especially the predefined conditional compilation symbols COMPILED and INTERACTIVE. The former is set whenever you compile code using the command-line compiler, fsc.exe, and the latter is set whenever you load code interactively using F# Interactive. A common use for these flags is to start the GUI event loop for a Windows Forms or other graphical application, such as using System.Windows.Forms.Application.Run. F# Interactive starts an event loop automatically, so you require a call to this function in the compiled code only:

```fsharp
open System.Windows.Forms

let form = new Form(Width = 400, Height = 300,
                      Visible = true, Text = "F# Forms Sample")

#if COMPILED
    // Run the main code
#elseendif
```

Note: You can specify additional conditional compilation directives by using the --define command-line compiler option.

Choosing Optimization Settings

The F# compiler comes with a simple choice of optimization levels. You nearly always want to compile your final code using --optimize, which applies maximum optimizations to your code. This is also the default optimization setting when using fsc.exe or fsi.exe directly, but it is not the default for compiled code using Visual Studio’s “Debug” mode.

The F# compiler is a cross-module, cross-assembly optimizing compiler, and it attaches optimization information to each assembly you create when using optimization. This information may contain some code fragments of your assembly, which may be inlined into later assemblies by the optimizing compiler. In some situations, you may not want this information included in your assembly. For example, you may expect to independently version assemblies, and in this case, you may want to ensure that code is never duplicated from one assembly to another during compilation. In this case, you can use the --nooptimizationdata switch to prevent optimization data being recorded with the assemblies that you create.