Add-ins (also known as plug-ins) are separately compiled components that your application can find, load, and use dynamically. Often an application is designed to use add-ins so that it can be enhanced in the future without needing to be modified, recompiled, and retested. Add-ins also give you the flexibility to customize separate instances of an application for a particular market or client. But the most common reason to use the add-in model is to allow third-party developers to extend the functionality of your application. For example, add-ins in Adobe Photoshop provide a wide range of picture-processing effects. Add-ins in Firefox provide enhanced web-surfing features and entirely new functionality. In both cases, the add-ins are created by third-party developers.

Since .NET 1.0, developers have had all the technology they need to create their own add-in system. The two basic ingredients are interfaces (which allow you to define the contracts through which the application interacts with the add-in and the add-in interacts with the application) and reflection (which allows your application to dynamically discover and load add-in types from a separate assembly). However, building an add-in system from scratch requires a fair bit of work. You need to devise a way to locate add-ins, and you need to ensure that they’re managed correctly (in other words, that they execute in a restricted security context and can be unloaded when necessary).

Fortunately, .NET has a prebuilt add-in model that saves you the trouble. It uses interfaces and reflection, like the add-in model that you’d probably write yourself. However, it handles the low-level plumbing for tedious tasks such as discovery and hosting. In this chapter, you’ll learn how to use the add-in model in a WPF application.

Choosing Between MAF and MEF

Before you can get started building an extensible application with add-ins, you need to deal with an unexpected headache. Namely, .NET doesn’t have just one add-in framework; it has two.

.NET 3.5 introduced an add-in model called the Managed Add-in Framework (MAF). But to make matters even more interesting (and a whole lot more confusing), .NET 4 added a new model called the Managed Extensibility Framework (MEF). Developers, who had to create their own add-in system not long ago, suddenly have two completely separate technologies that share the same ground. So, what’s the difference?

MAF is the more elaborate framework of the two. It allows you to decouple your add-ins from your application so they depend on nothing more than the interface you define. This gives you welcome flexibility if you want to handle versioning scenarios—for example, if you need to change the interface but continue to support old add-ins for backward compatibility. MAF also allows your application to load add-ins into a separate application domain so that they can crash harmlessly, without affecting the main application. All of these features mean that MAF works well if you have one development team working on an application and another one (or several) working on its add-ins. MAF is also particularly well suited for supporting third-party add-ins.

But MAF’s features come at a cost. MAF is a complex framework, and setting up the add-in pipeline is tedious, even for a simple application. This is where MEF comes in. It’s a lighter-weight option that aims to make extensibility as easy as copying related assemblies into the same folder. But MEF also has a different underlying philosophy than MAF. Whereas MAF is a strict, interface-driven add-in model, MEF is a free-wheelin’ system that allows an application to be built out of a collection of parts. Each part can export functionality, and any part can import the functionality
of any other part. This system gives developers far more flexibility, and it works particularly well for designing *composable applications* (modular programs that are developed by a single development team but need to be assembled in different ways, with differently implemented features, for separate releases). The obvious danger is that MEF is too loose, and a poorly designed application can quickly become a tangle of interrelated parts.

If you think MAF is the add-in system for you, keep reading—it’s the technology that’s discussed in this chapter. If you want to check out MEF, you can learn more at Microsoft’s MEF community site at [http://tinyurl.com/37s2jdx](http://tinyurl.com/37s2jdx). And if your real interest is not in add-ins but in composable applications, you’ll want to check out Microsoft’s Composite Application Library (CAL), which is also known by its old code name, Prism. Although MEF is a general-purpose solution for building any sort of modular .NET application, CAL is tailored for WPF. It includes UI-oriented features, such as the ability to let different modules communicate with events and show content in separate display regions. CAL also has support for creating “hybrid” applications that can be compiled for the WPF platform or the browser-based Silverlight platform. You can find the documentations and downloads for CAL at [http://tinyurl.com/5ljve8](http://tinyurl.com/5ljve8).

---

**Note** From this point on, when the text refers to *the add-in model*, it means the MAF add-in model.

---

**Understanding the Add-in Pipeline**

The key advantage of the add-in model is that you don’t need to write the underlying plumbing for tasks such as discovery. The key disadvantage is the add-in model’s sheer complexity. The designers of .NET have taken great care to make the add-in model flexible enough to handle a wide range of versioning and hosting scenarios. The end result is that you must create at least seven (!) separate components to implement the add-in model in an application, even if you don’t need to use its most sophisticated features.

The heart of the add-in model is the add-in *pipeline*, which is a chain of components that allow the hosting application to interact with an add-in (see Figure 32-1). At one end of the pipeline is the hosting application. At the other end is the add-in. In between are the five components that govern the interaction.

![Figure 32-1. Communicating through the add-in pipeline](image)

At first glance, this model seems a bit excessive. A simpler scenario would put a single layer (the contract) between the application and the add-in. However, the additional layers (the views and adapters) allow the add-in model to be much more flexible in certain situations (as described in the sidebar “More Advanced Adapters”).