A Framework for Building Smart Clients

Microsoft® Composite Application Block (CAB)

Author:
Sal Razzaq
razzaq.org

Version 0.92
March 2007

Contents

Preface

This book describes the Microsoft Composite Application Block (CAB). The Microsoft Composite Application Block is an application framework for building Smart Clients using Microsoft .NET Common Language Runtime. Smart Clients generally refer to applications that provide a rich user experience through a sophisticated graphical user interface.

The intended audience of this book are software architects, designers, and programmers engaged in developing Smart Client Solutions using Microsoft .NET and C#.

About the author Sal Razzaq is a Software Architect at Kofax. Sal is involved in the design and development of solutions that transform massive amounts of data buried in various paper, electronic and image formats into mission-critical information. Sal is an alumnus of the University of Illinois, Urbana-Champaign.

Carlsbad, California, USA

Chapter 1

What is CAB?

CAB stands for Composite Application Block. It is called *Composite* because CAB facilitates composition of a given set of components in multiple permutations to fulfill different requirements of an application. It is called an *Application Block* because it is provided as a block of source code that can be compiled and incorporated directly into your application. Availability of source code for CAB provides an opportunity to learn the innards of CAB as well as assurance that developers can modify the code or fix bugs, if necessary.

CAB is only available for the Microsoft .NET platform. There are two other technologies that are closely related to CAB: Microsoft Enterprise Library and Smart Client Software Factory (SCSF).

1.1 Microsoft Enterprise Library

Microsoft Enterprise Library is a library of application blocks helpful in tackling common challenges encountered in developing an Enterprise Application. The Enterprise Library consists of several application blocks:

Caching Application Block Provides a flexible and extensible caching mechanism for use in client and server-side .NET development projects.

Configuration Application Block Makes it easier to build applications that read and write configuration information from a variety of data sources.

Cryptography Application Block Makes it easier to include cryptographic functionality in .NET applications. The block provides a simple interface to Data Protection API (DPAPI), symmetric encryption and hashing, and uses the Enterprise Library configuration tool to simplify key management.

Data Access Application Block Reduces the amount of custom code that you need to create, test, and maintain when building data access layers in .NET applications.

Exception Handling Application Block Makes it easier to implement consistent exception handling policies at logical tiers in an application. Exception policies can be configured to perform tasks such as logging exceptions, wrapping or replacing exception types.

Logging & Instrumentation Application Block Allows developers to instrument their applications with logging and tracing calls. Log and trace messages can be routed to a choice of data sinks, including the Event Log, text files or Windows Management Instrumentation (WMI).

Security Application Block Builds on the capabilities of the Microsoft .NET Framework to help you perform authentication, authorization, check role membership and access profile information.

The Enterprise Library is very useful in its own right with or without the employment of CAB. However, based on the needs of your application you can easily incorporate one or more of these pre-fabricated Enterprise Application Blocks into your CAB solution and cut down your development time.

1.2 Smart Client Software Factory (SCSF)

The Smart Client Software Factory (SCSF) provides further guidance and tools to utilize CAB. At one level SCSF provides an additional layer of abstraction by providing wrapper classes for commonly used CAB functionality. On another level SCSF provides extensions to Visual Studio 2005 to streamline creation of various components in a CAB-based solution.

It is quite likely that you will find one or more of the Enterprise Library Application Blocks very useful in your CAB-based solution. You may also appreciate the extra layer of abstraction and guidance automation tools (GAT) provided by the Smart Client Software Factory (SCSF).

However, in this book we deal only with the CAB API.

1.3 Composite Application Block (CAB)

Before we dive into the details of the Composite Application Block, we pose two questions:

1. Why use a framework?
2. Why not create our own framework instead of learning the CAB Framework?

A framework provides a level of consistency in the design and implementation of an application. A good framework provides common facilities and services likely to be utilized by various parts of an application. Furthermore, a good framework is readily extensible allowing addition of facilities and services that cater to the particular needs of *your* application. Utilization of common infrastructure services and facilities encourages code reuse and curbs code duplication. A good framework also fosters a modular design that makes application maintenance and enhancements amenable. Instead of focusing efforts on building the application infrastructure, developers can spend bulk of their time implementing domain logic and organizing that logic into sensible modules.

The second question is a more difficult question to answer. Developing a custom application framework generally pays off if such a framework will be utilized for several (generally believed to be five or more) applications. Another consideration in pursuing the development of a custom application framework is not knowing exactly at the outset what services and facilities your target applications will eventually require. A few bad decisions in the framework design in early stages can have very crippling, long-term consequences. Last but not least is the cost, effort and logistics of developing a custom application framework. In practice, companies tend to isolate their framework development team from the application development team. Unfortunately, such a split causes the framework development team to be too far removed from the real challenges of developing an application in a particular domain.

Based on considerations mentioned above, why not use an existing application framework that meets your application development needs? For many Windows developers, this conjures up images of Microsoft Foundation Classes (MFC), Active Template Library (ATL) and the Windows Template Library (WTL). MFC tried to fit every application in the mold of a Document-View paradigm. That paradigm worked very well for a document-centric application but left others trying to fit a round peg into a square hole. ATL/WTL works well for the middle-tier applets, device drivers and applications that have a relatively modest user interface. However, ATL/WTL requires an excruciating amount code to accomplish a sophisticated user interface. To top it all, both MFC and ATL/WTL require programming in native C++ code providing ample of opportunities to introduce memory management related bugs.

.NET provides a very capable class library. Further, .NET does not impose any sort of application paradigm like MFC. However, in order to build a Smart Client Solution, the developers still need to build the necessary infrastructure of an application from scratch. Examples of infrastructure facilities needed to build a Smart Client include an event mechanism for inter-module communication and a scheme for creation, scoping and sharing of application data structures across modules, all without hard dependencies. In short, there is a gap between all the wonderful functionality .NET class library provides and what is needed to build a Smart Client. *CAB fills this gap.* In the next chapter, we will go over the design philosophies behind the CAB Framework. Understanding these design philosophies will help us utilize CAB Framework to its full potential.

Chapter 2

CAB Design Philosophies

CAB uses several novel ideas and concepts in its design. It is worthwhile to understand these ideas and concepts before examining the lower level details of CAB. This will allow us to effectively utilize all that CAB has to offer. A firm understanding of CAB design concepts will also help us decide what functionality of CAB is the most appropriate to use in a given situation.

2.1 Shell vs. Module

CAB separates application functionality into an Application Shell and one or more Modules. The Application Shell is simply a container that *hosts* user facing functionality provided by one or more Module(s). Generally, the Application Shell is very lightweight, providing just minimal facilities such as screen areas called *Workspaces* where Modules can display their visual parts. The Shell also exposes common user interface elements such as menu bars, toolbars and status bars that may be extended by the Modules.

Separating functionality into Modules is not a new idea [PARNAS]. What is special about CAB is that it also facilitates loose coupling between the Application Shell and the Module(s). In other words, there is no *compile time* dependency between the Shell and the Modules(s). The Shell simply loads *Modules* listed in a catalog at *runtime*. The late binding of the Modules with the Shell is facilitated by the *Reflection* capabilities of the .NET environment. Reflection allows a .NET program to have knowledge about its own structure at runtime. CAB exploits this capability to discover and manipulate CAB Module code at runtime.

The Shell lays out the general user interface paradigm for the application (for example, Multiple Document Interface) and makes appropriate areas of the application frame available to the Modules in the form of Workspaces and extensible UI elements. The loaded Modules may optionally make themselves visible in one or more Workspaces of the Shell and wait for user interaction. The Modules may also optionally extend Shell-exposed

UI elements such as menubars, toolbars and status bars. For example, a Module may add a new menu item and associate a command handler with the menu item that invokes some piece of code in the Module.

This dichotomy of the Shell and the Module(s) provides an opportunity to create variants of an application by simply packaging Modules in different combinations. There are also opportunities for late and piecemeal deployment of Modules.

2.2 Use Case Driven Development

The development model of CAB is centered around use cases that need to be fulfilled by an application. This paradigm lends itself well to User Centered Design (UCD) philosophy. The CAB artifact that embodies a use case is called a *WorkItem*. WorkItems provide means to aggregate and scope collaborating components that participate in a Use Case.

Generally, you will find one to one correspondence between the primary use cases of your application and the WorkItems in your CAB application. A WorkItem can be thought of a container that binds all objects that are needed to implement a given use case. WorkItem manages the lifetime and scope of all collaborating objects it contains. All objects in a WorkItem have access to each other. Objects contained in a WorkItem have at most the lifetime of the WorkItem.

When a CAB application starts, a *Root* WorkItem is created by the Shell. All WorkItems created by Modules are added as children of the *Root* WorkItem. As a further means of use case break-down, Sub-WorkItems may be added to WorkItems created by Modules forming a hierarchy. In line with the container model of a WorkItem, the scope of the Sub-WorkItems is limited to the scope of the containing (parent) WorkItem.

The code that implements the visual and functional parts of a Module lives as objects within a WorkItem. There are trigger mechanisms in CAB to invoke a WorkItem such when a user makes a menu choice or when an event is fired by a WorkItem. WorkItems can be arbitrarily organized into Modules per your application needs.

2.3 Client-Side Services

CAB allows factoring out of common functionality used by multiple parts of an application into a *Service*. CAB provides some basic services such as loading of Modules at runtime. An application can roll out its own custom Service that can be consumed by interested WorkItems. A CAB application may also opt to override a CAB-provided service. For example, a CAB application may specify where to get the list of Modules to load by re-implementing and registering the `IModuleEnumerator` Service.

A Service is contained within a WorkItem and therefore, its scope is determined by the scope of the WorkItem it is contained in. For example, a Service added to the

Root WorkItem will be visible to all Modules. To contrast a Service with a WorkItem, a Service encapsulates useful functionality that may cut across several use cases whereas a WorkItem is centered around fulfilling a single use case. Services primarily exist to facilitate code reuse or to encapsulate some external service such as a Web Service.

2.4 Event Broker

Since CAB Modules are loosely coupled by design, Modules should *not* make direct calls to code in other Modules. Doing so will result in compile time dependencies between the Modules which is the antithesis of what CAB strives for. However, it is conceivable that a Module may wish to react to changes in other Modules. For example, say a WorkItem in MODULE A deletes a record. Other WorkItems in other Modules unknown to the WorkItem in MODULE A that deleted the record may be interested in knowing about this event (for example, to update their views). What is needed here is a mechanism to announce a change and have the interested WorkItems receive and act upon the change notification without mutual, *a priori* knowledge of existence of parties involved. CAB employs the *Observer* [GOF] pattern to facilitate this communication. A WorkItem may *publish* an event and an interested WorkItem may *subscribe* to an event and act as desired. CAB provides a robust *Event Broker* component to propagate events to all subscribers in a CAB application. Furthermore, the event and the event arguments can be specifically tailored by the application. Events are named using URI's (Universal Resource Identifier) to avoid event name collisions. The event data is a class object of type `EventArgs` or a class derived from `EventArgs` type. Finally, *many* WorkItems can publish a given event and *many* WorkItems can subscribe to a given event. This kind of many-to-many event mechanism is referred to as a *multicast* event scheme.

2.5 Automatic Object Creation

CAB has a robust scheme in place to create and instantiate objects and object dependencies on the fly, as needed or as specified declaratively using .NET attributes. CAB uses *Builder* Design Pattern [GOF] that separates the construction of a complex object from its representation so that the same construction process can create different representations. CAB accomplishes this separation by requiring that concrete classes be based on well-defined interfaces. *Factory Method* Design Pattern [GOF] is utilized to define interfaces for creating objects. The implementations of these interfaces do the actual instantiation of concrete objects. The component of CAB responsible for this functionality is called the *ObjectBuilder*.

Another novel idea CAB uses is *Dependency Injection* which is an application of *Inversion of Control* [IOC] Design Pattern. .NET allows marking of classes, class constructors, methods and properties using attributes. The attributes allow detection and selection of the marked classes, class constructors, methods and properties using *reflec-*

tion capabilities of .NET at runtime. CAB uses its ability to construct complex objects in conjunction with .NET's ability to detect specially marked classes, class constructors, methods and properties to automatically construct objects and inject them into executing code. This capability of CAB is referred to as *Dependency Injection*. For example, a property of a CAB-managed class can be marked as a SERVICE DEPENDENCY. At runtime, CAB will lookup the Service specified by the type of the property and set the property to the running instance of the Service.

2.6 Runtime Configuration and Extension of Shell UI

When a CAB application starts and before it loads any Modules, the CAB application has no idea of what functionality (views, menubar and toolbar items, WorkItems, Services, etc.) the Modules will be adding to the application. What the application Shell *is* privy of are Workspaces (areas of the Main Window where Modules can display their user interface widgets) and shared UI elements such menubar, toolbar and status bar. The Shell exposes its Workspaces and shared UI elements by their *name*. A Module can obtain a reference to a Workspace or a common UI element using its *name* at runtime. A Module may display its user interface widgets or views in the Shell Workspace(s). A Module may add items to menubar and toolbar and associate command handlers with the items added. A Module may create and register Services that can be consumed internally by the Module or shared with other Modules. All of this happens at *runtime* as opposed to *compile* time.

2.7 Modular Design

All of the above mentioned design philosophies contribute to the modular design of a CAB-based application. Specifically:

1. CAB forces organization of functionality into WorkItems corresponding to application use cases
2. CAB allows organization of WorkItems into stand-alone, yet collaborating Modules
3. CAB provides an Event Broker that allows loose coupling between Modules
4. CAB allows Modules to hookup Commands at runtime
5. CAB enables code reuse through client-side Services
6. CAB encourages separation of the Presentation Layer from the Domain Model

2.8 Distributed Team Development

The CAB framework lends itself well for development by a distributed team. Once the Application Shell and common Services are in place, small teams can work fairly independently on different Modules. However, careful design and architecture is still needed to manage runtime dependencies between Modules and to spot and factor out common code as Services, shared libraries or perhaps as new WorkItems. CAB is not a substitute for careful design and architecture but CAB does provide a nice framework to facilitate such efforts.

Chapter 3

Quick Start

3.1 Tools and Libraries

You will need to obtain and install the following tools and libraries to develop CAB-based solutions.

1. Visual Studio 2005 Professional Edition or better with .NET Framework 2.0 or better
2. Composite UI Application Block in C#
   http://www.codeplex.com/smartclient/Wiki/View.aspx?title=Composite%20UI%20Application%20Block

This chapter will get you jump started in creating a CAB-based solution. Before we begin, download and install Composite UI Application Block in C#. In this chapter, we will be creating two separate Visual Studio Projects in two different Visual Studio Solutions: One for the Application Shell and the other for a Module.

3.2 Coding CAB Application Shell - HelloCABShell

1. Start Visual Studio 2005 and create a new Visual C#, Windows Application project called `HelloCABShell`

2. Add the following three *existing* CAB projects to your solution. These projects are installed and present in subdirectories under the `C:\Program Files\Microsoft Composite UI App Block\CSharp\Source` directory by default assuming you have already installed Composite UI Application Block in C#.

- CompositeUI.csproj
- CompositeUI.WinForms.csproj

- ObjectBuilder.csproj

3. Select `HelloCABShell` project from the solution and add REFERENCES to `CompositeUI`, `CompositeUI.Winforms`, and `ObjectBuilder` projects.

4. Add a new C# CODE FILE called `HelloCABWorkItem.cs` to the `HelloCABShell` project. Replace the *entire* contents of this file with the code in Table 3.1:

Table 3.1: HelloCABShellWorkItem (HelloCABShellWorkItem.cs)

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI;

namespace HelloCABShell
{
  public class HelloCABShellWorkItem : WorkItem
  {
  }
}
```

5. Add a menubar to the `HelloCABShell`'s main form that can be extended by any Module. Drag and drop a MENUSTRIP control onto `Form1`. Create a top level menu item called `File`.

6. Next we implement the *main* function for the CAB Shell and expose the `File` menu item so that other Modules can access and extend it. Replace *all* code in `Program.cs` with the code in Table 3.2.

7. Finally, we will add a WORKSPACE where hosted Modules can display their user interface. Drag and drop TABWORKSPACE from the Toolbox onto `Form1`. TABWORKSPACE is located under CompositeUI.Winforms Components tab of the Toolbox. Set the `Dock` property of `tabWorkspace1` to `Fill`.

We are now done with coding of a *generic* CAB Application Shell that allows any CAB Module to add commands to its `File` menu and allows any CAB Module to display Module's visual elements in its `tabWorkspace1` Workspace.

3.3 Coding CAB Module - HelloCABModule

1. Start *another* instance of Visual Studio 2005 and create a VISUAL C#, CLASS LIBRARY project called `HelloCABModule`

2. Add the following three existing CAB projects to your solution. These projects are installed and present in subdirectories under the `C:\Program Files\Microsoft Composite UI App Block\CSharp\Source` directory by default assuming you have already installed Composite UI Application Block in C#.

- CompositeUI.csproj
- CompositeUI.WinForms.csproj
- ObjectBuilder.csproj

3. Select `HelloCABModule` project from the solution and add REFERENCES to `CompositeUI`, `CompositeUI.Winforms`, and `ObjectBuilder` projects.

4. Create a *new* USER CONTROL to the project. Drag and drop a LABEL control onto `UserControl1`. Change TEXT property of `label1` to `Hello CAB!`. Open the source code file for the user control (`UserControl1.cs`) and add a USING directive for `using Microsoft.Practices.CompositeUI.SmartParts` and mark the `UserControl1` class with an attribute called `SmartPart` as shown in Table 3.3.

6. Add a *new* C# CODE FILE called `HelloWorkItem.cs` and copy the code from Table 3.4 into `HelloWorkItem.cs`.

7. Add a *new* C# CODE FILE called `ModuleInit.cs` and copy the code from Table 3.5 into `ModuleInit.cs`.

We have now completed coding a CAB Module, `HelloCABModule`, that we can plug into the CAB Application Shell, `HelloCABShell`.

3.4 Plugging HelloCABModule into HelloCABShell

1. Rebuild both `HelloCABShell` and `HelloCABModule` projects

2. So far `HelloCABShell` and `HelloCABModule` have no knowledge of each other. Now we will declaratively tell `HelloCABShell` to load `HelloCABModule`. Select `HelloCABShell` project and add a *new* XML FILE and name it `ProfileCatalog.xml`. Replace the *entire* contents of `ProfileCatalog.xml` with code from Table 3.6.

3. Create a new folder called `HelloCABApp` anywhere on your file system. Locate the files shown in Table 3.7 in respective `HelloCABShell` and `HelloCABModule` project `bin/debug` folders and copy these files into the `HelloCABApp` directory:

4. Double-click on `HelloCABShell.exe` to start the application. Choose File | Say Hello menu. You should see `Hello CAB!` displayed on a new tab in the Application Shell.

3.5 Summary

In this chapter we have covered the very basics of creating a CAB-based solution. You can of course place both the Application Shell and Module projects in the same Visual Studio Solution and specify POST BUILD commands to copy the built binaries to the desired deployment directory location. Furthermore, you can organize your code in `HelloModuleInit` and `HelloWorkItem` using Design Patterns such as Model View Presenter (MVP) or Model View Controller (MVC).

Our purpose in separating the Application Shell and the CAB Module into separate Visual Studio Solutions was to emphasize their decoupled nature and to bring the *Plug-in* Design Pattern utilized by CAB into light.

In the next chapter, we provide a quick reference for working with the CAB API.

Table 3.2: HelloCABShellApplication (Program.cs)

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using Microsoft.Practices.CompositeUI.WinForms;

namespace HelloCABShell
{
  public class HelloCABShellApplication :
    FormShellApplication<HelloCABShellWorkItem, Form1>
  {
    [STAThread]
    static void Main()
    {
      new HelloCABShellApplication().Run();
    }

    protected override void AfterShellCreated()
    {
      base.AfterShellCreated();
      // Expose File menu for use by Module(s)
      ToolStripMenuItem fileItem =
          (ToolStripMenuItem)Shell.MainMenuStrip.Items["fileToolStripMenuItem"];
      RootWorkItem.UIExtensionSites.RegisterSite
          ("FileDropDown", fileItem.DropDownItems);
    }
  }
}
```

Table 3.3: SmartPart attribute (UserControl1.cs)

```
...
using Microsoft.Practices.CompositeUI.SmartParts;

...

    [SmartPart]
    public partial class UserControl1 : UserControl

....
```

Table 3.4: HelloWorkItem (HelloWorkItem.cs)

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.SmartParts;
using Microsoft.Practices.CompositeUI.Commands;

namespace HelloCABModule
{
  public class HelloWorkItem : WorkItem
  {
    IWorkspace targetWorkspace;

    public void Run(IWorkspace targetWorkspace)
    {
      // save a reference to the target workspace for later use
      this.targetWorkspace = targetWorkspace;
      // add a menu item under File menu
      ToolStripMenuItem sayHelloMenuItem = new ToolStripMenuItem("Say Hello");
      RootWorkItem.UIExtensionSites["FileDropDown"].Add(sayHelloMenuItem);
      // associate a command handler with the "Say Hello" menu item
      this.Commands["SayHelloCmd"].AddInvoker(sayHelloMenuItem, "Click");
    }

    [CommandHandler("SayHelloCmd")]
    public void OnSayHello(object sender, EventArgs e)
    {
      UserControl control = this.Items.AddNew<UserControl1>();
      this.targetWorkspace.Show(control);
    }
  }
}
```

Table 3.5: HelloModuleInit (ModuleInit.cs)

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.Services;

namespace HelloCABModule
{
  public class HelloModuleInit : ModuleInit
  {
    private WorkItem parentWorkItem;

    [ServiceDependency]
    public WorkItem ParentWorkItem
    {
      set { parentWorkItem = value; }
      get { return parentWorkItem; }
    }

    public override void Load()
    {
      base.Load();
      HelloWorkItem workItem =
          ParentWorkItem.WorkItems.AddNew<HelloWorkItem>();
      workItem.Run(ParentWorkItem.Workspaces["tabWorkspace1"]);
    }
  }
}
```

Table 3.6: ProfileCatalog.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<SolutionProfile xmlns="http://schemas.microsoft.com/pag/cab-profile" >
  <Modules>
    <ModuleInfo AssemblyFile="HelloCABModule.dll" />
  </Modules>
</SolutionProfile>
```

Table 3.7: Deploying a CAB application

```
HelloCABShell.exe
Microsoft.Practices.CompositeUI.dll
Microsoft.Practices.CompositeUI.WinForms.dll
Microsoft.Practices.ObjectBuilder.dll
ProfileCatalog.xml
HelloCABModule.dll
```

Chapter 4

Working with the CAB API

One of the key design techniques used in the CAB Framework is *Dependency Injection*. This technique is paramount in enabling the loosely coupled nature of the Shell and the Modules. The enabling .NET technology that makes *Dependency Injection* possible is .NET *Reflection*. *Reflection* allows CAB Framework to inspect application code during execution.

4.1 Object Creation and Code Injection

The CAB framework injects code and object references into your CAB-based application at runtime. For this mechanism to work, CAB framework must be made aware of the code sections where injection needs to take place. This awareness is achieved through the following means:

1. The objects created by your application must be added to object collections managed by CAB. Alternatively, you can have the CAB software factories construct your application's objects thus implicitly adding your application objects to CAB-managed collections

2. The code injection points in the application code are marked using .NET attributes that CAB Framework can discover at runtime

Objects of a certain type being managed by the CAB Framework can be looked up as shown in Table 4.1.

CAB framework provides facilities to create new application objects. Table 4.2 shows an example of using CAB's *ObjectBuilder* component to construct and add a new object to a CAB-managed collection with one method call.

The code in Table 4.2 has the same effect as the code in Table 4.3 where we are now creating the `HelloWorkItem` object explicitly and using the `Add` method instead of

Table 4.1: Finding an object in CAB Items Collection

```
// workItem is a reference to a WorkItem object
ICollection<MyObject> coll = workItem.Items.FindByType<MyObject>();
```

Table 4.2: Constructing and adding objects using AddNew()

```
// Create HelloWorkItem object and add to ParentWorkItem
HelloWorkItem workItem =
        ParentWorkItem.WorkItems.AddNew<HelloWorkItem>();
```

`AddNew` method of `WorkItems` collection.

Table 4.3: Constructing and adding objects using Add()

```
HelloWorkItem workItem = new HelloWorkItem();
ParentWorkItem.WorkItems.Add<HelloWorkItem>();
```

We can also assign an identifier to an instance of a WorkItem added to CAB. Later, we can retrieve a particular instance of a WorkItem using the same identifier. Table 4.4 demonstrates this functionality.

How do we invoke a specific constructor when using the `AddNew` method of `WorkItems` collection? The answer is to mark a constructor of the target object with an attribute called `InjectionConstructor`. This attribute is used to specify which constructor of an object should be used for object construction by the `AddNew` method of `WorkItems` collection. Table 4.5 shows how to mark a class constructor with this attribute.

Table 4.6 is a slightly different implementation of `HelloModuleInit` that uses `InjectionConstructor` attribute in conjunction with the `ServiceDependency` attribute to get a reference to the *Root* WorkItem.

When CAB framework loads `HelloCABModule`, it uses the `InjectionConstructor` to construct an instance of `HelloModuleInit`. Further, CAB finds a `ServiceDependency` for the argument to this constructor. To honor this `ServiceDependency`, CAB sets an object reference for the `parentWorkItem` parameter. In this instance, this object reference is the *Root* WorkItem reference. In the sections to follow, you will see examples of the same pattern. That is, classes, class constructors, methods, and properties are marked with attributes. Subsequently, the CAB Framework discovers these classes, class constructors, methods, and properties and performs code injection.

Table 4.4: Constructing and adding objects with Identifiers

```
// Create HelloWorkItem object and add to ParentWorkItem
// Identify this WorkItem as "Item1"
HelloWorkItem workItem =
        ParentWorkItem.WorkItems.AddNew<HelloWorkItem>("Item1");

// Locate "Item1"
WorkItem targetWi = null;
foreach (KeyValuePair<string, WorkItem> wi in
 parentWorkItem.WorkItems)
{
  if (wi.Key == "Item1")
  {
    targetWi = wi.Value;
    break;
  }
}
if (targetWi != null)
  MessageBox.Show("Found it");
```

Table 4.5: Specifying default constructor for object creation

```
[InjectionConstructor]
public HelloWorkItem(type arg1, type arg2,...)
{
  ...
}
```

Table 4.6: Using Injection Constructor and Service Dependency

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.Services;
using Microsoft.Practices.ObjectBuilder;

namespace HelloCABModule
{
   public class HelloModuleInit : ModuleInit
  {
    private WorkItem parentWorkItem;

    [InjectionConstructor]
    public HelloModuleInit([ServiceDependency] WorkItem parentWorkItem)
    {
      this.parentWorkItem = parentWorkItem;
    }

    public override void Load()
    {
      base.Load();
      HelloWorkItem workItem =
          this.parentWorkItem.WorkItems.AddNew<HelloWorkItem>();
      workItem.Run(this.parentWorkItem.Workspaces["tabWorkspace1"]);
    }
  }
}
```

4.2 Module Initialization

At design time, the Shell and the Modules are not aware of each other. At runtime, the Shell becomes aware of what Modules to load by the Module listing in `ProfileCatalog.xml`.

Once CAB Framework knows what Modules to load, it needs an entry point into each Module. The CAB Framework looks for an application-defined class in the Module that implements an `IModule` interface. The easiest way to define such a class is to derive your Module initialization class from CAB's `ModuleInit` base class. Recall that in the `HelloCABApp` example in the previous chapter, `HelloModuleInit` was derived from `ModuleInit`. CAB Framework calls the overridden `Load` method of `HelloModuleInit`. Subsequently, CAB Framework calls the overridden `AddServices` method of `HelloModuleInit`. Typically a CAB Module instantiates its WorkItems in its `Load` method and creates and registers its Services in the `AddServices` method as evident from the code in Table 4.7.

Important Be sure to declare `HelloModuleInit` as a `public` class otherwise CAB Framework will not be able find and instantiate `HelloModuleInit` at runtime.

Care must be taken in listing the Modules in the *Profile Catalog*. For example, if Module A implements and registers a Service Module B relies on, Module A must be loaded first. CAB loads the Modules listed in `ProfileCatalog.xml` in the *reverse* order. That is, the Modules listed at the bottom of the catalog are loaded first. In the example cited, Module B should be listed before Module A in `ProfileCatalog.xml`.

Another way to manage Module dependencies is to specify the assembly a Module depends upon. This specification is done in the `Assembly.cs` file of the dependent Module as follows:

```
// An entry in Assembly.cs of ModuleB
[assembly: ModuleDependency("ModuleA")]
```

Alternatively, a CAB-based application can implement and register a class based on `IModuleEnumerator` interface. In this case, the Module list is obtained from the application-defined enumerator class instead of the `ProfileCatalog.xml`.

4.3 WorkItems

After a Module is loaded, the first order of business for the Module is to obtain a reference to the *Root* WorkItem. A Root WorkItem is created by the Application Shell when a CAB application starts. CAB Modules loaded by the Shell may add their WorkItems to the *Root* WorkItem. WorkItems can be organized into a hierarchy if your design calls for it. In other words, Sub-WorkItems may added to WorkItems created by Modules forming

Table 4.7: Module Initialization

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.Services;
using Microsoft.Practices.ObjectBuilder;

namespace HelloCABModule
{
   public class HelloModuleInit : ModuleInit
  {
    private WorkItem parentWorkItem;

    [InjectionConstructor]
    public HelloModuleInit([ServiceDependency] WorkItem parentWorkItem)
    {
      this.parentWorkItem = parentWorkItem;
    }

    public override void Load()
    {
      base.Load();
      HelloWorkItem workItem =
          this.parentWorkItem.WorkItems.AddNew<HelloWorkItem>();
      workItem.Run(this.parentWorkItem.Workspaces["tabWorkspace1"]);
    }

    public override void AddServices()
    {
      base.AddServices();
      // Add Module implemented Services
    }
  }
}
```

a hierarchy. Multiple instances of a WorkItem can be created and added as well. But before any of this can happen, a Module needs a reference to the *Root* WorkItem. The *Root* WorkItem is what gives a Module access to Shell's Workspaces, UI Extensions and Services. This is where CAB's *ObjectBuilder* component does its magic by providing a reference to the *Root* WorkItem to any Module that requests it.

Referring back to our code example of `HelloModuleInit`, the `ParentWorkItem` property of `HelloModuleInit` is set to the *Singleton* instance of the *Root* WorkItem by the CAB Framework. CAB knows about `HelloModuleInit` class object because `HelloModuleInit` implements the `IModule` interface. Recall `HelloModuleInit` is derived from `ModuleInit` that implements the `IModule` interface. An instance of `HelloModuleInit` is automatically created by the CAB Framework when a Module is loaded. Table 4.8 demonstrates how a reference to `Root` WorkItem is received by the Module.

Table 4.8: Obtaining a WorkItem Reference

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.Services;

namespace HelloCABModule
{
  public class HelloModuleInit : ModuleInit
  {
    private WorkItem parentWorkItem;

    [ServiceDependency]
    public WorkItem ParentWorkItem
    {
      set { parentWorkItem = value; }
      get { return parentWorkItem; }
    }

    ...

  }
}
```

Note that in Table 4.8, the `ParentWorkItem` property is marked with `ServiceDependency` attribute. That is how CAB knows to set this property to the reference of the WorkItem that instantiated `HelloModuleInit`. In this particular case, this reference will be the *Root* WorkItem. Once the Module has a reference to the *Root*

WorkItem, it has access to all the published facilities of the Shell such as Workspaces, UI Extensions and Services.

In the `Load` method of `HelloModuleInit` , we are creating and adding a new instance of `HelloWorkItem` to a CAB-managed object collection. Since `HelloWorkItem` is a `WorkItem`-derived class, we add `HelloWorkItem` instance to the `WorkItems` collection of the Parent WorkItem. By doing so CAB is now aware of the newly created WorkItem and can perform *Dependency Injection* as specified. The `Load` method also calls our *custom-defined* `Run` method of the `HelloWorkItem` passing to it a Shell's Workspace reference it retrieved from the *Root* WorkItem as shown in Table 4.9.

Table 4.9: Creating and invoking a WorkItem - Approach 1

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.SmartParts;
using Microsoft.Practices.CompositeUI.Commands;

namespace HelloCABModule
{
  public class HelloWorkItem : WorkItem
  {
    IWorkspace targetWorkspace;

    public void Run(IWorkspace targetWorkspace)
    {
      // save a reference to the target workspace for later use
      this.targetWorkspace = targetWorkspace;
      // add a menu item under File menu
      ToolStripMenuItem sayHelloMenuItem = new ToolStripMenuItem("Say Hello");
      RootWorkItem.UIExtensionSites["FileDropDown"].Add(sayHelloMenuItem);
      // associate a command handler with the "Say Hello" menu item
      this.Commands["SayHelloCmd"].AddInvoker(sayHelloMenuItem, "Click");
    }

    ...
  }
}
```

Alternatively, we can call the built-in `Run` method of the WorkItem instance as shown in Table 4.10.

Table 4.10: Creating and invoking a WorkItem - Approach 2

```
...

namespace HelloCABModule
{
  public class HelloModuleInit : ModuleInit
  {

    ...

    public override void Load()
    {
      base.Load();
      HelloWorkItem workItem =
          ParentWorkItem.WorkItems.AddNew<HelloWorkItem>();
      workItem.Run();
    }

    ...
  }
}
```

In this scenario, we would override the `OnRunStarted` method of the WorkItem as shown in Table 4.11.

Table 4.11: Overriding OnStarted method of a WorkItem

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.SmartParts;
using Microsoft.Practices.CompositeUI.Commands;

namespace HelloCABModule
{
  public class HelloWorkItem : WorkItem
{
  ...

  protected override void OnRunStarted()
  {
    base.OnRunStarted();
    // Assuming parentWorkItem is a reference to the Root WorkItem
    IWorkspace targetWorkspace =
      this.parentWorkItem.Workspaces["tabWorkspace1"];
    // save a reference to the target workspace for later use
    this.targetWorkspace = targetWorkspace;
    // add a menu item under File menu
    ToolStripMenuItem sayHelloMenuItem = new ToolStripMenuItem("Say Hello");
    RootWorkItem.UIExtensionSites["FileDropDown"].Add(sayHelloMenuItem);
    // associate a command handler with the "Say Hello" menu item
    this.Commands["SayHelloCmd"].AddInvoker(sayHelloMenuItem, "Click");
  }
  ...
}
```

The key difference between the two approaches is that in the first approach we are passing the *Workspace* reference from `HelloModuleInit` into the `HelloWorkItem` presumably to make `HelloWorkItem` agnostic to the actual Workspace used. In the latter case, we are retrieving the reference to the Workspace directly within `HelloWorkItem`.

Another possibility is *not* to call the `Run` method on the WorkItem at all at the time of WorkItem creation. In this case, a subsequent Command or Event Handler would lookup the WorkItem in CAB's `Item` collection and call the `Run` method on the WorkItem

object.

A reference to *parent* WorkItem may be retrieved inside an instance of a `WorkItem`-derived class object via its `Parent` property. In case of a *Sub*-WorkItem class object, the `Parent` WorkItem may be a reference to another WorkItem that is below the *Root* WorkItem in the WorkItem hierarchy. However, given any WorkItem, we can traverse our way up to the *Root* WorkItem. The *Parent* WorkItem of a WorkItem can be accessed via its `Parent` property.

A WorkItem can be activated (`Activate` method), deactivated (`Deactivate` method) and terminated (`Terminate` method). There are corresponding override-able methods that get invoked when WorkItems are activated, deactivated or terminated. The status of a WorkItem can be tested using its `Status` property. The following is a plausible usage scenario for these methods. A CAB Application Shell may iterate through all WorkItems and call `Terminate` on all WorkItems before closing down. The WorkItems may override `OnTerminating` method and perform a clean-up before the application closes down. Another usage might be when a user chooses a menu item that *Activates* a certain WorkItem. The target WorkItem overrides `OnActivated` method and shows its SmartPart to the user.

A WorkItem also provides several event notifications (delegates) that other objects can subscribe to such as `Activating, Activated, Deactivating, Deactivated, Disposed, IdChanged, Initialized, RunStarted, Terminating` and `Terminated`.

4.4 Adding a Command

WorkItems can extend UI elements of the Shell. The UI elements of the Shell typically invoke *Commands*. For example, in the `HelloWorkItem.Run()` method in Table 4.12 we add a new menu item called *Say Hello* to Shell's *File* menu. Furthermore, we instruct the CAB Framework to trigger `SayHelloCmd` Command when *Say Hello* menu item is chosen. Note that in the `Run` method, the `SayHelloCmd` is associated with the *Click* event of the *Say Hello* menu item added by `HelloWorkItem`.

Generally, commands are triggered when a user chooses a menu item or presses a toolbar button. However, it is possible to trigger a command programmatically as shown in Table 4.13.

A Command's status can be manipulated as shown in Table 4.14.

The code in Table 4.15 disables all commands.

4.5 Responding to a Command

CAB allows a Module to mark a method in the WorkItem (or any class object known to CAB) that should respond to a *Command*. The `HelloWorkItem` marks its `OnSayHello` method with `CommandHandler` attribute with `"SayHelloCmd"` as the parameter. CAB has

Table 4.12: Adding a Command

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.SmartParts;
using Microsoft.Practices.CompositeUI.Commands;

namespace HelloCABModule
{
  public class HelloWorkItem : WorkItem
  {
    IWorkspace targetWorkspace;

    public void Run(IWorkspace targetWorkspace)
    {
      // save a reference to the target workspace for later use
      this.targetWorkspace = targetWorkspace;
      // add a menu item under File menu
      ToolStripMenuItem sayHelloMenuItem = new ToolStripMenuItem("Say Hello");
      RootWorkItem.UIExtensionSites["FileDropDown"].Add(sayHelloMenuItem);
      // associate a command handler with the "Say Hello" menu item
      this.Commands["SayHelloCmd"].AddInvoker(sayHelloMenuItem, "Click");
    }

    ...

  }
}
```

Table 4.13: Invoking a Command programmatically

```
  this.Commands["FireEventWithArgs"].Execute();
```

Table 4.14: Setting Command status

```
  Command fireCmd = this.Commands["FireEventWithArgs"];
  fireCmd.Status = CommandStatus.Enabled;
```

Table 4.15: Disabling Commands

```
foreach(KeyValuePair<string, Command> cmd
  in WorkItem.Commands)
{
  cmd.Value.Status = CommandStatus.Disabled;
}
```

`HelloWorkItem` in its managed collection and is therefore aware of an attributed method it should call when a *Command* named `SayHelloCmd` is fired. Table 4.16 shows a typical command handler.

CAB takes on the responsibility of routing a Command to appropriate handlers when a Shell UI elements triggers a Command. There can be multiple Command handlers associated with a Command. Similarly, different UI Elements may trigger the same Command. For example, a Command may be triggered by a menu item as well as by a toolbar button. Likewise, a particular Command may be handled by multiple command handlers.

4.6 Creating a SmartPart

In Table 4.17, the `OnSayHello` method of `HelloWorkItem` adds a new instance of `UserControl1` and displays the newly created control on a Shell Workspace. The `Items` property is inherited by the `HelloWorkItem` from CAB-defined `WorkItem` base class. The `Items` property of WorkItem holds references to all objects that CAB framework is aware of.

It is possible to associate an identifier with each SmartPart instance added as shown in Table 4.18. This is useful if your code needs to refer to a particular instance of one of many SmartPart instances of the same type.

What CAB-managed object collection is the newly created instance of `UserControl1` added to?

Recall we marked `UserControl1` class with `SmartPart` attribute (Table 4.19). This tells CAB that an instance of `UserControl1` object is a *SmartPart* that should be added to `SmartParts` collection of a WorkItem.

Alternatively, we can let CAB know of a SmartPart as shown in Table 4.20 without attributing the `UserControl1` class. Here we are explicitly adding the user control instance to the SmartParts collection of the WorkItem.

Once in SmartPart collection, CAB Framework can keep track of the SmartPart object and throw events as the SmartPart object is manipulated. Event notifications regard-

Table 4.16: Responding to a Command

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.SmartParts;
using Microsoft.Practices.CompositeUI.Commands;

namespace HelloCABModule
{
  public class HelloWorkItem : WorkItem
  {

    ...

    [CommandHandler("SayHelloCmd")]
    public void OnSayHello(object sender, EventArgs e)
    {
      UserControl control = this.Items.AddNew<UserControl1>();
      SmartPartInfo smartPartInfo =
        new SmartPartInfo("Say Hello Tab", "Displays Say Hello");
      this.targetWorkspace.Show(control, smartPartInfo);
    }
  }
}
```

Table 4.17: Creating a SmartPart

```
 [CommandHandler("SayHelloCmd")]
  public void OnSayHello(object sender, EventArgs e)
  {
    UserControl control = this.Items.AddNew<UserControl1>();
    SmartPartInfo smartPartInfo =
      new SmartPartInfo("Say Hello Tab", "Displays Say Hello");
    this.targetWorkspace.Show(control, smartPartInfo);
  }
```

Table 4.18: Creating a SmartPart with an ID

```
[CommandHandler("SayHelloCmd")]
 public void OnSayHello(object sender, EventArgs e)
 {
   UserControl control =
     this.Items.AddNew<UserControl1>(Guid.NewGuid().ToString());
   ...
 }
```

Table 4.19: SmartPart attribute

```
...
using Microsoft.Practices.CompositeUI.SmartParts;
...

    [SmartPart]
    public partial class UserControl1 : UserControl

....
```

Table 4.20: Adding a SmartPart to SmartParts collection

```
 [CommandHandler("SayHelloCmd")]
 public void OnSayHello(object sender, EventArgs e)
 {
   UserControl control = this.SmartParts.AddNew<UserControl1>();
   SmartPartInfo smartPartInfo =
     new SmartPartInfo("Say Hello Tab", "Displays Say Hello");
   this.targetWorkspace.Show(control, smartPartInfo);
 }
```

ing SmartParts can be received within Modules by adding event handlers to various event delegates exposed by Workspaces (for example, `SmartPartClosing, SmartPartActivated` etc.). Table 4.21 shows an example of how a Module would subscribe to `SmartPartClosing` event of a Workspace.

Table 4.21: Subscribing to Workspace/SmartPart events

```
ParentWorkItem.Workspaces["TargetWorkspaceName"].SmartPartClosing +=
              new EventHandler<WorkspaceCancelEventArgs>
              (MyHandler_SmartPartClosing);
```

Table 4.22 shows a sample definition of `MyHandler_SmartPartClosing`. A SmartPart can be deterred from closing by setting `Cancel` property of `WorkspaceCancelEventArgs` argument.

Table 4.22: Workspace/SmartPart event handler

```
protected void MyHandler_SmartPartClosing(object sender,
  WorkspaceCancelEventArgs e)
{
  // this.View is an application-defined property that
  // provides a references to a SmartPart
  // Is this a SmartPart we are interested in?
  if (e.SmartPart == this.View)
  {
      ParentWorkItem.Workspaces["TargetWorkspaceName"].SmartPartClosing -=
          new EventHandler<WorkspaceCancelEventArgs>
          (MyHandler_SmartPartClosing);
      ParentWorkItem.SmartParts.Remove(this.View);
      this.View = null;
  }
}
```

A SmartPart instance previously added and shown on a Workspace can be hidden using `Hide` method of Workspace object. Likewise, a hidden SmartPart instance can be re-shown using the `Show` method of Workspace object. A SmartPart can be *closed* programmatically as shown in Table 4.23.

A SmartPart may implement `ISmartPartInfo` interface to provide more information about itself such as its title and description. If a SmartPart does not implement `ISmartPartInfo` interface, equivalent information about the SmartPart may be provided using *SmartPartInfo* object when calling the `Show` method on the Workspace for a SmartPart as is the case in Table 4.24.

Table 4.23: Closing a SmartPart

```
// this.View is a reference to a SmartPart
ParentWorkItem.Workspaces["TargetWorkspaceName"].Close(this.View);
ParentWorkItem.SmartParts.Remove(this.View);
this.View = null;
```

Table 4.24: Setting SmartPartInfo

```
[CommandHandler("SayHelloCmd")]
public void OnSayHello(object sender, EventArgs e)
{
  UserControl control = this.Items.AddNew<UserControl1>();
  SmartPartInfo smartPartInfo =
    new SmartPartInfo("Say Hello Tab", "Displays Say Hello");
  this.targetWorkspace.Show(control, smartPartInfo);
}
```

4.7 Creating a Service

Services in CAB also live within WorkItems. A Service is simply a class that implements functionality that other components may find useful. A Service that lives in the *Root* WorkItem is visible to all Modules. `Service` attribute is used to declaratively create an instance of class that provides services to other Modules. The Service class so marked is instantiated during application startup and is registered as Service in the *Root* WorkItem (Table 4.25). The constructor parameter for the `Service` attribute is the Service interface itself. The Service interface is used as the key for subsequent lookup of the service.

Table 4.25: Creating and registering a Service automatically

```
[Service(typeof(IUsefulService))]
public class UsefulService : IUsefulService
{
  ...
}
```

Services may also be added via code as shown in Table 4.26.

A specific constructor of `UsefulService` class may be invoked by `WorkItem.Services.AddNew` by marking the desired constructor with `[InjectionConstructor]` attribute.

Table 4.26: Creating and registering a Service via code

```
// WorkItem is an object reference to Root or to another WorkItem
WorkItem.Services.AddNew<UsefulService, IUsefulService>();

OR

// Create the service object and add it to service
UsefulService usefulService = new UsefulService();
WorkItem.Services.Add<usefulService, IUsefulService>;
```

At what point in your code do you create and add Services? CABApplication (in the Application Shell) has an override-able method called `AddServices`. Similarly, `ModuleInit` (or `IModule`-derived class) has an override-able method called `AddServices`. These are appropriate methods to override when adding custom Services. Table 4.27 demonstrates a sample `AddServices` method.

Table 4.27: Where to add Services (AddServices method)

```
// method overridden in FormShellApplication-derived class
protected override void AddServices()
{
  base.AddServices();
  RootWorkItem.Services.AddNew<UsefulService, IUsefulService>();

}
```

Note we call the `AddServices` method on the *base* class before adding our own Services to ensure that all CAB-defined Services are created and registered. For an application-provided Service that requires a reference to a User Interface object, you will have to instantiate the Service from the overridden `AfterShellCreated` of the `FormShellApplication`. This is necessary to ensure that the UI elements have been instantiated before they are referenced by the Service.

4.8 Consuming a Service

A *property* of an object that CAB is aware of can be code injected by CAB at runtime by marking the property with `ServiceDependency` attribute. CAB searches for the requested Service (identified by its interface) in the current WorkItem. If the requested service is not found in the current WorkItem, the parent WorkItem is searched until the Root

WorkItem is reached. In case the requested Service is not located, an exception is thrown. Therefore, it is important that your Modules and WorkItems are loaded in the correct sequence such that dependent Modules are loaded *after* the Modules implementing the requested Service(s) are loaded.

Table 4.28 demonstrates how an object obtains a reference to a Service. CAB knows by the type of the property, `IUsefulService`, what Service to look for in the WorkItems.

Table 4.28: Getting a reference to a Service via Service Dependency

```
private IUsefulService service;

[ServiceDependency]
public IUsefulService MyService
{
  set { service = value; }
}
```

It is also possible to force setting of a Service object reference before a dependent object is created. The `ServiceDependency` attribute can be used in a class object constructor as shown in Table 4.29. In this case, CAB passes the requested service reference to the dependent object constructor.

Table 4.29: Getting a reference to a Service at object construction

```
private IUsefulService service;

[InjectionConstructor]
public SomeApplicatonDefinedClass
([ServiceDependency] IUsefulService service)
{
  this.service = service;
}
```

Finally, a reference to a Service may be obtained via code as shown in Table 4.30.

Table 4.30: Getting a reference to a Service via code

```
  // WorkItem is an object reference to Root or to another WorkItem
  IUsefulService usefulService = WorkItem.Services.Get<IUsefulService>();
```

4.9 Maintaining WorkItem State

When you create WorkItem you may want to keep the current state of the WorkItem along with the WorkItem. For this purpose, WorkItems provide a state bag where a state of a WorkItem may be kept. A WorkItem can hold an object representing its state in its *State* bag. The *State* bag maps an application-defined name to the state object as shown in Table 4.31.

Table 4.31: Holding WorkItem State

```
WorkItem.State["mystatename"] = new MyWorkItemState();
```

Once again CAB is able to automatically set references to *State* properties of objects that are known to CAB as shown in Table 4.32.

Table 4.32: Obtaining WorkItem State automatically

```
MyWorkItemState workItemStateData;

[State("mystatename")]
public MyWorkItemState MyWorkItemStateData
{
  get { return this.workItemStateData; }
  set { this.workItemStateData = value; }
}
```

Alternatively, a state object can be retrieved from a WorkItem as shown in Table 4.33.

Table 4.33: Obtaining WorkItem State via code

```
// WorkItem is an object reference to Root or to another WorkItem
MyWorkItemState workItemStateData =
              WorkItem.State["mystatename"];
```

4.10 Publishing an Event

Ideally, CAB Modules should *not* have design time dependencies on each other. In other words, Modules should not make direct calls to other Modules. Then how do Modules

communicate with each other? CAB provides three primary ways for Modules to collaborate:

1. Services can be created by one Module and utilized by others
2. All Modules can have an object reference to the *Root* WorkItem. As such, the *Root* WorkItem can be thought of a central, shared data structure among Modules
3. Modules can publish events; Interested Modules can listen to events and act upon receiving an event notification

CAB has an *Event Broker* component that allows Modules to publish and subscribe to events. Each event should be uniquely identified by a URI (Universal Resource Identifiers). By using URIs to identify events we can avoid event name conflicts when events are named by various collaborating Modules.

To define an event, three decisions need to be made:

1. What is the name of the event? As discussed previously, URIs are a good choice for this purpose
2. The scope of the event or to whom should this event be announced to. There are three options here: Global (fired on all WorkItem instances), WorkItem (fired on WorkItem instance that publishes the event), Descendants (fired on WorkItem instance that publishes the event and all descendant WorkItems)
3. The data to accompany the Event

Table 4.34 shows a simple event declaration.

Table 4.34: Declaring an Event

```
[EventPublication("topic://HelloCABModule/DataChanged",
  PublicationScope.Global)]
public event EventHandler NewDataAvailable;
```

The declaration of the EventHandler is attributed with `EventPublication`. The `EventPublication` constructor takes two arguments: the first is the name of the event (URI) and the second is the scope of event (Global). Now all that is left is to fire this event whenever desirable. Code in Table 4.35 does just that.

In Table 4.35, we are first checking whether there are any subscribers to this event at all by comparing `NewDataAvailable` to `null`. CAB would have added subscribers (event handlers) via its code injection facilities if there were in fact subscribers to this event.

Table 4.35: Firing an Event

```
// Assuming NewDataAvailable event handler is defined in this class
if (this.NewDataAvailable != null)
{
  NewDataAvailable(this, null);
}
```

CAB looks for subscribers in all of its object (item) collections. The rest of the code is standard .NET code for firing events.

So far we have published an event that does not provide any data to the subscriber of the event. Passing data as part of an event is simply a matter of defining an `EventArgs`-derived class and passing an object instance of the derived class when firing the event. CAB provides a template called `DataEventArgs` to make this derivation easier. The `DataEventArgs` is defined in the `Microsoft.Practices.CompositeUI.Utility` namespace. Table 4.36 shows how this scheme works.

Table 4.36: Declaring an Event with Data

```
[EventPublication("topic://HelloCABModule/DataChangedWithArgs",
  PublicationScope.Global)]
public event EventHandler<DataEventArgs<int>> NewDataAvailableWithArgs;
```

In Table 4.36, we have defined an event handler that expects an `EventArgs` object of type `DataEventArgs<int>`. Here we have defined our data type to be `int`. The data type can be any data type that fits your application needs. Table 4.37 shows how this event is fired with accompanying data.

Table 4.37: Firing an Event with data

```
if (this.NewDataAvailableWithArgs != null)
 {
   DataEventArgs<int> eventArg = new DataEventArgs<int>(1000);
   NewDataAvailableWithArgs(this, eventArg);
 }
```

Next, we will see how Modules subscribe to events.

4.11 Subscribing to an Event

In CAB Framework, any method with an appropriate signature that is part of an object known to CAB can subscribe to an event. Further, more than one method can subscribe to the same event. However, the order of execution of event handler methods is not guaranteed. The signature of the subscriber method must match event handler signature defined by the publisher. The first argument is the sender of the event and the second argument is an object of type `EventArgs` or a class object derived from `EventArgs`. The second argument may be `null` if the publisher calls the event handlers with no data. It is possible to stipulate on what thread an event handler method executes. Table 4.38 shows an example of a method that subscribes to an event.

Table 4.38: Subscribing to an Event

```
[EventSubscription("topic://HelloCABModule/DataChanged",
  ThreadOption.UserInterface)]
public void HandleDataChanged(object sender, EventArgs e)
{
  MessageBox.Show("Event Fired!");
}
```

In Table 4.38, the method has subscribed to *topic://HelloCABModule/DataChanged* event. The `HandleDataChanged` method receives two arguments: the first is the sender object of the event and the second argument is the data sent by the publisher along with the event. The thread options are as follow:

Background The call is done asynchronously on a background thread

Publisher The call is done on the same thread on which the EventTopic was fired

UserInterface The call is done on the UI thread

You would want to use *UserInterface* thread option, if the handler code makes any calls to manipulate UI elements of the application.

Table 4.39 shows how to retrieve data passed along with the event. This subscriber method corresponds to the example of event publication in Table 4.37 that sends data (`DataEventArgs<int>`) along with the event.

Table 4.39: Subscribing to an Event and retrieving Event Data

```
[EventSubscription("topic://HelloCABModule/DataChangedWithArgs",
 ThreadOption.UserInterface)]
public void HandleDataChanged(object sender, DataEventArgs<int> e)
{
  MessageBox.Show(
   String.Format("Event Fired With Arguments! Data Value = {0}", e.Data));
}
```

Bibliography

[GOF] Erich Gamma...[et al.], *Design Patterns: elements of reusable object-oriented software*, Addison-Wesley professional computing series, 1995.

[IOC] Martin Fowler, *Inversion of Control Containers and the Dependency Injection pattern*, http://www.martinfowler.com/articles/injection.html, 2004.

[PARNAS] Hoffman...[et al.], *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley professional computing series, 2001.

List of Tables

Index