Bonus Chapter 1

Composition

WHAT'S IN THIS CHAPTER?

- Understanding the architecture of the Composition framework
- Using attributes for composition
- Registering parts using conventions
- Defining contracts
- Exporting and importing parts
- Lazy loading of parts

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The Wrox.com code downloads for this chapter are found at `www.wrox.com` on the Download Code tab. The source code is also available at `https://github.com/ProfessionalCSharp/ProfessionalCSharp7` in the directory `Composition`.

The code for this chapter is divided into the following major examples:

- Attribute-Based Sample
- Convention-Based Sample
- UWP UI Calculator

INTRODUCTION

Microsoft Composition is a framework for creating independency between parts and containers. Parts can be used from containers without the need for the container to know the implementation or other details. The container just needs a contract—for example, an interface to use a part.

Microsoft Composition can be used with different scenarios, such as a dependency injection container, or you can even use it for adding functionality to an application after the application is released by dynamically loading add-ins into the application. To get into these scenarios, you need a foundation.

For making development of apps easier, it's a good practice to have separation of concerns (SoC). SoC is a design principle for separating a program into different sections where each section has its own responsibility. Having different sections allows you to reuse and update these sections independently of each other.

Having a tight coupling between these sections or components makes it hard to reuse and update these components independently of each other. Low coupling—for example, by using interfaces—helps this goal of independence.

Using interfaces for coupling and allowing them to develop independent of any concrete implementation, is known as the *dependency injection* design pattern. Dependency injection implements inversion of control where the control to define what implementation is used is reversed. The component for using an interface receives the implementation via a property (property injection) or via a constructor (constructor injection). Using a component just by an interface, it's not necessary to know about the implementation. Different implementations can be used for different scenarios—for example, with unit testing, a different implementation can be used that supplies test data.

Dependency injection can be implemented by using a *dependency injection container*. When you use a dependency injection container, the container defines for what interface which implementation should be used. Microsoft Composition can take the functionality of the container. This is one use case of this technology among the others.

> **NOTE** *Dependency injection is explained in detail in Chapter 20, "Dependency Injection." Chapter 20 shows the use of the dependency injection container* `Microsoft.Framework.DependencyInjection`.

Add-ins (or plug-ins) enable you to add functionality to an existing application. You can create a hosting application that gains more and more functionality over time—such functionality might be written by your team of developers, but different vendors can also extend your application by creating add-ins.

Today, add-ins are used with many different applications, such as Internet Explorer and Visual Studio. Internet Explorer is a hosting application that offers an add-in framework that is used by many companies to provide extensions when viewing web pages. The Shockwave Flash Object enables you to view web pages with Flash content. The Google toolbar offers specific Google features that can be accessed quickly from Internet Explorer. Visual Studio also has an add-in model that enables you to extend Visual Studio with different levels of extensions. Visual Studio add-ins makes use of the Managed Extensibility Framework (MEF), the first version of Microsoft Composition.

For your custom applications, it has always been possible to create an add-in model to dynamically load and use functionality from assemblies. However, all the issues associated with finding and using add-ins need to be resolved. You can accomplish that automatically by using Microsoft Composition. This technology helps to create boundaries and to remove dependencies between parts and the clients or callers that make use of the parts.

> **NOTE** *The first version of Microsoft Composition was known as Microsoft Extensibility Framework (MEF). MEF 1.x is still available with the full .NET Framework in the namespace* `System.ComponentModel.Composition`. *The new namespace for Microsoft Composition is* `System.Composition`. *Microsoft Composition is available with NuGet packages.*
>
> *MEF 1.x offers different catalogs—for example, an* `AssemblyCatalog` *or a* `DirectoryCatalog`*—to find types within an assembly or within a directory. The new version of Microsoft Composition doesn't offer this feature. However, you can build this part on your own. Chapter 16, "Reflection, Metadata, and Dynamic Programming," shows you how to load assemblies dynamically. You can use this information to build your own directory catalog.*

> **NOTE** *MEF (or Composition) has been available since .NET Framework 4.0 for creating add-ins with .NET. The .NET Framework offers another technology for writing flexible applications that load add-ins dynamically: the* Managed Add-in Framework (MAF). *MAF has been available since .NET 3.5. MAF uses a pipeline for communication between the add-in and the host application that makes the development process more complex but offers separation of add-ins via app domains or even different processes. In that regard, Composition is the simpler of these technologies. MAF and MEF can be combined to get the advantage of each, but it doubles the work. MAF was not ported to .NET Core and is only available with the full framework.*

The major namespace covered in this chapter is `System.Composition`.

ARCHITECTURE OF THE COMPOSITION LIBRARY

Microsoft Composition is built with parts and containers, as shown in Figure BC1-1. A container finds parts that are exported and connects imports to exports, thereby making parts available to the hosting application.

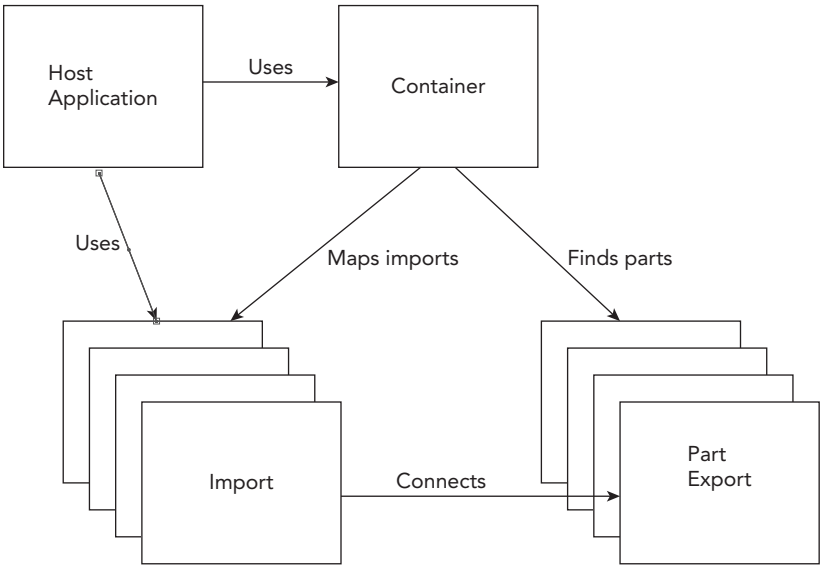

FIGURE BC1-1

Here's the full picture of how parts are loaded. As mentioned, parts are found with exports. Exports can be defined using attributes, or with a fluent API from C# code. Multiple export providers can be connected in chains for customizing exports—for example, with a custom export provider to only allow parts for specific users or roles. The container uses export providers to connect imports to exports and is itself an export provider.

Microsoft Composition consists of the NuGet packages shown in the following table. However, you don't need to deal with all the specific packages. You can add a dependency to `System.Composition` that itself references all these packages.

NUGET PACKAGE	DESCRIPTION
System.Composition.AttributedModel	This NuGet package contains Export and Import attributes. This package allows using attributes to export and import parts.
System.Composition.Convention	With this NuGet package it's possible to use plain old CLR objects (POCO) as parts. Rules can be applied programmatically to define exports.
System.Composition.Runtime	This NuGet package contains the runtime and thus is needed from the hosting application. The class CompositionContext is contained in this package. CompositionContext is an abstract class that allows getting exports for the context.
System.Composition.Hosting	This NuGet package contains the CompositionHost. CompositionHost derives from the base class CompositionContext and thus gives a concrete class to retrieve exports.
System.Composition.TypedParts	This NuGet package defines the class ContainerConfiguration. With ContainerConfiguration you can define what assemblies and parts should be used for exports. The class CompositionContextExtensions defines the extension method SatisfyImports for the CompositionContext to make it easy to match imports with exports.

Composition Using Attributes

Let's start with a simple example to demonstrate the Composition architecture. The hosting application can load add-ins. With Microsoft Composition, an add-in is referred to as a *part*. Parts are defined as *exports* and are loaded into a container that *imports* parts.

The sample code for AttributeBasedSample defines these dependencies and namespaces:

CalculatorContract (.NET Standard Class Library)

Namespace

System.Collections.Generic

SimpleCalculator (.NET Standard Class Library)

Dependencies

System.Compositon

CalculatorContract

Namespaces

System

System.Collections.Generic

System.Composition

AdvancedCalculator (.NET Standard Class Library)

Dependencies

System.Compositon

CalculatorContract

Namespaces

```
System
System.Collections.Generic
System.Composition
```

SimpleHost (Console App .NET Core)

Dependencies

```
CalculatorContract
System.Composition
```

Namespaces

```
System
System.Collections.Generic
System.Composition
System.Composition.Hosting
```

In this example, a Console App (.NET Core) is created to host calculator parts from a library. To create independence from the host and the calculator part, three projects are required. One project, `CalculatorContract`, holds the contracts that are used by both the add-in assembly and the hosting executable. The project `SimpleCalculator` contains the part and implements the contract defined by the contract assembly. The host uses the contract assembly to invoke the part.

The contracts in the assembly `CalculatorContract` are defined by two interfaces: `ICalculator` and `IOperation`. The `ICalculator` interface defines the methods `GetOperations` and `Operate`. The `GetOperations` method returns a list of all operations that the add-in calculator supports, and with the `Operate` method an operation is invoked. This interface is flexible in that the calculator can support different operations. If the interface defined `Add` and `Subtract` methods instead of the flexible `Operate` method, a new version of the interface would be required to support `Divide` and `Multiply` methods. With the `ICalculator` interface as it is defined in this example, however, the calculator can offer any number of operations with any number of operands (code file `AttributeBasedSample/CalculatorContract/ICalculator.cs`):

```
public interface ICalculator
{
  IList<IOperation> GetOperations();
  double Operate(IOperation operation, double[] operands);
}
```

The `ICalculator` interface uses the `IOperation` interface to return the list of operations and to invoke an operation. The `IOperation` interface defines the read-only properties `Name` and `NumberOperands` (code file `AttributeBasedSample/CalculatorContract/IOperation.cs`):

```
public interface IOperation
{
  string Name { get; }
  int NumberOperands { get; }
}
```

The `CalculatorContract` .NET Standard library doesn't require any reference to `System.Composition` assemblies. Only simple .NET interfaces are contained within it.

The add-in assembly `SimpleCalculator` contains classes that implement the interfaces defined by the contracts. The class `Operation` implements the interface `IOperation`. This class contains just two properties as defined by the interface. The interface defines `get` accessors of the properties; internal `set` accessors are

used to set the properties from within the assembly (code file AttributeBasedSample/SimpleCalculator/Operation.cs):

```
public class Operation: IOperation
{
  public string Name { get; internal set; }
  public int NumberOperands { get; internal set; }
}
```

The Calculator class provides the functionality of this add-in by implementing the ICalculator interface. The Calculator class is exported as a part as defined by the Export attribute. This attribute is defined in the System.Composition namespace in the NuGet package System.Composition.AttributedModel (code file AttributeBasedSample/SimpleCalculator/Calculator.cs):

```
[Export(typeof(ICalculator))]
public class Calculator: ICalculator
{
  public IList<IOperation> GetOperations() =>
    new List<IOperation>()
    {
      new Operation { Name="+", NumberOperands=2},
      new Operation { Name="-", NumberOperands=2},
      new Operation { Name="/", NumberOperands=2},
      new Operation { Name="*", NumberOperands=2}
    };

  public double Operate(IOperation operation, double[] operands)
  {
    double result = 0;
    switch (operation.Name)
    {
      case "+":
        result = operands[0] + operands[1];
        break;
      case "-":
        result = operands[0]—operands[1];
        break;
      case "/":
        result = operands[0] / operands[1];
        break;
      case "*":
        result = operands[0] * operands[1];
        break;
      default:
        throw new InvalidOperationException(
          $"invalid operation {operation.Name}");
    }
    return result;
  }
}
```

The hosting application is a Console App (.NET Core). The part uses an Export attribute to define what is exported; with the hosting application, the Import attribute defines what is used. Here, the Import attribute annotates the Calculator property that sets and gets an object implementing ICalculator. Therefore, any calculator add-in that implements this interface can be used here (code file AttributeBasedSample/SimpleHost/Program.cs):

```
class Program
{
  [Import]
  public ICalculator Calculator { get; set; }
  //...
}
```

In the entry method Main of the console application, a new instance of the Program class is created, and then the Bootstrapper method is invoked. In the Bootstrapper method, a ContainerConfiguration is created. With the ContainerConfiguration, a fluent API can be used to configure this object. The method WithPart<Calculator> finds the exports of the Calculator class to have it available from the composition host. The CompositionHost instance is created using the CreateContainer method of the ContainerConfiguration (code file AttributeBasedSample/SimpleHost/Program.cs):

```
static void Main()
{
  var p = new Program();
  p.Bootstrapper();
  p.Run();
}

public void Bootstrapper()
{
  var configuration = new ContainerConfiguration()
    .WithPart<Calculator>();

  using (CompositionHost host = configuration.CreateContainer())
  {
    //...
  }
}
```

Besides using the method WithPart (which has overloads and generic versions as well as non-generic versions), you can also use WithParts to add a list of parts and use WithAssembly or WithAssemblies to add the exports of an assembly.

Using the CompositionHost, you can access exported parts with the GetExport and GetExports methods:

You can also use more "magic." Instead of specifying all the export types you need to access, you can use the SatisfyImports method that is an extension method for the CompositionHost. The first parameter requires an object with imports. Because the Program class itself defines a property that has an Import attribute applied, the instance of the Program class can be passed to the SatisfyImports method. After invoking SatisfyImports, you will see that the Calculator property of the Program class is filled (code file AttributeBasedSample/SimpleHost/Program.cs):

```
using (CompositionHost host = configuration.CreateContainer())
{
  host.SatisfyImports(this);
}
```

With the Calculator property, you can use the methods from the interface ICalculator. GetOperations invokes the methods of the previously created add-in, which returns four operations. After asking the user what operation should be invoked and requesting the operand values, the add-in method Operate is called:

```
public void Run()
{
  var operations = Calculator.GetOperations();
  var operationsDict = new SortedList<string, IOperation>();
  foreach (var item in operations)
  {
    Console.WriteLine($"Name: {item.Name}, number operands: " +
      $"{item.NumberOperands}");
    operationsDict.Add(item.Name, item);
  }
  Console.WriteLine();

  string selectedOp = null;
  do
  {
```

```
      try
      {
        Console.Write("Operation? ");
        selectedOp =ReadLine();
        if (selectedOp.ToLower() == "exit" ||
          !operationsDict.ContainsKey(selectedOp))
          continue;

        var operation = operationsDict[selectedOp];
        double[] operands = new double[operation.NumberOperands];
        for (int i = 0; i < operation.NumberOperands; i++)
        {
          Console.Write($"\t operand {i + 1}? ");
          string selectedOperand = ReadLine();
          operands[i] = double.Parse(selectedOperand);
        }
        Console.WriteLine("calling calculator");
        double result = Calculator.Operate(operation, operands);
        Console.WriteLine($"result: {result}");
      }
      catch (FormatException ex)
      {
        Console.WriteLine(ex.Message);
        Console.WriteLine();
        continue;
      }
    } while (selectedOp != "exit");
  }
```

The output of one sample run of the application is shown here:

```
Name: +, number operands: 2
Name: -, number operands: 2
Name: /, number operands: 2
Name: *, number operands: 2
Operation? +
operand 1? 3
operand 2? 5
calling calculator
result: 8
Operation? -
operand 1? 7
operand 2? 2
calling calculator
result: 5
Operation? exit
```

Without any code changes in the host application, it is possible to use a completely different library for the parts. The project `AdvancedCalculator` defines a different implementation for the `Calculator` class to offer more operations. You can use this calculator in place of the other one by referencing the project `AdvancedCalculator` with the `SimpleHost` project.

Here, the Calculator class implements the additional operators %, ++, and -- (code file `AttributeBasedSample/AdvancedCalculator/Calculator.cs`):

```
[Export(typeof(ICalculator))]
public class Calculator: ICalculator
{
  public IList<IOperation> GetOperations() =>
    new List<IOperation>()
  {
    new Operation { Name="+", NumberOperands=2},
```

```
    new Operation { Name="-", NumberOperands=2},
    new Operation { Name="/", NumberOperands=2},
    new Operation { Name="*", NumberOperands=2},
    new Operation { Name="%", NumberOperands=2},
    new Operation { Name="++", NumberOperands=1},
    new Operation { Name="—", NumberOperands=1}
  };

  public double Operate(IOperation operation, double[] operands)
  {
    double result = 0;
    switch (operation.Name)
    {
      case "+":
        result = operands[0] + operands[1];
        break;
      case "-":
        result = operands[0]—operands[1];
        break;
      case "/":
        result = operands[0] / operands[1];
        break;
      case "*":
        result = operands[0] * operands[1];
        break;
      case "%":
        result = operands[0] % operands[1];
        break;
      case "++":
        result = ++operands[0];
        break;
      case "—":
        result =—operands[0];
        break;
      default:
        throw new InvalidOperationException(
          $"invalid operation {operation.Name}");
    }
    return result;
  }
}
```

> **NOTE** *With the* `SimpleHost` *you can't use both implementations of the* `Calculator` *at one time. You need to remove the reference* `SimpleCalculator` *before using the* `AdvancedCalculator`*, and the other way around. Later in this chapter, you see how multiple exports of the same type can be used with one container.*

Now you've seen imports, exports, and catalogs from the Composition architecture. In case you want to use existing classes where you can't add an attribute with Composition, you can use convention-based part registration, which is shown in the next section.

Convention-Based Part Registration

Convention-based registration not only allows exporting parts without using attributes, it also gives you more options to define what should be exported—for example, using naming conventions such as the class name ends with `PlugIn`, or `ViewModel`, or using the suffix name `Controller` to find all controllers.

This introduction to convention-based part registration builds the same example code shown previously using attributes, but attributes are no longer needed; therefore, the same code is not repeated here. The same contract interfaces `ICalculator` and `IOperation` are implemented, and nearly the same part with the class `Calculator`. The difference with the `Calculator` class is that it doesn't have the `Export` attribute applied to it.

The solution `ConventionBasedSample` contains the following projects with these references and namespaces. With the `SimpleCalculator` project, a NuGet package for Microsoft Composition is not needed, as exports are not defined by this project.

CalculatorContract (Class Library .NET Standard)

Namespace

```
System.Collections.Generic
```

SimpleCalculator (Class Library .NET Standard)

Dependency

```
CalculatorContract
```

Namespaces

```
System
System.Collections.Generic
System.Composition
```

SimpleHost (Console App .NET Core)

Dependencies

```
CalculatorContract
System.Composition
```

Namespaces

```
System
System.Collections.Generic
System.Composition
System.Composition.Hosting
```

> **NOTE** *You need to create a directory* `c:/addins` *before compiling the solution. The hosting application of this sample solution loads assemblies from the directory* `c:/addins`. *That's why a post-build command is defined with the project* `SimpleCalculator` *to copy the library to the* `c:/addins` *directory.*

When you create the host application, all this becomes more interesting. Like before, a property of type `ICalculator` is created as shown in the following code snippet—it just doesn't have an `Import` attribute applied to it (code file `ConventionBasedSample/SimpleHost/Program.cs`):

```
public ICalculator Calculator { get; set; }
```

You can apply the `Import` attribute to the property `Calculator` and use only conventions for the exports. You can mix this, using conventions only with exports or imports, or with both—as shown in this example.

The `Main` method of the `Program` class looks like before; a new instance of `Program` is created, and because the `Calculator` property is an instance property of this class, the `Bootstrap` and `Run` methods are invoked (code file `ConventionBasedSample/SimpleHost/Program.cs`):

```
public static void Main()
{
  var p = new Program();
  p.Bootstrap();
  p.Run();
}
```

The `Bootstrap` method now creates a new `ConventionBuilder`. `ConventionBuilder` derives from the base class `AttributedModelBuilder`; thus, it can be used everywhere this base class is needed. Instead of using the `Export` attribute, convention rules are defined for types that derive from `ICalculator` to export `ICalculator` with the methods `ForTypesDerivedFrom` and `Export`. `ForTypesDerivedFrom` returns a `PartConventionBuilder`, which allows using the fluent API to continue with the part definition to invoke the `Export` method on the part type. Instead of using the `Import` attribute, the convention rule for the `Program` class is used to import a property of type `ICalculator`. The property is defined using a lambda expression (code file `ConventionBasedSample/SimpleHost/Program.cs`):

```
public void Bootstrap()
{
  var conventions = new ConventionBuilder();
  conventions.ForTypesDerivedFrom<ICalculator>()
    .Export<ICalculator>();
  conventions.ForType<Program>()
    .ImportProperty<ICalculator>(p => p.Calculator);
  //...
}
```

After the convention rules are defined, the `ContainerConfiguration` class is instantiated. With the container configuration to use the conventions defined by the `ConventionsBuilder`, the method `WithDefaultConventions` is used. `WithDefaultConventions` requires any parameter that derives from the base class `AttributedModelProvider`, which is the class `ConventionBuilder`. After defining to use the conventions, you could use the `WithPart` method like before to specify the part or parts where the conventions should be applied. For making this more flexible than before, now the `WithAssemblies` method is used to specify the assemblies that should be applied. All the assemblies that are passed to this method are filtered for types that derive from the interface `ICalculator` to apply the export. After the container configuration is in place, the `CompositionHost` is created like in the previous sample (code file `ConventionBasedSample/SimpleHost/Program.cs`):

```
public void Bootstrap()
{
  //...
  var configuration = new ContainerConfiguration()
    .WithDefaultConventions(conventions)
    .WithAssemblies(GetAssemblies("c:/addins"));

  using (CompositionHost host = configuration.CreateContainer())
  {
    host.SatisfyImports(this, conventions);
  }
}
```

The `GetAssemblies` method loads all assemblies from the given directory (code file `ConventionBasedSample/SimpleHost/Program.cs`):

```
private IEnumerable<Assembly> GetAssemblies(string path)
{
```

```
    IEnumerable<string> files = Directory.EnumerateFiles(path, "*.dll");
    var assemblies = new List<Assembly>();
    foreach (var file in files)
    {
      Assembly assembly = Assembly.LoadFile(file);
      assemblies.Add(assembly);
    }
    return assemblies;
  }
```

As you've seen, the `ConventionBuilder` is the heart of convention-based part registration and Microsoft Composition. It uses a fluent API and offers all the flexibility you'll see with attributes as well. Conventions can be applied to a specific type with `ForType`; or for types that derive from a base class or implement an interface, `ForTypesDerivedFrom`. `ForTypesMatching` enables specifying a flexible predicate. For example, `ForTypesMatching(t => t.Name.EndsWith("ViewModel"))` applies a convention to all types that end with the name `ViewModel`.

The methods to select the type return a `PartBuilder`. With the `PartBuilder`, exports and imports can be defined, as well as metadata applied. The `PartBuilder` offers several methods to define exports: `Export` to export a specific type, `ExportInterfaces` to export a list of interfaces, and `ExportProperties` to export properties. Using the export methods to export multiple interfaces or properties, a predicate can be applied to further define a selection. The same applies to importing properties or constructors with `ImportProperty`, `ImportProperties`, and `SelectConstructors`.

Now that we have briefly looked at the two ways of using Microsoft Composition with attributes and conventions, the next section digs into the details by using Windows applications to host parts.

DEFINING CONTRACTS

The following sample application extends the first one. The hosting application is a UWP (Universal Windows Platform) app that loads calculator parts for calculation functionality; other add-ins bring their own user interfaces into the host.

> **NOTE** *For more information about writing UWP applications, see Chapters 33 to 36.*

The `UICalculator` is a somewhat bigger solution, at least for a book. It demonstrates using Microsoft Composition with UWP. Of course, you can focus on one of these technologies, and use this framework from other applications, such as WPF or Windows Forms.

The projects and their dependencies of the solution are shown in Figure BC1-2. The `UWPCalculatorHost` project loads and manages parts. This project contains two types of parts. Parts without a user interface, and parts with UWP user interfaces. `SimpleCalculator` is a part like the one you've seen before. This part implements methods for calculations. What's different from the earlier calculation sample is that this part makes use of another part: `AdvancedOperations`. A part itself can make use of parts. Parts with XAML user interfaces are `FuelEconomy` and `TemperatureConversion`.

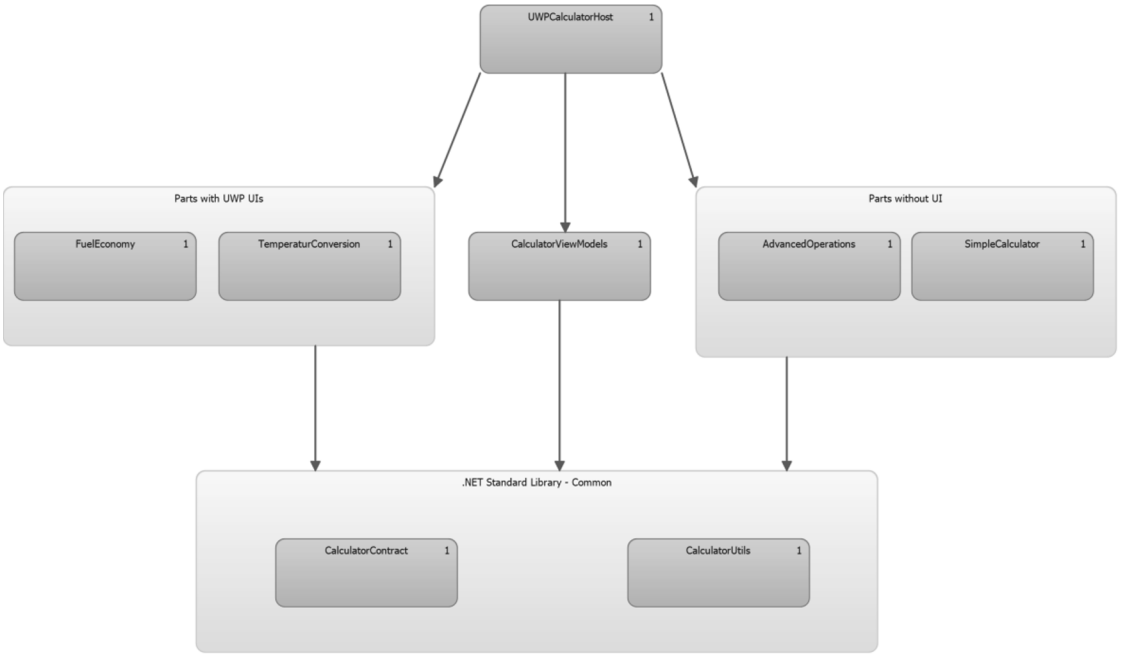

FIGURE BC1-2

You need the following projects with dependencies and namespaces:

CalculatorContract (Class Library .NET Standard)

Namespace

```
System.Collections.Generic
```

CalculatorUtils (Class Library .NET Standard)

Dependency

```
System.Composition
```

Namespaces

```
System
System.Collections.Generic
System.ComponentModel
System.Composition
System.Runtime.CompilerServices
System.Windows.Input
```

SimpleCalculator (Class Library .NET Standard)

Dependency

```
System.Composition
```

Namespaces

```
System
System.Collections.Generic
System.Composition
```

AdvancedOperations (Class Library .NET Standard)

Dependency

```
System.Composition
```

Namespaces

```
System.Composition
System.Threading.Tasks
```

Fuel Economy and Temp. Conversion (Class Library Universal Windows)

Dependency

```
System.Composition
```

Namespaces

```
System.Collections.Generic
System.Composition
Windows.UI.Xaml.Controls
```

Calculator View Models (Class Library .NET Standard)

Dependency

```
System.Composition
```

Namespaces

```
System
System.Collections.Generic
System.Collections.ObjectModel
System.Composition
System.Composition.Hosting
System.Linq
System.Windows.Input
```

UWP Calculator Host (UWP Application)

Dependencies

```
CalculatorContract
CalculatorUtils
CalculatorViewModels
FuelEconomy
```

SimpleCalculator

System.Composition

TemperatureConversion

Namespaces

System

Windows.ApplicationModel

Windows.ApplicationModel.Activation

Windows.UI.Xaml

Windows.UI.Xaml.Controls

Windows.UI.Xaml.Navigation

For the calculation, the same contracts that were defined earlier are used: `ICalculator` and `IOperation`. Added to this example is another contract: `ICalculatorExtension`. This interface defines the `UI` property that can be used by the hosting application. The `get` accessor of this property returns a `FrameworkElement`. The property type is defined to be of type `object` to support UWP and other UI technologies with this interface. For example, with WPF, the `FrameworkElement` is defined in the namespace `System.Windows`; with UWP it's in the namespace `Windows.UI.Xaml`. Defining the property of type object also doesn't require adding UWP dependencies to the library.

The `UI` property enables the add-in to return any user interface element that derives from `FrameworkElement` to be shown as the user interface within the host application (code file `UICalculator/CalculatorContract/ICalculatorExtension.cs`):

```
public interface ICalculatorExtension
{
  object UI { get; }
}
```

.NET interfaces are used to remove the dependency between one that implements the interface and one that uses it. This way, a .NET interface is also a good contract for Composition to remove a dependency between the hosting application and the add-in. If the interface is defined in a separate assembly, as with the `CalculatorContract` assembly, the hosting application and the add-in don't have a direct dependency. Instead, the hosting application and the add-in just reference the contract assembly.

From a Composition standpoint, an interface contract is not required at all. The contract can be a simple string. To avoid conflicts with other contracts, the name of the string should contain a namespace name—for example, `Wrox.ProCSharp.Composition.SampleContract`, as shown in the following code snippet. Here, the class `Foo` is exported by using the `Export` attribute, and a string passed to the attribute instead of the interface:

```
[Export("Wrox.ProCSharp.Composition.SampleContract")]
public class Foo
{
  public string Bar()
  {
    return "Foo.Bar";
  }
}
```

The problem with using a contract as a string is that the methods, properties, and events provided by the type are not strongly defined. Either the caller needs a reference to the type `Foo` to use it, or .NET reflection can be used to access its members. The C# `dynamic` keyword makes reflection easier to use and can be very helpful in such scenarios.

The hosting application can use the `dynamic` type to import a contract with the name `Wrox.ProCSharp.Composition.SampleContract`:

```
[Import("Wrox.ProCSharp.MEF.SampleContract")]
public dynamic Foo { get; set; }
```

With the `dynamic` keyword, the `Foo` property can now be used to access the `Bar` method directly. The call to this method is resolved during runtime:

```
string s = Foo.Bar();
```

Contract names and interfaces can also be used in conjunction to define that the contract is used only if both the interface and the contract name are the same. This way, you can use the same interface for different contracts.

> **NOTE** *The* `dynamic` *type is explained in Chapter 16, "Reflection, Metadata, and Dynamic Programming."*

EXPORTING PARTS

The previous example showed the part `SimpleCalculator`, which exports the type `Calculator` with all its methods and properties. The following example contains the `SimpleCalculator` as well, with the same implementation that was shown previously; and two more parts, `TemperatureConversion` and `FuelEconomy`, are exported. These parts offer a UI for the hosting application.

Creating Parts

The Class Library Universal Windows named `TemperatureConversion` defines a user interface as shown in Figure BC1-3. This control provides conversion between Celsius, Fahrenheit, and Kelvin scales. You use the first and second combo box to select the conversion source and target. Clicking the Calculate button starts the calculation to do the conversion.

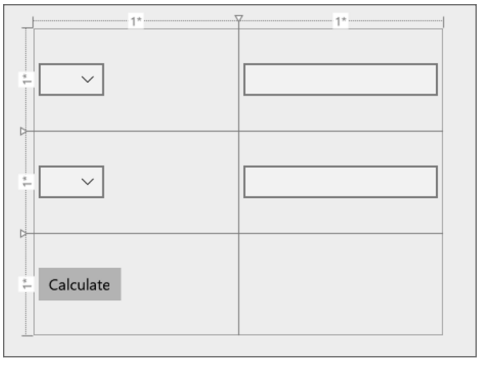

FIGURE BC1-3

The user control has a simple implementation for temperature conversion. The enumeration `TempConversionType` defines the different conversions that are possible with that control. The enumeration values shown in the two combo boxes are bound to the `TemperatureConversionTypes` property in the `TemperatureConversionViewModel`. The method `ToCelsiusFrom` converts the argument `t` from its original value to Celsius. The temperature source type is defined with the second argument,

`TempConversionType`. The method `FromCelsiusTo` converts a Celsius value to the selected temperature scale. The method `OnCalculate` is assigned to the Calculate command and invokes the `ToCelsiusFrom` and `FromCelsiusTo` methods to do the conversion according to the user's selected conversion type (code file `TemperatureConversion/TemperatureConversionViewModel.cs`):

```
public enum TempConversionType
{
  Celsius,
  Fahrenheit,
  Kelvin
}

public class TemperatureConversionViewModel: Observable
{
  public TemperatureConversionViewModel()
  {
    CalculateCommand = new RelayCommand(OnCalculate);
  }

  public RelayCommand CalculateCommand { get; }

  public IEnumerable<string> TemperatureConversionTypes =>
    Enum.GetNames(typeof(TempConversionType));

  private double ToCelsiusFrom(double t, TempConversionType conv)
  {
    switch (conv)
    {
      case TempConversionType.Celsius:
        return t;
      case TempConversionType.Fahrenheit:
        return (t-32) / 1.8;
      case TempConversionType.Kelvin:
        return (t-273.15);
      default:
        throw new ArgumentException("invalid enumeration value");
    }
  }

  private double FromCelsiusTo(double t, TempConversionType conv)
  {
    switch (conv)
    {
      case TempConversionType.Celsius:
        return t;
      case TempConversionType.Fahrenheit:
        return (t * 1.8) + 32;
      case TempConversionType.Kelvin:
        return t + 273.15;
      default:
        throw new ArgumentException("invalid enumeration value");
    }
  }

  private string _fromValue;
  public string FromValue
  {
    get => _fromValue;
    set => SetProperty(ref _fromValue, value);
  }

  private string _toValue;
```

```
    public string ToValue
    {
      get => _toValue;
      set => SetProperty(ref _toValue, value);
    }

    private string _fromType;
    public string FromType
    {
      get => _fromType;
      set => SetProperty(ref _fromType, value);
    }

    private string _toType;
    public string ToType
    {
      get => _toType;
      set => SetProperty(ref _toType, value);
    }

    public void OnCalculate()
    {
      double celsius = ToCelsiusFrom(double.Parse(FromValue),
        Enum.Parse<TempConversionType>(FromType));
      double result = FromCelsiusTo(celsius,
        Enum.Parse<TempConversionType>(ToType));
      ToValue = result.ToString();
    }
  }
```

So far, this control is just a simple user interface control with a view model. To create a part, the class `TemperatureCalculatorExtension` is exported by using the `Export` attribute. The class implements the interface `ICalculatorExtension` to return the user control `TemperatureConversion` from the `UI` property (code file `UICalculator/TemperatureConversion/TemperatureCalculatorExtension.cs`):

```
[Export(typeof(ICalculatorExtension))]
[CalculatorExtensionMetadata(
  Title = "Temperature",
  Description = "Temperature conversion",
  ImageUri = "ms-appx:///TemperatureConversion/Images/Temperature.png")]
public class TemperatureConversionExtension: ICalculatorExtension
{
  private object _control;
  public object UI =>
    _control ?? (_control = new TemperatureConversionUC());
}
```

For now, ignore the `CalculatorExtensionMetadata` attribute used in the previous code snippet. It is explained in the section "Exporting Metadata" later in this chapter.

The second user control that implements the interface `ICalculatorExtension` is `FuelEconomy`. With this control, either miles per gallon or liters per 100 km can be calculated. The user interface is shown in Figure BC1-4.

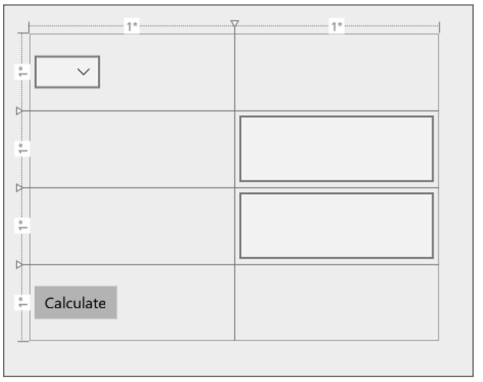

FIGURE BC1-4

The next code snippet shows the class `FuelEconomyViewModel`, which defines several properties that are bound from the user interface, such as a list of `FuelEcoTypes` that enables the user to select between miles and kilometers, and the `Fuel` and `Distance` properties, which are filled by the user (code file `UICalculator/FuelEconomyShared/FuelEconomyViewModel.cs`):

```
public class FuelEconomyViewModel: Observable
{
  public FuelEconomyViewModel()
  {
    InitializeFuelEcoTypes();
    CalculateCommand = new RelayCommand(OnCalculate);
  }

  public RelayCommand CalculateCommand { get; }

  //...

  public List<FuelEconomyType> FuelEcoTypes { get; } =
    new List<FuelEconomyType>();

  private void InitializeFuelEcoTypes()
  {
    var t1 = new FuelEconomyType
    {
      Id = "lpk",
      Text = "L/100 km",
      DistanceText = "Distance (kilometers)",
      FuelText = "Fuel used (liters)"
    };
    var t2 = new FuelEconomyType
    {
      Id = "mpg",
      Text = "Miles per gallon",
      DistanceText = "Distance (miles)",
      FuelText = "Fuel used (gallons)"
    };
```

```
    FuelEcoTypes.AddRange(new FuelEconomyType[] { t1, t2 });
  }

  private FuelEconomyType _selectedFuelEcoType;
  public FuelEconomyType SelectedFuelEcoType
  {
    get => _selectedFuelEcoType;
    set => SetProperty(ref _selectedFuelEcoType, value);
  }

  private string _fuel;
  public string Fuel
  {
    get => _fuel;
    set => SetProperty(ref _fuel, value);
  }

  private string _distance;
  public string Distance
  {
    get => _distance;
    set => SetProperty(ref _distance, value);
  }

  private string _result;
  public string Result
  {
    get => _result;
    set => SetProperty(ref _result, value);
  }
}
```

> **NOTE** *The base class* `Observable` *that is used with the sample code offers an implementation of the interface* `INotifyPropertyChanged`*. This class is found in the* `CalculatorUtils` *project.*

The calculation is within the `OnCalculate` method. `OnCalculate` is invoked via the `Command` assignment of the Calculate button (code file `FuelEconomy/FuelEconomyViewModel.cs`):

```
public void OnCalculate()
{
  double fuel = double.Parse(Fuel);
  double distance = double.Parse(Distance);
  FuelEconomyType ecoType = SelectedFuelEcoType;
  double result = 0;
  switch (ecoType.Id)
  {
    case "lpk":
      result = fuel / (distance / 100);
      break;
    case "mpg":
      result = distance / fuel;
      break;
    default:
      break;
  }
  Result = result.ToString();
}
```

Again, the interface `ICalculatorExtension` is implemented and exported with the `Export` attribute (code file `FuelEconomy/FuelCalculatorExtension.cs`):

```
[Export(typeof(ICalculatorExtension))]
[CalculatorExtensionMetadata(
  Title = "Fuel Economy",
  Description = "Calculate fuel economy",
  ImageUri = "ms-appx:///FuelEconomy/Images/Fuel.png")]
public class FuelCalculatorExtension: ICalculatorExtension
{
  private object _control;
  public object UI => _control ?? (_control = new FuelEconomyUC());
}
```

Before continuing the hosting applications to import the user controls, let's look at what other options you have with exports. A part itself can import other parts, and you can add metadata information to the exports.

Parts Using Parts

The `Calculator` class now doesn't directly implement the `Add` and `Subtract` methods but uses other parts that do this. To define parts that offer a single operation, the interface `IBinaryOperation` is defined (code file `CalculatorContract/IBinaryOperation.cs`):

```
public interface IBinaryOperation
{
  double Operation(double x, double y);
}
```

The class `Calculator` defines a property where a matching part of the `Subtract` method will be imported. The import is named `Subtract`, as not all exports of `IBinaryOperation` are needed—just the exports named `Subtract` (code file `SimpleCalculator/Calculator.cs`):

```
[Import("Subtract")]
public IBinaryOperation SubtractMethod { get; set; }
```

The `Import` in the class `Calculator` matches the `Export` of the `SubtractOperation` (code file `AdvancedOperations/Operations.cs`):

```
[Export("Subtract", typeof(IBinaryOperation))]
public class SubtractOperation: IBinaryOperation
{
  public double Operation(double x, double y) => x - y;
}
```

Now only the implementation of the `Operate` method of the `Calculator` class needs to be changed to make use of the inner part. There's no need for the `Calculator` itself to create a container to match the inner part. This is already automatically done from the hosting container if the exported parts are available within the registered types or assemblies (code file `SimpleCalculator/Calculator.cs`):

```
public double Operate(IOperation operation, double[] operands)
{
  double result = 0;
  switch (operation.Name)
  {
    //...
    case "-":
      result = SubtractMethod.Operation(operands[0], operands[1]);
      break;
    //...
}
```

Exporting Metadata

With exports, you can also attach metadata information. Metadata enables you to provide information in addition to a name and a type. This can be used to add capability information and to determine, on the import side, which of the exports should be used.

The `Calculator` class uses an inner part not only for the `Subtract` method, but also for the `Add` method. The `AddOperation` from the following code snippet uses the `Export` attribute named `Add` in conjunction with the `SpeedMetadata` attribute. The `SpeedMetadata` attribute specifies the `Speed` information `Speed.Fast` (code file `AdvancedOperations/Operations.cs`):

```
[Export("Add", typeof(IBinaryOperation))]
[SpeedMetadata(Speed = Speed.Fast)]
public class AddOperation: IBinaryOperation
{
  public double Operation(double x, double y) => x + y;
}
```

There's another export for an `Add` method with `SpeedMetadata Speed.Slow` (code file `AdvancedOperations/Operations.cs`):

```
[Export("Add", typeof(IBinaryOperation))]
[SpeedMetadata(Speed = Speed.Slow)]
public class SlowAddOperation: IBinaryOperation
{
  public double Operation(double x, double y)
  {
    Task.Delay(3000).Wait();
    return x + y;
  }
}
```

`Speed` is just an enumeration with two values (code file `CalculatorUtils/SpeedMetadata.cs`):

```
public enum Speed
{
  Fast,
  Slow
}
```

You can define metadata by creating an attribute class with the `MetadataAttribute` applied. This attribute is then applied to a part as you've seen with the `AddOperation` and `SlowAddOperation` types (code file `CalculatorUtils/SpeedMetadataAttribute.cs`):

```
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class)]
public class SpeedMetadataAttribute: Attribute
{
  public Speed Speed { get; set; }
}
```

> **NOTE** *For more information about how to create custom attributes, read Chapter 16.*

To access the metadata with the import, the class `SpeedMetadata` is defined. `SpeedMetadata` defines the same properties as the `SpeedMetadataAttribute` (code file `CalculatorUtils/SpeedMetadata.cs`):

```
public class SpeedMetadata
{
  public Speed Speed { get; set; }
}
```

With multiple `Add` exports defined, using the `Import` attribute as shown previously fails during runtime. Multiple exports cannot match just one import. The attribute `ImportMany` is used if more than one export of the same name and type is available. This attribute is applied to a property of type array or `IEnumeration<T>`.

Because metadata is applied with the export, the type of the property that matches the `Add` export is an array of `Lazy<IBinaryOperation, SpeedMetadata>` (code file `SimpleCalculator/Calculator.cs`):

```
[ImportMany("Add")]
public Lazy<IBinaryOperation, SpeedMetadata>[] AddMethods { get; set; }
```

`ImportMany` is explained with more detail in the next section. The `Lazy` type allows accessing metadata with the generic definition `Lazy<T, TMetadata>`. The class `Lazy<T>` is used to support lazy initialization of types on first use. `Lazy<T, TMetadata>` derives from `Lazy<T>` and supports, in addition to the base class, access to metadata information with the `Metadata` property.

The call to the `Add` method is now changed to iterate through the collection of `Lazy<IBinaryOperation, SpeedMetadata>` elements. With the `Metadata` property, the key for the capability is checked; if the `Speed` capability has the value `Speed.Fast`, the operation is invoked by using the `Value` property of `Lazy<T>` to invoke the operation (code file `SimpleCalculator/Calculator.cs`):

```
public double Operate(IOperation operation, double[] operands)
{
  double result = 0;
  switch (operation.Name)
  {
    case "+":
      foreach (var addMethod in AddMethods)
      {
        if (addMethod.Metadata.Speed == Speed.Fast)
        {
          result = addMethod.Value.Operation(operands[0], operands[1]);
        }
      }
      break;
      //...
```

Using Metadata for Lazy Loading

Using metadata with Microsoft Composition is not only useful for selecting parts based on metadata information. Another great use is providing information to the host application about the part before the part is instantiated.

The following example is implemented to offer a title, a description, and a link to an image for the calculator extensions `FuelEconomy` and `TemperatureConversion` (code file `CalculatorUtils/CalculatorExtensionMetadataAttribute.cs`):

```
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class)]
public class CalculatorExtensionMetadataAttribute: Attribute
{
  public string Title { get; set; }
  public string Description { get; set; }
  public string ImageUri { get; set; }
}
```

With a part, the `CalculatorExtensionMetadata` attribute is applied. The following is an example—the `FuelCalculatorExtension` (code file `FuelEconomy/FuelCalculatorExtension.cs`):

```
[Export(typeof(ICalculatorExtension))]
[CalculatorExtensionMetadata(
  Title = "Fuel Economy",
```

```
    Description = "Calculate fuel economy",
    ImageUri = "ms-appx:///FuelEconomy/Images/Fuel.png")]
public class FuelCalculatorExtension: ICalculatorExtension
{
  private object _control;
  public object UI => _control ?? (_control = new FuelEconomyUC());
}
```

Parts can consume a large amount of memory. If the user does not instantiate the part, there's no need to consume this memory. Instead, the title, description, and image can be accessed to give the user information about the part before instantiating it.

IMPORTING PARTS

Now let's look at using the parts with a hosting application. For the calculator part that only offers functionality, UWP needs to add a user interface. For the other parts with a user interface, just the panels that load this user interface are needed.

The design view of the UWP user control for the calculator part is shown in Figure BC1-5.

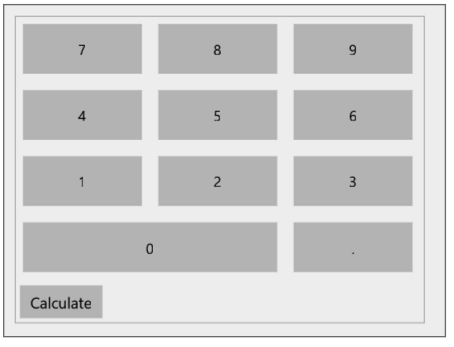

FIGURE BC1-5

For every part type, a separate import, manager, and view model is created. For using the part implementing the `ICalculator` interface, the `CalculatorImport` is used to define the `Import`, the `CalculatorManager` is used to create the `CompositionHost` and load the parts, and the `CalculatorViewModel` is used to define the properties and commands that are bound to the user interface. For using the part implementing the `ICalculatorExtension` interface, the `CalculatorExtensionImport`, `CalculatorExtensionManager`, and `CalculatorExtensionViewModel` are defined accordingly.

Let's start with the `CalculatorImport` class. With the first sample, just a property has been defined with the `Program` class to import a part. It's a good practice to define a separate class for imports. With this class, you can also define a method that is annotated with the attribute `OnImportsSatisfied`. This attribute marks the method that is called when imports are matched. In the sample code, the event `ImportsSatisfied` is fired. The `Calculator` property has the `Import` attribute applied. Here, the type is `Lazy<ICalculator>` for late instantiation. The part is instantiated only when the `Value` of the `Lazy` type is accessed (code file `CalculatorViewModels/CalculatorImport.cs`):

```
public class CalculatorImport
{
  public event EventHandler<ImportEventArgs> ImportsSatisfied;

  [Import]
  public Lazy<ICalculator> Calculator { get; set; }
```

```
  [OnImportsSatisfied]
  public void OnImportsSatisfied()
  {
    ImportsSatisfied?.Invoke(this,
      new ImportEventArgs
      {
        StatusMessage = "ICalculator import successful"
      });
  }
}
```

The `CalculatorManager` class instantiates the `CalculatorImport` class in the constructor. With the `InitializeContainer` method, the `ContainerConfiguration` class is instantiated to create the `CompositionHost` container with the types passed to the method. The method `SatisfyImports` matches exports to imports (code file `CalculatorViewModels/CalculatorManager.cs`):

```
public class CalculatorManager
{
  private CalculatorImport _calcImport;
  public event EventHandler<ImportEventArgs> ImportsSatisfied;

  public CalculatorManager()
  {
    _calcImport = new CalculatorImport();
    _calcImport.ImportsSatisfied += (sender, e) =>
    {
      ImportsSatisfied?.Invoke(this, e);
    };
  }

  public void InitializeContainer(params Type[] parts)
  {
    var configuration = new ContainerConfiguration().WithParts(parts);
    using (CompositionHost host = configuration.CreateContainer())
    {
      host.SatisfyImports(_calcImport);
    }
  }
  //...
}
```

The `GetOperators` method of the `CalculatorManager` invokes the `GetOperations` method of the `Calculator`. This method is used to display all the available operators in the user interface. As soon as a calculation is defined, the `InvokeCalculator` method is invoked to pass the operation and operands, and in turn invoke the `Operate` method in the calculator (code file `CalculatorViewModels/CalculatorManager.cs`):

```
public class CalculatorManager
{
  //...
  public IEnumerable<IOperation> GetOperators() =>
    _calcImport.Calculator.Value.GetOperations();

  public double InvokeCalculator(IOperation operation, double[] operands) =>
    _calcImport.Calculator.Value.Operate(operation, operands);
}
```

What's needed by the `CalculatorViewModel`? This view model defines several properties: the `CalcAddInOperators` property to list available operators, the `Input` property that contains the calculation entered by the user, the `Result` property that shows the result of the operation, and the `CurrentOperation` property that contains the current operation. It also defines the `_currentOperands` field that contains the operands selected. With the `Init` method, the container is initialized, and operators are retrieved

from the Calculator part. The `OnCalculate` method does the calculation using the part (code file `CalculatorViewModels/CalculatorViewModel.cs`):

```
public class CalculatorViewModel: Observable
{
  public CalculatorViewModel()
  {
    _calculatorManager = new CalculatorManager();
    _calculatorManager.ImportsSatisfied += (sender, e) =>
    {
      Status += $"{e.StatusMessage}\n";
    };
    CalculateCommand = new RelayCommand(OnCalculate);
  }

  public void Init(params Type[] parts)
  {
    _calculatorManager.InitializeContainer(parts);
    var operators = _calculatorManager.GetOperators();
    CalcAddInOperators.Clear();
    foreach (var op in operators)
    {
      CalcAddInOperators.Add(op);
    }
  }

  private CalculatorManager _calculatorManager;
  public ICommand CalculateCommand { get; set; }

  public void OnCalculate()
  {
    if (_currentOperands.Length == 2)
    {
      string[] input = Input.Split(' ');
      _currentOperands[1] = double.Parse(input[2]);
      Result = _calculatorManager.InvokeCalculator(_currentOperation,
        _currentOperands);
    }
  }

  private string _status;
  public string Status
  {
    get => _status;
    set => SetProperty(ref _status, value);
  }

  private string _input;
  public string Input
  {
    get => _input;
    set => SetProperty(ref _input, value);
  }

  private double _result;
  public double Result
  {
    get => _result;
    set => SetProperty(ref _result, value);
  }

  private IOperation _currentOperation;
```

```
  public IOperation CurrentOperation
  {
    get => _currentOperation;
    set => SetCurrentOperation(value);
  }

  private double[] _currentOperands;
  private void SetCurrentOperation(IOperation op)
  {
    try
    {
      _currentOperands = new double[op.NumberOperands];
      _currentOperands[0] = double.Parse(Input);
      Input += $" {op.Name} ";
      SetProperty(ref _currentOperation, op, nameof(CurrentOperation));
    }
    catch (FormatException ex)
    {
      Status = ex.Message;
    }
  }

  public ObservableCollection<IOperation> CalcAddInOperators { get; } =
    new ObservableCollection<IOperation>();
}
```

Importing Collections

An import connects to an export. When using exported parts, an import is needed to make the connection. With the `Import` attribute, it's possible to connect to a single export. If more than one part should be loaded, the `ImportMany` attribute is required and needs to be defined as an array type or `IEnumerable<T>`. Because the hosting calculator application allows many calculator extensions that implement the interface `ICalculatorExtension` to be loaded, the class `CalculatorExtensionImport` defines the property `CalculatorExtensions` of type `IEnumerable<ICalculatorExtension>` to access all the calculator extension parts (code file `CalculatorViewModels/CalculatorExtensionsImport.cs`):

```
public class CalculatorExtensionsImport
{
  public event EventHandler<ImportEventArgs> ImportsSatisfied;

  [ImportMany()]
  public IEnumerable<Lazy<ICalculatorExtension,
    CalculatorExtensionMetadataAttribute>>

  CalculatorExtensions { get; set; }

  [OnImportsSatisfied]
  public void OnImportsSatisfied()
  {
    ImportsSatisfied?.Invoke(this, new ImportEventArgs
    {
      StatusMessage = "ICalculatorExtension imports successful"
    });
  }
}
```

The `Import` and `ImportMany` attributes enable the use of `ContractName` and `ContractType` to map the import to an export.

The event `ImportsSatisfied` of the `CalculatorExtensionsImport` is connected to an event handler on creation of the `CalculatorExtensionsManager` to route firing the event, and in turn write

a message to a `Status` property that is bound in the UI for displaying status information (code file `CalculatorViewModels/CalculatorExtensionsManager.cs`):

```
public sealed class CalculatorExtensionsManager
{
  private CalculatorExtensionsImport _calcExtensionImport;
  public event EventHandler<ImportEventArgs> ImportsSatisfied;

  public CalculatorExtensionsManager()
  {
    _calcExtensionImport = new CalculatorExtensionsImport();
    _calcExtensionImport.ImportsSatisfied += (sender, e) =>
    {
      ImportsSatisfied?.Invoke(this, e);
    };
  }

  public void InitializeContainer(params Type[] parts)
  {
    var configuration = new ContainerConfiguration().WithParts(parts);
    using (CompositionHost host = configuration.CreateContainer())
    {
      host.SatisfyImports(_calcExtensionImport);
    }
  }

  public IEnumerable<Lazy<ICalculatorExtension,
    CalculatorExtensionMetadataAttribute>> GetExtensionInformation() =>
      _calcExtensionImport.CalculatorExtensions.ToArray();
}
```

Lazy Loading of Parts

By default, parts are loaded from the container—for example, by calling the extension method `SatisfyImports` on the `CompositionHost`. With the help of the `Lazy<T>` class, the parts can be loaded on first access. The type `Lazy<T>` enables the late instantiation of any type `T` and defines the properties `IsValueCreated` and `Value`. `IsValueCreated` is a Boolean that returns the information if the contained type `T` is already instantiated. `Value` initializes the contained type `T` on first access and returns the instance.

The import of an add-in can be declared to be of type `Lazy<T>`, as shown in the `Lazy<ICalculator>` example (code file `CalculatorViewModels/CalculatorImport.cs`):

```
[Import]
public Lazy<ICalculator> Calculator { get; set; }
```

Calling the imported property also requires some changes to access the `Value` property of the `Lazy<T> type. calcImport` is a variable of type `CalculatorImport`. The `Calculator` property returns `Lazy<ICalculator>`. The `Value` property instantiates the imported type lazily and returns the `ICalculator` interface, enabling the `GetOperations` method to be invoked to get all supported operations from the calculator add-in (code file `CalculatorViewModels/CalculatorManager.cs`):

```
public IEnumerable<IOperation> GetOperators() =>
  _calcImport.Calculator.Value.GetOperations();
```

Reading Metadata

The parts `FuelEconomy` and `TemperatureConversion`—all the parts that implement the interface `ICalculatorExtension`—are lazy loaded as well. As you've seen earlier, a collection can be imported

with a property of `IEnumerable<T>`. Instantiating the parts lazily, the property can be of type `IEnumerable<Lazy<T>>`. Information about these parts is needed before instantiation to display information to the user about what can be expected with these parts. These parts offer additional information using metadata, as shown earlier. Metadata information can be accessed using a `Lazy` type with two generic type parameters. Using `Lazy<ICalculatorExtension, CalculatorExtensionMetadataAttribute>`, the first generic parameter, `ICalculatorExtension`, is used to access the members of the instantiated type; the second generic parameter, `ICalculatorExtensionMetadataAttribute`, is used to access metadata information (code file `CalculatorViewModels/CalculatorExtensionsImport.cs`):

```
[ImportMany()]
public IEnumerable<Lazy<ICalculatorExtension,
  CalculatorExtensionMetadataAttribute>> CalculatorExtensions { get; set; }
```

The method `GetExtensionInformation` returns an array of `Lazy<ICalculatorExtension, CalculatorExtensionMetadataAttribute>`, which can be used to access metadata information about the parts without instantiating the part (code file `CalculatorViewModels/CalculatorExtensionsManager.cs`):

```
public IEnumerable<Lazy<ICalculatorExtension,
  CalculatorExtensionMetadataAttribute>> GetExtensionInformation() =>
    _calcExtensionImport.CalculatorExtensions.ToArray();
```

The `GetExtensionInformation` method is used in the `CalculatorExtensionsViewModel` class on initialization to fill the `Extensions` property (code file `CalculatorViewModels/CalculatorExtensionsViewModel.cs`):

```
public class CalculatorExtensionsViewModel: BindableBase
{
  private CalculatorExtensionsManager _calculatorExtensionsManager;

  public CalculatorExtensionsViewModel()
  {
    _calculatorExtensionsManager = new CalculatorExtensionsManager();
    _calculatorExtensionsManager.ImportsSatisfied += (sender, e) =>
    {
      Status += $"{e.StatusMessage}\n";
    };
  }

  public void Init(params Type[] parts)
  {
    _calculatorExtensionsManager.InitializeContainer(parts);
    foreach (var extension in
    _calculatorExtensionsManager.GetExtensionInformation())
    {
      var vm = new ExtensionViewModel(extension);
      vm.ActivatedExtensionChanged += OnActivatedExtensionChanged;
      Extensions.Add(vm);
    }
  }

  public ObservableCollection<ExtensionViewModel> Extensions { get; } =
    new ObservableCollection<ExtensionViewModel>();
  //...
```

Within the XAML code, metadata information is bound. The `Lazy` type has a `Metadata` property that returns `CalculatorExtensionMetadataAttribute`. This way, `Description`, `Title`, and `ImageUri`

can be accessed for data binding without instantiating the add-ins (code file `UWPCalculatorHost/MainPage.xaml`):

```
<ListView ItemsSource="{x:Bind ViewModel.Extensions}">
  <ListView.ItemsPanel>
    <ItemsPanelTemplate>
      <StackPanel Orientation="Horizontal" />
    </ItemsPanelTemplate>
  </ListView.ItemsPanel>
  <ListView.ItemTemplate>
    <DataTemplate x:DataType="comp:ExtensionViewModel">
      <AppBarButton Label="{x:Bind Extension.Metadata.Title, Mode=OneTime}"
        ToolTipService.ToolTip="{x:Bind Extension.Metadata.Description,
          Mode=OneTime}"
        Width="120"
        Command="{x:Bind ActivateCommand}" IsCompact="False" >
        <AppBarButton.Icon>
          <BitmapIcon UriSource="{x:Bind Extension.Metadata.ImageUri,
            Mode=OneTime}" />
        </AppBarButton.Icon>
      </AppBarButton>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Figure BC1-6 shows the running application where metadata from the calculator extensions is read—it includes the image, the title, and the description. With Figure BC1-7 you can see an activated calculator extension.

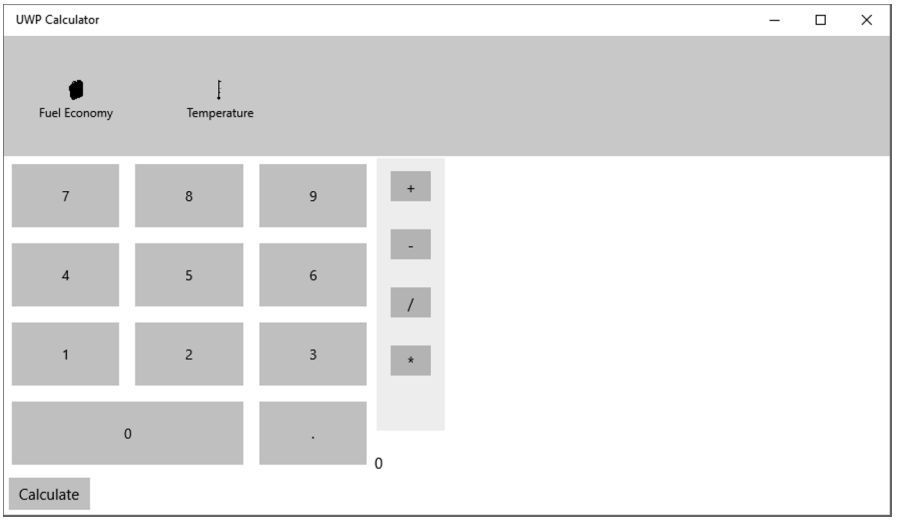

FIGURE BC1-6

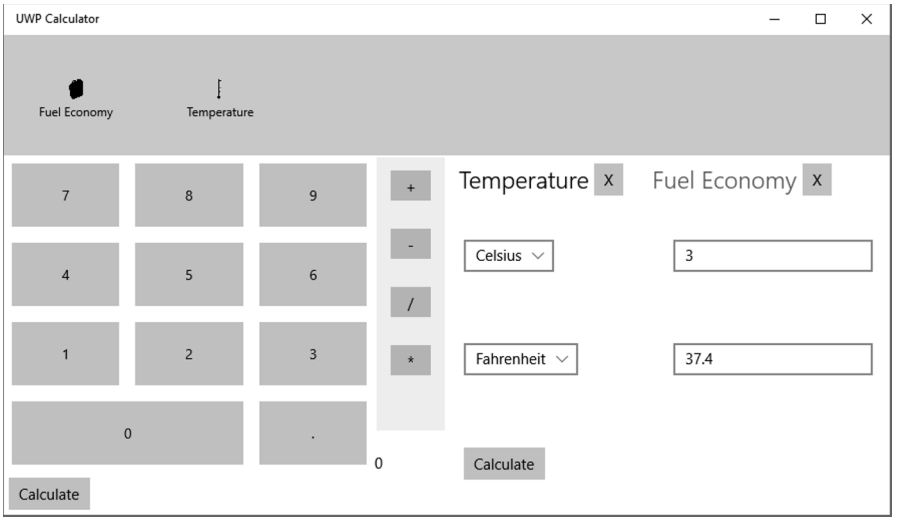

FIGURE BC1-7

SUMMARY

In this chapter, you learned about the parts, exports, imports, and containers of Microsoft Composition. You've learned how an application can be built up with complete independency of its parts and dynamically load parts that can come from different assemblies.

You've seen how you can use either attributes or conventions to match exports and imports. Using conventions allows using parts where you can't change the source code to add attributes, and it also gives the option to create a framework based on Composition that doesn't require the user of your framework to add attributes for importing the parts.

You've also learned how parts can be lazy loaded to instantiate them only when they are needed. Parts can offer metadata that can give enough information for the client to decide whether the part should be instantiated.