A. Leonard, *Building Custom Tasks for SQL Server Integration Services*
https://doi.org/10.1007/978-1-4842-6482-9_10

# 10. Expanding Editor Functionality

Andy Leonard[1]
(1) Farmville, VA, USA

The first edition of Building Custom Tasks for SQL Server Integration Services was published in 2017. Chapter 10 covered troubleshooting tips. Chapter 11 contained notes from my experience. This edition will include those chapters… only later.



editor with General and Settings pages" views

- Enable the use of SSIS Expressions
- Add design-time validation of task settings
- Surface more SSIS Catalog execution properties in the editor

We begin these efforts in this chapter, in which we refactor existing ExecuteCatalogPackageTask code and replace the existing task editor project with a new version.

## Refactoring

Developers are sometimes given the opportunity to rewrite or update older versions of their code. The longer said code has been in use, the more experience a developer has with the code. *Refactoring* is defined in a few ways, including cleaning, simplifying, and restructuring existing code.

In this chapter, we begin with some refactoring of the ExecuteCatalogPackageTask code, starting with managing the scope of variables.

## Adding and Updating Task Properties

One change will directly address one of the bullets above and move us closer to solving a few others: adding SSIS Catalog, Server, and other SSIS package related objects as private properties in the ExecuteCatalogPackageTask object.

Examining the ExecuteCatalogPackageTask class in the ExecuteCatalogPackageTask project, we find the *beginning* of an SSIS Catalog object in the PackageCatalog property as shown in Figure **10-1**:

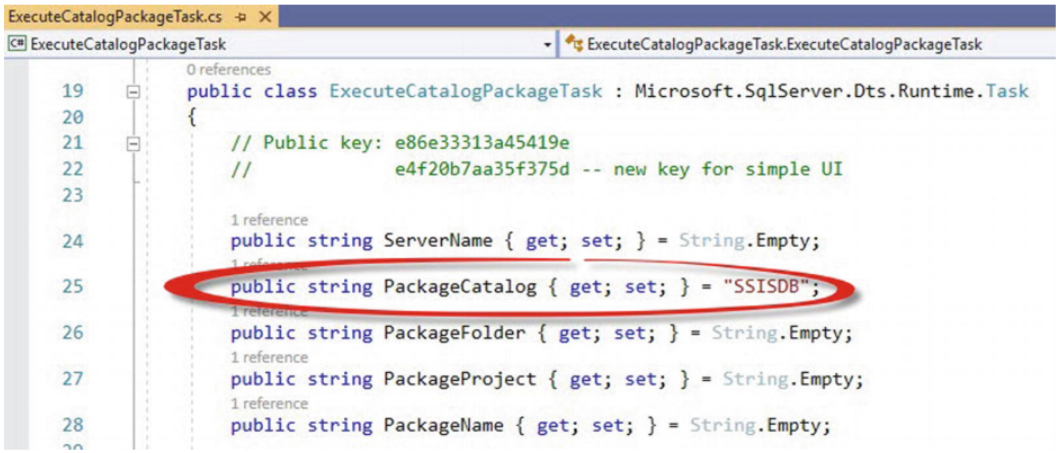



***Figure 10-1*** The PackageCatalog property

The PackageCatalog property is a string and contains the name of the SSIS Catalog. At the time of this writing, SSIS Catalog names have remained consistent

and unchanged, “SSISDB.” The property is included as an example of future-proofing the Execute Catalog Package Task in case future versions of the SSIS Catalog permit editing the SSIS Catalog name, but I do not like the name of the Execute Catalog Package Task’s PackageCatalog property.

Rename the Execute Catalog Package Task PackageCatalog property to “PackageCatalogName,” click the “Quick Actions” screwdriver icon in the left margin, and then click “Rename ‘PackageCatalog’ to ‘PackageCatalogName’,” as shown in Figure 10-2:

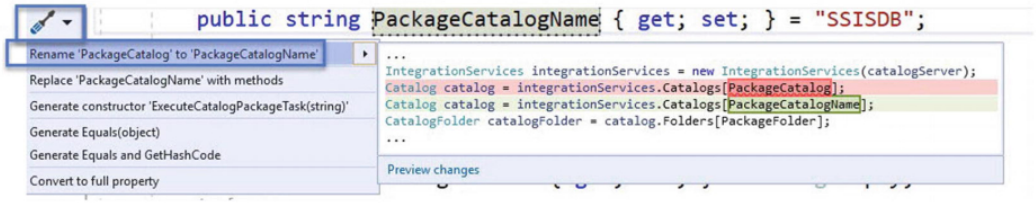



***Figure 10-2*** Renaming the PackageCatalog property to “PackageCatalogName”

PackageCatalogName is a more accurate name for the property formerly known as PackageCatalog.

The current version of the Execute Catalog Package Task includes a variable of type Microsoft.SqlServer.Management.IntegrationServices.Catalog in the Execute method, as shown in Figure 10-3:

```
public override DTSExecResult Execute(
    Connections connections,
    VariableDispenser variableDispenser,
    IDTSComponentEvents componentEvents,
    IDTSLogging log,
    object transaction)
{
    Server catalogServer = new Server(ServerName);
    IntegrationServices integrationServices = new IntegrationServices(catalogServer);
    Catalog catalog = integrationServices.Catalogs[PackageCatalog];
    CatalogFolder catalogFolder = catalog.Folders[PackageFolder];
    ProjectInfo catalogProject = catalogFolder.Projects[PackageProject];
    Microsoft.SqlServer.Management.IntegrationServices.PackageInfo catalogPackage = catalogProject.Packages[PackageName];

    catalogPackage.Execute(false, null);

    return DTSExecResult.Success;
}
```

***Figure 10-3*** A variable of type Microsoft.SqlServer.Management.IntegrationServices.Catalog

How difficult is it to cut this property from the `Execute` method and use a new private property for the task of the Microsoft.SqlServer.Management.IntegrationServices.Catalog type at the class scope? We may assign a value to the `catalog` property once we have enough information available during task configuration. After all, the Execute Catalog Package Task will connect to one and only one SSIS catalog.

Begin this process by adding a new property named `catalog` of the Microsoft.SqlServer.Management.IntegrationServices type to the ExecuteCatalogPackageTask properties list as shown in Listing **10-1** and Figure **10-4**:

```
public Catalog catalog { get; set; } = null;
```

***Listing 10-1*** Code to add the Catalog property

```
public class ExecuteCatalogPackageTask : Microsoft.SqlServer.Dts.Runtime.Task
{
    // Public key: e86e33313a45419e
    //             e4f20b7aa35f375d -- new key for simple UI

    3 references
    public string ServerName { get; set; } = String.Empty;
    1 reference
    public string PackageCatalogName { get; set; } = "SSISDB";
    0 references
    private Catalog catalog { get; set; } = null;
    1 reference
    public string PackageFolder { get; set; } = String.Empty;
    1 reference
    public string PackageProject { get; set; } = String.Empty;
    1 reference
    public string PackageName { get; set; } = String.Empty;
```

***Figure 10-4*** Declaring a Catalog property

**Note** The name of this property is not a capital letter. That's intentional. This property will be internal to the task. Also, the object type does not need to include

"`Microsoft.SqlServer.Management.IntegrationServices`" because the code includes *using* `Microsoft.SqlServer.Management.IntegrationServices` at the top of the class code.

Because the `catalog` property is internal to the ExecuteCatalogPackageTask class, it is declared as `private`.

While we're at it, let's migrate other variables from the `Execute` method to private properties, such as `catalogServer` (`Microsoft.SqlServer.Management.Smo.Server` type), `integrationServices` (`Microsoft.SqlServer.Management.IntegrationServices` type), `catalogFolder` (`Microsoft.SqlServer.Management.IntegrationServices.CatalogFolder` type), `catalogProject` (`Microsoft.SqlServer.Management.IntegrationServices.ProjectInfo` type), `catalogPackage` (`Microsoft.SqlServer.Management.IntegrationServices.PackageInfo` type), as shown in Listing **10-2** and Figure **10-5**:

```
private Server catalogServer { get; set; } = null;
private IntegrationServices integrationServices { get; set; } = null;
private CatalogFolder catalogFolder { get; set; } = null;
private ProjectInfo catalogProject { get; set; } = null;
private Microsoft.SqlServer.Management.IntegrationServices.PackageInfo c
```

***Listing 10-2*** Adding additional properties

Please note the full data type specification for the catalogPackage private property. The full name of the variable type must be declared because there is a different `PackageInfo` type in the Microsoft.SqlServer.Dts.Runtime assembly. Visual Studio complains if it is not specific because Visual Studio has no way to determine which `PackageInfo` type we wish to use.

```
public class ExecuteCatalogPackageTask : Microsoft.SqlServer.Dts.Runtime.Task
```

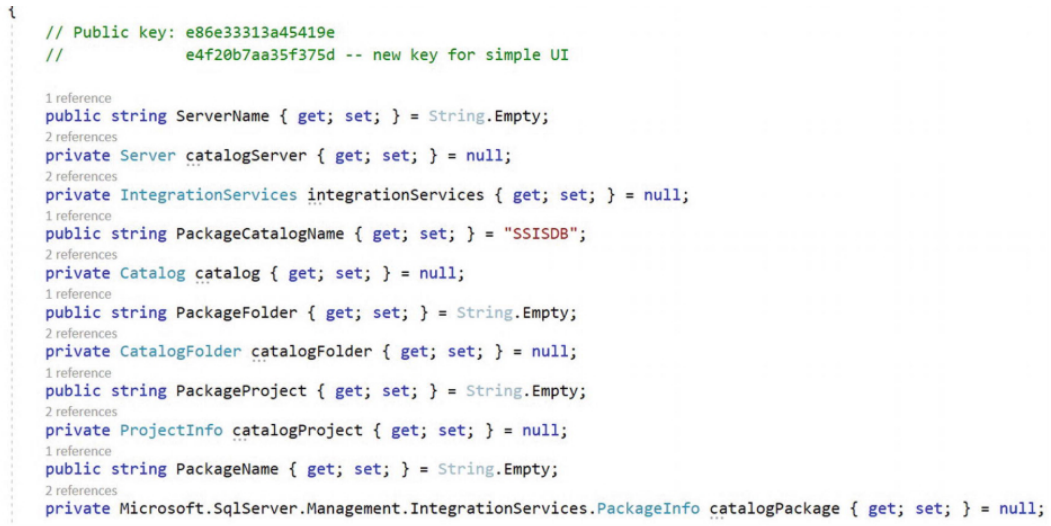



***Figure 10-5*** Additional properties added

Let's next update the `Execute` method to initialize the class-scoped variables we just added.

## Updating the Execute Method

To continue our refactoring effort, update the variables declared in the `Execute` method to use and initialize the new internal task properties.

The current state of the `Execute` method is listed in Listing **10-3** and shown in Figure **10-6**:

```
public override DTSExecResult Execute(
            Connections connections,
            VariableDispenser variableDispenser,
            IDTSComponentEvents componentEvents,
            IDTSLogging log,
            object transaction)
        {
            Server catalogServer = new Server(ServerName);
```

```
            IntegrationServices integrationServices = new IntegrationSer
            Catalog catalog = integrationServices.Catalogs[PackageCatalo
            CatalogFolder catalogFolder = catalog.Folders[PackageFolder]
            ProjectInfo catalogProject = catalogFolder.Projects[PackageP
            Microsoft.SqlServer.Management.IntegrationServices.PackageIn
            catalogPackage.Execute(false, null);
            return DTSExecResult.Success;
        }
```

***Listing 10-3*** The Execute method, currently

```
public override DTSExecResult Execute(
    Connections connections,
    VariableDispenser variableDispenser,
    IDTSComponentEvents componentEvents,
    IDTSLogging log,
    object transaction)
{
    Server catalogServer = new Server(ServerName);
    IntegrationServices integrationServices = new IntegrationServices(catalogServer);
    Catalog catalog = integrationServices.Catalogs[PackageCatalogName];
    CatalogFolder catalogFolder = catalog.Folders[PackageFolder];
    ProjectInfo catalogProject = catalogFolder.Projects[PackageProject];
    Microsoft.SqlServer.Management.IntegrationServices.PackageInfo catalogPackage = catalogProject.Packages[PackageName];

    catalogPackage.Execute(false, null);

    return DTSExecResult.Success;
}
```

***Figure 10-6*** The current version of the Execute method

We want to initialize the new class-scoped variables in the `Execute` method, so we need to remove the type declarations. Update the type declaration and initialization code to *remove* type declarations, as shown in Listing **10-4**:

```
catalogServer = new Server(ServerName);
integrationServices = new IntegrationServices(catalogServer);
catalog = integrationServices.Catalogs[PackageCatalogName];
catalogFolder = catalog.Folders[PackageFolder];
catalogProject = catalogFolder.Projects[PackageProject];
catalogPackage = catalogProject.Packages[PackageName];
```

***Listing 10-4*** Removing Execute method variable type declarations

At this point in our efforts to expand the functionality of the Execute Catalog Package Task, we may test by building the solution and testing the task (see Chapter 9, sections "Building the Task" and "Testing the Task"). This is a good test of our changes to the `Execution` method.

## Adding a New Editor

Two bullets from the introduction of this chapter are

- Surface a more "SSIS-y" experience for users of the task including a "pretty" editor with General and Settings pages "views."
- Enable the use of SSIS Expressions.

> Before moving forward in this section, the author owes a debt of gratitude to Kirk Haselden for his outstanding early SSIS book titled *Microsoft SQL Server 2005 Integration Services* (amazon.com/Microsoft-Server-2005-Integration-Services-dp-0672327813/dp/0672327813), Sams Publishing, 2006.
>
> I know lots of people who learned about custom SSIS tasks from Kirk's book.
>
> Thank you, Kirk!

### Complex UI Overview

This section of the book is difficult and advanced. At the heart of the complex UI is the Microsoft.DataTransformationServices.Controls assembly. Inheriting from the Microsoft.DataTransformationServices.Controls assembly is challenging and can be nonintuitive. That said, the functionality surfaced in the Microsoft.DataTransformationServices.Controls assembly is more than worth the time required to overcome any challenges and manage its nonintuitive-ness.

The Microsoft.DataTransformationServices.Controls assembly requires development of *views*, which SSIS developers encounter when we open a task editor. Each view represents a high-level collection of task properties. Views are listed on the left side of the editor and

outlined by the green box in Figure 10-7:

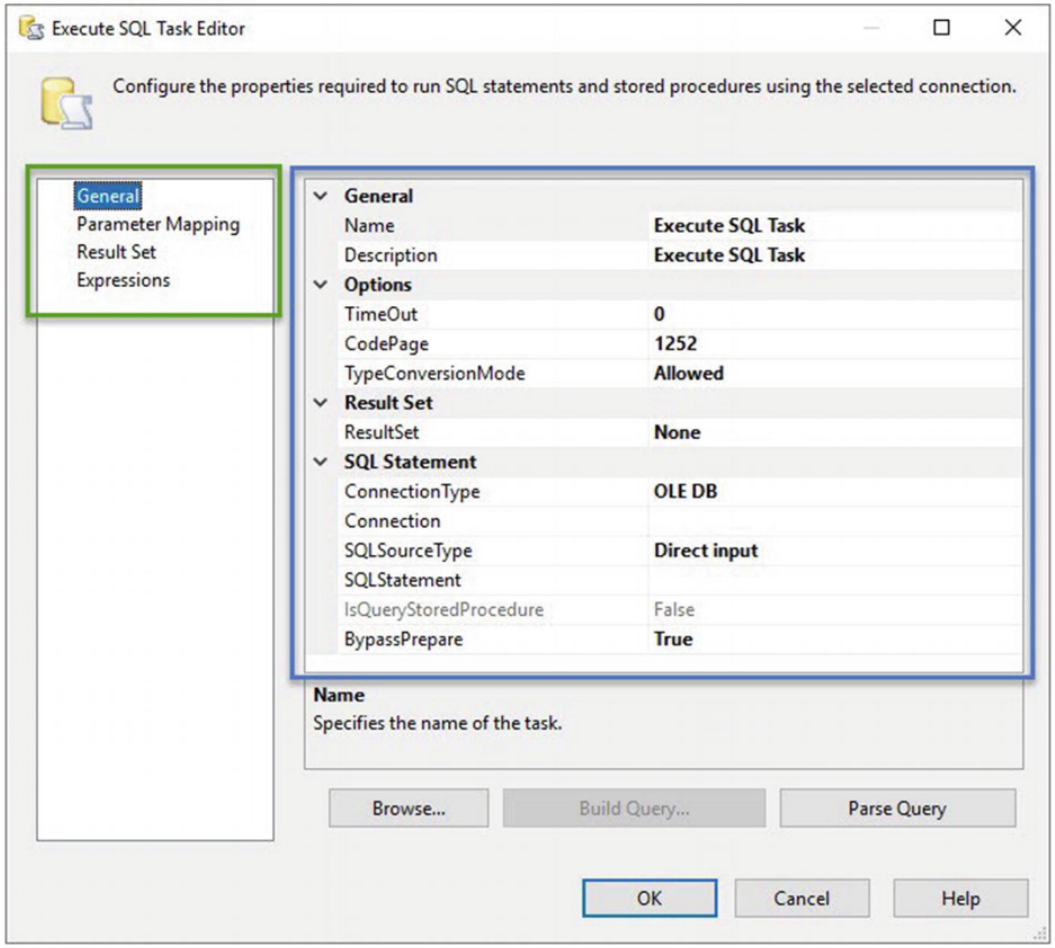



***Figure 10-7*** The Execute SQL Task's General page

Under the hood of each view is a *node*, defined as an internal C# class instantiated for each view. Each node surfaces a PropertyGrid control, which is shown in Figure 10-7 inside the blue box.

A node manages a collection of task properties organized by property category. Each task property is a C# property in the node class. An example we will build out later is the Name property in the General node of the General view. The code represents a View ➤

Node ➤ Property Category ➤ Property hierarchy. The code flows as shown in Listing 10-5:

```
namespace ExecuteCatalogPackageTaskComplexUI
{
  public partial class GeneralView
  {
    private GeneralNode generalNode = null;
  }
  internal class GeneralNode
  {
    [
      Category("General"),
      Description("Specifies the name of the task.")
    ]
    public string Name
    {
      get { return taskHost.Name; }
      set { taskHost.Name = value; }
    }
  }
}
```

***Listing 10-5*** Partial listing for General view, General node, Name property

The View ➤ Node ➤ Property Category ➤ Property hierarchy shown in the partial listing in Listing 10-5 begins with the view named `GeneralView`. `GeneralView` declares a private member of the `GeneralNode` type named `generalNode`. The GeneralNode class contains a string type property named `Name`. The Name property is decorated with `Category` and `Description` decorations, and the Category decoration contains the name for the property category ("General").

Figure 10-8 visualizes the hierarchy for the Execute SQL Task Editor. As mentioned earlier, the General view (in the green box on the left) displays the General node – represented by the propertygrid control (in the blue box on the right). The property category named "General" is shown in the red box. The Name property is shown enclosed in

the yellow box, as shown in Figure 10-8:

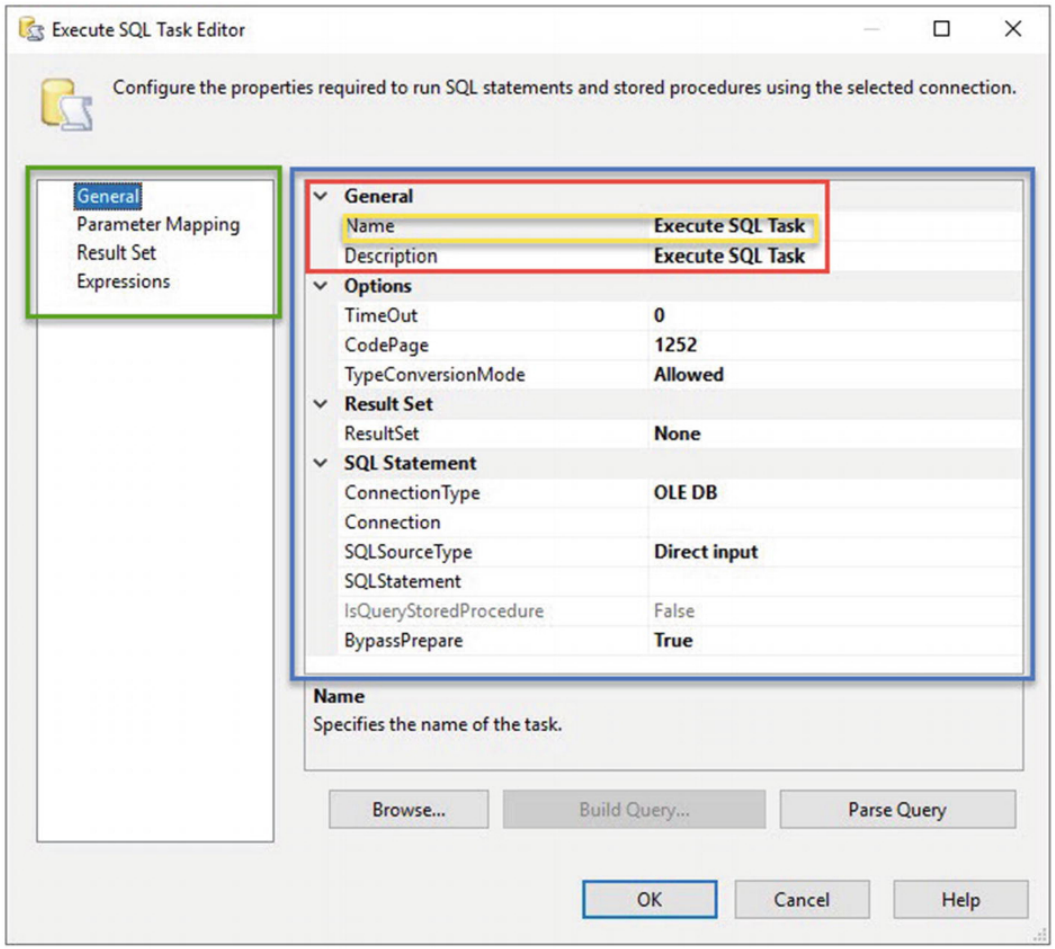



***Figure 10-8*** Visualizing View ➤ Node ➤ Property Category ➤ Property

Applied to the View ➤ Node ➤ Property Category ➤ Property hierarchy, the names of the entities for the Execute SQL Task Editor – shown in Figure 10-8 – are as follows: General (View) ➤ General (Node) ➤ General (Property Category) ➤ Name (Property). That's a lot of General's, and one – the General node – is hiding beneath the propertygrid control.

## Adding the Editor Project

To begin adding a new editor, add a new project to the ExecuteCatalogPackageTask Visual Studio solution by right-clicking the solution in Solution Explorer, hovering over "Add," and then clicking "New project…," as shown in Figure 10-9:

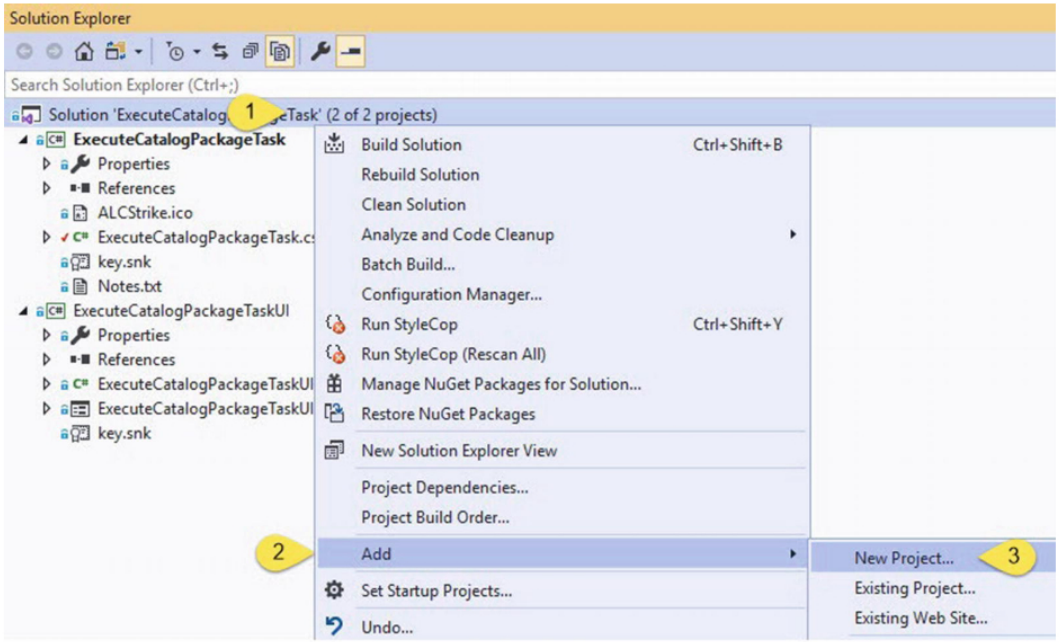
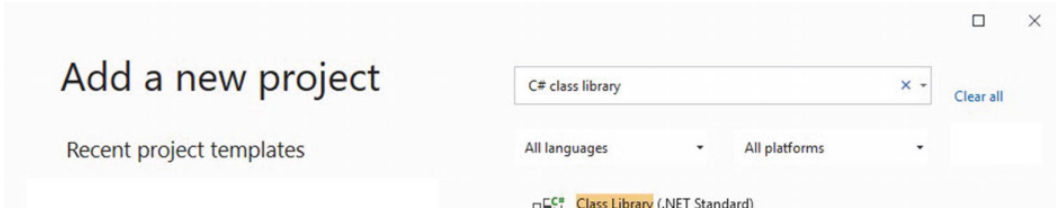



***Figure 10-9*** Adding a new project

When the "Create a new project" window displays, search for, and then select, a C# Class Library (.Net Framework) project type, or in the .Net language of your choosing. I chose C# for the language and named the project ExecuteCatalogPackageTask as shown in Figure 10-10:



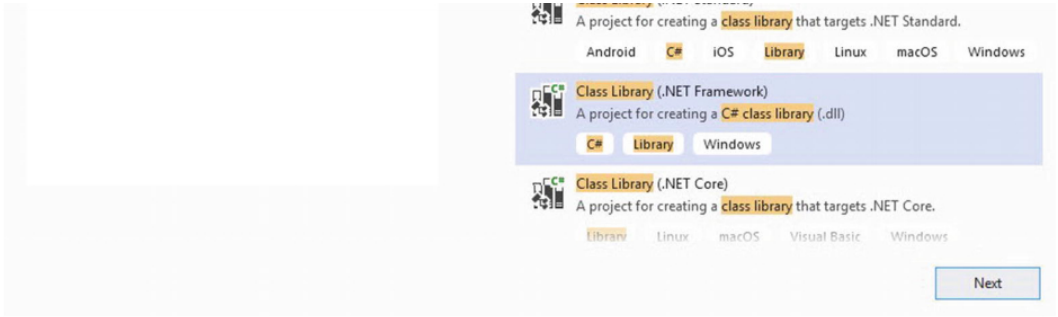



***Figure 10-10*** Selecting a project type

Select “Class Library (.NET Framework)” and click the Next button.

When the “Configure your new project” window displays, enter “ExecuteCatalogPackageTaskComplexUI ”as the name for the new project, and set the project files location as shown in Figure **10-11**:

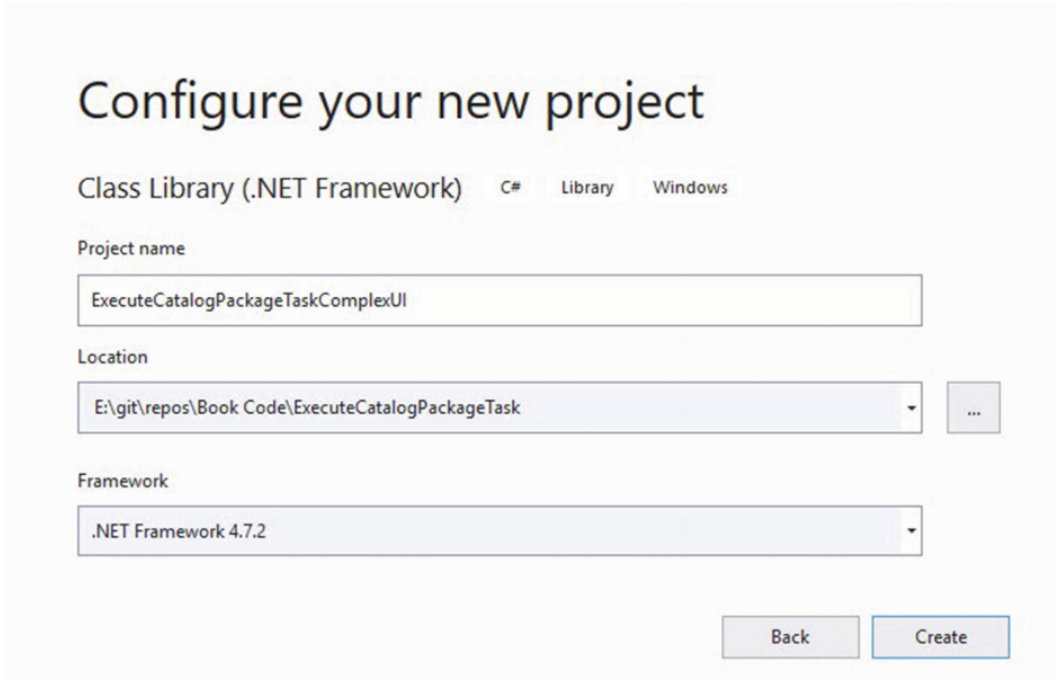
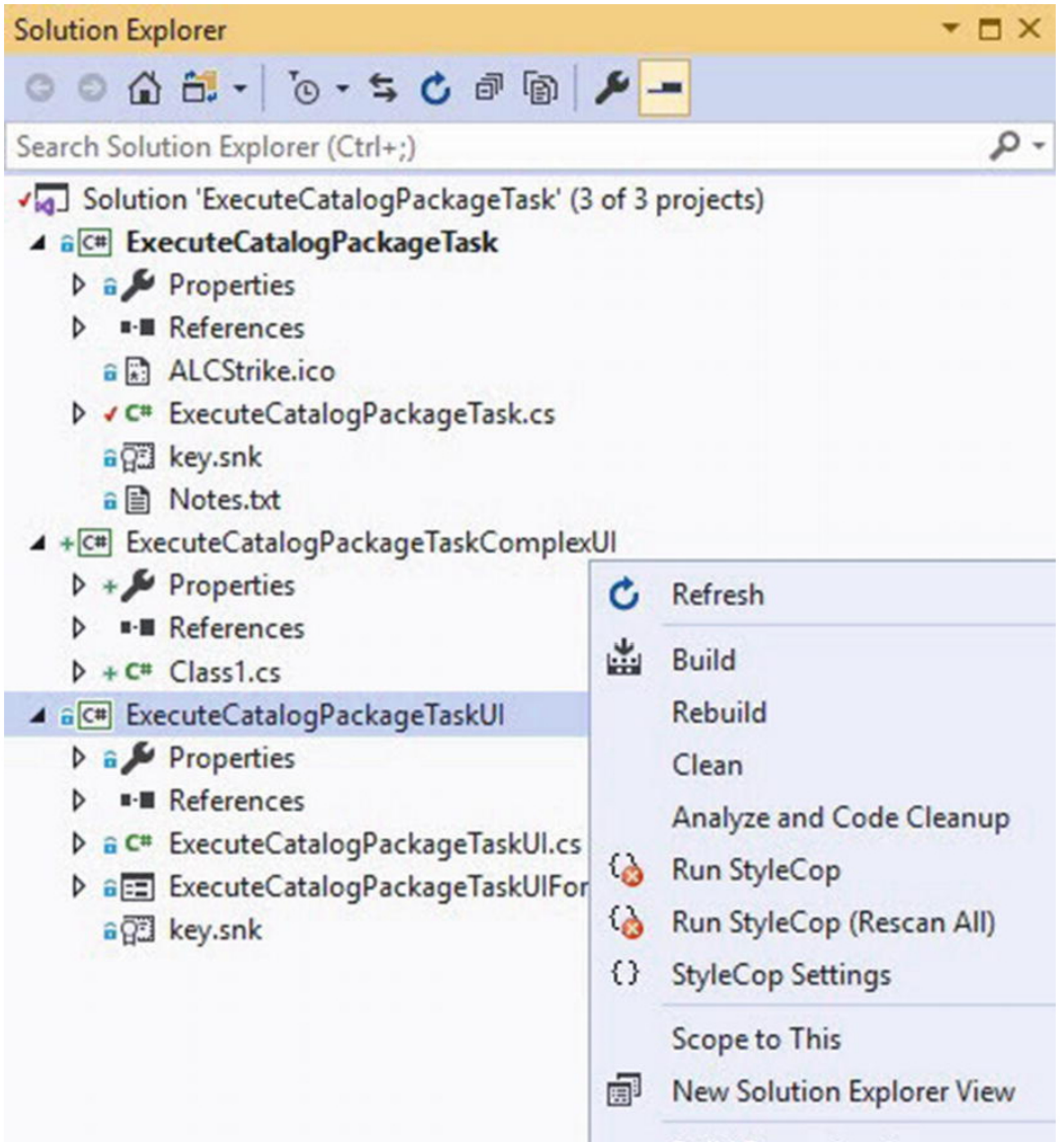



***Figure 10-11*** Configuring and naming the new project

Click the Create button to add the new ExecuteCatalogPackageTaskComplexUI project.

To remove the existing ExecuteCatalogPackageTaskUI project, right-click the ExecuteCatalogPackageTaskUI project in Solution Explorer, and then click “Remove,” as shown in Figure **10-12**:



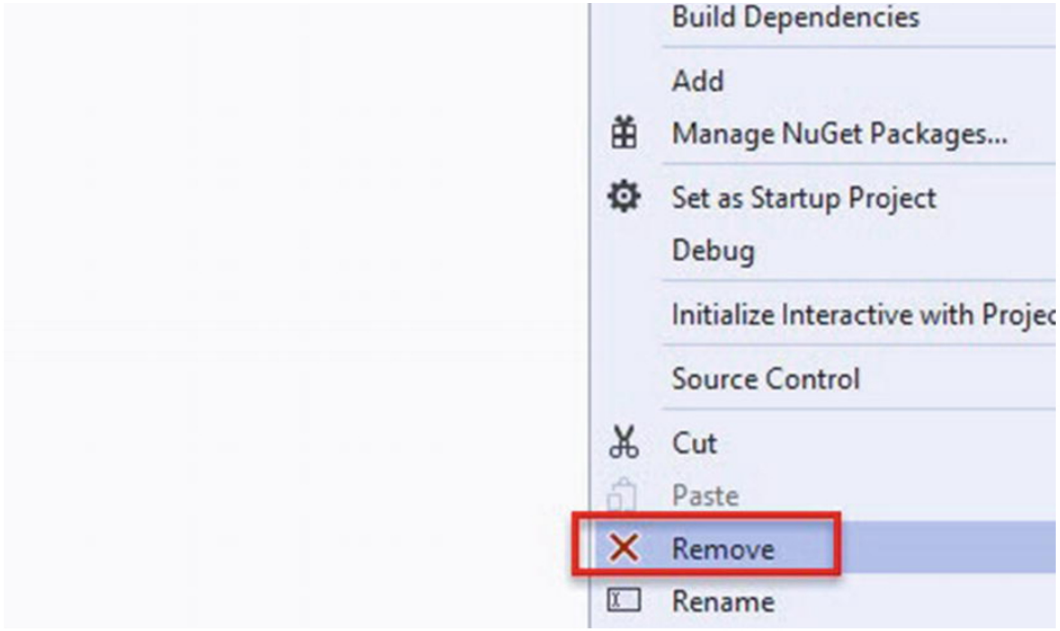



***Figure 10-12*** Removing the ExecuteCatalogPackageTaskUI project

In Solution Explorer, rename the ExecuteCatalogPackageTaskComplexUI project's Class1.cs file "ExecuteCatalogPackageTaskComplexUI," as shown in Figure **10-13**:

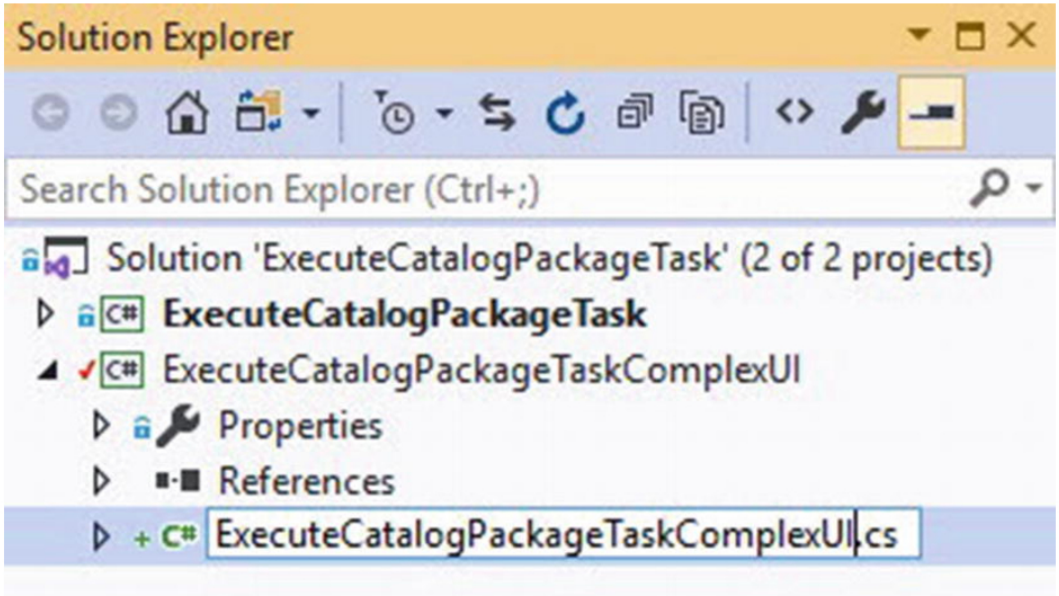


*Figure 10-13* Renaming Class1.cs to ExecuteCatalogPackageTaskComplexUI.cs

Visual Studio will prompt, asking if you would also like to rename all references to "Class1" in the project, as shown in Figure 10-14:

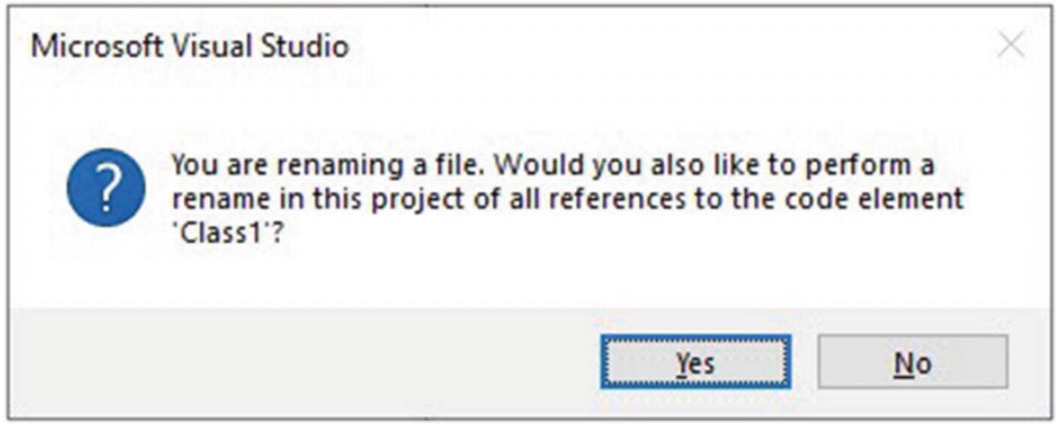



*Figure 10-14* Prompting to rename all Class1 references to ExecuteCatalogPackageTaskComplexUI

Click the Yes button. Note the "`public class Class1`" declaration in the ExecuteCatalogPackageTaskComplexUI.cs file now reads "`public class ExecuteCatalogPackageTaskComplexUI`," as shown in Figure 10-15:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ExecuteCatalogPackageTaskComplexUI
{
    0 references
    public class ExecuteCatalogPackageTaskComplexUI
    {
```

*Figure 10-15* Class1 is now ExecuteCatalogPackageTaskComplexUI

The ExecuteCatalogPackageTaskComplexUI class is now ready for references and additional coding.

## Adding References

To add references to the new ExecuteCatalogPackageTaskComplexUI project, right-click References and click Add reference, as shown in Figure **10-16**:

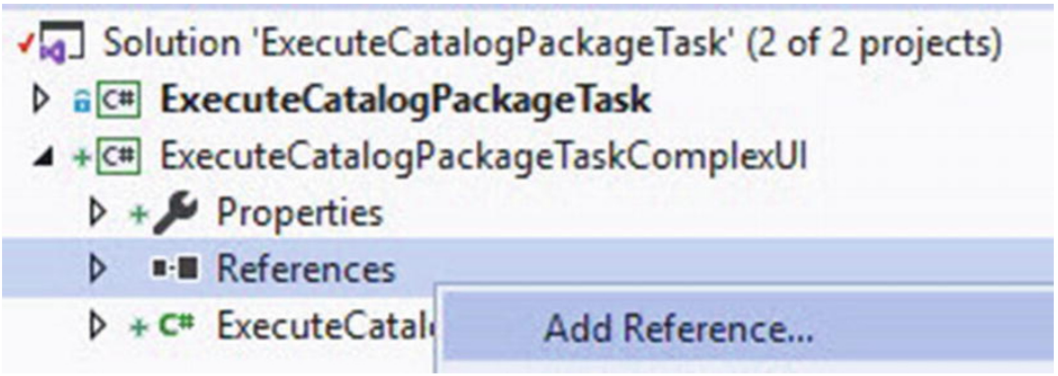
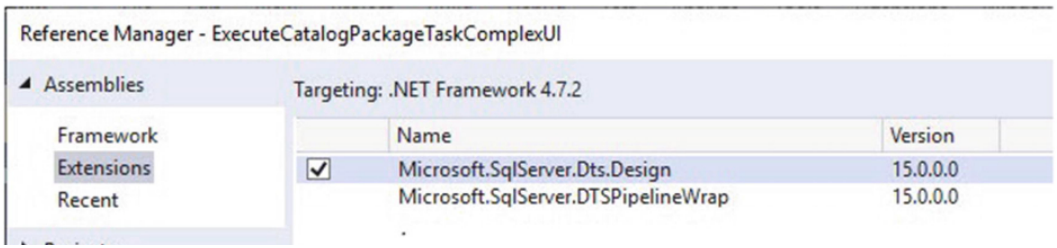



*Figure 10-16* Preparing to add references

When Reference Manager displays, navigate to Assemblies ➤ Extensions and check the checkboxes beside Microsoft.SqlServer.Dts.Design and Microsoft.SqlServer.ManagedDTS, as shown in Figure **10-17**:



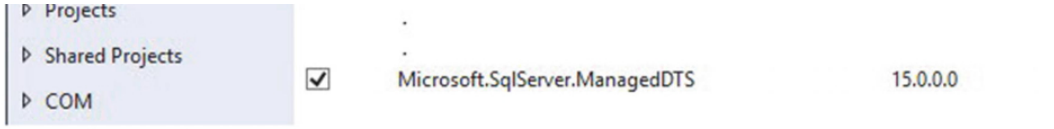



***Figure 10-17*** Adding Microsoft.SqlServer.Dts.Design and Microsoft.SqlServer.ManagedDTS references

Click Assemblies ➤ Framework and select Systems.Windows.Forms, as shown in Figure 10-18:

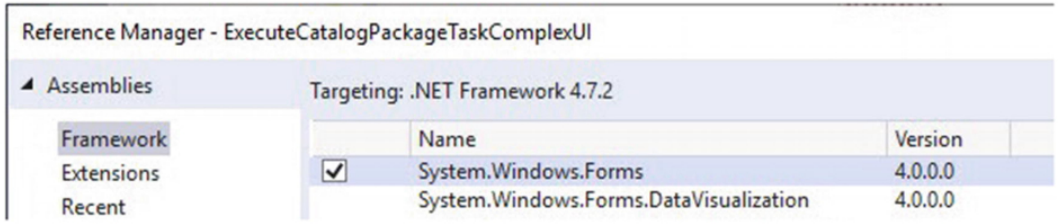



***Figure 10-18*** Referencing System.Windows.Forms

Click the Browse button and search for the latest version of the Microsoft.SqlServer.Management.IntegrationServices assembly folder in the <*installation drive*>:\Windows\Microsoft.NET\assembly\GAC_MSIL\ folder. On my development virtual machine, the full path is C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Microsoft.SqlServer.Management.IntegrationServices\v4.0_15.0.0

Click the Browse button again and search for the latest version of the Microsoft.SqlServer.Management.Sdk.Sfc assembly folder in the <*installation drive*>:\Windows\Microsoft.NET\assembly\GAC_MSIL\ folder. On my development virtual machine, the full path is C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Microsoft.SqlServer.Management.Sdk.Sfc\v4.0_15.0.0.0__89845dc

Click the Browse button again and search for the latest version of the Microsoft.DataTransformationServices.Controls assembly folder in the <*installation drive*>:\Windows\Microsoft.NET\assembly\GAC_MSIL\ folder. On my development virtual machine, the full path is

C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Microsoft.DataTransformationServices.Controls\v4.0_15.0.0.0__89

The Microsoft.DataTransformationServices.Controls assembly is the source of both the “SSIS-y” experience and the ability to use Expressions in the custom task.

Once the assemblies are referenced, The References virtual folder in the ExecuteCatalogPackageTaskComplexUI project should appear similar to Figure 10-19:

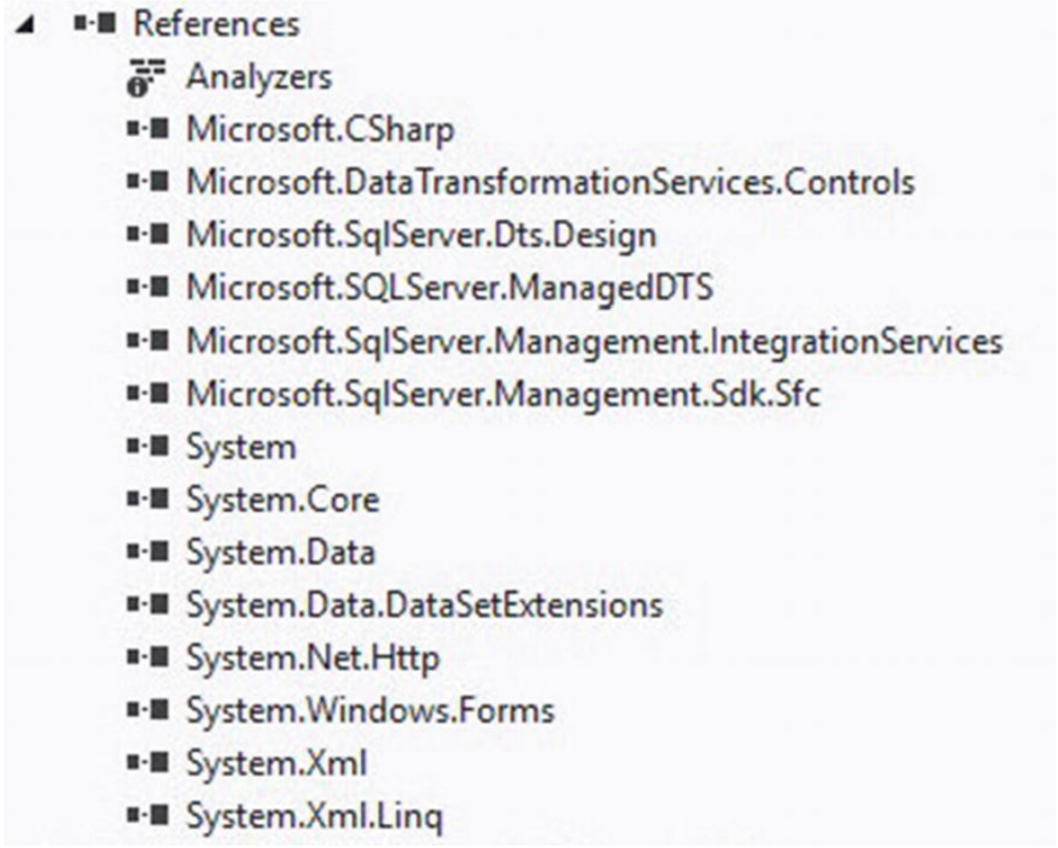



***Figure 10-19*** ExecuteCatalogPackageTaskComplexUI References

Now that the assemblies we need are referenced, let’s prepare to use them in .Net code.

### Using Referenced Assemblies

### Using Referenced Assemblies

To access the assemblies referenced in the previous section, add the `using` statements shown in Listing **10-6** near the existing using statements in the ExecuteCatalogPackageTaskComplexUI class:

```
using Microsoft.SqlServer.Dts.Runtime;
using Microsoft.SqlServer.Dts.Runtime.Design;
using System.Windows.Forms;
```

***Listing 10-6*** Use the referenced assemblies

Once added, the ExecuteCatalogPackageTaskComplexUI class should appear as shown in Figure **10-20**:

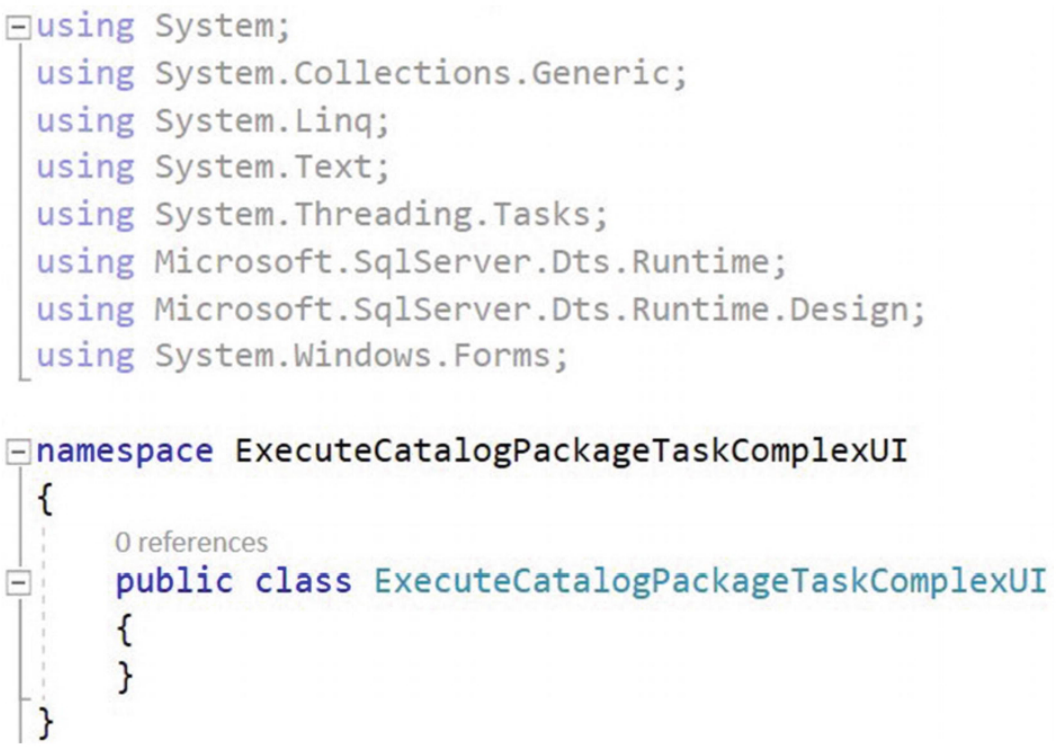



***Figure 10-20*** ExecuteCatalogPackageTaskComplexUI class after adding additional using statements

Next, inherit the IDtsTaskUI interface by editing the ExecuteCatalogPackageTaskComplexUI class declaration so it appears as shown in Listing 10-7 and Figure 10-21:

```
public class ExecuteCatalogPackageTaskComplexUI : IDtsTaskUI
```

***Listing 10-7*** ExecuteCatalogPackageTaskComplexUI inheriting IDtsTaskUI

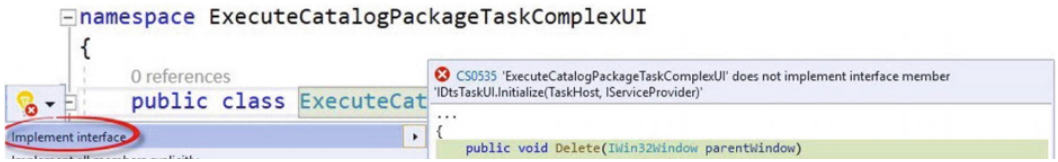
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.SqlServer.Dts.Runtime;
using Microsoft.SqlServer.Dts.Runtime.Design;
using System.Windows.Forms;

namespace ExecuteCatalogPackageTaskComplexUI
{
    0 references
    public class ExecuteCatalogPackageTaskComplexUI : IDtsTaskUI
    {
    }
}
```

***Figure 10-21*** ExecuteCatalogPackageTaskComplexUI inheriting IDtsTaskUI

When implemented, the class should appear as shown in Figure 10-21:

The red squiggly line beneath "IDtsTaskUI" in Figure 10-21 indicates the inherited interface is not correctly – or *completely*, in this case – implemented. Click the line and then expand Quick Actions to view the issues, as shown in Figure 10-22:



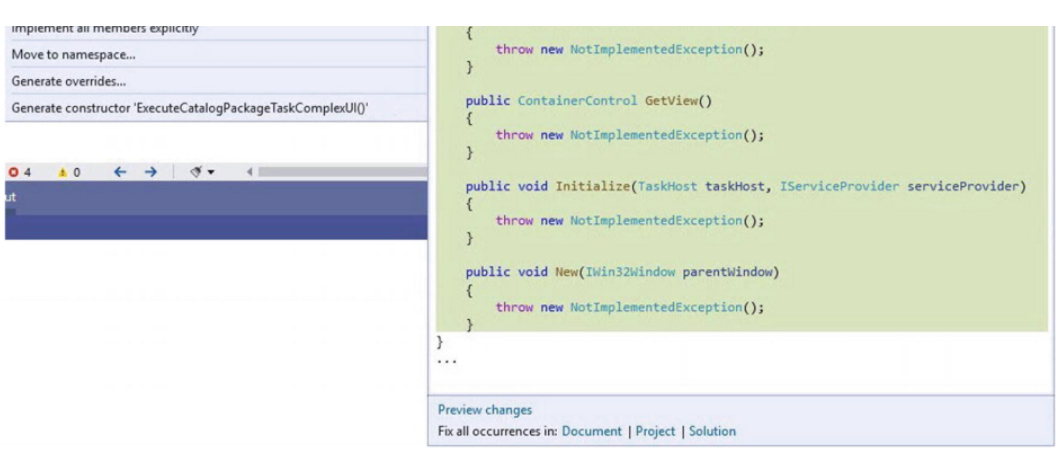



***Figure 10-22*** IDtsTaskUI implementation issues

Clicking the "Implement interface" option will implement a barebones version of the required interface members, as shown in Figure **10-23**:

```
public class ExecuteCatalogPackageTaskComplexUI : IDtsTaskUI
{
    0 references
    public void Delete(IWin32Window parentWindow)
    {
        throw new NotImplementedException();
    }

    0 references
    public ContainerControl GetView()
    {
        throw new NotImplementedException();
    }

    0 references
    public void Initialize(TaskHost taskHost, IServiceProvider serviceProvider)
    {
        throw new NotImplementedException();
    }

    0 references
    public void New(IWin32Window parentWindow)
    {
        throw new NotImplementedException();
    }
}
```

*Figure 10-23* The IDtsTaskUI interface with members implemented

Each member is implemented to throw an exception by default. Please note the red squiggly line is no longer present beneath IDtsTaskUI.

## Conclusion

In this chapter, we started the process of developing a new task editor. We started by refactoring existing ExecuteCatalogPackageTask code and replacing the existing task editor project with a new version.

Now would be an excellent time to check in this solution.