A. Leonard, *Building Custom Tasks for SQL Server Integration Services*
https://doi.org/10.1007/978-1-4842-6482-9_17

17. Refactoring the SSIS Package Hierarchy

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

The goal of this chapter is to refactor ExecuteCatalogPackageTask methods to prepare for adding and testing the Reference property. The goals are

- Refactor the `SettingsView` catalog, folder, project, and package properties.
- Test folder, project, and package property expressions.

Refactor the SettingsView Catalog, Folder, Project, and Package Properties

Are you tired of typing the folder name, project name, and package name each time we configure an instance of the Execute Catalog Package Task? I am. Let's add properties and functionality to allow us to *select* folders, projects, and packages.

In this section, we add `object` type members to manage three collections – folders, projects, and packages – to the SettingsNode class. To do so, we will follow this pattern for each collection:

- Declare SettingsNode object type members (properties) to contain the collection.
- Add a new property for each level in the Folder ➤ Project ➤ Package hierarchy.
- Add a hidden (non-browsable) property for each collection.
- Add a method to populate a list for each object collection.
- Add a method to update each object collection.
- Centralize the call to update all related collections.
- Add calls to the propertyGridSettings_PropertyValueChanged to populate the *next* property collections in the hierarchy when certain property hierarchy values change.
- Add calls to the propertyGridSettings_PropertyValueChanged to clear prop-

erties *lower* in the hierarchy when certain property hierarchy values change.

- Hide the original property.

We begin by declaring the three objects for _folders, _projects, and _packages in the `SettingsNode` class using the code in Listing 17-1:

```
private object _folders = null;
private object _projects = null;
private object _packages = null;
```

Listing 17-1 Adding _folders, _projects, and _packages collection objects to SettingsN

Once added, the new objects appear as shown in Figure 17-1:

```
internal class SettingsNode
{
    internal ExecuteCatalogPackageTask.ExecuteCatalogPackageTask _task = null;
    private TaskHost _taskHost = null;
    private object _connections = null;

    private object _folders = null;
    private object _projects = null;
    private object _packages = null;

    private const string NEW_CONNECTION = "<New Connection...>";
```

Figure 17-1 Adding _folders, _projects, and _packages collection objects

The next step is to add hidden (non-browsable) properties for each collection.

The Folder Property and Folders Collection

The relationship between the Folder property and the Folders collection is similar to the relationship between the SourceConnection property and the connections collection. One difference is the connections collection is passed to the task from the SSIS package hosting the Execute Catalog Package Task. Our code sets the value of a `Connections` property in the `ExecuteCatalogPackageTask.InitializeTask` method (see Figures 16-10 and 16-11 in Chapter 16). The SSIS package is unaware of a collection of SSIS Catalog Folders. We must, therefore, manage the collection of SSIS Catalog Folders in our code.

Add a new Folder property to the `SettingsNode` class using the code in Listing 17-2:

```
[
  Category("SSIS Catalog Package Properties"),
```

```
    Description("Select SSIS Catalog Package folder name."),
    TypeConverter(typeof(Folders))
  ]
  public string Folder {
    get { return _task.PackageFolder; }
    set {
      if (value == null)
      {
        throw new ApplicationException("Folder name cannot be empty");
      }
      _task.PackageFolder = value;
    }
  }
```

Listing 17-2 Add the SettingsNode.Folder property

Once added, the code appears as shown in Figure 17-2:

```
[
    Category("SSIS Catalog Package Properties"),
    Description("Select SSIS Catalog Package folder name."),
    TypeConverter(typeof(Folders))
]
0 references | Andy Leonard, 4 days ago | 1 author, 3 changes
public string Folder {
    get { return _task.PackageFolder; }
    set {
        if (value == null)
        {
            throw new ApplicationException("Folder name cannot be empty");
        }

        _task.PackageFolder = value;
    }
}
```

Figure 17-2 Adding the SettingsNode.Folder property

Note the Folders TypeConverter has a red squiggly line beneath it, indicating an error condition. In this case, the Folders TypeConverter does not yet exist. Add the Folders TypeConverter – modeled after the LoggingLevels TypeConverter – using the code in Listing 17-3:

```
internal class Folders : StringConverter
{
  private object GetSpecializedObject(object contextInstance)
  {
    DTSLocalizableTypeDescriptor typeDescr = contextInstance as DTSLocal
    if (typeDescr == null)
    {
      return contextInstance;
```

```
    }
    return typeDescr.SelectedObject;
  }




  public override StandardValuesCollection GetStandardValues(ITypeDescri
  {
    object retrievalObject = GetSpecializedObject(context.Instance) as o
    return new StandardValuesCollection(getFolders(retrievalObject));
  }
  public override bool GetStandardValuesExclusive(ITypeDescriptorContext
  {
    return true;
  }
  public override bool GetStandardValuesSupported(ITypeDescriptorContext
  {
    return true;
  }
  private ArrayList getFolders(object retrievalObject)
  {
    SettingsNode node = (SettingsNode)retrievalObject;
    ArrayList list = new ArrayList();
    ArrayList listFolders;
    listFolders = (ArrayList)node.Folders;
    if (listFolders == null)
    {
      listFolders = new ArrayList();
    }
    if (listFolders != null)
    {
      // adds each folder
      foreach (string fld in listFolders)
      {
        list.Add(fld); // Folder name
      }
      // sorts the folder list
      if ((list != null) && (list.Count > 0))
      {
        list.Sort();
      }
    }
    return list;
  }
}
```

Listing 17-3 The Folders TypeConverter

Once added, the Folders TypeConverter code appears as shown in Figure 17-3:

```
internal class Folders : StringConverter
{
    1 reference | Andy Leonard, 20 hours ago | 1 author, 3 changes
    private object GetSpecializedObject(object contextInstance)
    {
        DTSLocalizableTypeDescriptor typeDescr = contextInstance as DTSLocalizableTypeDescriptor;

        if (typeDescr == null)
        {
            return contextInstance;
        }

        return typeDescr.SelectedObject;
    }

    3 references | Andy Leonard, 20 hours ago | 1 author, 3 changes
    public override StandardValuesCollection GetStandardValues(ITypeDescriptorContext context)
    {
        object retrievalObject = GetSpecializedObject(context.Instance) as object;

        return new StandardValuesCollection(getFolders(retrievalObject));
    }

    3 references | Andy Leonard, 20 hours ago | 1 author, 3 changes
    public override bool GetStandardValuesExclusive(ITypeDescriptorContext context)
    {
        return true;
    }

    3 references | Andy Leonard, 20 hours ago | 1 author, 3 changes
    public override bool GetStandardValuesSupported(ITypeDescriptorContext context)
    {
        return true;
    }

    1 reference | Andy Leonard, 20 hours ago | 1 author, 3 changes
    private ArrayList getFolders(object retrievalObject)
    {
        SettingsNode node = (SettingsNode)retrievalObject;
        ArrayList list = new ArrayList();
        ArrayList listFolders;

        listFolders = (ArrayList)node.Folders;
        if (listFolders == null)
        {
            listFolders = new ArrayList();
        }

        if (listFolders != null)
        {
            // adds each folder
            foreach (string fld in listFolders)
            {
                list.Add(fld); // Folder name
            }

            // sorts the folder list
            if ((list != null) && (list.Count > 0))
            {
                list.Sort();
            }
        }

        return list;
    }
}
```

Figure 17-3 The Folders TypeConverter

Note the getFolders method displays an error – a red squiggly line under `node.Folders`. We correct that error in the coming pages by adding the Folders property to the `SettingsNode` class.

When the Folders TypeConverter is added, the error in the Folder property declaration is resolved, as shown in Figure 17-4 (compare to Figure 17-2):

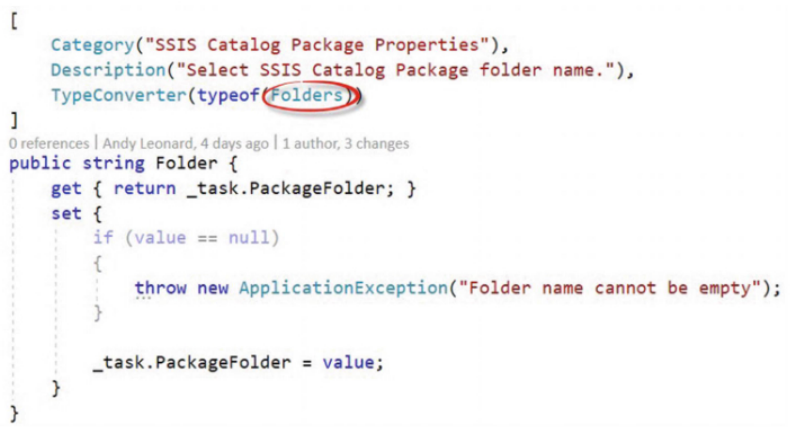


Figure 17-4 Folders TypeConverter error cleared

Add the Folders property using the code in Listing 17-4:

```
[
  Category("SSIS Catalog Package Path Collections"),
  Description("Enter SSIS Catalog Package folders collection."),
  Browsable(false)
]
public object Folders {
  get { return _folders; }
  set { _folders = value; }
}
```

Listing 17-4 Adding the Folders property

Once added, the Folders property appears as shown in Figure 17-5:

```
        _task.PackageFolder = value;
    }
}

[
    Category("SSIS Catalog Package Path Collections"),
    Description("Enter SSIS Catalog Package folders collection."),
    Browsable(false)
]
1 reference
internal object Folders {
    get { return _folders; }
    set { _folders = value; }
}
```

Figure 17-5 Folders property added

The next step is to add a method to populate a list of folders using the code in Listing 17-5:

```
internal ArrayList GetFoldersFromCatalog()
{
  ArrayList foldersList = new ArrayList();
  if ((_task.ServerName != null) && (_task.ServerName != ""))
  {
    Catalog catalog = _task.returnCatalog(_task.ServerName);
    if (catalog != null)
    {
      foreach (CatalogFolder cf in catalog.Folders)
      {
        foldersList.Add(cf.Name);
      }
    }
  }
  return foldersList;
}
```

Listing 17-5 Populate the Folders collection

Once added, the code appears as shown in Figure 17-6:

```
internal ArrayList GetFoldersFromCatalog()
{
    ArrayList foldersList = new ArrayList();

    if ((_task.ServerName != null) && (_task.ServerName != ""))
    {
        Catalog catalog = _task.returnCatalog(_task.ServerName);
```

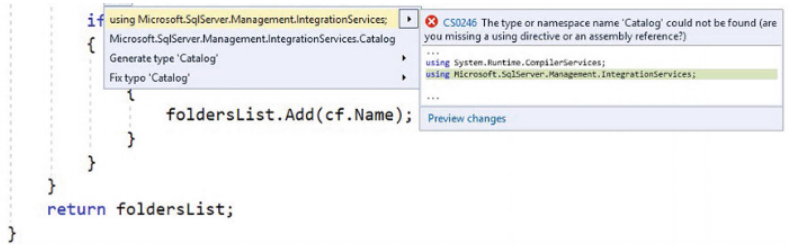


Figure 17-6 Adding the GetFoldersFromCatalog method

The `Catalog` (and hidden `CatalogFolder`) objects are underlined with red squiggly lines. Click the Quick Actions dropdown and then click `using Microsoft.SqlServer.Management.IntegrationServices;` to add the directive. Once the directive is added, the code appears as shown in Figure 17-7:

```
internal ArrayList GetFoldersFromCatalog()
{
    ArrayList foldersList = new ArrayList();

    if ((_task.ServerName != null) && (_task.ServerName != ""))
    {
        Catalog catalog = _task.returnCatalog(_task.ServerName);

        if (catalog != null)
        {
            foreach (CatalogFolder cf in catalog.Folders)
            {
                foldersList.Add(cf.Name);
            }
        }
    }
    return foldersList;
}
```

Figure 17-7 The GetFoldersFromCatalog method after adding the using directive

The `GetFoldersFromCatalog` method constructs and populates an `ArrayList` type variable that contains a list of SSIS Catalog Folders hosted in the SSIS Catalog surfaced by the `SourceConnection` property.

The next step is to add a method to populate the Folders collection property with the contents returned from the `GetFoldersFromCatalog` method, and then refresh the `SettingsView` propertygrid. The code for the new method named `updateFolders` is in Listing 17-6:

```
private void updateFolders()
{
```

```
    if (settingsNode.SourceConnection != "")
    {
      settingsNode.Folders = settingsNode.GetFoldersFromCatalog();
      this.settingsPropertyGrid.Refresh();
    }
  }
```

Listing 17-6 Adding the updateFolders method

Once added, the code appears as shown in Figure 17-8:

```
private void updateFolders()
{
    if (settingsNode.SourceConnection != "")
    {
        settingsNode.Folders = settingsNode.GetFoldersFromCatalog();
        this.settingsPropertyGrid.Refresh();
    }
}
```

Figure 17-8 The updateFolders method, added

The next step is to begin centralizing the call to update all related collections using the code in Listing 17-7:

```
  private void updateCollections()
  {
    Cursor = Cursors.WaitCursor;
    updateFolders();



    Cursor = Cursors.Default;
  }
```

Listing 17-7 Centralize the call to update the folders collection

Once added, the code appears as shown in Figure 17-9:

```
private void updateCollections()
{
    Cursor = Cursors.WaitCursor;
```

```
        updateFolders();

        Cursor = Cursors.Default;
    }
```

Figure 17-9 Centralizing the call to update the folders collection

We start the `updateCollections` method by setting the cursor to a wait cursor because later, when we connect to Azure-SSIS catalogs, populating the folders collection will take a few seconds. After calling the `updateFolders` method, the cursor is reset to the default cursor.

Updates to the `Folders` collection make sense when the `SourceConnection` property is initially selected or updated. Add a call to the `updateCollections` method at the bottom of the `SourceConnection` property section of the `SettingsView.propertyGridSettings_PropertyValueChanged` method, as shown in Figure 17-10:

```
            if (e.ChangedItem.PropertyDescriptor.Name.CompareTo("Folder") == 0)
            {
                updateCollections();
            }
        }

        else // if not a new connection
        {
            theTask.ServerName = returnSelectedConnectionManagerDataSourceValue(settingsNode.SourceConnection);
            theTask.ConnectionManagerName = settingsNode.SourceConnection;
            theTask.ConnectionManagerId = theTask.GetConnectionID(theTask.Connections, theTask.ConnectionManagerName);
            settingsNode.Connections = ConnectionService.GetConnectionsOfType("ADO.Net");
        }
        updateCollections();
    }
}
```

Figure 17-10 Adding a call to updateCollections to SourceConnection section of the SettingsView.propertyGridSettings_PropertyValueChanged

The next step is to add a call to the updateCollections method in the `propertyGridSettings_PropertyValueChanged` method, a call that populates the folders property collection when the `Folder` property value changes, by adding the code in Listing 17-8 to the `SettingsView.propertyGridSettings_PropertyValueChanged` method:

```
if (e.ChangedItem.PropertyDescriptor.Name.CompareTo("Folder") == 0)
{
  updateCollections();
```

```
}
```

Listing 17-8 Add code to respond to changes in the Folder property

Once added, the code appears as shown in Figure 17-11:

```
if (e.ChangedItem.PropertyDescriptor.Name.CompareTo("Folder") == 0)
{
    updateCollections();
}
}
```

Figure 17-11 Adding code to respond to changes in the Folder property

Add a call to the `updateCollections` method at the bottom of the `SettingsView.OnInitialize` method, as shown in Figure 17-12:

```
    settingsPropertyGrid.SelectedObject = this.settingsNode;

    ConnectionService = (IDtsConnectionService)connections;

    updateCollections();
}
```

Figure 17-12 Adding a call to updateCollections to SettingsView.OnInitialize

To test the functionality, build the ExecuteCatalogPackageTask solution, open a test SSIS package in a test SSIS project, add an Execute Catalog Package Task to the control flow canvas, open the editor, and configure a SourceConnection. After configuring a SourceConnection, the Folder property should be populated with the names of SSIS Catalog Folders, as shown in Figure 17-13:

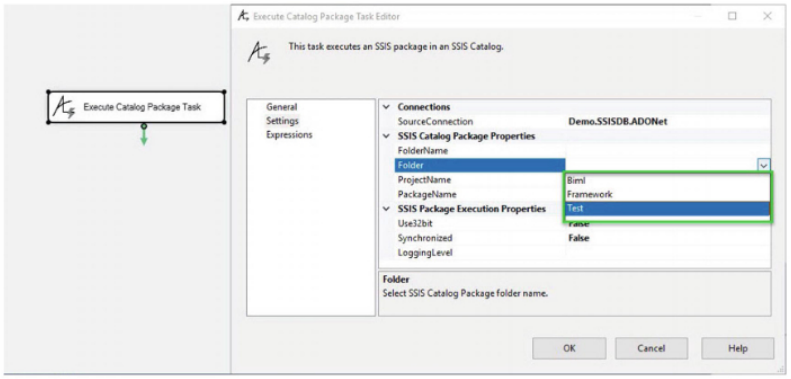


Figure 17-13 Folders

The next step is to hide the FolderName property. To hide the FolderName property,

simply comment out the FolderName property declaration by selecting the code, holding the Ctrl key, and pressing the K key, followed by the C key. The selected code is commented out as shown in Figure 17-14:

```
//[
//    Category("SSIS Catalog Package Properties"),
//    Description("Enter SSIS Catalog Package folder name.")
//]
//public string FolderName {
//    get { return _task.PackageFolder; }
//    set {
//        if ((value == null) || (value.Trim().Length == 0))
//        {
//            throw new ApplicationException("Folder name cannot be empty");
//        }

//        _task.PackageFolder = value;
//    }
//}
```

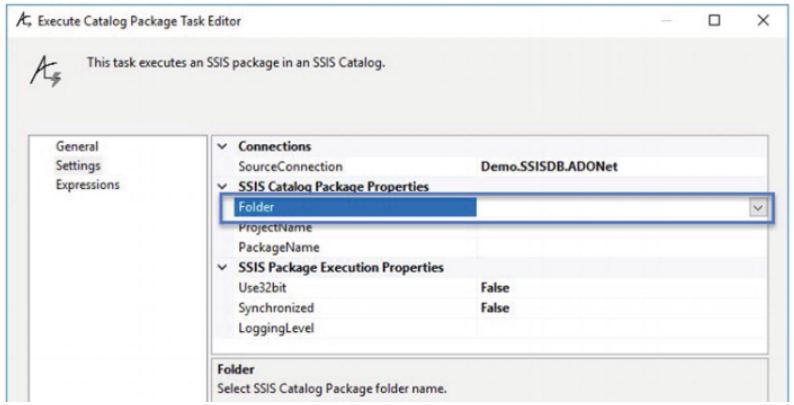
Figure 17-14 The FolderName property, commented out

The next step is to update the `SettingsView.OnCommit` method, updating the line `theTask.PackageFolder = settingsNode.FolderName;` with `theTask.PackageFolder = settingsNode.Folder;`, as shown in Figure 17-15:

```
public virtual void OnCommit(object taskHost)
{
    theTask.ConnectionManagerName = settingsNode.SourceConnection;
    theTask.ServerName = returnSelectedConnectionManagerDataSourceValue(settingsNode.SourceConnection);
    theTask.PackageFolder = settingsNode.Folder;
    theTask.PackageProject = settingsNode.ProjectName;
    theTask.PackageName = settingsNode.PackageName;
}
```

Figure 17-15 Updating the PackageFolder property assignment

Build and test the ExecuteCatalogPackageTask solution. The Settings view on the task editor should now appear as shown in Figure 17-16:



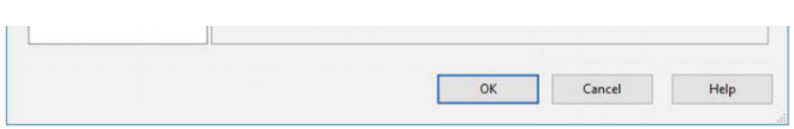


Figure 17-16 The Folder property

The Folder and Folders collection properties are now configured. The next steps are to configure the Project and Projects collection properties and the Package and Packages collection properties.

Now would be an excellent time to check in your code.

The Project Property and Projects Collection

The relationship between the Project property and the Projects collection is identical to the relationship between the Folder property and the Folders collection. The only difference is the objects themselves. As with the Folders collection, we must manage the collection of SSIS Catalog Projects in our code.

Add a new Project property to the `SettingsNode` class using the code in Listing 17-9:

```
[
  Category("SSIS Catalog Package Properties"),
  Description("Select SSIS Catalog Package project name."),
  TypeConverter(typeof(Projects))
]
public string Project {
  get { return _task.PackageProject; }
  set {
    if (value == null)
    {
      throw new ApplicationException("Project name cannot be empty");
    }



    _task. PackageProject = value;
  }
}
```

Listing 17-9 Add the SettingsNode.Project property

Once added, the code appears as shown in Figure 17-17:

```
    Category("SSIS Catalog Package Properties"),
    Description("Select SSIS Catalog Package project name."),
    TypeConverter(typeof(Projects))
]
0 references | Andy Leonard, 5 days ago | 1 author, 3 changes
public string Project {
    get { return _task.PackageProject; }
    set {
        if (value == null)
        {
            throw new ApplicationException("Project name cannot be empty");
        }

        _task.PackageProject = value;
    }
}
```

Figure 17-17 Adding the SettingsNode.Project property

As before with the Folder TypeConverter, please note the Projects TypeConverter has a red squiggly line beneath it, indicating the same error condition as before: The Projects TypeConverter does not yet exist. Add the Projects TypeConverter using the code in Listing 17-10:

```
internal class Projects : StringConverter
{
  private object GetSpecializedObject(object contextInstance)
  {
    DTSLocalizableTypeDescriptor typeDescr = contextInstance as DTSLocal
    if (typeDescr == null)
    {
      return contextInstance;
    }
    return typeDescr.SelectedObject;
  }
  public override StandardValuesCollection GetStandardValues(ITypeDescri
  {
    object retrievalObject = GetSpecializedObject(context.Instance) as o
    return new StandardValuesCollection(getProjects(retrievalObject));
  }
  public override bool GetStandardValuesExclusive(ITypeDescriptorContext
  {
    return true;
  }
  public override bool GetStandardValuesSupported(ITypeDescriptorContext
  {
    return true;
  }
  private ArrayList getProjects(object retrievalObject)
  {
    SettingsNode node = (SettingsNode)retrievalObject;
```

```
      ArrayList list = new ArrayList();
      ArrayList listProjects;
      listProjects = (ArrayList)node.Projects;
      if (listProjects == null)
      {
        listProjects = new ArrayList();
      }
      if (listProjects!= null)
      {
        // adds each project
        foreach (string fld in listProjects)
        {
          list.Add(fld); // Project name
        }
        // sorts the project list
        if ((list != null) && (list.Count > 0))
        {
          list.Sort();
        }
      }
      return list;
    }
  }
```

Listing 17-10 The Projects TypeConverter

Once added, the Projects TypeConverter code appears as shown in Figure 17-18:

```
internal class Projects : StringConverter
{
    1 reference | Andy Leonard, 5 days ago | 1 author, 2 changes
    private object GetSpecializedObject(object contextInstance)
    {
        DTSLocalizableTypeDescriptor typeDescr = contextInstance as DTSLocalizableTypeDescriptor;

        if (typeDescr == null)
        {
            return contextInstance;
        }

        return typeDescr.SelectedObject;
    }

    3 references | Andy Leonard, 5 days ago | 1 author, 2 changes
    public override StandardValuesCollection GetStandardValues(ITypeDescriptorContext context)
    {
        object retrievalObject = GetSpecializedObject(context.Instance) as object;

        return new StandardValuesCollection(getProjects(retrievalObject));
    }

    3 references | Andy Leonard, 5 days ago | 1 author, 2 changes
    public override bool GetStandardValuesExclusive(ITypeDescriptorContext context)
    {
        return true;
    }

    3 references | Andy Leonard, 5 days ago | 1 author, 2 changes
    public override bool GetStandardValuesSupported(ITypeDescriptorContext context)
    {
```

```
        return true;
    }

    1 reference | Andy Leonard, 5 days ago | 1 author, 2 changes
    private ArrayList getProjects(object retrievalObject)
    {
        SettingsNode node = (SettingsNode)retrievalObject;
        ArrayList list = new ArrayList();
        ArrayList listProjects;

        listProjects = (ArrayList)node.Projects;
        if (listProjects == null)
        {
            listProjects = new ArrayList();
        }

        if (listProjects != null)
        {
            // adds each project
            foreach (string fld in listProjects)
            {
                list.Add(fld); // Project name
            }

            // sorts the project list
            if ((list != null) && (list.Count > 0))
            {
                list.Sort();
            }
        }

        return list;
    }
```

Figure 17-18 The Projects TypeConverter

Note the getProjects method displays an error – a red squiggly line under `node.Projects`. We correct that error in the coming pages by adding the Projects property to the `SettingsNode` class.

When the Projects TypeConverter is added, the error in the Project property declaration is resolved, as shown in Figure 17-19 (compare to Figure 17-17):

```
[
    Category("SSIS Catalog Package Properties"),
    Description("Select SSIS Catalog Package project name."),
    TypeConverter(typeof(Projects))
]
0 references | Andy Leonard, 5 days ago | 1 author, 3 changes
public string Project {
    get { return _task.PackageProject; }
    set {
        if (value == null)
        {
            throw new ApplicationException("Project name cannot be empty");
        }

        _task.PackageProject = value;
    }
}
```

Figure 17-19 Projects TypeConverter error cleared

Add the Projects property using the code in Listing 17-11:

```
[
  Category("SSIS Catalog Package Path Collections"),
  Description("Enter SSIS Catalog Package projects collection."),
  Browsable(false)
]
public object Projects {
  get { return _projects; }
  set { _projects = value; }
}
```

Listing 17-11 Adding the Projects property

Once added, the Projects property appears as shown in Figure 17-20:

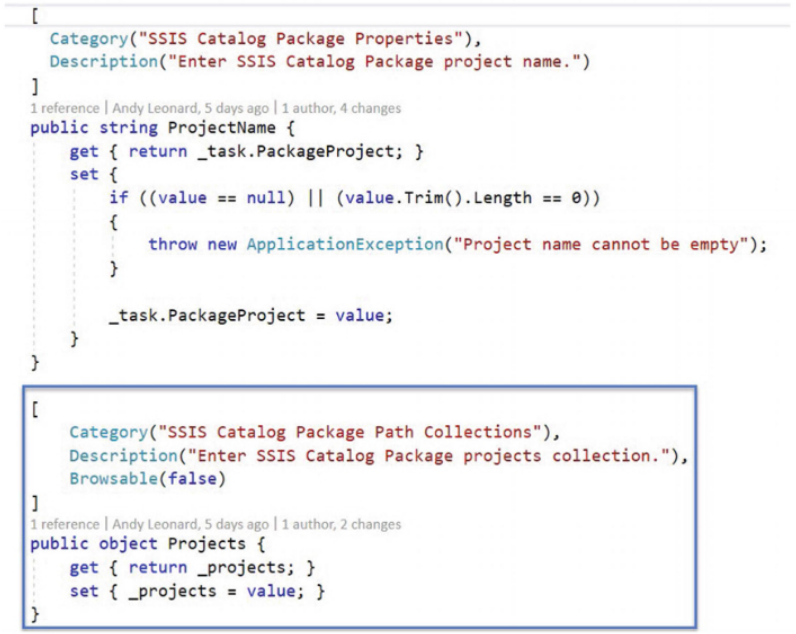


Figure 17-20 Projects property added

The next step is to add a method to populate a list of projects using the code in Listing 17-12:

```
internal ArrayList GetProjectsFromCatalog()
{
  ArrayList projectsList = new ArrayList();
  if (((_task.ServerName != null) && (_task.ServerName != ""))
   && ((_task.PackageFolder != null) && (_task.PackageFolder != "")))
```

```
    {
      CatalogFolder catalogFolder = _task.returnCatalogFolder(_task.Server
      if (catalogFolder != null)
      {
        foreach (ProjectInfo pr in catalogFolder.Projects)
        {
          projectsList.Add(pr.Name);
        }
      }
    }
    return projectsList;
  }
```

Listing 17-12 Populate the Projects collection

Once added, the code appears as shown in Figure 17-21:

```
internal ArrayList GetProjectsFromCatalog()
{
    ArrayList projectsList = new ArrayList();

    if (((_task.ServerName != null) && (_task.ServerName != ""))
     && ((_task.PackageFolder != null) && (_task.PackageFolder != "")))
    {
        CatalogFolder catalogFolder = _task.returnCatalogFolder(_task.ServerName
                                                              , _task.PackageFolder);

        if (catalogFolder != null)
        {
            foreach (ProjectInfo pr in catalogFolder.Projects)
            {
                projectsList.Add(pr.Name);
            }
        }
    }
    return projectsList;
}
```