# Exercise: A Calculator Application Using MEF

## 1 Overview

In this exercise you will develop a Calculator application using MEF provided by .NET Framework 4.7. 2The Calculator will have four basic arithmetic operations, i.e. Add, Subtract, Multiply and Division.

Estimated time to complete this tutorial: 30 minutes.

## 2 Environment & Tools

* Visual Studio 2019

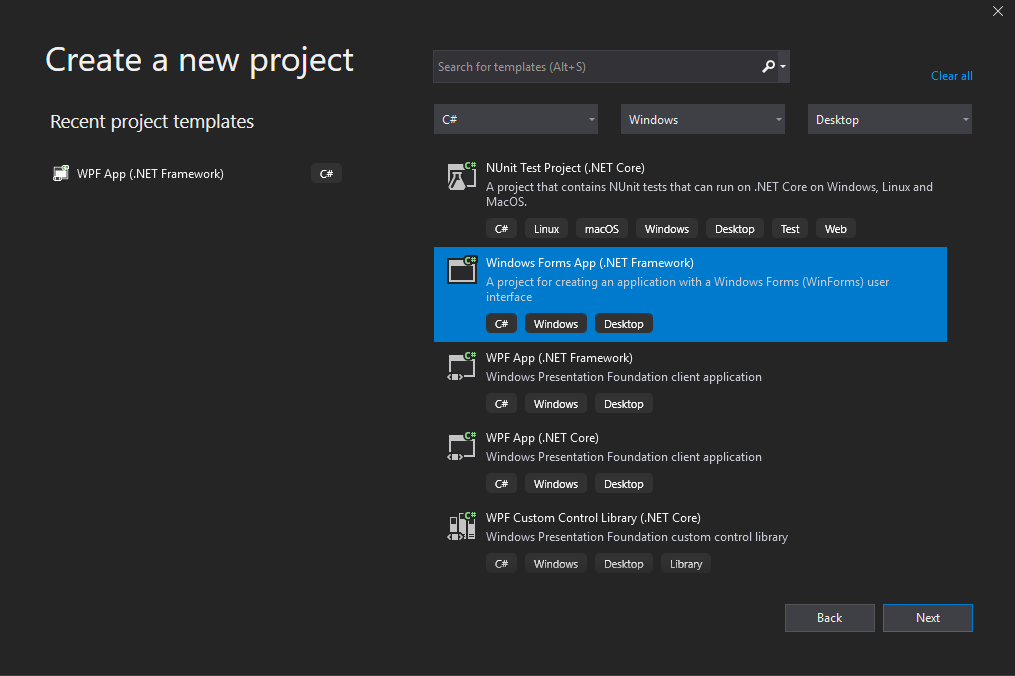
## 3 Tasks

To complete these tasks, you can choose to either follow the tasks of creating the entire project from scratch or executing the task from existing source codes provided.

**Option 1: Creating Project from Scratch**

### Task 1: Create a Windows Forms application

Open Visual Studio 2019. On the menu, choose **C#,** **Windows, Desktop, Windows Forms Application (.NET Framework)**. Name the application CalculatorUsingMEF, and click **OK**.



Add a few controls to the form, so it looks like Figure 1 below.

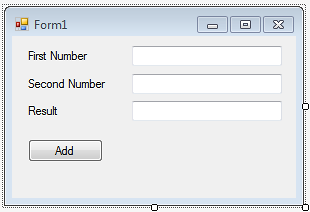
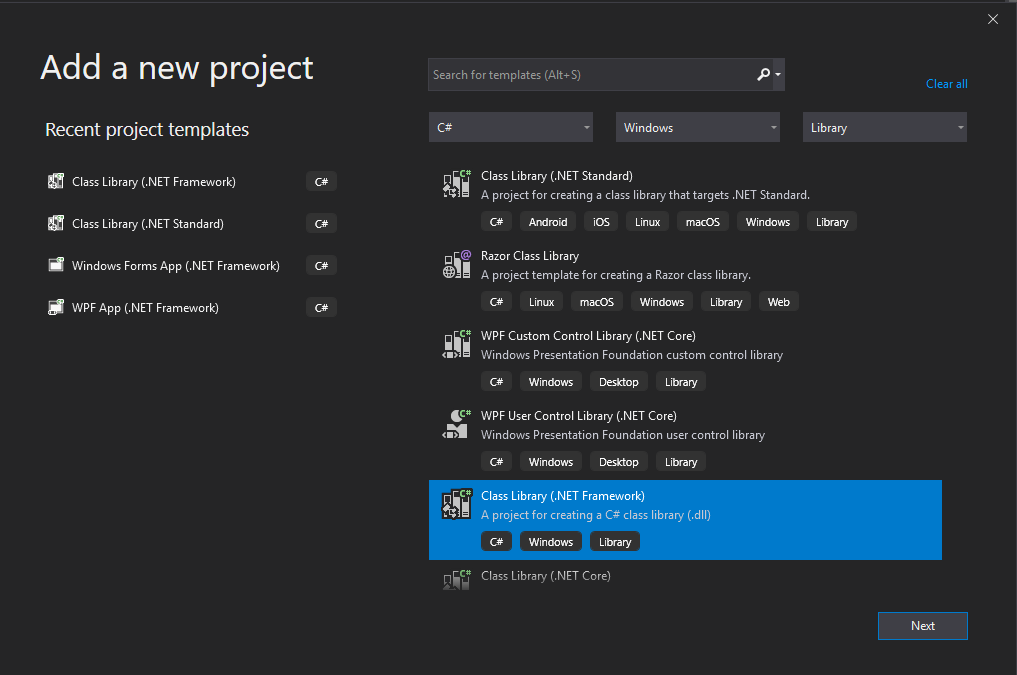


Figure 1: The application GUI

### Task 2: Create a Contract

Right click on the solution, and choose **C#,** **Windows, Library, Class Library (.NET Framework)**. Name the project Contract, and click **OK**.



Right click on the project, and choose **Add** -> **New Item** -> **Interface**. Name the Interface ICalculator, and click **OK**.

Now, the structure of the Solution should look like Figure 2 below.

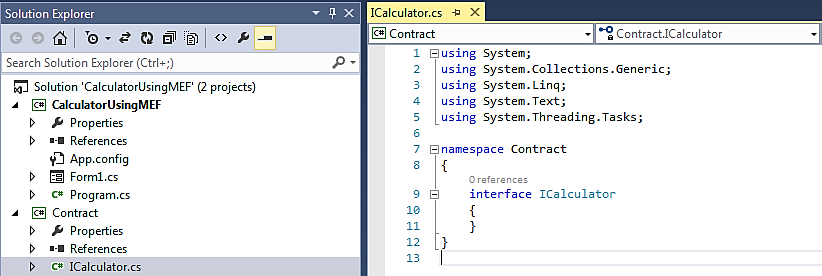


Figure 2: Structure of the Solution

Modify the Interface by adding a GetResult method:

public interface ICalculator

{

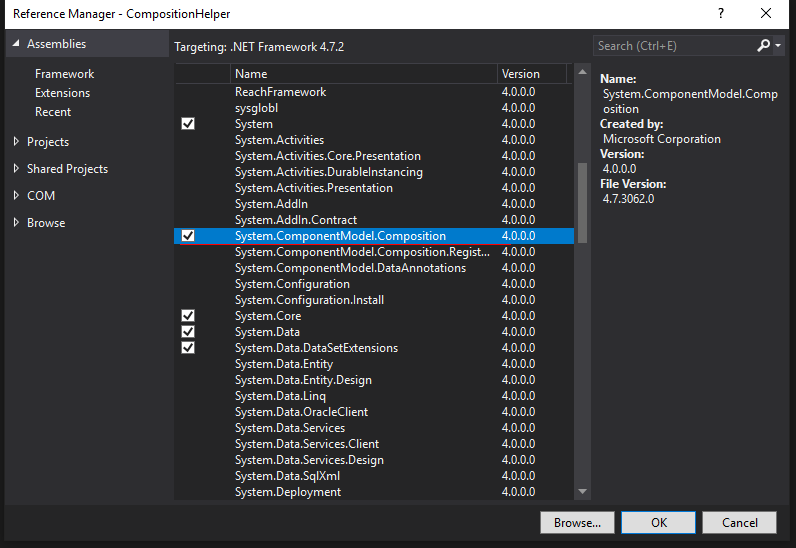
double GetResult(double FirstNumber, double SecondNumber);

}

### Task 3: Create an Export Part in the current assembly

Again, add another Class Library project to the Solution, and name it CompositionHelper.

Right click on **References** of the project, and choose **Add Reference**. In the Reference Manager dialog, select System.ComponentModel.Composition under the **Assemblies** category, as shown in Figure 3. Then, select the Contract project under the **Solution** category. Click **OK**.



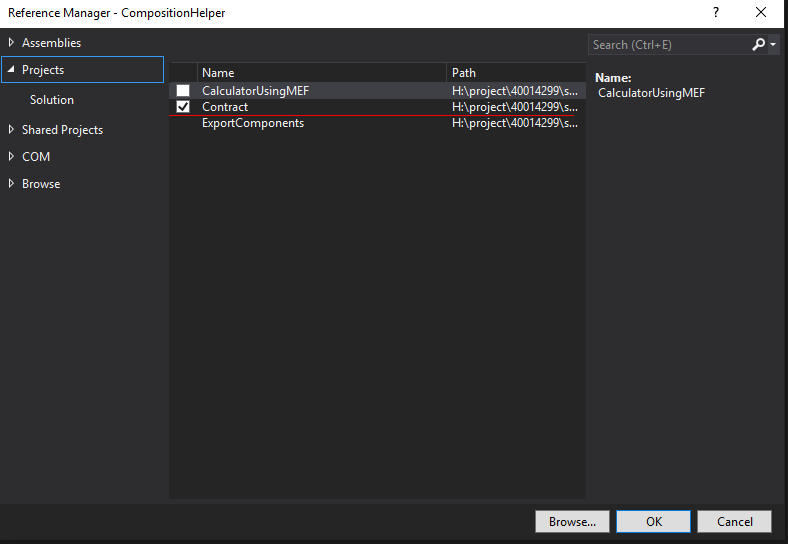


Figure 3: Add MEF reference

Next, create a new **Class**, namely Add. The source code of the class is listed below:

using System;

using System.ComponentModel.Composition;

using Contract;

namespace CompositionHelper

{

[Export(typeof(ICalculator))]

public class Add : ICalculator

{

public double GetResult(double num1, double num2)

{

return num1 + num2;

}

}

}

Note: The class is annotated by the Export attribute which suggests that this class will be exported to the composition model and it fulfils the ICalculator interface type.

### Task 4: Create an Import Part

Continue to add another new Class, namely CalculatorCompositionHelper. The source code of the class is listed below:

using System;

using System.ComponentModel.Composition;

using System.ComponentModel.Composition.Hosting;

using System.Reflection;

using Contract;

namespace CompositionHelper

{

public class CalculatorCompositionHelper

{

[Import(typeof(ICalculator))]

public ICalculator CalPlugin { get; set; }

public void AssembleCalculatorComponents()

{

try

{

//Step 1: Initialise a new instance of the

//System.ComponentModel.Composition.Hosting.AssemblyCatalog

//class with the current executing assembly.

var catalog = new AssemblyCatalog(

Assembly.GetExecutingAssembly());

//Step 2: Add The assemblies obtained in step 1 to the

//CompositionContainer

var container = new CompositionContainer(catalog);

//Step 3: Create composable parts, i.e. the Import and

//Export components

container.ComposeParts(this);

} catch (Exception ex)

{

throw ex;

}

}

public double Execute(double num1, double num2)

{

return CalPlugin.GetResult(num1, num2);

}

}

}

Note: The class member CalPlugin is annotated by the Import attribute which suggests that it will be injected by a component at runtime.

Now, the structure of the Solution should look like Figure 4 below.

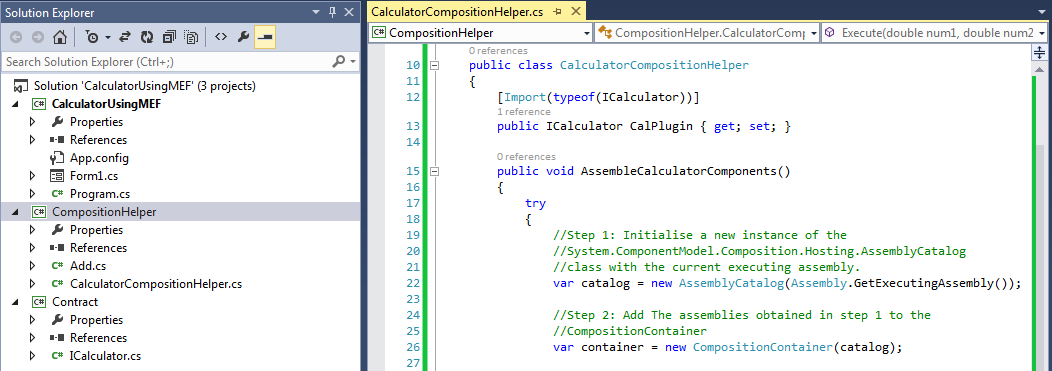


Figure 4: Updated structure of the Solution

### Task 5: Update the Calculator application to consume the Exported part

Right click on **References** of the **CalculatorUsingMEF** project, and choose **Add Reference**. In the Reference Manager dialog, select CompositionHelper under the **Solution** category. Click **OK**.

Update the event handler of the Add button with the following source code:

... ...

using CompositionHelper;

namespace CalculatorUsingMEF

{

public partial class Form1 : Form

{

CalculatorCompositionHelper helper;

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

helper = new CalculatorCompositionHelper();

helper.AssembleCalculatorComponents();

var result = helper.Execute(Convert.ToDouble(textBox1.Text),

Convert.ToDouble(textBox2.Text));

textBox3.Text = result.ToString();

}

}

}

Run the application and try out the Add function.

### Task 5: Create multiple Export parts in the current assembly

Well by this time, we have learnt something about MEF, how it works and also we have successfully added one component in our application. Now let’s enhance our application so that it can handle all the other components like Subtraction, Multiplication and Division.

Visit the Add class in the CompositionHelper project. Insert the following line into the source code:

... ...

[Export(typeof(ICalculator))]

[ExportMetadata("CalculatorMetaData", "Add")]

public class Add : ICalculator

{ ... ...

Note: The ExportMetadata attribute will help us to determine at runtime which implementation to use. It is essentially a form of name value pair.

Next, update the CalculatorCompositionHelper class with the following source code:

using System;

using System.ComponentModel.Composition;

using System.ComponentModel.Composition.Hosting;

using System.Reflection;

using System.Collections.Generic;

using Contract;

namespace CompositionHelper

{

public class CalculatorCompositionHelper

{

[ImportMany]

public System.Lazy<ICalculator, IDictionary<string, object>>[]

CalPlugins { get; set; }

public void AssembleCalculatorComponents()

{

... ...

}

public double Execute(double num1, double num2, string operationType)

{

double result = 0;

foreach (var CalPlugin in CalPlugins)

{

if ((string)CalPlugin.Metadata["CalculatorMetaData"]

== operationType)

{

result = CalPlugin.Value.GetResult(num1, num2);

break;

}

}

return result;

}

}

}

Note:

* The first change is that now we have ImportMany attribute instead of Import attribute. The Import attribute is always fulfilled by a single Export attribute while ImportMany attribute can be fulfilled by any number of Export attributes.
* One more noticeable point is the Lazy<T> class basically defers the creation of large objects till the time we need them. In the sample code, the second parameter is the metadata that will be handled by MEF at runtime.
* Correspondingly, the Execute function has been changed a little bit. It now accepts a third parameter which specifies the operation type and based on that, the Exported parts will be invoked.

Now, create Subtract, Multiply and Divide classes in the CompositionHelper project using the following source code respectively:

using System;

using System.ComponentModel.Composition;

using Contract;

namespace CompositionHelper

{

[Export(typeof(ICalculator))]

[ExportMetadata("CalculatorMetaData", "Subtract")]

public class Subtract : ICalculator

{

public double GetResult(double num1, double num2)

{

return num1 - num2;

}

}

}

using System;

using System.ComponentModel.Composition;

using Contract;

namespace CompositionHelper

{

[Export(typeof(ICalculator))]

[ExportMetadata("CalculatorMetaData", "Multiply")]

public class Multiply : ICalculator

{

public double GetResult(double num1, double num2)

{

return num1 \* num2;

}

}

}

using System;

using System.ComponentModel.Composition;

using Contract;

namespace CompositionHelper

{

[Export(typeof(ICalculator))]

[ExportMetadata("CalculatorMetaData", "Divide")]

public class Divide : ICalculator

{

public double GetResult(double num1, double num2)

{

return num1 / num2;

}

}

}

To try out these components, let’s add a few more buttons to the application GUI. The result may look like Figure 5.

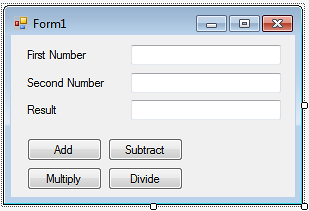


Figure 5: Updated application GUI

Also, update the buttons’ event handlers with the following source code:

... ...

using CompositionHelper;

namespace CalculatorUsingMEF

{

public partial class Form1 : Form

{

CalculatorCompositionHelper helper;

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

DoOperation("Add");

}

private void button2\_Click(object sender, EventArgs e)

{

DoOperation("Subtract");

}

private void button3\_Click(object sender, EventArgs e)

{

DoOperation("Multiply");

}

private void button4\_Click(object sender, EventArgs e)

{

DoOperation("Divide");

}

private void DoOperation(string operationType)

{

helper = new CalculatorCompositionHelper();

helper.AssembleCalculatorComponents();

var result = helper.Execute(Convert.ToDouble(textBox1.Text),

Convert.ToDouble(textBox2.Text), operationType);

textBox3.Text = result.ToString();

}

}

}

Run the application and try out the Add, Subtract, Multiply and Divide functions.

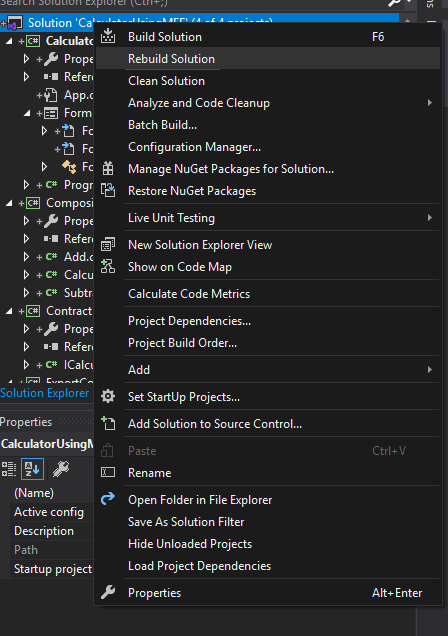
### Task 6: Use Aggregate and Directory Catalogs

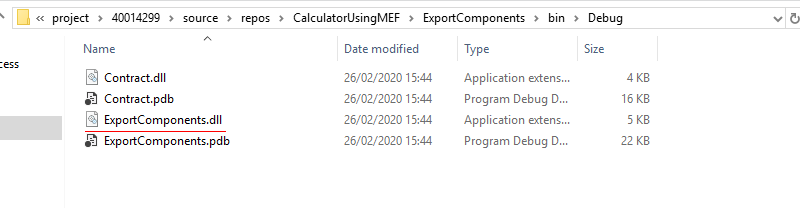
Until now, we have seen that our composable parts were residing in the current assembly. However, what if they reside in different assemblies or in different locations? We will see how Aggregate and Directory Catalogs help us in such situations.

Firstly, create another Class Library project, namely ExportComponents. By default, the project uses the ExportComponents namespace, which is not referred by the other projects in the solution, and thus can be regarded as an isolated third-party codebase.

Secondly, move the Multiply and Divide classes from the CompositionHelper project to the ExportedComponents project, and remember to change their namespace.

Thirdly, compile the ExportedComponents project to obtain ExportedComponents.dll. As explained above, this file can be regarded as a third-party assembly. Copy this file to any directory, e.g. d:\lib\.





OK, now let’s find out whether our Calculator application is able to discover and consume the Multiply and Divide components residing in such a third-party assembly.

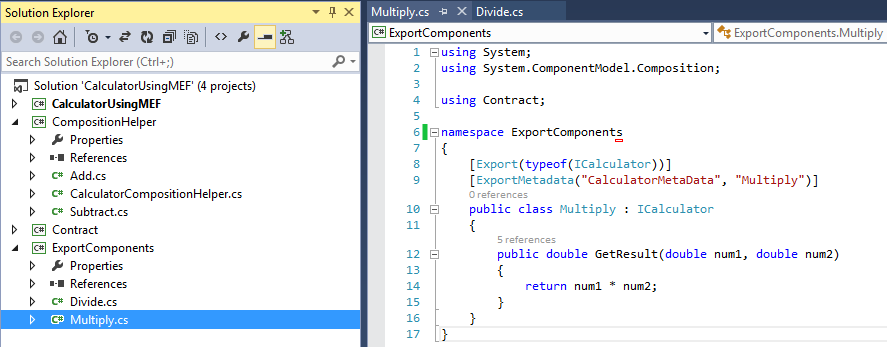


Figure 6: Moved Multiply and Divide components to an external project

Visit the CalculatorCompositionHelper class in the CompositionHelper project. Update its AssembleCalculatorComponents function with the following source code:

public void AssembleCalculatorComponents() {

try {

var aggregateCatalog = new AggregateCatalog();

var directoryPath = @"d:\lib";

var directoryCatalog = new DirectoryCatalog(directoryPath, "\*.dll");

var asmCatalog = new AssemblyCatalog(Assembly.GetExecutingAssembly());

aggregateCatalog.Catalogs.Add(directoryCatalog);

aggregateCatalog.Catalogs.Add(asmCatalog);

var container = new CompositionContainer(aggregateCatalog);

container.ComposeParts(this);

} catch (Exception ex) {

throw ex;

}

}

Note: As can be seen, initially we have created an AggregateCatalog that will aggregate other catalogs. Then we are picking up the third-party DLLs from d:\lib directory.

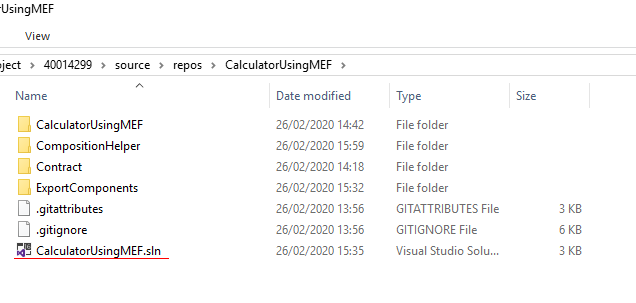
Run the application and try out the Add, Subtract, Multiply and Divide functions.

To make sure that the Multiply and Divide functions really come from the third-party DLL, remove the DLL file from d:\lib and run the application again.

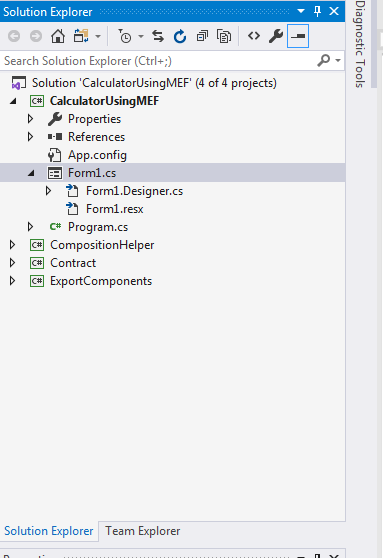
**Option 2: Implementing from the Existing Source Code**

### **Step 1:** Extract the existing source folder to a choice location on the local computer

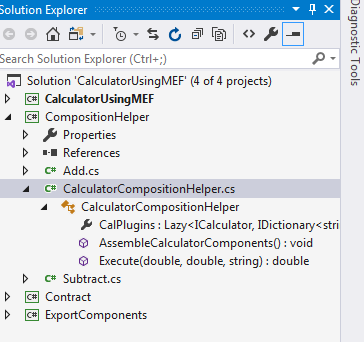
**Step 2:** In the extracted folder navigate to a file named “CalculatorUsingMEF.sln”, double-click or right-click to open in the recent Microsoft Visual Studio version installed on your local computer.

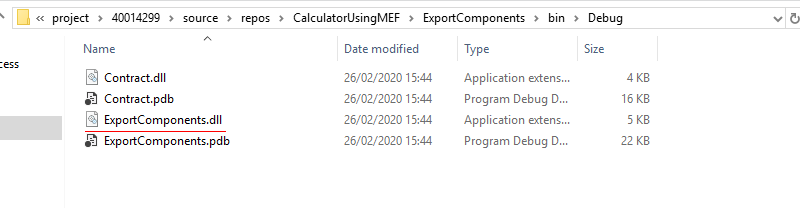


**Step 3:** Expand the project named “CalculatorUsingMEF” to open “Form1.cs”

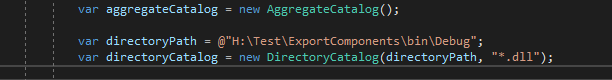


**Step 4:** Expand the project named “CompositionHelper” in order to open the “CalculatorCompositionHelper.cs” class



**Step 5:** Navigate to the extracted folder to copy the ExportComponents library. 

**Step 6:** Save the library file in any location on your local computer and set the value of the variable directoryPath within the CalculatorCompositionHelper class to the new location of the ExportComponents.dll library.



**Step 6:** Execute the project, try out the Add, Subtract, Multiply, and Divide functions.

To make sure that the Multiply and Divide functions really come from the third-party DLL, remove the DLL file from the current location and run the application again.