|  |  |
| --- | --- |
|  | LAB | Dependency Injection with Mef |
|  | WORKSHOP | Design Patterns |
|  | Crystal Tenn  crtenn@microsoft.com |

# SETUP LAB ENVRIOMENT

## Prerequisites

To perform the tasks in this lab you need following:

* Visual Studio 2015 or 2017, any version

## Objectives

In this lab you will complete following tasks:

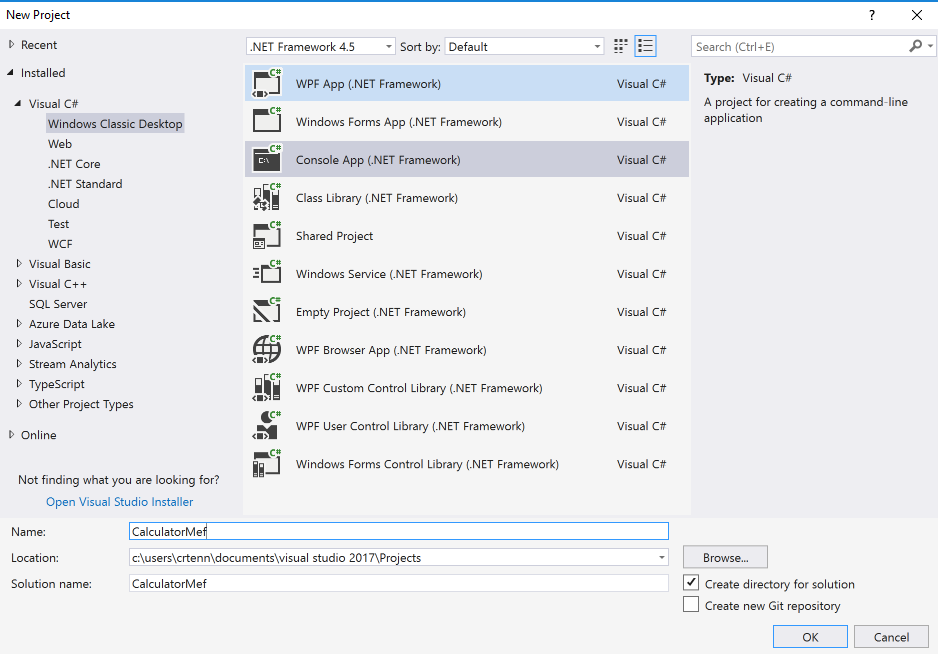
* Create a simple Console calculator application with dependency injection using Mef

### Estimated Completion Time: 30 minutes.

## Task: Create a Console Application

In this task you will create a new Console application.

1. Open Visual Studio.
2. Choose New Project. Select a .NET Standard Framework Console App.
3. Name it: CalculatorMef
4. Hit OK to create your new application.



**Note**: Dependency Injection is based off programming to an interface, not an implementation. We need to create interfaces for all our classes that will contain application logic. This is effectively the Dependency Inversion Principle. Doing so allows you to replace, intercept or decorate dependencies without the need to change consumers of such dependency. This is useful for if you need to change out certain frameworks or sections of decoupled code as you can remove the implementation and just rewrite your new code / implement your new framework into the pre-existing interface setup. Additionally, dependency injection allows you to do more effective unit testing as you can dynamically mock out code using the interfaces.Please complete the following steps to create the interfaces we will need to make our application work.

1. Add a new class called ICalculator. Remove the existing class code, and add the following code to it:

public interface ICalculator

{

String Calculate(String input);

}

1. Add a new class called IOperation. Remove the existing class code, and add the following code to it:

public interface IOperation

{

int Operate(int left, int right);

}

1. Add a new class called IOperationData. Remove the existing class code, and add the following code to it:

public interface IOperationData

{

Char Symbol { get; }

}

**Note**: Next, we will need to add a class for implementing the actual operations, such as addition.

1. Add a new class called: Add.cs. Put the following code in this new class:

[Export(typeof(IOperation))]

[ExportMetadata("Symbol", '+')]

class Add : IOperation

{

public int Operate(int left, int right)

{

return left + right;

}

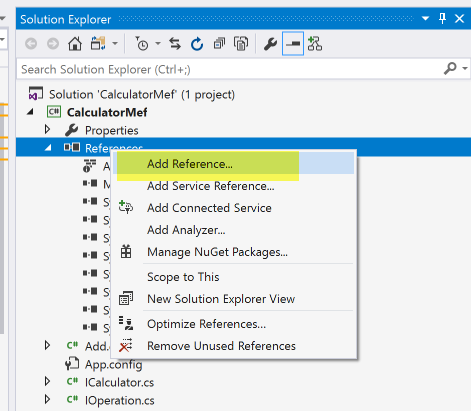
}

**Note**: Let’s talk about Mef before we add it into our project. The Managed Extensibility Framework or MEF is a library for creating lightweight, extensible applications. It allows application developers to discover and use extensions with no configuration required. It also lets extension developers easily encapsulate code and avoid fragile hard dependencies. MEF not only allows extensions to be reused within applications, but across applications as well. It was introduced by in the .NET Framework 4. Since it is part of the .NET Framework it is not a NuGet package, but a reference to add and it is part of System.

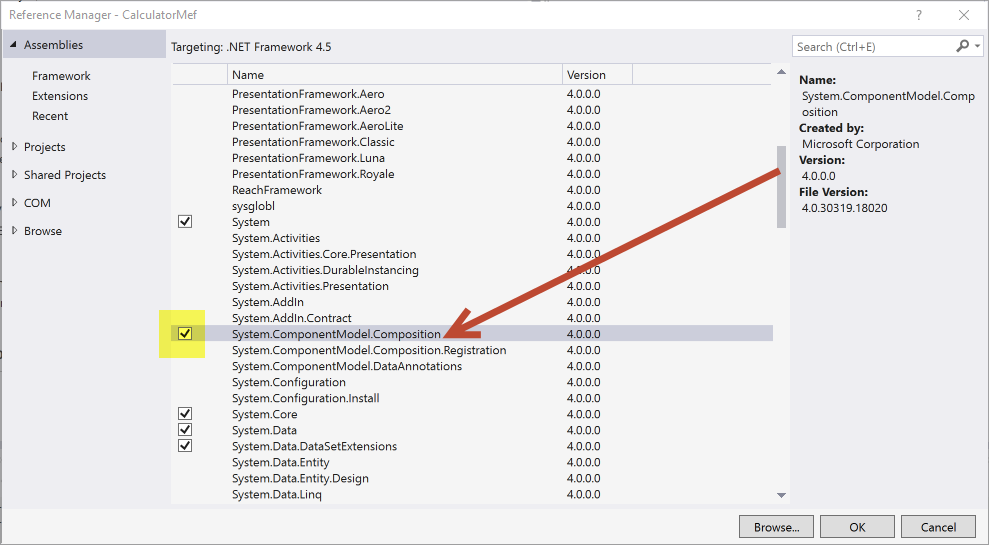
**How does it work?**

* Roughly speaking, MEF's core consists of a catalog and a **CompositionContainer**. A catalog is responsible for discovering extensions and the container coordinates creation and satisfies dependencies.
* MEF's first-class citizen is the **ComposablePart** class. A composable part offers up one or more **Exports**, and may also depend on one or more externally provided services or Imports. A composable part also manages an instance, which can be an object instance of a given type (it is in the default MEF implementation)[clarification needed]. MEF, however, is extensible and additional **ComposablePart** implementations can be provided as long as they adhere to the **Import/Export** contracts.
* Exports and imports each have a **Contract**. **Contracts** are the bridge between exports and imports. An export contract can consist of further metadata that can be used to filter on its discovery. For example, it might indicate a specific capability that the export offers.
* MEF's container interacts with **Catalogs** to have access to composable parts. The container itself resolves a part's dependencies and exposes Exports to the outside world. Composable part instances may be added directly to the container.
* A **ComposablePart** returned by a catalog will likely be an extension to the application. It might have **Imports** (dependencies) on components the host application offers, and it's likely to **Export** others.
* The default MEF composable part implementation uses attribute-based metadata to declare exports and imports. This allows MEF to determine which parts, imports, and exports are available through discovery.

1. Right click on your project’s references. Hit Add References.



1. Click on Assemblies on the left. Choose System.ComponentModel.Composition



1. Verify it has been added to your references in your project.
2. Add this using statement to the top of your new Add.cs class: using System.ComponentModel.Composition;
3. Your full Add.cs class should look like the following:

using System.ComponentModel.Composition;

namespace CalculatorMef

{

[Export(typeof(IOperation))]

[ExportMetadata("Symbol", '+')]

public class Add : IOperation

{

public int Operate(int left, int right)

{

return left + right;

}

}

}

**Note**: I want to explain what is going on in the Add.cs class above. We are implementing the IOperation interface, so that we can sub out the Add functionality as needed if it’s implementation needed to change in the future. For example, we could rewrite the a brand new Add class to deal with multiple numbers or check for order of operations, ensure it had an Operate method, and implemented the interface.. then we would delete the existing Add.cs class and have a brand new one to work off of. It makes swapping out parts very simple!

Additionally, we could mock out this Add class in the class it will be used in and have it return anything we wanted. While this is not as useful here, since it is a simple calculator application.. this would be useful if you were creating an implementation of a service that called a database or another application and you wanted to mock it out and have it return a specific subset of data of your choice.

Lastly, let’s go over the attributes used at the top of this class. We talked about Mef consisting of conceptual objects include parts, imports, and **exports**. We are utilizing the export attribute in this class.

An export is a value that a part provides to other parts in the container, and an import is a requirement that a part expresses to the container, to be filled from the available exports. In the attributed programming model, imports and exports are declared by decorating classes or members with the Import and Export attributes. The Export attribute can decorate a class, field, property, or method, while the Import attribute can decorate a field, property, or constructor parameter.

To use metadata, you typically declare an interface known as a metadata view, which declares what metadata will be available. The metadata view interface must have only properties, and those properties must have get accessors.

It is also possible to use a generic collection, IDictionary<string, object>, as a metadata view, but this forfeits the benefits of type checking and should be avoided.

Ordinarily, all of the properties named in the metadata view are required, and any exports that do not provide them will not be considered a match. The DefaultValue attribute specifies that a property is optional. If the property is not included, it will be assigned the default value specified as a parameter of DefaultValue.

For an import to be matched with an export, the import and export must have the same contract. The contract consists of a string, called the contract name, and the type of the exported or imported object, called the contract type. Only if both the contract name and contract type match is an export considered to fulfill an import.

**Best Practices:** Exports and imports should usually be declared on public classes or members. Other declarations are supported, but exporting or importing a private, protected, or internal member breaks the isolation model for the part and is therefore not recommended. And remember, the contract type must match exactly for the export and import to be considered a match!!!

1. Let’s create a similar class for Subtract that will be called Subtract.cs and look like this:

using System.ComponentModel.Composition;

namespace CalculatorMef

{

[Export(typeof(IOperation))]

[ExportMetadata("Symbol", '-')]

class Subtract : IOperation

{

public int Operate(int left, int right)

{

return left - right;

}

}

}

1. Create a new class called MySimpleCalculator.cs, add the following to it.

Note to explain new code that is added: The Import attribute will only be successfully composed when it matches one and only one export. Other cases will produce a composition error. To import more than one export that matches the same contract, use the ImportMany attribute. Imports marked with this attribute are always optional. For example, composition will not fail if no matching exports are present. In many cases, you will want to combine metadata with the ImportMany attribute, to parse through the available imports and choose and instantiate only one, or filter a collection to match a certain condition.

Lazy loading should be used when you want to instantiate something the first time it’s actually used. This delays the cost of creating it till if/when it's needed instead of always incurring the cost. Usually this is preferable when the object may or may not be used and the cost of constructing it is non-trivial.

using System;

using System.Collections.Generic;

using System.ComponentModel.Composition;

namespace CalculatorMef

{

[Export(typeof(ICalculator))]

public class MySimpleCalculator : ICalculator

{

[ImportMany] IEnumerable<Lazy<IOperation, IOperationData>> operations;

public String Calculate(String input)

{

}

}

}

1. Add the following inside of the Calculate method to parse input that a user would add to the Console application:

int left;

int right;

Char operation;

int fn = FindFirstNonDigit(input); //finds the operator

if (fn < 0) return "Could not parse command.";

try

{

//separate out the operands

left = int.Parse(input.Substring(0, fn));

right = int.Parse(input.Substring(fn + 1));

}

catch

{

return "Could not parse command.";

}

1. Add the following code into your Calculate method under your try catch statement:

operation = input[fn];

foreach (Lazy<IOperation, IOperationData> i in operations)

{

if (i.Metadata.Symbol.Equals(operation)) return i.Value.Operate(left, right).ToString();

}

return "Operation Not Found!";

1. Add a private method below your Calculate method:

private int FindFirstNonDigit(String s)

{

for (int i = 0; i < s.Length; i++)

{

if (!(Char.IsDigit(s[i]))) return i;

}

return -1;

}

1. Your full class should look like the following:

using System;

using System.Collections.Generic;

using System.ComponentModel.Composition;

namespace CalculatorMef

{

[Export(typeof(ICalculator))]

public class MySimpleCalculator : ICalculator

{

[ImportMany]

IEnumerable<Lazy<IOperation, IOperationData>> operations;

public String Calculate(String input)

{

int left;

int right;

Char operation;

int fn = FindFirstNonDigit(input); //finds the operator

if (fn < 0) return "Could not parse command.";

try

{

//separate out the operands

left = int.Parse(input.Substring(0, fn));

right = int.Parse(input.Substring(fn + 1));

}

catch

{

return "Could not parse command.";

}

operation = input[fn];

foreach (Lazy<IOperation, IOperationData> i in operations)

{

if (i.Metadata.Symbol.Equals(operation)) return i.Value.Operate(left, right).ToString();

}

return "Operation Not Found!";

}

private int FindFirstNonDigit(String s)

{

for (int i = 0; i < s.Length; i++)

{

if (!(Char.IsDigit(s[i]))) return i;

}

return -1;

}

}

}

1. Go to your Program.cs file. Add the following code to the top of the class:

Reminder Note: For an import to be matched with an export, the import and export must have the same contract. The contract consists of a string, called the contract name, and the type of the exported or imported object, called the contract type. Only if both the contract name and contract type match is an export considered to fulfill a import. The Import attribute will only be successfully composed when it matches one and only one export. Other cases will produce a composition error.

private CompositionContainer \_container;

[Import(typeof(ICalculator))]

public ICalculator calculator;

1. Add the following constructor to your Program.cs file underneath the code you added in the last step. Make sure you change the highlighted section to wherever your “SimpleCalculator3\Extensions” path is located. Also escape the slashes, so you have a double slash “\\”. Note, while this folder is empty, it will break the program if you do not enter the correct path. Also this DirectoryCatalog addition is here to be illustrative of how you can add a specific directory in if necessary so you have that option. In the code below, we are displaying how to use both the AssemblyCatalog and the DirectoryCatalog.

**Note:** In some cases, you may want to prevent a part from being discovered as part of a catalog. For example, the part may be a base class intended to be inherited from, but not used. There are two ways to accomplish this. First, you can use the abstract keyword on the part class. Abstract classes never provide exports, although they can provide inherited exports to classes that derive from them.

If the class cannot be made abstract, you can decorate it with the PartNotDiscoverable attribute. A part decorated with this attribute will not be included in any catalogs.

private Program()

{

//An aggregate catalog that combines multiple catalogs

var catalog = new AggregateCatalog();

//Adds all the parts found in the same assembly as the Program class

catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));

catalog.Catalogs.Add(new DirectoryCatalog("C:\\Users\\crtenn\\05 OCC Design Patterns\\Day 1\\Simple Calculator MEF Application\\C#\\SimpleCalculator3\\Extensions"));

//Create the CompositionContainer with the parts in the catalog

\_container = new CompositionContainer(catalog);

//Fill the imports of this object

try

{

this.\_container.ComposeParts(this);

}

catch (CompositionException compositionException)

{

Console.WriteLine(compositionException.ToString());

}

}

1. Add the following using statements to the top of your Program.cs file:

using System;

using System.ComponentModel.Composition;

using System.ComponentModel.Composition.Hosting;

1. Edit your Main method so it looks like this:

static void Main(string[] args)

{

Program p = new Program(); //Composition is performed in the constructor

String s;

Console.WriteLine("Enter Command:");

while (true)

{

s = Console.ReadLine();

Console.WriteLine(p.calculator.Calculate(s));

}

}

1. Your full Program.cs file should look like this:

using System;

using System.ComponentModel.Composition;

using System.ComponentModel.Composition.Hosting;

namespace CalculatorMef

{

class Program

{

private CompositionContainer \_container;

[Import(typeof(ICalculator))]

public ICalculator calculator;

private Program()

{

//An aggregate catalog that combines multiple catalogs

var catalog = new AggregateCatalog();

//Adds all the parts found in the same assembly as the Program class

catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));

//catalog.Catalogs.Add(new DirectoryCatalog("C:\\Users\\crtenn\\Google Drive\\Microsoft\\2017-2018 Q1 Work\\05 OCC Design Patterns\\Day 1\\Simple Calculator MEF Application\\C#\\SimpleCalculator3\\Extensions"));

//Create the CompositionContainer with the parts in the catalog

\_container = new CompositionContainer(catalog);

//Fill the imports of this object

try

{

this.\_container.ComposeParts(this);

}

catch (CompositionException compositionException)

{

Console.WriteLine(compositionException.ToString());

}

}

static void Main(string[] args)

{

Program p = new Program(); //Composition is performed in the constructor

String s;

Console.WriteLine("Enter Command:");

while (true)

{

s = Console.ReadLine();

Console.WriteLine(p.calculator.Calculate(s));

}

}

}

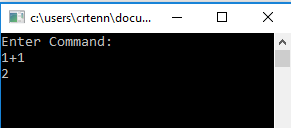
}

## Task: Test the Project

In this task you will create a new Console application.

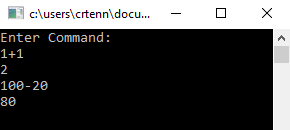
1. Click on the Start button or press F5.
2. When the console window opens, type the following and hit enter when you are done:

1 + 1

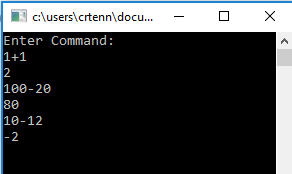


1. Type the following and hit enter when you are done:

100-20



1. **Enter: 10-12**

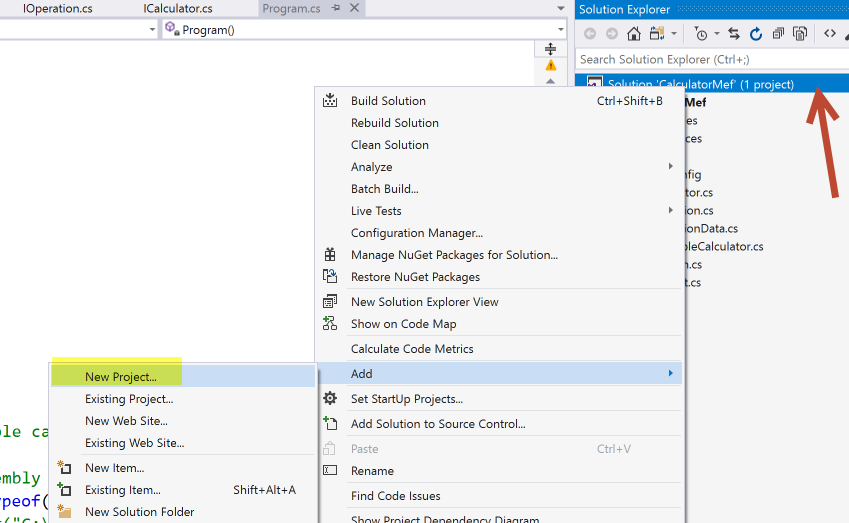


1. **When you are done testing, close the window to stop running the console app.**

## Task: Create a Unit Test with Moq

In this task you will create a new Console application.

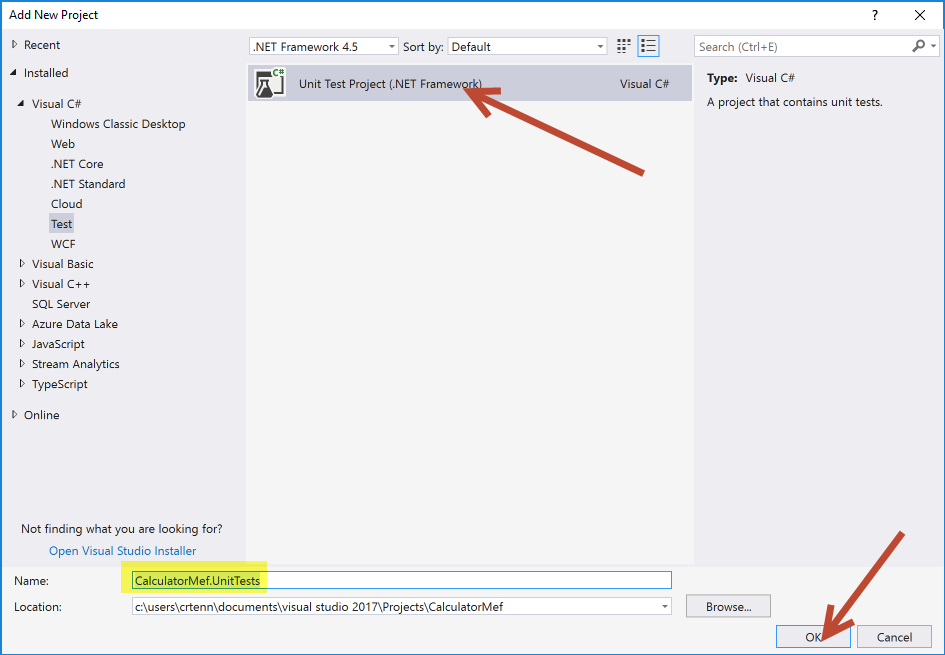
1. Right click on your solution and Add a new project.



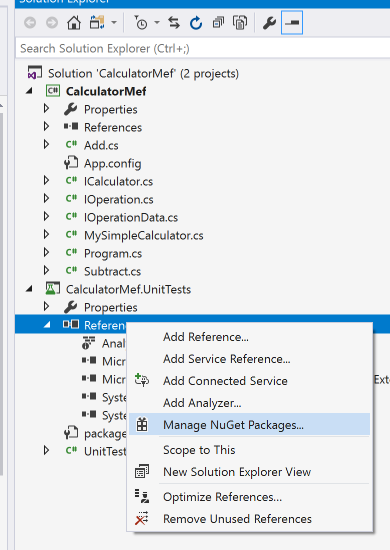
**Note**: Let’s add a unit test to prove that it is easier to test using dependency injection. In compiled languages like C# this runtime manipulation of type definitions is much more difficult. Changing the definition of what a class is doing at runtime would require a deep and low-level understanding of the code and may not be possible in many cases.

So, instead of messing with any of that we will use a mocking framework called Moq which is a common and popular choice! We will leverage the interfaces and the dependency injection framework that we just created in order to use Moq. Moq works fine with any dependency injection framework.

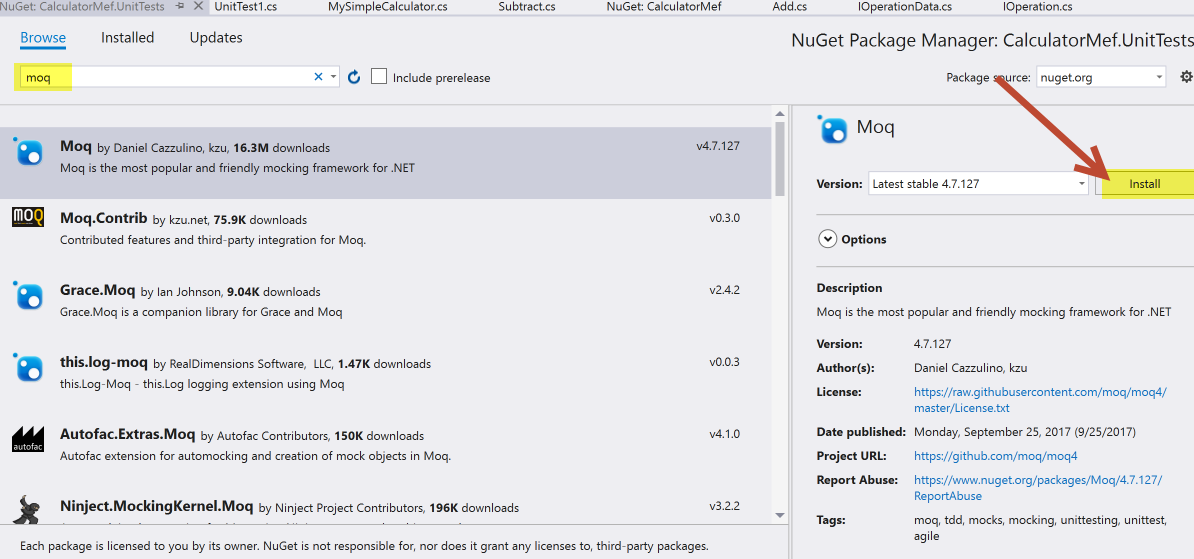
1. Under Test, choose Unit Test project. Call it: CalculatorMef.UnitTests



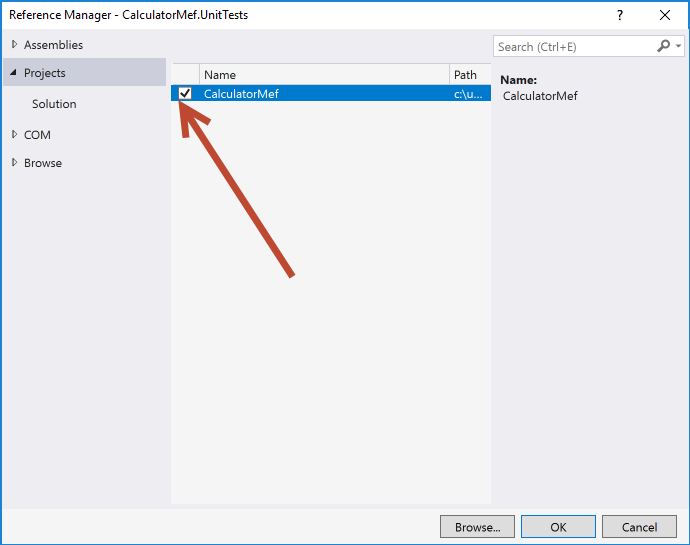
1. In the test project, right click references and manage your NuGet packages:



1. Click on Browse on the top left. Type in Moq. On the right pane, install the latest version of Moq (latest version is selected by default).



1. Once you are done installing Moq, ensure it shows up in your References of your test project.
2. Add a reference from your unit test project to the CalculatorMef project. Right click on references on your test project, then go to Projects, then check the box for CalculatorMef.



1. Rename your UnitTest1.cs class to AddTests.cs