* 1. 

Hands-On Lab

Introduction to the Managed Extensibility Framework

* 1. Lab version: 1.0.0
  2. Last updated: 1/4/2011
  3. 

Contents

[Overview 3](#_Toc281932539)

[Exercise 1: Using MEF to Extend an Application 5](#_Toc281932540)

[Task 1 – Updating Your Application to Load Composable Parts 6](#_Toc281932541)

[Task 2 – Updating Your Composable Part to Export Contracts 11](#_Toc281932542)

[Task 3 – Creating a New Extension 16](#_Toc281932543)

[Exercise 1: Verification 23](#_Toc281932544)

[Exercise 2: Using Metadata and Lazy-Loading Modules 23](#_Toc281932545)

[Task 1 – Attaching Metadata to an Export 24](#_Toc281932546)

[Task 2 – Using a Custom Export Attribute 25](#_Toc281932547)

[Task 3 – Importing Metadata 32](#_Toc281932548)

[Exercise 2: Verification 38](#_Toc281932549)

[Exercise 3: Using MEF and Silverlight 4 40](#_Toc281932550)

[Task 1 – Convert Extensions to Silverlight 40](#_Toc281932551)

[Task 2 – Create Silverlight Cash Maker UI 48](#_Toc281932552)

[Task 3 – Update Your Silverlight Application to Load Composable Parts 55](#_Toc281932553)

[Task 4 – Load Silverlight Extensions Dynamically 57](#_Toc281932554)

[Exercise 3: Verification 61](#_Toc281932555)

[Summary 64](#_Toc281932556)

Overview

* 1. The Managed Extensibility Framework (MEF) allows developers to provide hooks into their .NET applications for extensions by first and third parties. MEF can be thought of as a general application extension facility.
  2. MEF enables developers to create extensions dynamically, without the extending application or the extension requiring any specific knowledge of the other. This ensures no coupling exists between the two at compile time, enabling applications to be extended at runtime, without the need to recompile. MEF also provides the ability to examine the metadata of an extension assembly before loading the extension into the application -- a much faster approach.
  3. Several key concepts relating to extensibility are covered in this lab:
  + **Composition** is the practice of combining several objects with discrete functionality into one or more complex objects. Composition is not inheritance of functionality from a parent class, but is instead the assembling of several different objects into one. For example, **Wing**, **Propeller**, **Fuselage**, and ***VerticalStabilizer*** objects could be composed as part of an **Aircraft** object.
  + **ComposableParts** are the key building blocks of MEF. **ComposableParts** enable applications to expose and consume component extensions via **Exports** and **Imports.**
  + **Contracts** are the avenue for communication between **Export** and **Import** components. A **Contract** is normally implemented via an **Interface** class. **Contracts** enable MEF **ComposableParts** to avoid dependencies or tight coupling with other components.
  + **Conditional Binding** allows loading of components, which meet specific metadata criteria. Following the example above, you might choose to load **VerticalStabilizer** components, which were made only of composite graphite.
  1. Extensibility is achieved, primarily, by adding **Import** attributes at appropriate points in the application and adding corresponding **Export** attributes to the extensions. **Import** and **Export** attributes can be thought of in the same aspect as a supplier and consumer respectively. **Export** components supply some value; **Import** components consume that value. Other options for extensibility are open to developers, including completely custom extensibility approaches; however, this lab focuses solely on the primary approach discussed above. Moreover, the components can be imported using the **ImportMany** attribute, which loads all the components that match the given contract as a collection.
  2. With the release of Silverlight 4, MEF has been included in it, allowing you to extend rich internet applications. This does not imply that MEF can only be used with Silverlight 4; using the version available in codeplex (<http://mef.codeplex.com>), MEF can be used to compose applications created in Silverlight 3.

# Objectives

* 1. In this Hands-On Lab, you will learn how to:
  + Create extensibility modules and components
  + Import Extended assemblies while an application is running
  + Extend a Silverlight 4 application (optional)

# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2010
  + .NET Framework 4
  + Silverlight 4 (optional)
  + Silverlight 4 SDK (optional)
  + Silverlight 4 Tools for Visual Studio 2010 (optional)

# Setup

* 1. All the requisites for this lab are verified using the **Configuration Wizard**. To make sure that everything is correctly configured, follow these steps.
  2. Run the **Configuration Wizard** for the Training Kit if you have not done it previously. To do this, browse to **Source\Setup** folder of this lab, and double-click the **Dependencies.dep** file. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.
     1. **Note:** The Configuration Wizard is used for checking dependencies and setting up the environment. If the Configuration Wizard is not installed on your machine, you must install it running the DependencyChecker.msi file located on the %VS2010TKInstallationFolder%\Assets\DependencyChecker folder (e.g. by default the Training Kit is installed under C:\VS2010TrainingKit).
     2. For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The **Dependencies.dep** file launches the Visual Studio installer file that installs the code snippets.

# Exercises

* 1. This Hands-On Lab comprises the following exercises:
  2. Exercise 1: Using MEF to Extend an Application
  3. Exercise 2: Using Metadata and Lazy-Loading Modules
  4. Exercise 3: Using MEF and Silverlight 4

# Starting Materials

* 1. This Hands-On Lab includes the following starting materials.
  + **Visual Studio solutions.** Depending on the exercise you will find Visual Studio solutions that you can use as starting point for the exercises.
  1. **Note:** Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

**Note:** Each exercise contains a Visual Basic and a C# version; Inside the **End/Begin** solution folder, you will find two folders: **VB**, containing the Visual Basic version of the exercise, and **C#**, containing the C# version of it.

* 1. Estimated time to complete this lab: **30 minutes**.

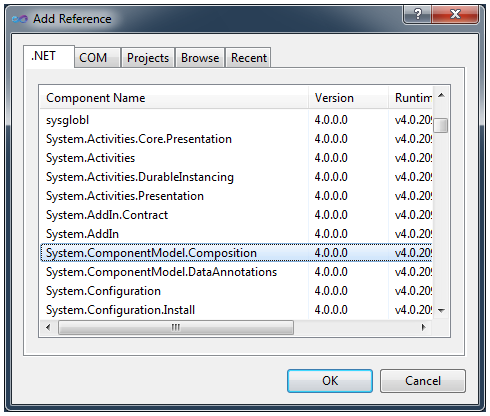
# Next Step

* 1. Exercise 1: Using MEF to Extend an Application

Exercise 1: Using MEF to Extend an Application

* 1. One practical use of the Managed Extensibility Framework is adding modules to an application at runtime. This is useful in a scenario in which users chooses specific modules to purchase or install originally and may add more modules at a later time. Using MEF, you can configure your application to monitor a well-known directory and add any module assemblies found in that directory. Dropping module assemblies into a directory allows your application to load those assemblies without explicitly setting references to them.

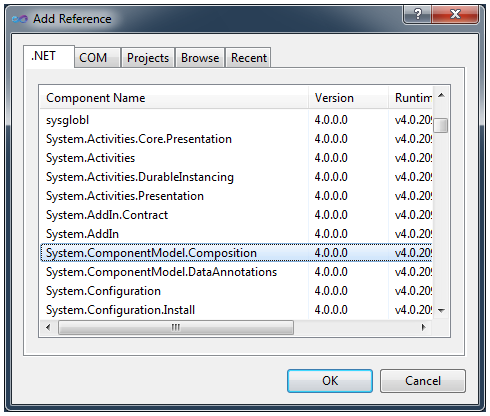
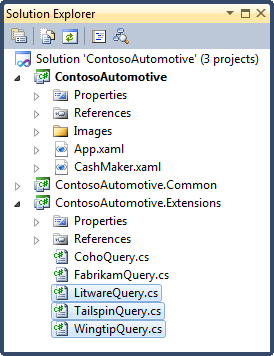
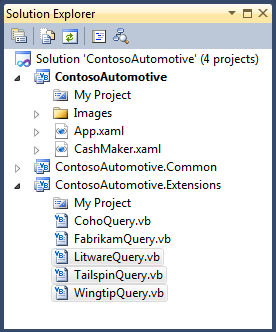
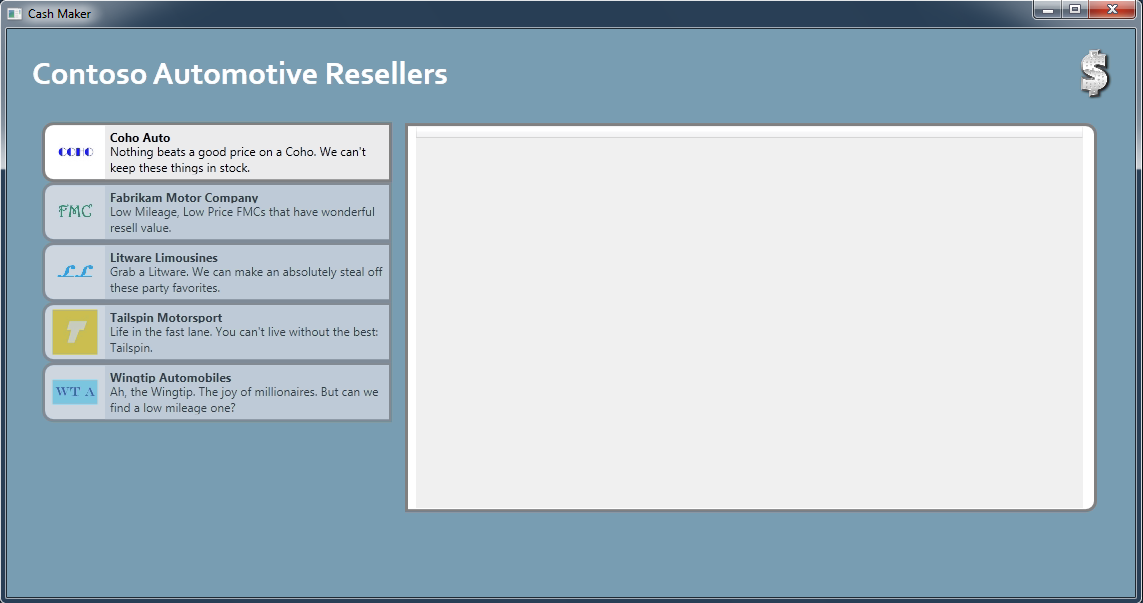
Task 1 – Updating Your Application to Load Composable Parts

* 1. In this task, you will modify an existing WPF Window class to create extension hooks that will allow dynamically importing queries. You will update this query classes later in this exercise.
  2. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  3. Open the **ContosoAutomotive.sln** solution file. By default, this file is located in the folder ***Source\Ex1\begin* (choosing the folder that matches the language of your preference.)**
  4. Add a reference to the MEF library on the **ContosoAutomotive** project. To do this:
     1. Select the **ContosoAutomotive** project in the **Solution Explorer** and select **Project | Add Reference**… The **Add References** dialog appears.
     2. Select the **.NET** tab and then select the **System.ComponentModel.Composition** component. Click the **OK** button to add a reference to this library.
        1. 
        2. Figure 1
        3. Add a Reference to the MEF library.
  5. Open the App class in code view. To do this, right-click on the **App.xaml** file in the Solution Explorer and select **View Code**.
  6. Update the App class to use the MEF library. To do this, add the following statement on top of the using clause list above the **App** class definition.
     1. C#
     2. **using System.ComponentModel.Composition.Hosting;**
     3. Visual Basic
     4. **Imports System.ComponentModel.Composition.Hosting**
  7. Create an aggregate catalog to load parts from the current application assembly and from assemblies located in the current folder in the file system. Also, create a composition container instance based on it and create an exported CashMaker instance. To do this, modify the **AppStartup** method with the following code.
     1. *(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step6 - StartupCatalogAndContainer CSharp)*
     2. C#
     3. void AppStartup(object sender, StartupEventArgs args)
     4. {
     5. **var catalog = new AggregateCatalog(new DirectoryCatalog("."),**
     6. **new AssemblyCatalog(Assembly.GetExecutingAssembly()));**
     7. **var container = new CompositionContainer(catalog);**
     8. **var window = container.GetExportedValue<CashMaker>();**
     9. window.Show();
     10. }
     11. *(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step6 - StartupCatalogAndContainer VB)*
     12. Visual Basic
     13. Sub AppStartup(ByVal sender As Object, ByVal args As StartupEventArgs)
     14. {
     15. **Dim catalog = New AggregateCatalog(New DirectoryCatalog("."),**
     16. **New AssemblyCatalog(Assembly.GetExecutingAssembly()))**
     17. **Dim container = New CompositionContainer(catalog)**
     18. **Dim window = container.GetExportedValue(Of CashMaker)()**
     19. window.Show()
     20. }
     21. **Note:** The starting point for enabling MEF composability is creating a composition container based on one or several catalogs. 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. In addition, you are free to add composable part instances directly to the container if you wish.
     22. **Note:** There are different kinds of Catalogs, which are able to discover parts on different ways. This exercise uses an Aggregate Catalog combining a Directory Catalog searching on the Application directory (represented by the “.”) and Assembly Catalog for look at the types inside the application assembly (retrieved using the Assembly.GetExecutingAssembly method).
     23. The full list of Catalogs available on MEF is:
     24. **- Assembly Catalog:** Discovers the different parts on a specific assembly.
     25. **- Directory Catalog:** Discovers parts inside the assemblies on a specific directory.
     26. **- Aggregate Catalog:** Allows use more than one catalog combining them.
     27. **- Type Catalog:** Looks inside the assemblies for specific types.
     28. For more information on Catalogs, see [Using catalogs](http://mef.codeplex.com/wikipage?title=Using%20catalogs).
     29. **Note:** Application’s main window is also located and loaded using MEF through the container’s **GetExportedValue** method. In the following steps, you will decorate the **CashMaker** class (which is the main window) with the **Export** attribute to be discoverable by MEF.
     30. The **GetExportedValue** method will throw an exception if no parts or more than one implement the solicited contract. This is not a problem in the exercise implementation because we are not using an interface to discover the type, we are using the class itself so it has to be defined and cannot be duplicated, ensuring that MEF will find only one instance.
  8. Open the CashMaker Window in code view. To do this, right-click on the **CashMaker.xaml** file in the Solution explorer and select **View Code**.
  9. Update the CashMaker class to use the MEF library. To do this, add the following statement at the top of the **CashMaker** class definition.
     1. C#
     2. **using System.ComponentModel.Composition;**
     3. Visual Basic
     4. **Imports System.ComponentModel.Composition**
  10. Since the CashMaker class is now a composable part you will export the class making it available to the composition container. To do this, decorate the **CashMaker** class with the **Export** attribute as shown in the code below.
      1. C#
      2. **[Export]**
      3. public partial class CashMaker : Window
      4. {
      5. ...
      6. }
      7. Visual Basic
      8. **<Export()>**
      9. Class CashMaker
      10. Inherits Window
      11. ...
      12. End Class

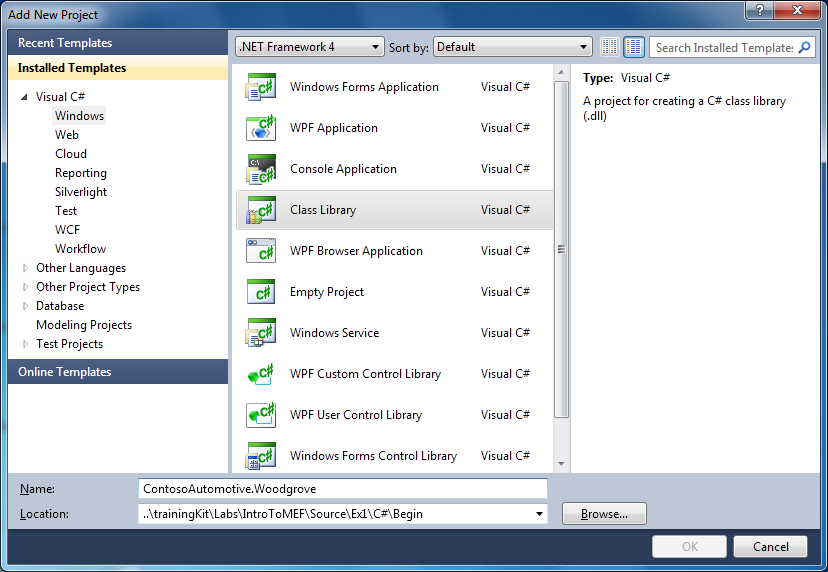
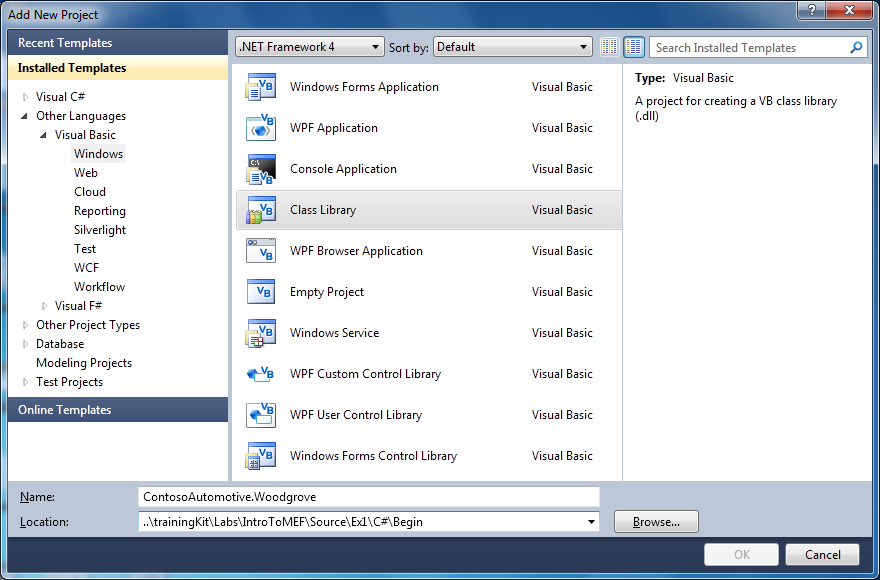
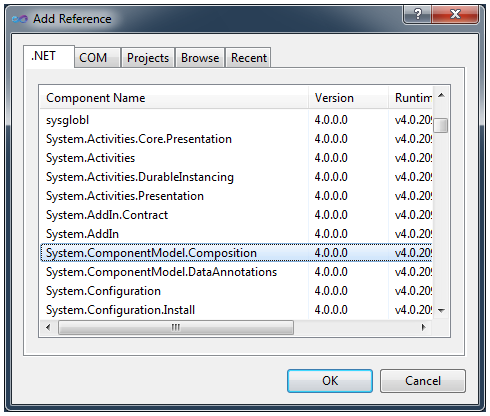
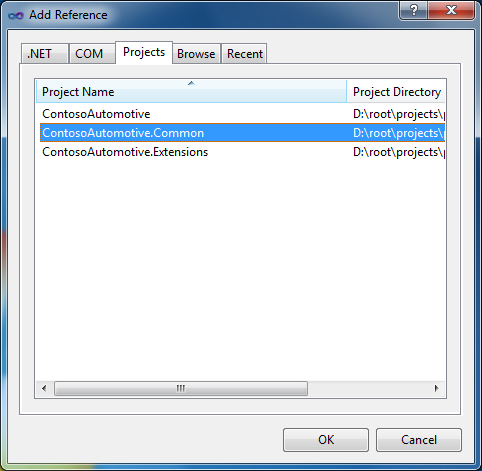
**NOTE: Import** and **Export** values are interconnected with unique Contracts. Contracts serve as uni-directional bridges. An export contract can consist of further metadata used to filter on its discovery. For example, it might indicate a specific capability that the export offers.

1. As you decorate the **CashMaker** class with the **Export** attribute, it will discover composable parts exporting the **ICarQuery** contract, defined in the **ContosoAutomotive.Common** project. You will declare a collection to hold the **Imports** and expose them through a property. MEF's composition container will resolve applicable **Exports** and **Imports** loading the collection for you.
   1. Add the following code to the **CashMaker.xaml.cs** (C#) or **CashMaker.xaml.vb** (Visual Basic) file, just above its constructor.
      1. *(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step10 - ImportedCarQueries CSharp)*
      2. C#
      3. **[ImportMany(AllowRecomposition = true)]**
      4. **public ObservableCollection<ICarQuery> CarQueries { get; set; }**
      5. *(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step10 - ImportedCarQueries VB)*
      6. Visual Basic
      7. **<ImportMany(AllowRecomposition:=True)>**
      8. **Public Property CarQueries() As ObservableCollection(Of ICarQuery)**
      9. **Note:** You can import collections with the **ImportMany** attribute. This means that all instances of the specific contract will be imported from the container.
      10. MEF parts can also support recomposition. This means that as new exports become available in the container, collections are automatically updated with the new set.
   2. Once MEF completes the CarQueries collection import process, you want your application to be notified and so it can take a particular action. To do this, make the CashMaker class to implement the IPartImportsSatisfiedNotification interface as shown in the following code.
      1. C#
      2. [Export]
      3. public partial class CashMaker : Window, **IPartImportsSatisfiedNotification**
      4. {
      5. ...
      6. }
      7. Visual Basic
      8. <Export>
      9. Public Partial Class CashMaker
      10. Inherits Window
      11. **Implements IPartImportsSatisfiedNotification**
      12. ...
      13. End Class
   3. Implement the IPartImportsSatisfiedNotification interface and bind the CarQueries collection to the UI by setting the DataContext property of the commandGrid control. To do this, paste the following **OnImportsSatisfied** method code inside the CashMaker class.
      1. *(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step12 - OnImportsSatisfied CSharp)*
      2. C#
      3. **public void OnImportsSatisfied()**
      4. **{**
      5. **this.commandGrid.DataContext = this.CarQueries;**
      6. **}**
      7. *(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step 12 - OnImportsSatisfied VB)*
      8. Visual Basic
      9. **Public Sub OnImportsSatisfied() Implements IPartImportsSatisfiedNotification. OnImportsSatisfied**
      10. **Me.commandGrid.DataContext = Me.CarQueries**
      11. **End Sub**
      12. **Note:** After MEF’s Container fills the different imports inside the class, it calls the OnImportsSatisfied method if the class implements the **IPartImportsSatisfiedNotification** interface. This method is used as a notification when the container has filled all the imports. It is also called after a recomposition occurs.
   4. Now you can remove the old code used to explicitly create and bind the query instances. To do this, in the CashMaker constructor remove the code used to populate the commandGrid.DataContext property as shown below.
      1. C#
      2. public CashMaker()
      3. {
      4. this.InitializeComponent();
      5. new Thread(() => this.GenerateCars()).Start();
      6. // Removed code
      7. }
      8. Visual Basic
      9. Public Sub New()
      10. Me.InitializeComponent()
      11. Dim thread = New Thread(() => Me.GenerateCars())
      12. thread.Start()
      13. ' Removed code
      14. End Sub

Task 2 – Updating Your Composable Part to Export Contracts

* 1. In this task, you will modify an existing Windows Class Library project with services classes to query a collection of cars. You will use MEF to mark the queries to export a contract to another application.
  2. Add a reference to the MEF library on the **ContosoAutomotive.Extensions** project. To do this:
     1. Select the **ContosoAutomotive.Extensions** project in the **Solution Explorer** and select **Project | Add Reference**… The **Add References** dialog appears.
     2. Select the **.NET** tab and then select the **System.ComponentModel.Composition** component. Click the **OK** button to add a reference to this library.
        1. 
        2. Figure 1
        3. Add a Reference to the MEF library.
  3. Open the **CohoQuery.cs** file (for Visual C# projects) or **CohoQuery.vb** file (for Visual Basic projects) in the **ContosoAutomotive.Extensions** project. Add the following namespace declaration to import the types contained in the MEF library at the top of the **CohoQuery** class definition.
     1. C#
     2. **using System.ComponentModel.Composition;**
     3. Visual Basic
     4. **Imports System.ComponentModel.Composition**
  4. The *CohoQuery* class is one of several parts available in your application. Take into account that you used the *ImportMany* attribute in the *CashMaker* class to pull in only components that implement the **ICarQuery** contract. You will decorate the CohoQuery class indicating that it matches that contract. To do this, decorate the **CohoQuery** class with the **Export** attribute as shown in the code below:
     1. C#
     2. **[Export(typeof(ICarQuery))]**
     3. public class CohoQuery : CarQueryBase
     4. {
     5. ...
     6. }
     7. Visual Basic
     8. **<Export(GetType(ICarQuery))>**
     9. Public Class CohoQuery
     10. Inherits CarQueryBase
     11. ...
     12. End Class
     13. **Note:** You are using the **Export** attribute to define that the **CohoQuery** class is a part of the application and it implements the **ICarQuery** contract, which is the type that the import property inside the **CashMaker** class will look for.
  5. Now, you will apply the same changes to the FabrikamQuery class. To do this, open the **FabrikamQuery.cs** file (for Visual C# projects) or **FabrikamQuery.vb** file (for Visual Basic projects) in the **ContosoAutomotive.Extensions** project. Add the following namespace declaration to import the types contained in the MEF library at the top of the FabrikamQuery class definition.
     1. C#
     2. **using System.ComponentModel.Composition;**
     3. Visual Basic
     4. **Imports System.ComponentModel.Composition**
  6. You will decorate the FabrikamQuery class indicating that it matches the ICarQuery contract. To do this, decorate the **FabrikamQuery** class with the **Export** attribute as shown in the code below:
     1. C#
     2. **[Export(typeof(ICarQuery))]**
     3. public class FabrikamQuery : CarQueryBase
     4. {
     5. ...
     6. }
     7. Visual Basic
     8. **<Export(GetType(ICarQuery))>**
     9. Public Class FabrikamQuery
     10. Inherits CarQueryBase
     11. ...
     12. End Class
  7. At this point, you can start a new instance of the ContosoAutomotive project to check that the two queries are shown in the UI. To do this, right click the **ContosoAutomotive** project and select **Debug** | **Start new instance**.
     1. 
     2. Figure 1
     3. MEF loaded queries in ContosoAutomotive application.
  8. To verify the MEF extensibility, you will add several pre-built query classes. To do this, right-click **ContosoAutomotive.Extensions** in **Solution Explorer**, point to **Add** and select **Existing Item**. In the **Add Existing Item** dialog, browse to the **ContosoAutomotive.Extensions** in **Source\Assets** for the language of your project, hold the **CTRL** key down while you select every file in this folder and click **Add**.
     1. 
     2. Figure 2
     3. Added new queries in ContosoAutomotive.Extensions project (C#)
     4. 
     5. Figure 3
     6. Added new queries in ContosoAutomotive.Extensions project (Visual Basic)
  9. At this point, you can start a new instance of the ContosoAutomotive project to check that the two queries are shown in the UI. To do this, right click the **ContosoAutomotive** project and select **Debug** | **Start new instance**.
     1. 
     2. Figure 4
     3. MEF loaded queries in ContosoAutomotive application

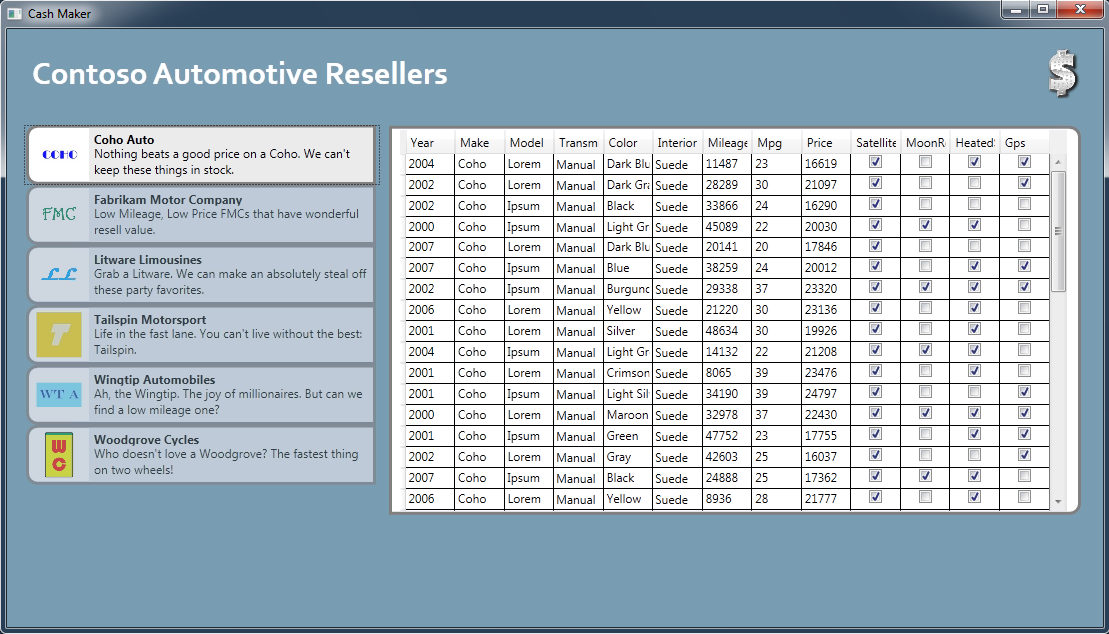
Task 3 – Creating a New Extension

* 1. In this task, you will create a new application extension with a query services class. You will use MEF to Export contracts and a build action to avoid references between projects. Additionally, you can mix CLR languages and create the new project in different CLR language than your main application.
  2. Create a new application extension project named Woodgrove. To do this, right click the solution node in the Solution Explorer and select **Add** | **New Project.** In the Add New Project dialog select **Windows Class Library** project type and name it **ContosoAutomotive.Woodgrove**.
     1. 
     2. Figure 5
     3. New Woodgrove extension project (C#)
     4. 
     5. Figure 6
     6. New Woodgrove extension project (Visual Basic)
  3. Delete the default class file **Class1.cs** (C#) or **Class1.vb** (Visual Basic).
  4. Add a reference to the MEF library on the **ContosoAutomotive.Woodgrove** project. To do this:
     1. Select the **ContosoAutomotive.Woodgrove** project in the **Solution Explorer** and select **Project | Add Reference**… The **Add References** dialog appears.
     2. Select the **.NET** tab and then select the **System.ComponentModel.Composition** component. Click the **OK** button to add a reference to this library.
        1. 
        2. Figure 1
        3. Add a Reference to the MEF library.
  5. Add a reference to the ContosoAutomotive.Common library on the **ContosoAutomotive.Woodgrove** project. To do this:
     1. Select the **ContosoAutomotive.Woodgrove** project in the **Solution Explorer** and select **Project | Add Reference**… The **Add References** dialog appears.
     2. Select the **Projects** tab and then select the **ContosoAutomotive.Common** component. Click the **OK** button to add a reference to this library.
        1. 
        2. Figure 1
        3. Add a Reference to the ContosoAutomotive.Common library.
  6. Create a new class named WoodgroveQuery. To do this, right click the ContosoAutomotive.Woodgrove project and select **Add** | **Class** and name it **WoodgroveQuery**.
  7. Open the **WoodgroveQuery.cs** file (for Visual C# projects) or **WoodgroveQuery.vb** file (for Visual Basic projects) in the **ContosoAutomotive.Woodgrove** project. Add the following namespace declaration to import the types contained in the MEF library at the top of the **WoodgroveQuery** class definition.
     1. *(Code Snippet – Intro to MEF Lab - Ex1 Task3 Step6 – WoodgroveQueryNamespaces CSharp)*
     2. C#
     3. **using System.ComponentModel.Composition;**
     4. **using ContosoAutomotive.Common;**
     5. *(Code Snippet – Intro to MEF Lab - Ex1 Task3 Step6 - WoodgroveQueryNamespaces VB)*
     6. Visual Basic
     7. **Imports System.Collections.Generic**
     8. **Imports System.ComponentModel.Composition**
     9. **Imports ContosoAutomotive.Common**
  8. The *WoodgroveQuery* class matches the same contract and implements the same interface as the previous modified classes. You will implement a new query class similar to the previous one. To do this, replace the class implementation with the following code.
     1. *(Code Snippet – Intro to MEF Lab - Ex1 Task3 Step7 - WoodgroveQueryClass CSharp)*
     2. C#
     3. **[Export(typeof(ICarQuery))]**
     4. **public class WoodgroveQuery : CarQueryBase**
     5. **{**
     6. **public WoodgroveQuery()**
     7. **{**
     8. **this.Name = "Woodgrove Cycles";**
     9. **this.Description = "Who doesn't love a Woodgrove? The fastest thing on two wheels!";**
     10. **this.ImagePath = "Images/woodgrove.jpg";**
     11. **}**
     12. **protected override IEnumerable<Car> RunQuery(IEnumerable<Car> cars)**
     13. **{**
     14. **var results = from c in cars**
     15. **where c.Make == "Woodgrove"**
     16. **&& c.Price >= 50000**
     17. **&& c.Year >= 2000**
     18. **&& c.Transmission == Transmission.Manual**
     19. **&& c.SatelliteRadio == true**
     20. **&& c.Mpg >= 20**
     21. **&& c.Interior == InteriorType.Suede**
     22. **&& c.MoonRoof == false**
     23. **&& c.Mileage <= 40000**
     24. **select c;**
     25. **return results;**
     26. **}**
     27. **}**
     28. *(Code Snippet – Intro to MEF Lab - Ex1 Task3 Step7 - WoodgroveQueryClass VB)*
     29. Visual Basic
     30. **<Export(GetType(ICarQuery))>**
     31. **Public Class WoodgroveQuery**
     32. **Inherits CarQueryBase**
     33. **Public Sub New()**
     34. **Me.Name = "Woodgrove Cycles"**
     35. **Me.Description = "Who doesn't love a Woodgrove? The fastest thing on two wheels!"**
     36. **Me.ImagePath = "Images/woodgrove.jpg"**
     37. **End Sub**
     38. **Protected Overrides Function RunQuery(ByVal cars As IEnumerable(Of Car)) As IEnumerable(Of Car)**
     39. **Dim results = From c In cars \_**
     40. **Where c.Make = "Woodgrove" \_**
     41. **AndAlso c.Price >= 50000 \_**
     42. **AndAlso c.Year >= 2000 \_**
     43. **AndAlso c.Transmission = Transmission.Manual \_**
     44. **AndAlso c.SatelliteRadio = True \_**
     45. **AndAlso c.Mpg >= 20 \_**
     46. **AndAlso c.Interior = InteriorType.Suede \_**
     47. **AndAlso c.MoonRoof = False \_**
     48. **AndAlso c.Mileage <= 40000**
     49. **Return results**
     50. **End Function**
     51. **End Class**
  9. For the new extension project to be discoverable by the catalog, you should place the assembly in the same folder as the main application. You can achieve this in the development environment by configuring the build events. To do this, right click on the **ContosoAutomotive.Woodgrove** project and select **Properties**. In the Properties page, select the **Build Events tab** (for C# projects) or select the **Compile tab** and press the **Build Events…** button (for Visual Basic projects) and enter the following value in the **Post-build event command line**:
     1. Post-Build Command
     2. **copy "$(TargetPath)" "$(SolutionDir)ContosoAutomotive\bin\$(ConfigurationName)"**
     3. **Note:** MEF’s Container will be able to discover the application parts inside the new assembly since you are copying it to the application directory, and a Directory Catalog is looking at it.

# Next Step

* 1. Exercise 1: Verification

### Exercise 1: Verification

* 1. In this verification, you will run the application to verify the new queries are shown in the main application.
  2. Compile the solution (**CTRL+SHIFT+B**).
  3. Set **ContosoAutomotive** as the startup project. In the Solution Explorer, right-click **ContosoAutomotive** and select **Set as startup project**.
  4. Press F5 to run the application. The parent CashMaker window should appear with a list of manufacturers on it.
     1. 
     2. Figure
     3. The CashMaker window shows manufacturers.
  5. Click the close button () in the top right of the **Cash Maker** window to close it.

# Next Step

* 1. Exercise 2: Using Metadata and Lazy-Loading Modules

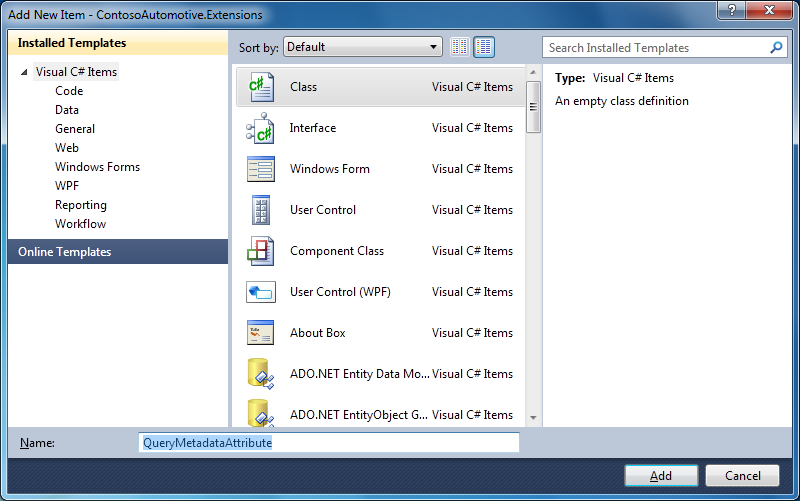
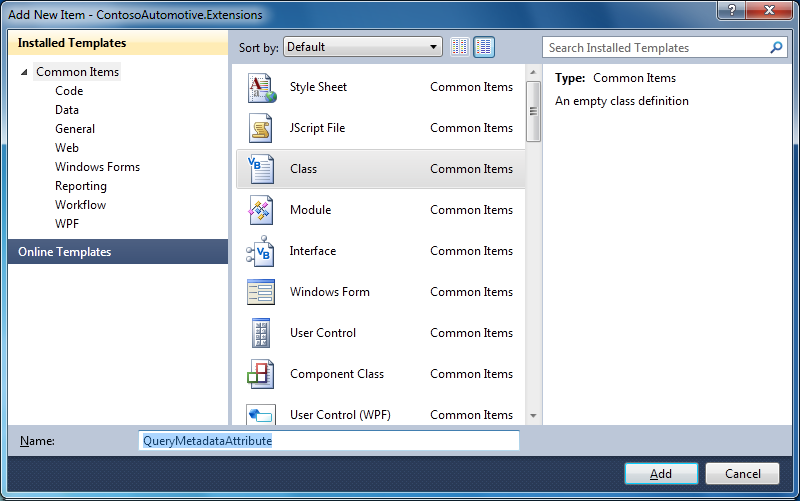
Exercise 2: Using Metadata and Lazy-Loading Modules

* 1. During composition of a part, an import will trigger the instantiation of a part (or parts) that expose the necessary exports required for the original requested part. For some applications, delaying this instantiation and preventing the recursive composition down the graph, may be an important factor to consider, since the creation of a long and complex graph of objects can be expensive and unnecessary.
  2. This is the motivation for MEF to support what we call lazy exports including the Lazy generic type, this type allows delay the load process of the objects until the moment it is first called, minimizing the time required to initialize the application.
  3. In this type of scenarios, associating information with the export becomes critical to inform the importer about the capabilities of a specific implementation of a common contract.
  4. In this exercise, we will tackle both problems in a single scenario.

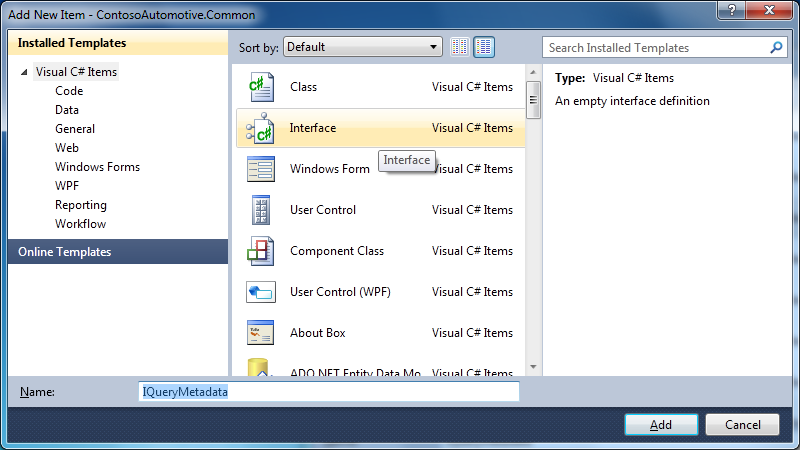
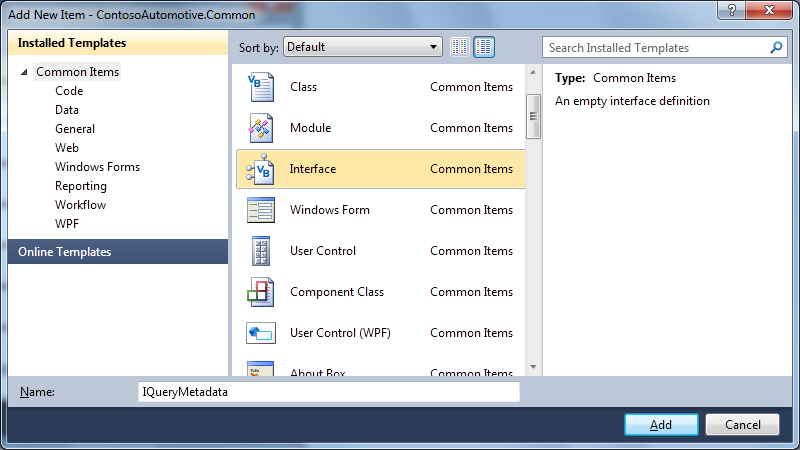
Task 1 – Attaching Metadata to an Export

* 1. In some cases, it is necessary to associate information with exports for a variety of reasons. Commonly you use metadata to explain about the capabilities of a specific implementation of a common contract. In this task, you will modify an existing Query implementation to expose some properties as metadata.
  2. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  3. Open the **ContosoAutomotive.sln** solution file. By default, this file is located in the folder ***Source\Ex2\begin* (choosing the folder that matches the language of your preference.)** Optionally, you can continue working the solution you created in the previous exercise.
  4. You will start adding metadata to the WoodgroveQuery implementation. To do this, open the **WoodgroveQuery.c**s (C#) or **WoodgroveQuery.vb** (Visual Basic) file under the **ContosoAutomotive.Woodgrove** project.
  5. Since you will attach metadata by using attributes, you no longer required the initialization code. Remove the WoodgroveQuery constructor that initializes the Name, Description and ImagePath properties.
  6. To attach metadata to an Export implementation, you decorate the class with the ExportMetadata attribute. In this case, you will attach the same information you removed from the constructor initialization. To do this, decorate the WoodgroveQuery class with the following attributes.
     1. *(Code Snippet – Intro to MEF Lab - Ex2 Task1 Step5 -WoodgroveQueryMetadata CSharp)*
     2. C#
     3. [Export(typeof(ICarQuery))]
     4. **[ExportMetadata("Name", "Woodgrove Cycles")]**
     5. **[ExportMetadata("Description", "Who doesn't love a Woodgrove? The fastest thing on two wheels!")]**
     6. **[ExportMetadata("ImagePath", "Images/woodgrove.jpg")]**
     7. public class WoodgroveQuery : CarQueryBase
     8. {
     9. // Removed code
     10. ...
     11. }
     12. *(Code Snippet – Intro to MEF Lab - Ex2 Task1 Step5 - WoodgroveQueryMetadata VB)*
     13. Visual Basic
     14. <Export(GetType(ICarQuery))>
     15. **<ExportMetadata("Name", "Woodgrove Cycles")>**
     16. **<ExportMetadata("Description", "Who doesn't love a Woodgrove? The fastest thing on two wheels!")>**
     17. **<ExportMetadata("ImagePath", "Images/woodgrove.jpg")>**
     18. Public Class WoodgroveQuery
     19. Inherits CarQueryBase
     20. ' Removed code
     21. ...
     22. End Class
     23. **Note:** You decorate a class with the ExportMetada attribute to include metadata to it. This attribute receives two parameters defining the name and the value of the class’ metadata respectively.

Task 2 – Using a Custom Export Attribute

* 1. In order to attach metadata to an export in a more strongly typed fashion than using the ExportMetadata attribute, you can create your own attribute and decorate it with the Metadata attribute. In this task, you will create a custom metadata attribute and derive it from Export attribute, thus creating a custom Export attribute that also specifies metadata.
  2. Create a new QueryMetadata class that will serve to both define an export and attach metadata at the same time. To do this, right click on the **ContosoAutomotive.Extensions** project and select **Add** | **Class**. In the Add New Item dialog, name the class as **QueryMetadataAttribute** and click OK.
     1. 
     2. Figure
     3. Adding QueryMetadataAttribute class (C#)
     4. 
     5. Figure
     6. Adding QueryMetadataAttribute class (Visual Basic)
  3. Update the QueryMetadataAttribute class to use the MEF library. To do this, add the following statement at the top of the **QueryMetadataAttribute** class definition.
     1. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step2 - Namespaces CSharp)*
     2. C#
     3. **using System.ComponentModel.Composition;**
     4. **using ContosoAutomotive.Common;**
     5. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step2 - Namespaces VB)*
     6. Visual Basic
     7. **Imports System.ComponentModel.Composition**
     8. **Imports ContosoAutomotive.Common**
  4. Make the QueryMetadataAttribute class definition public and decorate it with the **MetadataAttribute** and **AttributeUsage** attributes. In addition, make the class inherits from the **ExportAttribute** class. To do this, replace the default class definition with the following code.
     1. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step3 - QueryMetadataAttribute CSharp)*
     2. C#
     3. **[MetadataAttribute]**
     4. **[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]**
     5. **public class QueryMetadataAttribute : ExportAttribute**
     6. **{**
     7. **}**
     8. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step3 - QueryMetadataAttribute VB)*
  5. Visual Basic
     1. **<MetadataAttribute(), AttributeUsage(AttributeTargets.Class, AllowMultiple:=False)>**
  6. **Public Class QueryMetadataAttribute**
  7. **Inherits ExportAttribute**
  8. **End Class**
     1. **Note:** Both attributes, MetadataAttribute and AttributeUsage, are required to be included as decorators of the typed Metadata class.
     2. MetadataAttribute defines that the class can be used to define typed metadata of another class.
     3. AttributeUsage sets where the attribute can be placed. In this exercise, the attribute can decorate a class, and cannot be used more than one time on the same class.
  9. Add the required metadata properties and setup the inherited constructor to export the ICarQuery interface explicitly. To do this, insert the following code inside the QueryMetadataAttribute class.
     1. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step4 - QueryMetadataImplementation CSharp)*
     2. C#
     3. [MetadataAttribute]
     4. [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
     5. public class QueryMetadataAttribute : ExportAttribute
     6. {
     7. **public QueryMetadataAttribute()**
     8. **: base(typeof(ICarQuery))**
     9. **{**
     10. **}**
     11. **public string Name { get; set; }**
     12. **public string Description { get; set; }**
     13. **public string ImagePath { get; set; }**
     14. }
     15. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step4 - QueryMetadataImplementation VB)*
     16. Visual Basic
     17. <MetadataAttribute(), AttributeUsage(AttributeTargets.Class, AllowMultiple:=False)>
     18. Public Class QueryMetadataAttribute
     19. Inherits ExportAttribute
     20. **Public Sub New()**
     21. **MyBase.New(GetType(ICarQuery))**
     22. **End Sub**
     23. **Public Property Description As String**
     24. **Public Property ImagePath As String**
     25. **Public Property Name As String**
     26. End Class
     27. **Note:** Inheritance from ExportAttribute and calling the ExportAttribute class’ constructor passing the ICarQuery type automatically sets that the contract implemented by the class is the ICarQuery. This allows avoiding specifying it when you decorate the class as you did when you used the ExportAttribute directly.
  10. Now, you can attach metadata and export a contract using the **QueryMetadata** attribute in your classes removing both the Export decorating attribute and the initialization constructor. To do this, remove the constructor and replace the class definition in the **CohoQuery** class.
      1. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step5 - CohoQueryMetadata CSharp)*
      2. C#
      3. **[QueryMetadata(Name = "Coho Auto",**
      4. **Description = "Nothing beats a good price on a Coho. We can't keep these things in stock.",**
      5. **ImagePath = "Images/coho.jpg")]**
      6. public class CohoQuery : CarQueryBase
      7. {
      8. // Removed code
      9. ...
      10. }
      11. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step5 - CohoQueryMetadata VB)*
  11. Visual Basic
  12. **<QueryMetadata(Name:="Coho Auto",**
  13. **Description:="Nothing beats a good price on a Coho. We can't keep these things in stock.",**
  14. **ImagePath:="Images/coho.jpg")>**
  15. Public Class CohoQuery
  16. Inherits CarQueryBase
      1. // Removed code
      2. ...
  17. End Class
  18. You will repeat the same process on the FabrikamQuery class. To do this, remove the constructor and replace the class definition in the **FabrikamQuery** class.
      1. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step6 - FabrikamQueryMetadata CSharp)*
      2. C#
      3. **[QueryMetadata(Name = "Fabrikam Motor Company",**
      4. **Description = "Low Mileage, Low Price FMCs that have wonderful resell value.",**
      5. **ImagePath = "Images/fabrikam.jpg")]**
      6. public class FabrikamQuery: CarQueryBase
      7. {
      8. // Removed code
      9. ...
      10. }
      11. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step6 - FabrikamQueryMetadata VB)*
  19. Visual Basic
  20. **<QueryMetadata(Name:="Fabrikam Motor Company",**
  21. **Description:="Low Mileage, Low Price FMCs that have wonderful resell value.",**
  22. **ImagePath:="Images/fabrikam.jpg")>**
  23. Public Class FabrikamQuery
  24. Inherits CarQueryBase
      1. // Removed code
      2. ...
  25. End Class
  26. You will repeat the same process on the LitwareQuery class. To do this, remove the constructor and replace the class definition in the **LitwareQuery** class.
      1. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step7 - LitwareQueryMetadata CSharp)*
      2. C#
      3. **[QueryMetadata(Name = "Litware Limousines",**
      4. **Description = "Grab a Litware. We can make an absolutely steal off these party favorites.",**
      5. **ImagePath = "Images/litware.jpg")]**
      6. public class LitwareQuery: CarQueryBase
      7. {
      8. // Removed code
      9. ...
      10. }
      11. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step7 - LitwareQueryMetadata VB)*
  27. Visual Basic
  28. **<QueryMetadata(Name:="Litware Limousines",**
  29. **Description:="Grab a Litware. We can make an absolutely steal off these party favorites.",**
  30. **ImagePath:="Images/litware.jpg")>**
  31. Public Class LitwareQuery
  32. Inherits CarQueryBase
      1. // Removed code
      2. ...
  33. End Class
  34. You will repeat the same process on the TailspinQuery class. To do this, remove the constructor and replace the class definition in the **TailspinQuery** class.
      1. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step8 - TailspinQueryMetadata CSharp)*
      2. C#
      3. **[QueryMetadata(Name = "Tailspin Motorsport",**
      4. **Description = "Life in the fast lane. You can't live without the best: Tailspin.",**
      5. **ImagePath = "Images/tailspin.jpg")]**
      6. public class TailspinQuery: CarQueryBase
      7. {
      8. // Removed code
      9. ...
      10. }
      11. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step8 - TailspinQueryMetadata VB)*
  35. Visual Basic
  36. **<QueryMetadata(Name:="Tailspin Motorsport",**
  37. **Description:="Life in the fast lane. You can't live without the best: Tailspin.",**
  38. **ImagePath:="Images/tailspin.jpg")>**
  39. Public Class TailspinQuery
  40. Inherits CarQueryBase
      1. // Removed code
      2. ...
  41. End Class
  42. You will repeat the same process on the WingtipQuery class. To do this, remove the constructor and replace the class definition in the **WingtipQuery** class.
      1. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step9 - WingtipQueryMetadata CSharp)*
      2. C#
      3. **[QueryMetadata(Name = "Wingtip Automobiles",**
      4. **Description = "Ah, the Wingtip. The joy of millionaires. But can we find a low mileage one?",**
      5. **ImagePath = "Images/wingtip.jpg")]**
      6. public class WingtipQuery: CarQueryBase
      7. {
      8. // Removed code
      9. ...
      10. }
      11. *(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step9 - WingtipQueryMetadata VB)*
  43. Visual Basic
  44. **<QueryMetadata(Name:="Wingtip Automobiles",**
  45. **Description:="Ah, the Wingtip. The joy of millionaires. But can we find a low mileage one?",**
  46. **ImagePath:="Images/wingtip.jpg")>**
  47. Public Class WingtipQuery
  48. Inherits CarQueryBase
      1. // Removed code
      2. ...
  49. End Class

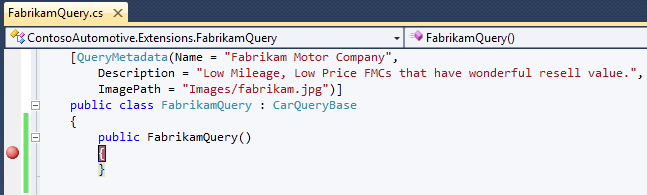
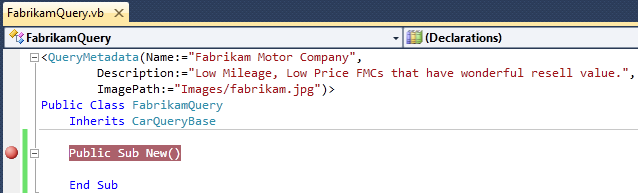
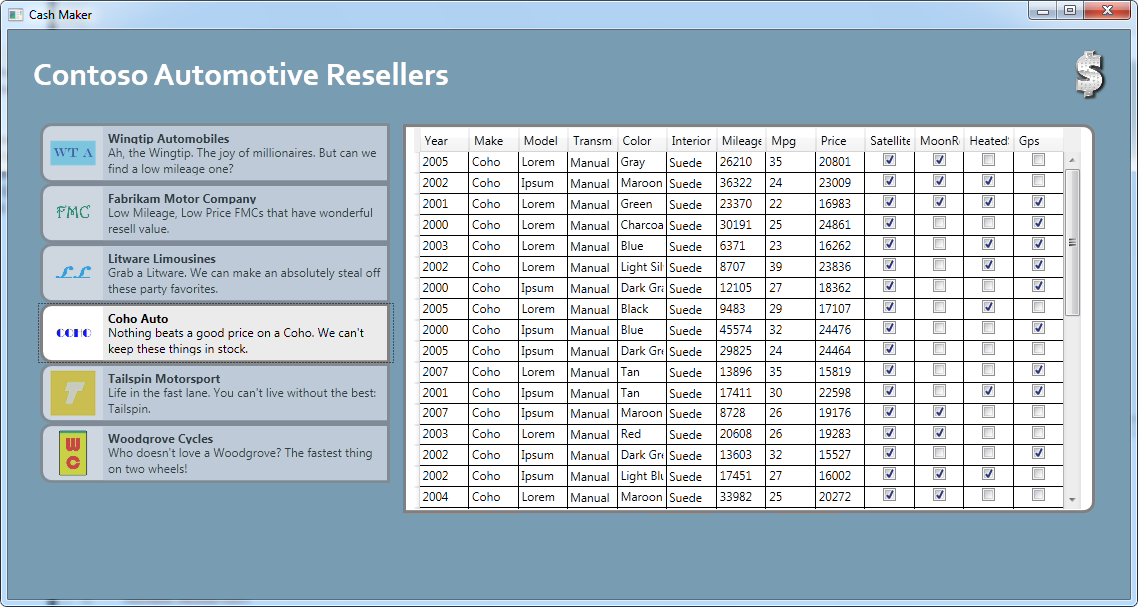
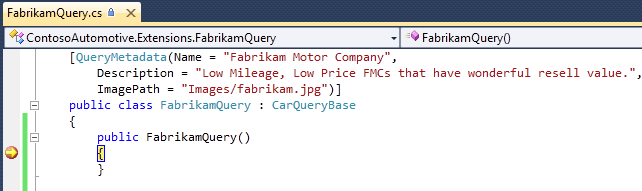
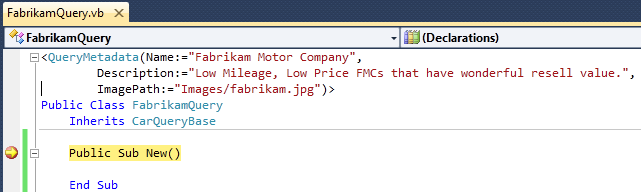
Task 3 – Importing Metadata

* 1. In this task, you will modify the way import occurs to let importers access the metadata attached to the exports.
  2. To access metadata in a strongly typed fashion create a metadata view by defining an interface with matching read only properties (names and types). To do this, right click on the **ContosoAutomotive.Common** project and select **Add** | **New Item**. In the Add New Item dialog select Interface type and name it **IQueryMetadata**.
     1. 
     2. Figure
     3. Adding IQueryMetadata interface (C#)
     4. 
     5. Figure
     6. Adding IQueryMetadata interface (Visual Basic)
  3. Make the interface public and add the desired metadata properties as read only. To do this, replace the default interface implementation with the following code.
     1. *(Code Snippet – Intro to MEF Lab - Ex2 Task3 Step2 - IQueryMetadata CSharp)*
     2. C#
     3. **public interface IQueryMetadata**
     4. **{**
     5. **string Name { get; }**
     6. **string Description { get; }**
     7. **string ImagePath { get; }**
     8. **}**
     9. *(Code Snippet – Intro to MEF Lab - Ex2 Task3 Step2 - IQueryMetadata VB)*
     10. Visual Basic
     11. **Public Interface IQueryMetadata**
     12. **ReadOnly Property Name As String**
     13. **ReadOnly Property Description As String**
     14. **ReadOnly Property ImagePath As String**
     15. **End Interface**
  4. Since you are no longer using the Name, Description and ImagePath properties you can remove them from the CarQueryBase class. To do this, open the **CarQueryBase.cs** (C#) or **CarQueryBase.vb** (Visual Basic) and remove the **Name**, **Description** and **ImagePath** properties and their associated member fields.
  5. Modify the main application to retrieve the query information from metadata instead of retrieving it from the query instance. To do this, open the CashMaker class by right clicking on the **CashMaker.xaml** file under the **ContosoAutomotive** project and select **View Code** in the contextual menu.
  6. Now you can start importing using the type System.Lazy<T, TMetadata> where T is the contract type and TMetadata is the interface you have created. To do this, replace the CarQueries field definition inside the CashMaker class with the following code.
     1. *(Code Snippet – Intro to MEF Lab - Ex2 Task3 Step5 – CarQueries Property CSharp)*
     2. C#
     3. **[ImportMany(AllowRecomposition = true)]**
     4. **public ObservableCollection<Lazy<ICarQuery, IQueryMetadata>> CarQueries { get; set; }**
     5. *(Code Snippet – Intro to MEF Lab - Ex2 Task3 Step5 – CarQueries Property VB)*
     6. Visual Basic
     7. **<ImportMany(AllowRecomposition:=True)>**
     8. **Public Property CarQueries As ObservableCollection(Of Lazy(Of ICarQuery, IQueryMetadata))**
     9. **Note:** As it was explained, Lazy type allows delaying the object instantiation until it is used, specifically, the object will be created when the **Value** property of the Lazy object will be called.
     10. You are declaring the Lazy type using the contract type and the metadata type. The last one is not required but you are defining it because you want to access the metadata information avoiding the object instantiation.
  7. Since you have modified the bound collection, you should adapt the SelectionChange event handler to deal with a Lazy Query object. To do this, replace the bolded lines of code below to use a proper approach to run queries.
     1. *(Code Snippet – Intro to MEF Lab - Ex2 Task3 Step6 – CommandList Event CSharp)*
     2. C#
     3. private void commandList\_SelectionChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e)
     4. {
     5. if (this.SearchEnabled)
     6. {
     7. this.DisableSearch();
     8. var thread = new Thread(() =>
     9. {
     10. if (e.AddedItems.Count > 0)
     11. {
     12. **var lazyQuery = e.AddedItems[0] as Lazy<ICarQuery, IQueryMetadata>;**
     13. **if (lazyQuery != null)**
     14. **{**
     15. **lazyQuery.Value.Run(this.cars, true);**
     16. **}**
     17. }
     18. this.EnableSearch();
     19. });
     20. thread.Start();
     21. thread.Join();
     22. }
     23. }
     24. *(Code Snippet – Intro to MEF Lab - Ex2 Task3 Step6 – CommandList Event VB)*
     25. Visual Basic
     26. Private Sub commandList\_SelectionChanged(ByVal sender As System.Object, ByVal e As System.Windows.Controls.SelectionChangedEventArgs)
     27. If (Me.SearchEnabled) Then
     28. Me.DisableSearch()
     29. Dim thread = New Thread(Sub()
     30. If (e.AddedItems.Count > 0) Then
     31. **Dim lazyQuery = TryCast(e.AddedItems(0), Lazy(Of ICarQuery, IQueryMetadata))**
     32. **If (lazyQuery IsNot Nothing) Then**
     33. **lazyQuery.Value.Run(Me.cars, True)**
     34. **End If**
     35. End If
     36. Me.EnableSearch()
     37. End Sub)
     38. thread.Start()
     39. thread.Join()
     40. End If
     41. End Sub
  8. Modify the UI XAML code as well to adapt the UI to the new lazy model. To do this, right click on the **CashMaker.xaml** file and select **View Designer** in the contextual menu. For simplicity, expand on the XAML tab to show only the markup code.
  9. Scroll down to the **<DataTemplate>** element with the **QueryItem** key and adapt the Path property of the Binding objects to retrieve the **Name**, **Description** and **ImagePath** from the **Metadata** property of the Lazy objects. To do this, replace the highlighted bolded lines of markup code as shown below.
     1. XAML
     2. <DataTemplate x:Key="QueryItem">
     3. <Border Style="{StaticResource RoundedBorder}" Width="350" Height="60" Margin="0" Padding="0">
     4. <Grid>
     5. <Grid.ColumnDefinitions>
     6. <ColumnDefinition Width="60" />
     7. <ColumnDefinition Width="\*" />
     8. </Grid.ColumnDefinitions>
     9. <Image Grid.Column="0" VerticalAlignment="Center" HorizontalAlignment="Center" Width="45" Height="45" Source="**{Binding Path=Metadata.ImagePath }**"></Image>
     10. <Grid Grid.Column="1" Background="#E8ECED" >
     11. <Grid>
     12. <Grid.RowDefinitions>
     13. <RowDefinition Height="\*" />
     14. <RowDefinition Height="2\*" />
     15. </Grid.RowDefinitions>
     16. <TextBlock Grid.Row="0" Margin="5 5 5 0" FontWeight="Bold" Text="**{Binding Path=Metadata.Name }**" VerticalAlignment="Center" />
     17. <TextBlock Grid.Row="1" Margin="5 0 5 5" Text="**{Binding Path=Metadata.Description }"** VerticalAlignment="Center" TextWrapping="Wrap"/>
     18. </Grid>
     19. </Grid>
     20. </Grid>
     21. </Border>
     22. </DataTemplate>
     23. **Note:** The Lazy type expose a property called **Metadata**, which is the metadata type defined on the Lazy type declaration (in this case, IQueryMetadata). This property is used in the XAML above to access the metadata information of the different query items instead of use the object properties as it was implemented in the previous exercise.
  10. Scroll down to the **<ContentControl>** element at the bottom of the file and adapt the Path property of the Binding object to retrieve the **Results** object from the **Value** instance of the Lazy objects. To do this, replace the bolded lines of markup code as shown below.
      1. XAML
      2. <ContentControl Width="700" Margin="1 1 5 1" VerticalAlignment="Top">
      3. <Border Padding="1 1 5 1" Style="{StaticResource RoundedBorderRight}" Grid.Row="0" >
      4. <DataGrid Name="Results" ItemsSource="**{Binding Path=Value.Results}**" AutoGenerateColumns="True" ColumnWidth="\*" BorderThickness="0" Width="667" Height="381" />
      5. </Border>
      6. </ContentControl>
      7. **Note:** As it was explained for the Lazy’s Metadata property. The **Value** property matches the contract type on the lazy type definition (in this case, ICarQuery) and allows accessing the different object’s properties and methods.

# Next Step

* 1. Exercise 2: Verification

### Exercise 2: Verification

* 1. In this verification, you will run the application to verify that all the queries are shown in the main application. In addition, you will verify the lazy behavior of the query objects.
  2. In order to verify the lazy instantiation of an object you will add an empty constructor to any of the ICarQuery implementations and set a breakpoint on it to check when it is being instantiated. To do this, open the **FabrikamQuery.cs** (C#) or **FabrikamQuery.vb** (Visual Basic), add a new constructor to it and set a breakpoint by pressing **F9** as shown in the following screenshot.
     1. 
     2. Figure 12
     3. FabrikamQuery constructor breakpoint (C#)
     4. 
     5. Figure 13
     6. FabrikamQuery constructor breakpoint (C#)
  3. Compile the solution (**CTRL+SHIFT+B**).
  4. Set **ContosoAutomotive** as the startup project. In the Solution Explorer, right-click **ContosoAutomotive** and select **Set as startup project**.
  5. Press F5 to run the application. The parent CashMaker window should appear with a list of manufacturers on it. You should note the FabrikamQuery constructor has not been executed.
     1. 
     2. Figure 14
     3. The CashMaker window shows manufacturers.
  6. Click on the Fabrikam Motor Company icon on the list of manufacturers in the Cash Maker application. Now, the code will stop in the breakpoint you have set in the FabrikamQuery constructor, because the instance is being created on demand.
     1. 
     2. Figure 15
     3. FabrikamQuery constructor breakpoint (C#)
     4. 
     5. Figure 16
     6. FabrikamQuery constructor breakpoint (C#)
  7. Click the close button () in the top right of the **Cash Maker** window to close it.
  8. Remove the breakpoint and the constructor you just created in the FabrikamQuery class.

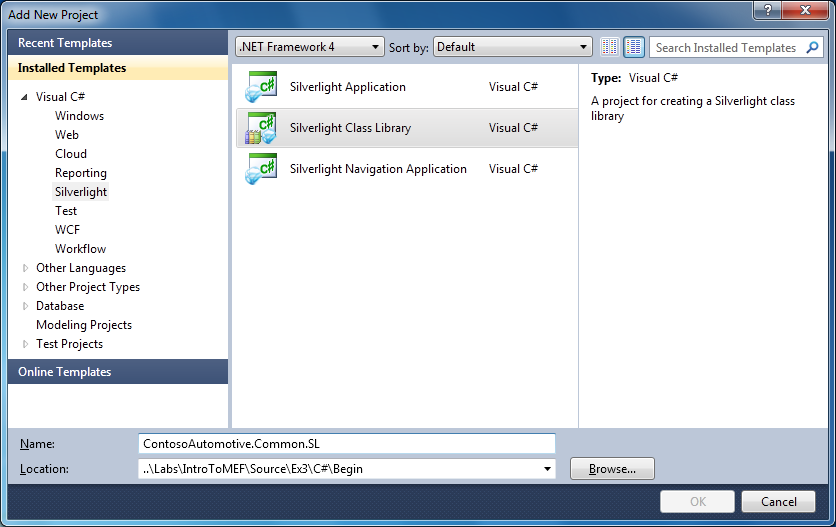
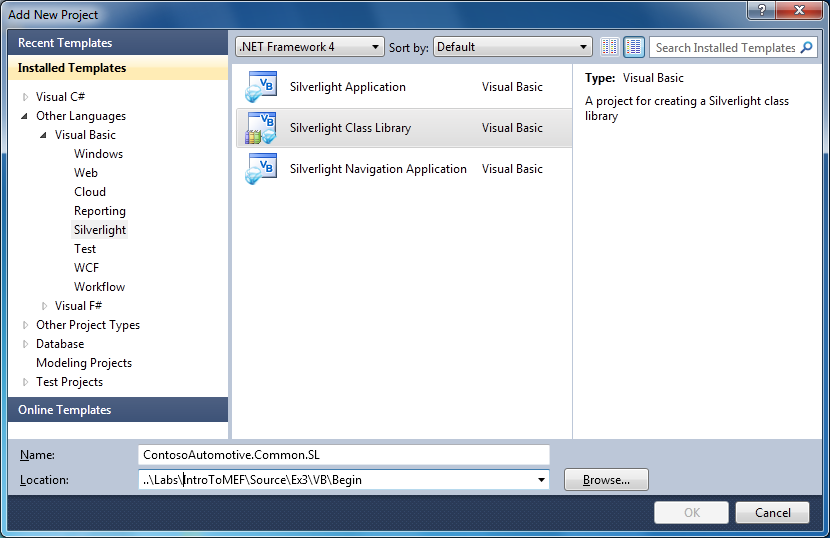
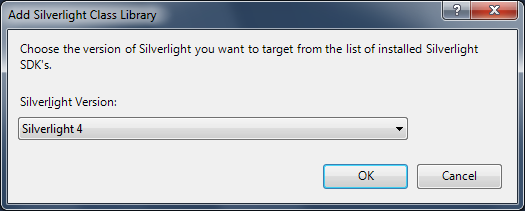
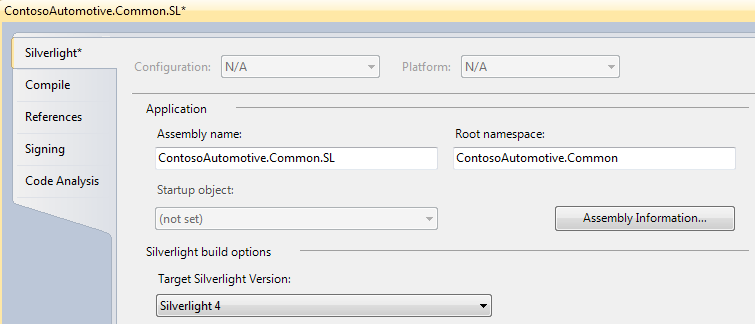
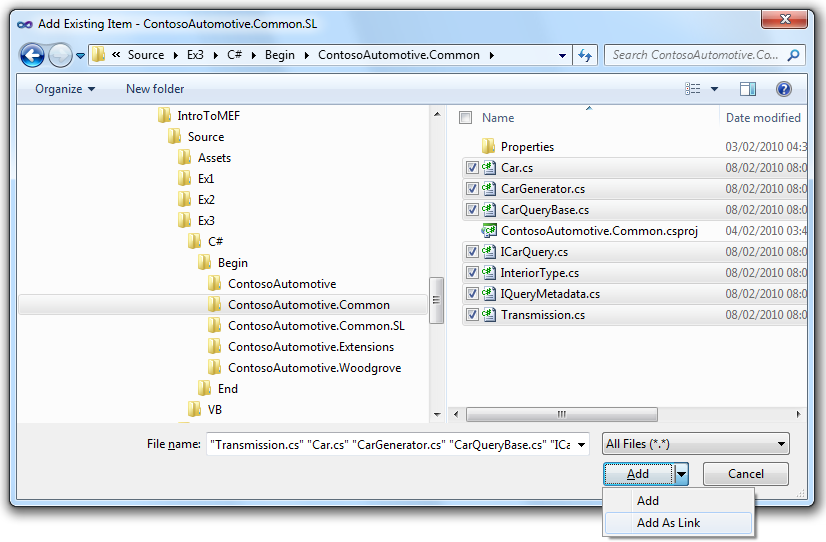
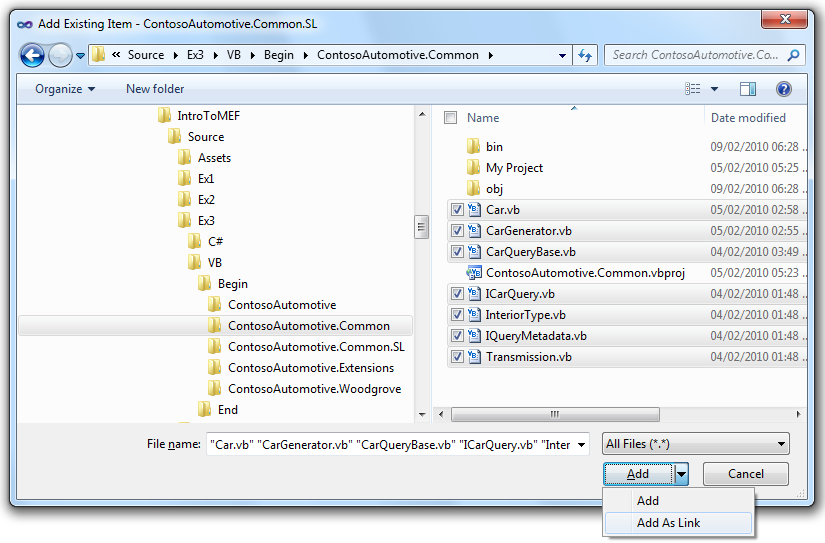
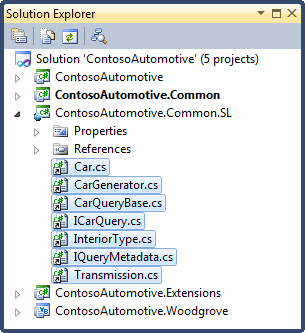
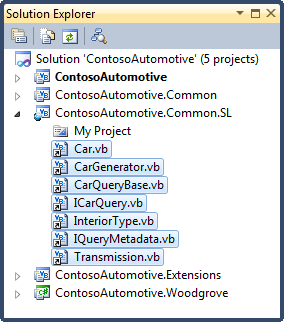
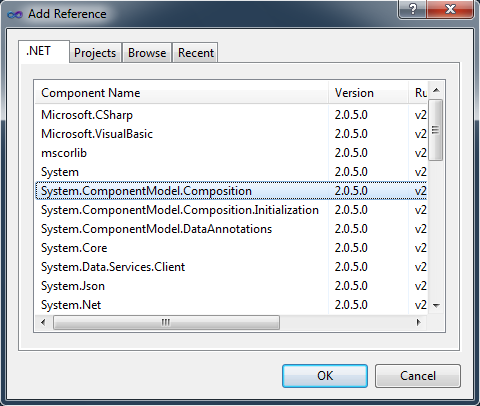
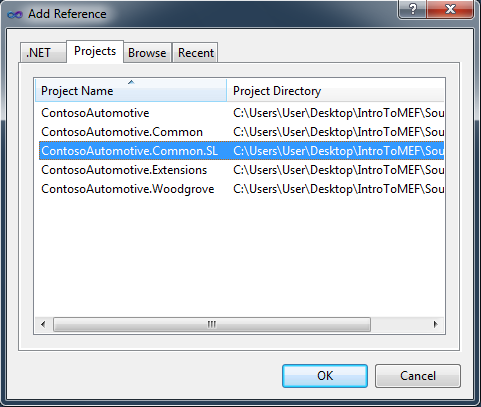
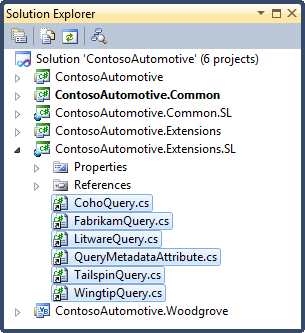
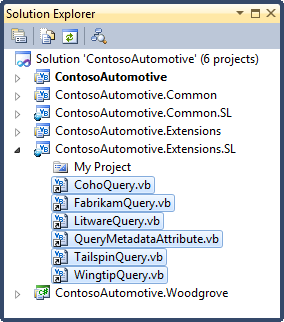
# Next Step

* 1. Exercise 3: Using MEF and Silverlight 4

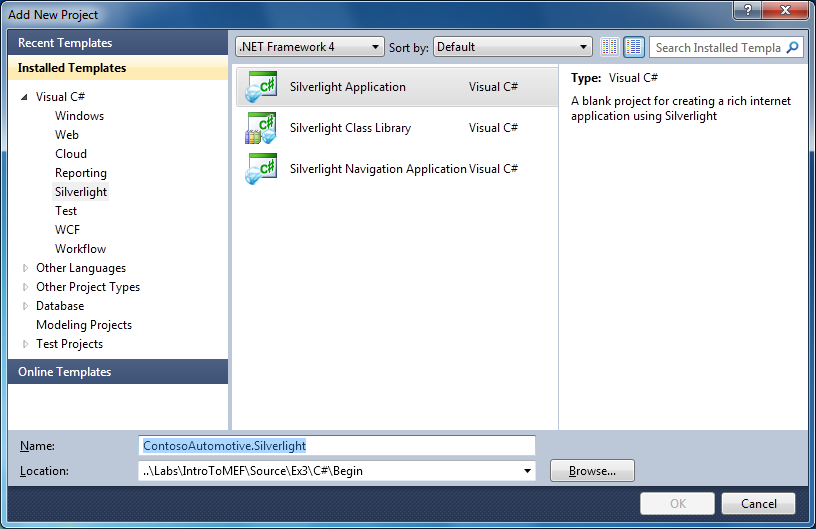
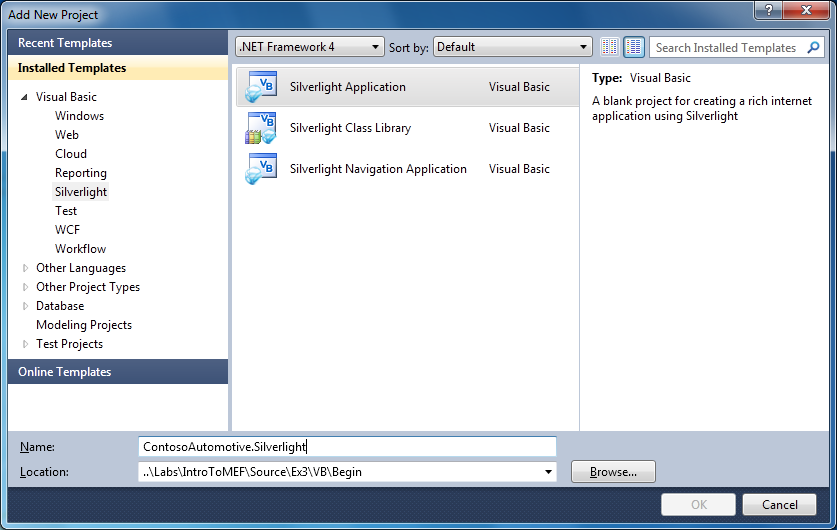
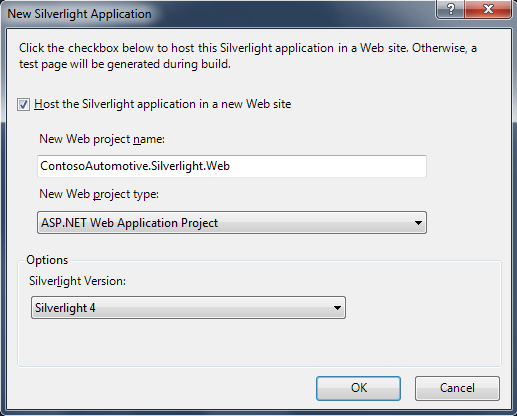
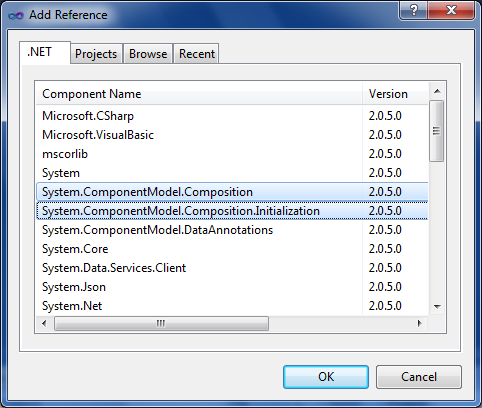
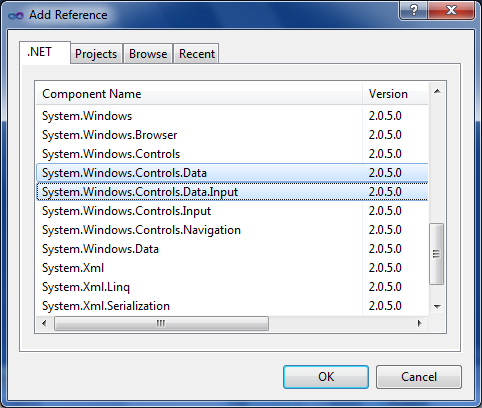
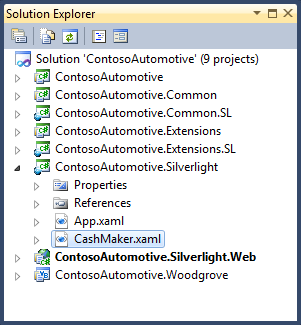
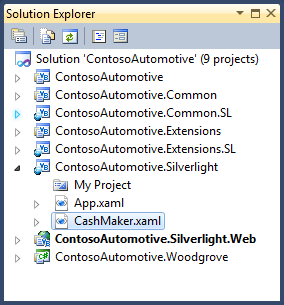
Exercise 3: Using MEF and Silverlight 4

* 1. Latest version of Silverlight 4 will include MEF out of the box. This means all the functionality described in the previous exercises are available for Silverlight 4 applications.
  2. Some minor tweaks are required because of the different deployment model of a Silverlight application.
  3. In this exercise, you will go through the creation of a Silverlight version of the Cash Maker application reusing most of the code and logic already present in the WPF version of the application.

Task 1 – Convert Extensions to Silverlight

* 1. In this task, you will convert all the Class Library projects used by the WPF version of the Cash Maker application to Silverlight. You will reuse all the code present in the WPF version by adding links to the existing files.
  2. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  3. Open the **ContosoAutomotive.sln** solution file. By default, this file is located in the folder ***Source\Ex3\begin* (choosing the folder that matches the language of your preference.)** Optionally, you can continue working the solution you created in the previous exercise.
  4. Because a Silverlight 4 application can only reference Silverlight Class Libraries, you will replicate all the extension and contract definitions for Silverlight 4. You will start adding the Silverlight 4 version of Contoso.Automotive.Common library. To do this, right click on the ContosoAutomotive solution node and select **Add** | **New Project**. In the Add New Project dialog, select the **Silverlight Class Library** project type and enter **ContosoAutomotive.Common.SL** in the Name field.
     1. 
     2. Figure 17
     3. Add ContosoAutomotive.Common.SL Silverlight class library project (C#)
     4. 
     5. Figure 18
     6. Add ContosoAutomotive.Common.SL Silverlight class library project (Visual Basic)
  5. In the New Silverlight Class Library options dialog accept the default settings. Ensure **Silverlight 4** is selected in the **Silverlight Version** list.
     1. 
     2. Figure 19
     3. New Silverlight Class Library options dialog
  6. Only in **Visual Basic**, change the root namespace of the ContosoAutomotive.Common.SL project to ContosoAutomotive.Common. To do this, right click the ContosoAutomotive.Common.SL project node and select Properties. In the **Properties** page, select the **Silverlight** tab and change the **Root namespace** field value to **ContosoAutomotive.Common**.
     1. 
     2. Figure 20
     3. Change Root namespace to ContosoAutomotive.Common (Visual Basic)
  7. Remove the auto-generated class file. To do this, right click on the **Class1.cs** (C#) or the **Class1.vb** (Visual Basic) file and select **Delete**.
  8. You will reuse all the code already present in the existing projects by linking the previous class files in the new projects. To do this, right click the ContosoAutomotive.Common.SL project node and select **Add** | **Existing Item**. In the Add Existing Item dialog browse one folder up and select all the class files inside the **ContosoAutomotive.Common** folder. Then, from the Open button drop-down list, select **Add As Link**.
     1. 
     2. Figure 21
     3. Add linked files to the ContosoAutomotive.Common.SL project (C#)
     4. 
     5. Figure 22
     6. Add linked files to the ContosoAutomotive.Common.SL project (Visual Basic)
  9. After adding the link to the class files, your solution should like the following image.
     1. 
     2. Figure 23
     3. Linked files in the ContosoAutomotive.Common.SL project (C#)
     4. 
     5. Figure 24
     6. Linked files in the ContosoAutomotive.Common.SL project (Visual Basic)
  10. Repeat the same process for the ContosoAutomotive.Extensions project creating the ContosoAutomotive.Extensions.SL Silverlight Class Library project. To do this, right click the ContosoAutomotive solution node and select **Add** | **New Project**. In the Add New Project dialog, select the **Silverlight Class Library** project type and enter **ContosoAutomotive.Extensions.SL** in the name field. Remember to select **Silverlight 4** in the Silverlight Version list.
  11. Add a reference to the MEF library on the ContosoAutomotive.Extensions.SL project. To do this:
      1. Select the **ContosoAutomotive.Extensions.SL** project in the **Solution Explorer** and select **Project | Add Reference**… The **Add References** dialog appears.
      2. Select the **.NET** tab and then select the **System.ComponentModel.Composition** component. Click the **OK** button to add a reference to this library.
         1. 
         2. Figure 1
         3. Add a Reference to the MEF library.
  12. Also, add a reference to the ContosoAutomotive.Common.SL library on the same project. To do this:
      1. Select the **ContosoAutomotive.Extensions.SL** project in the **Solution Explorer** and select **Project | Add Reference**… The **Add References** dialog appears.
      2. Select the **Projects** tab and then select the **ContosoAutomotive.Common.SL** component. Click the **OK** button to add a reference to this library.
         1. 
         2. Figure 1
         3. Add a Reference to the ContosoAutomotive.Common.SL library.
  13. Remove the auto-generated class file **Class1.cs** (C#) or **Class1.vb** (Visual Basic) and create a link to all the class files in the **ContosoAutomotive.Extensions** folder.
      1. 
      2. Figure 25
      3. Linked files in the ContosoAutomotive.Common.SL project (C#)
      4. 
      5. Figure 26
      6. Linked files in the ContosoAutomotive.Common.SL project (Visual Basic)

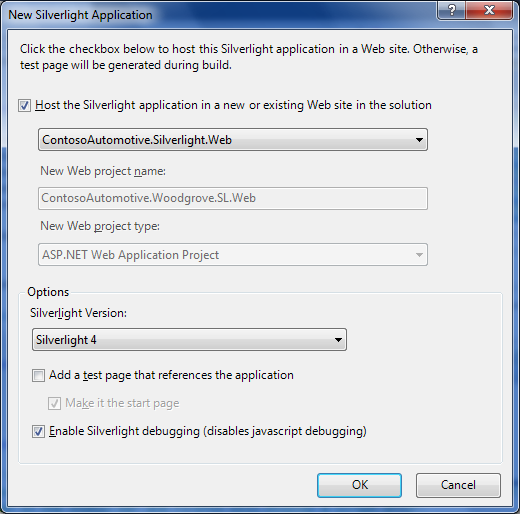
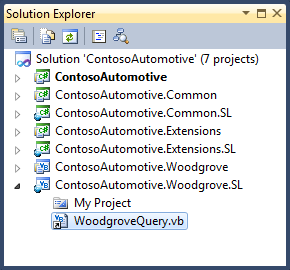
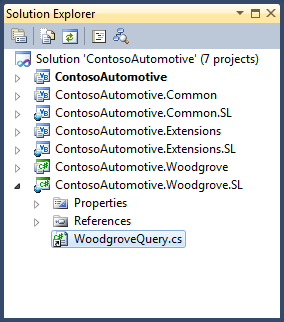
Task 2 – Create Silverlight Cash Maker UI

* 1. In this task, you will create a new Silverlight application to reproduce the same user experience we found in the WPF Cash Maker application.
  2. You will start adding a new Silverlight 4 application project to your solution. To do this, right click on the ContosoAutomotive solution node and selecting **Add** | **New project**. In the Add New Project dialog, select **Silverlight Application** as the project type and enter ContosoAutomotive.Silverlight in the project name field.
     1. 
     2. Figure 27
     3. Adding Silverlight 4 project (C#)
     4. 
     5. Figure 28
     6. Adding Silverlight 4 project (Visual Basic)
  3. In the New Silverlight Application options dialog accept all defaults. Ensure **Silverlight 4** is selected in the **Silverlight Version** list.
     1. 
     2. Figure 29
     3. New Silverlight Application options dialog
  4. Copy all the images files from the ContosoAutomotive WPF project into the Silverlight version. To do this, right click on the **Images** folder in the **ContosoAutomotive** project and select **Copy**. Then, right click the **ContosoAutomotive.Silverlight** project and select **Paste**.
  5. Add a reference to the MEF library on the ContosoAutomotive.Silverlight project. You will include the Initialization assembly as well. To do this:
     1. Select the **ContosoAutomotive.Silverlight** project in the **Solution Explorer** and select **Project | Add Reference**… The **Add References** dialog appears.
     2. Select the **.NET** tab and then select the **System.ComponentModel.Composition** and the **System.ComponentModel.Composition.Initialization** components. Click the **OK** button to add these references to the project.
        1. 
        2. Figure 1
        3. Add a Reference to the MEF library.
  6. Add a reference to the System.Windows.Controls.Data and System.Windows.Controls.Data.Input on the ContosoAutomotive.Silverlight project. The Cash Maker UI will use these libraries to show data. To do this:
     1. Select the **ContosoAutomotive.Silverlight** project in the **Solution Explorer** and select **Project | Add Reference**… The **Add References** dialog appears.
     2. Select the **.NET** tab and then select the **System.Windows.Controls.Data** and the **System.Windows.Controls.Data.Input** components. Click the **OK** button to add a reference to this library.
        1. 
        2. Figure 1
        3. Add a Reference to the System.Windows.Controls.Data and Data.Input libraries.
  7. Also, add a reference to the ContosoAutomotive.Common.SL and ContosoAutomotive.Extensions.SL the library on the ContosoAutomotive.Silverlight project. To do this, right click on the project and select Add Reference. In the **Add Reference** dialog, select the **Projects** tab and choose the **ContosoAutomotive.Common.SL** and **ContosoAutomotive.Extensions.SL** libraries.
  8. Remove the default MainPage.xaml file in order to replace it with the Cash Maker UI. To do this, right click on the **MainPage.xaml** file and select **Delete**.
  9. Include the prebuilt version of the Cash Maker UI located in the Assets folder. To do this, right click the ContosoAutomotive.Silverlight project node and select **Add** | **Existing item**. In the Add Existing Item dialog, browse the **Assets** folder and selecting the folder that matches the language of your choice add the **CashMaker.xaml** file. The code behind file will be automatically included as well.
     1. 
     2. Figure 30
     3. Add prebuilt Silverlight Cash Maker UI (C#)
     4. 
     5. Figure 31
     6. Add prebuilt Silverlight Cash Maker UI (Visual Basic)
     7. **Note:** The only differences between the version that you were using on the previous step and the recently added from the Assets folder are that the last one uses the Silverlight control (instead of the WPF version of those), and does not have implemented some code that you will add in the following steps.

Task 3 – Update Your Silverlight Application to Load Composable Parts

* 1. In this task, you will modify a Silverlight 4 application to load composable parts using MEF. You will use the same approach used in the WPF version of the Cash Maker application.
  2. Open the App class in Code View. To do this, right-click on the **App.xaml** file in the Solution explorer and select **View Code**.
  3. Update the App class to use the MEF library. To do this, add the following statement on top of the using clause list above the **App** class definition.
     1. C#
     2. **using System.ComponentModel.Composition.Hosting;**
     3. Visual Basic
     4. **Imports System.ComponentModel.Composition.Hosting**
  4. You now will modify the Application class to load the new CashMaker UI. In addition, you will prepare you application to load any available Composable Part from the Silverlight package. To do this, replace the code inside the Application\_Startup method with the following code.
     1. *(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step3 - Application\_Startup CSharp)*
     2. C#
     3. private void Application\_Startup(object sender, StartupEventArgs e)
     4. {
     5. **var catalog = new AggregateCatalog(new DeploymentCatalog());**
     6. **var container = new CompositionContainer(catalog);**
     7. **this.RootVisual = container.GetExportedValue<CashMaker>();**
     8. }
     9. *(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step3 - Application\_Startup VB)*
     10. Visual Basic
     11. Private Sub Application\_Startup(ByVal o As Object, ByVal e As StartupEventArgs) Handles Me.Startup
     12. **Dim catalog = New AggregateCatalog(New DeploymentCatalog())**
     13. **Dim container = New CompositionContainer(catalog)**
     14. **Me.RootVisual = container.GetExportedValue(Of CashMaker)()**
     15. End Sub
     16. **Note:** As you did for the WPF version, you create a container which discovers the composable parts through a Catalog. The main window is also retrieved using the GetExportedValue of the container.
     17. **Note:** Silverlight applications runs and are deployed in a different way that WPF applications, it causes that there is a new catalog on MEF Silverlight version. This exercise uses an Aggregate Catalog (which is also available on the WPF version) including the new Deployment Catalog which download a Silverlight Package (.xap), if it was not downloaded before, and searches for composable parts inside it. As the Deployment Catalog is instantiated without any parameter, it will look for parts inside the Silverlight Package that contains the application.
  5. Since the CashMaker class is now a composable part, you will export the class making it available to the composition container. To do this, decorate the **CashMaker** class with the **Export** attribute as shown in the code below.
     1. C#
     2. **[Export]**
     3. public partial class CashMaker : UserControl, IPartImportsSatisfiedNotification
     4. {
     5. ...
     6. }
     7. Visual Basic
     8. **<Export()>**
     9. Class CashMaker
     10. Inherits UserControl
     11. Implement IPartImportsSatisfiedNotification
     12. ...
     13. End Class
  6. Add the collection of Lazy object containing the contract implementations that will be populated by MEF. To do this, add the following code to the **CashMaker.xaml.cs** (C#) or **CashMaker.xaml.vb** (Visual Basic) file, just above its constructor.
     1. *(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step5 - SilverlightCarQueries CSharp)*
     2. C#
     3. **[ImportMany(AllowRecomposition = true)]**
     4. **public ObservableCollection<Lazy<ICarQuery, IQueryMetadata>> CarQueries { get; set; }**
     5. *(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step5 - SilverlightCarQueries VB)*
     6. Visual Basic
     7. **<ImportMany(AllowRecomposition:=True)>**
     8. **Public Property CarQueries As ObservableCollection(Of Lazy(Of ICarQuery, IQueryMetadata))**
  7. Modify the implementation of the IPartImportsSatisfiedNotification interface and bind the CarQueries collection to the UI by setting the DataContext property of the commandGrid control. To do this, paste the following code inside the CashMaker’s **OnImportsSatisfied** method.
     1. *(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step5 - OnImportsSatisfied CSharp)*
     2. C#
     3. public void OnImportsSatisfied()
     4. {
     5. **this.commandGrid.DataContext = this.CarQueries;**
     6. }
     7. *(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step5 - OnImportsSatisfied VB)*
     8. Visual Basic
     9. Public Sub OnImportsSatisfied() Implements IPartImportsSatisfiedNotification. OnImportsSatisfied
     10. **Me.commandGrid.DataContext = Me.CarQueries**
     11. End Sub

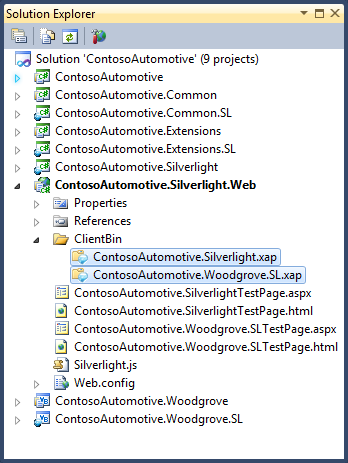
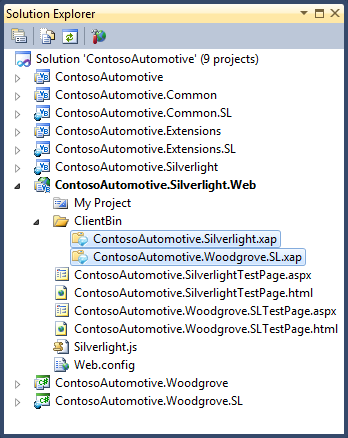
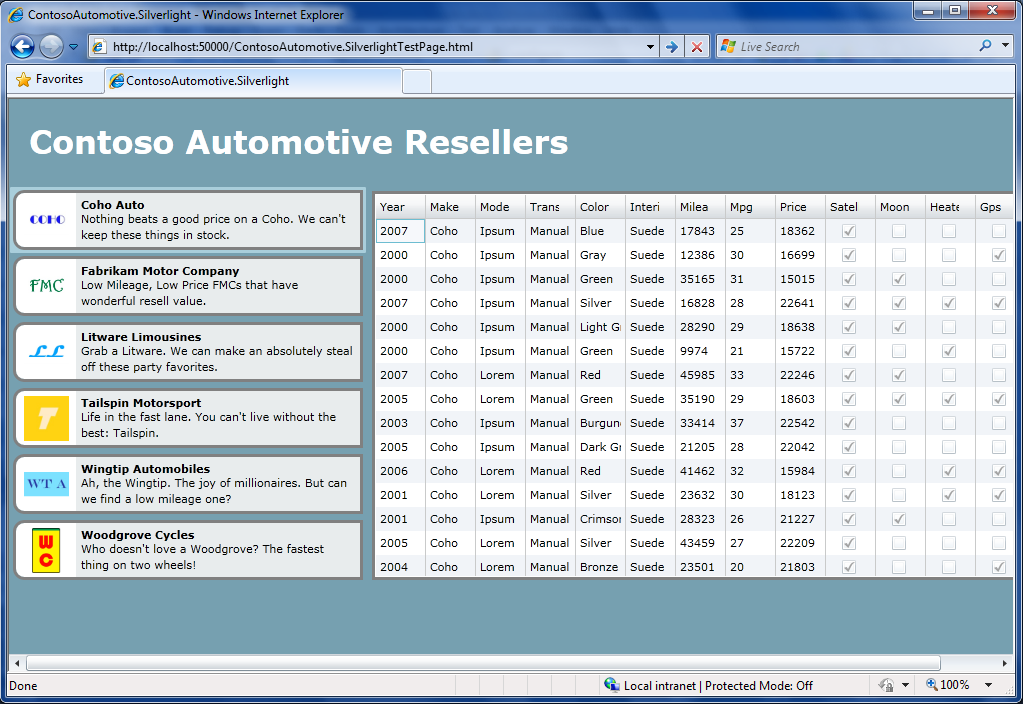
Task 4 – Load Silverlight Extensions Dynamically

* 1. In this task, you will create a new Silverlight 4 application and setup your application to load MEF parts dynamically by downloading a new Silverlight package. Additionally, you can mix CLR languages and create the new project in different CLR language than your main application.
  2. Create the ContosoAutomotive.Woodgrove.SL project reusing the classes in the ContosoAutomotive.Woodgrove Class Library project selecting a different language for the project than the main application. To do this, right click the ContosoAutomotive solution node and select **Add** | **New Project**. In the Add New Project dialog, select the **Silverlight Application** project type and enter **ContosoAutomotive.Woodgrove.SL** in the name field.
  3. In the New Silverlight Application options dialog select the existing ContosoAutomotive.Silverlight.Web project to host the Silverlight application and ensure **Silverlight 4** is selected in the **Silverlight Version** list. Also, remove the **Add a test page** option in the option section and press **OK**.
     1. 
     2. Figure 32
     3. New Silverlight Application options dialog
     4. **Note:** Remember that Silverlight Application projects are packaged into Silverlight Packages (.xap). In the other hand, Silverlight Class Library projects are included inside the application package that references the library. For that reason, you create the ContosoAutomotive.Woodgrove.SL project as a Silverlight application to be able to deploy it in a separated way allowing download it while the Silverlight Main application is running.
  4. After creating the project remove the auto-generated files **App.xaml** and **MainPage.xaml** and create a link to the WoodgroveQuery class files in the **ContosoAutomotive.Woodgrove** folder.
  5. Add a reference to the MEF library on the ContosoAutomotive.Woodgrove.SL project. To do this, right click on the project and select Add Reference. In the **Add Reference** dialog, select the **.Net** tab and choose the **System.ComponentModel.Composition** library.
  6. Add a reference to the ContosoAutomotive.Common.SL library on the ContosoAutomotive.Woodgrove.SL project. To do this, right click on the project and select Add Reference. In the **Add Reference** dialog, select the **Projects** tab and choose the **ContosoAutomotive.Common.SL** library.
  7. Now, your solution is prepared for creating a Silverlight 4 version of the Cash Maker application.
     1. 
     2. Figure 33
     3. All projects converted to Silverlight Class Library projects (C#)
     4. 
     5. Figure 34
     6. All projects converted to Silverlight Class Library projects (Visual Basic)
  8. Modify the ContosoAutomotive.Silverlight application class code to create a catalog to download the new Silverlight package. To do this, replace the catalog definition code inside the **Application\_Startup** method with the following.
     1. *(Code Snippet – Intro to MEF Lab - Ex3 Task4 Step7 - UriDeploymentCatalog CSharp)*
     2. C#
     3. private void Application\_Startup(object sender, StartupEventArgs e)
     4. {
     5. **var xapCatalog = new DeploymentCatalog("/ClientBin/ContosoAutomotive.Woodgrove.SL.xap");**
     6. **xapCatalog.DownloadAsync();**
     7. **var catalog = new AggregateCatalog(new DeploymentCatalog(), xapCatalog);**
     8. var container = new CompositionContainer(catalog);
     9. this.RootVisual = container.GetExportedValue<CashMaker>();
     10. }
     11. *(Code Snippet – Intro to MEF Lab - Ex3 Task4 Step7 - UriDeploymentCatalog VB)*
     12. Visual Basic
     13. Private Sub Application\_Startup(ByVal o As Object, ByVal e As StartupEventArgs) Handles Me.Startup
     14. **Dim xapCatalog = New DeploymentCatalog("/ClientBin/ContosoAutomotive.Woodgrove.SL.xap")**
     15. **xapCatalog.DownloadAsync()**
     16. **Dim catalog = New AggregateCatalog(New DeploymentCatalog(), xapCatalog)**
     17. Dim container = New CompositionContainer(catalog)
     18. Me.RootVisual = container.GetExportedValue(Of CashMaker)()
     19. End Sub
     20. **Note:** In the code above, a new Deployment Catalog is included to the Aggregate Catalog. Because it is instantiated specifying a url where a Silverlight Package (.xap) is available, it will be in charge of download it and find the composable parts inside it.
     21. The call to the DownloadAsync method of the DeploymentCatalog object forces to download the package (it is performed in a separated thread). Take into account that the package will not be downloaded until this method is called.

# Next Step

* 1. Exercise 3: Verification

### Exercise 3: Verification

* 1. In this verification, you will run the application to verify that all the queries are shown in the main application. The Woodgrove manufacturer will be loaded from a different Silverlight package downloaded from the website.
  2. Compile the solution (**CTRL+SHIFT+B**).
  3. Expand the **ClientBin** folder under the **ContosoAutomotive.Silverlight.Web** project and verify that there are two Silverlight packages present.
     1. 
     2. Figure
     3. Silverlight packages in the ContosoAutomotive.Silverlight.Web project (C#)
     4. 
     5. Figure
     6. Silverlight packages in the ContosoAutomotive.Silverlight.Web project (Visual Basic)
  4. You will modify the way the application loads the Woodgrove package to emphasize the fact it is being downloaded from the website. Open the **App** class in the **ContosoAutomotive.Silverlight** project and modify the **Application\_Startup** method including the bolded line of code below.
     1. *(Code Snippet – Intro to MEF Lab - Ex3 Verification Step3 - DownloadCompleteAddHandler CSharp)*
     2. C#
     3. private void Application\_Startup(object sender, StartupEventArgs e)
     4. {
     5. var xapCatalog = new DeploymentCatalog("/ClientBin/ContosoAutomotive.Woodgrove.SL.xap");
     6. **xapCatalog.DownloadCompleted += new EventHandler<System.ComponentModel.AsyncCompletedEventArgs>(xapCatalog\_DownloadCompleted);**
     7. xapCatalog.DownloadAsync();
     8. var catalog = new AggregateCatalog(new DeploymentCatalog(), xapCatalog);
     9. var container = new CompositionContainer(catalog);
     10. this.RootVisual = container.GetExportedValue<CashMaker>();
     11. }
     12. *(Code Snippet – Intro to MEF Lab - Ex3 Verification Step3 - DownloadCompleteAddHandler VB)*
     13. Visual Basic
     14. Private Sub Application\_Startup(ByVal o As Object, ByVal e As StartupEventArgs) Handles Me.Startup
     15. Dim xapCatalog = New DeploymentCatalog("/ClientBin/ContosoAutomotive.Woodgrove.SL.xap")
     16. **AddHandler xapCatalog.DownloadCompleted, AddressOf xapCatalog\_DownloadCompleted**
     17. xapCatalog.DownloadAsync()
     18. Dim catalog = New AggregateCatalog(New DeploymentCatalog(), xapCatalog)
     19. Dim container = New CompositionContainer(catalog)
     20. Me.RootVisual = container.GetExportedValue(Of CashMaker)()
     21. End Sub
  5. Add the **xapCatalog\_DownloadCompleted** method handler to increment in two seconds the delay experienced by downloading a new package from internet. In the **App** class, add the following methods below the **Application\_Startup** method.
     1. *(Code Snippet – Intro to MEF Lab - Ex3 Verification Step4 - DownloadCompleteMethod CSharp)*
     2. C#
     3. **private void xapCatalog\_DownloadCompleted(object sender, System.ComponentModel.AsyncCompletedEventArgs e)**
     4. **{**
     5. **System.Threading.Thread.Sleep(2000);**
     6. **}**
     7. *(Code Snippet – Intro to MEF Lab - Ex3 Verification Step4 - DownloadCompleteMethod VB)*
     8. Visual Basic
     9. **Private Sub xapCatalog\_DownloadCompleted(ByVal sender As Object, ByVal e As System.ComponentModel.AsyncCompletedEventArgs)**
     10. **System.Threading.Thread.Sleep(2000)**
     11. **End Sub**
  6. Set **ContosoAutomotive.Silverlight.Web** as the startup project. To do this, in the Solution Explorer, right-click **ContosoAutomotive.Silverlight.Web** and select **Set as Startup Project**.
  7. Press F5 to run the application. The CashMaker application should appear with a list of manufacturers on it. You should note that the Woodgrove manufacturer item would appear after some delay.
     1. 
     2. Figure
     3. The CashMaker Silverlight application shows manufacturers.
  8. Close the browser by clicking on the close button ().

## Next Step

* 1. Summary

Summary

* 1. In this lab you have used the Microsoft Managed Extensiblity Framework to add extensibility points to an application and build extensions to plug into those extensibility points.
  2. Using MEF, you set exported properties, loaded external assemblies without explicit references and dynamically extended your application at runtime. You have seen the interaction of Imports and Exports, and used Contracts via Interfaces to specify what Export components to deal with.
  3. In addition, you created a Silverlight 4 application and enabled loading composable parts using MEF.