VanSLUG

Introduction to MEF (mef.codeplex.com)

Speaker: Jeremiah Redekop
Date: January 26, 2010

vanslug.net
forum.vanslug.net

About Jeremiah Redekop

- email: jeremiah@geniuscode.net
- blogs.geniuscode.net/JeremiahRedekop
- twitter: @JRedekop

Outline

- What problems does MEF address?
- How does MEF work?
- What are some good scenarios for MEF?
  - .Net
  - Silverlight
- Demos
- Additional Resources
- Q&A

Problem:

Managing apps that are monolithic in nature

Monolithic Applications

- components are "tightly coupled" and there is no clear separation between them
- difficult for developers to **maintain**
- difficult to add **new features** to the system or replace existing features
- difficult to **resolve bugs** without breaking other portions of the system
- difficult to **test** and **deploy**
- difficult for designer and developers to **work together**
- difficult == costly == $$

Solution:

Extensible Applications

Extensible Applications

- Extensible: the **E** in M**E**F
- aka Composite, Plugins, Modular, etc

- Modules can be **individually** developed, tested, and deployed by **different individuals or teams**
- **Separation** of teams and responsibilities
- Recompile modules **individually**
- **Independent** modules
- Reduces cost of development and maintenance for long term

How does MEF work?

Magic!

"The good kind of Magic..."
Glenn Block, MS Project Manager

Quick Code Preview

- What will happen when an composition occurs?

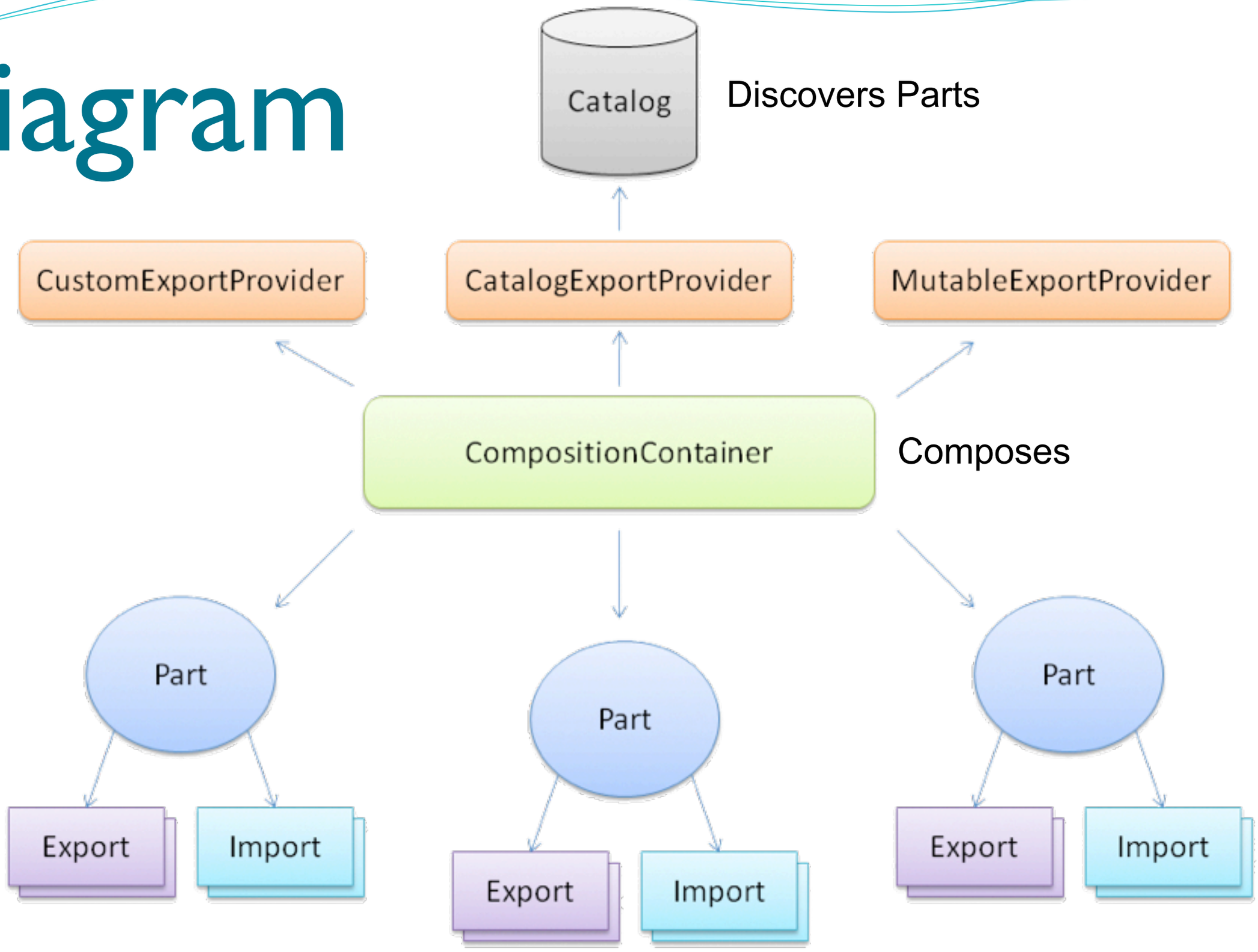
```
public class ToCompose
{
    [Import]
    public int IntegerToImport { get; set; }
}


public class ClassWithInteger
{
    [Export]
    public int IntegerToExport
    {
        get { return 5; }
    }
}
```

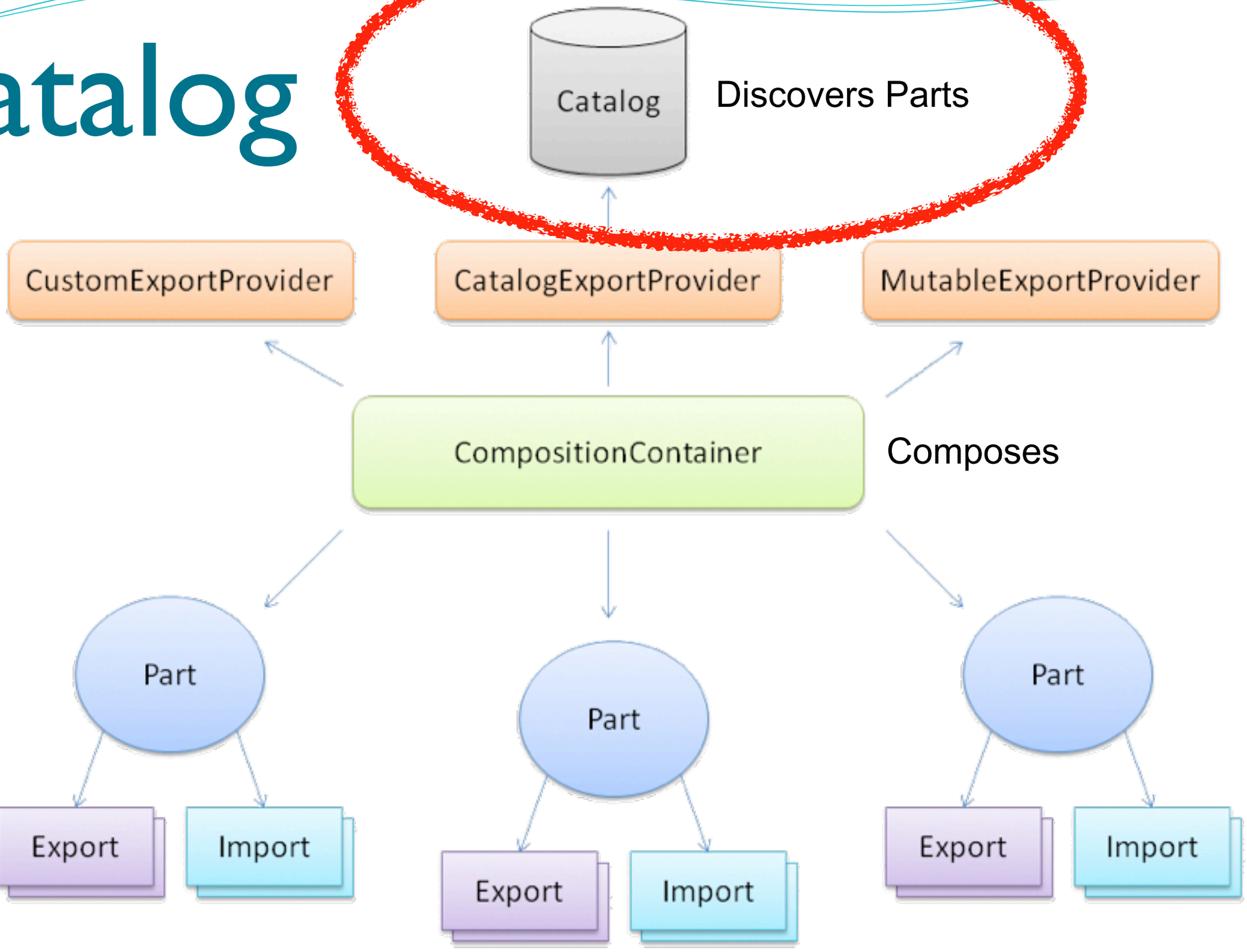
3 Main Parts of MEF

- Catalog
  - source of discoverable MEF parts
- Container
  - performs composition for an object
- Parts (imports and exports)
  - Exports and Imports that are to be discovered
    - Exports are discovered by the catalog
    - Imports are passed in to the container

Diagram


Catalog
Discovers Parts
CustomExportProvider
CatalogExportProvider
MutableExportProvider
CompositionContainer
Composes
Part
Part
Part
Export
Import
Export
Import
Export
Import

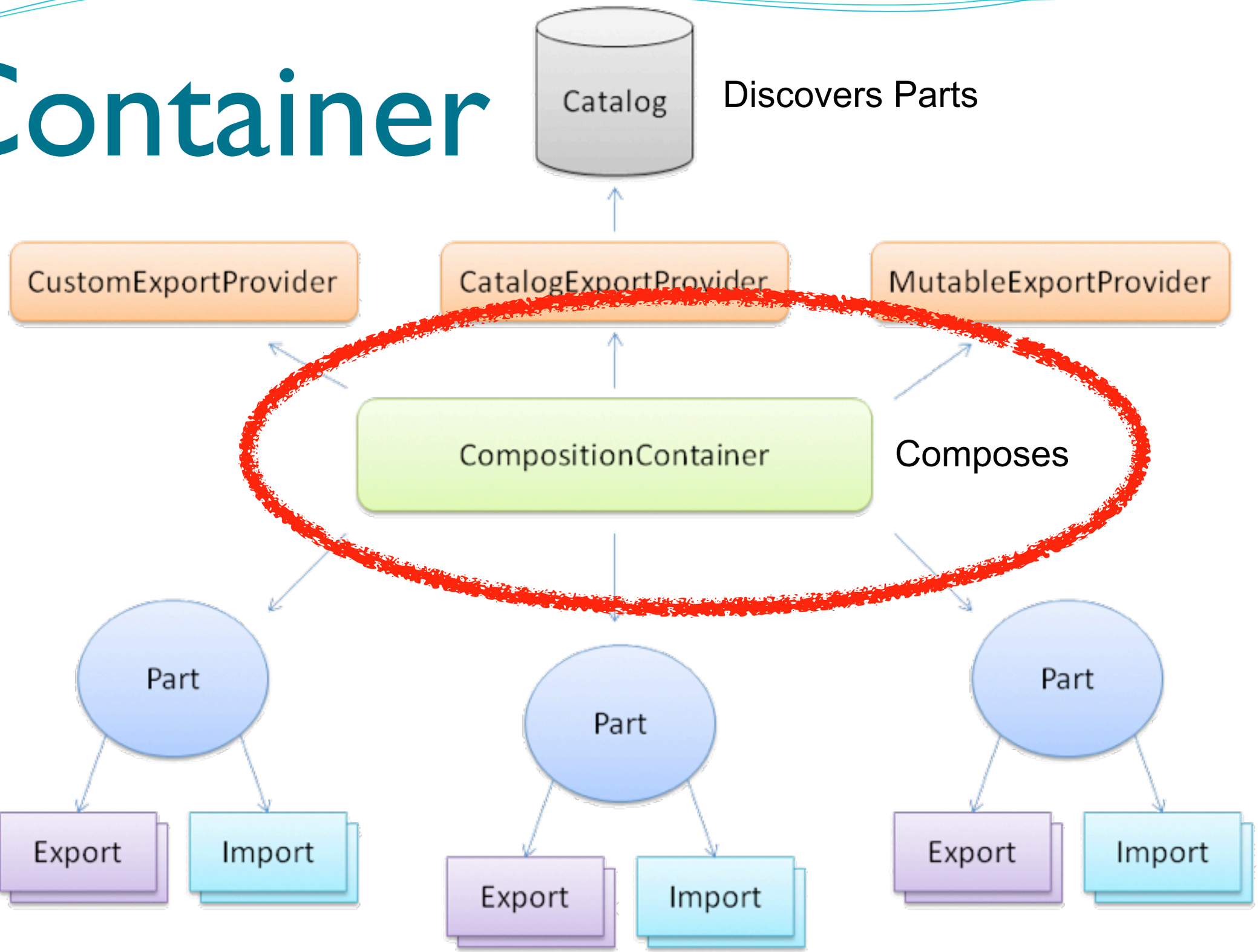
Catalog


Catalog
Discovers Parts
CustomExportProvider
CatalogExportProvider
MutableExportProvider
CompositionContainer
Composes
Part
Part
Part
Export
Import
Export
Import
Export
Import

Catalogs

- Where Parts are discovered
- Types of Catalogs:
  - Assembly Catalog
    - discovers exports in a given assembly
  - Deployment Catalog (*Silverlight only*)
    - uses dynamically downloaded XAPs
  - Type Catalog
    - declared with an array of Types to be used
  - Aggregate Catalog
    - collection of catalogs
    - Useful as a container can only have 1 catalog
  - Directory Catalog (*not supported in Silverlight*)
    - discovers exports in dlls in a given directory

Container

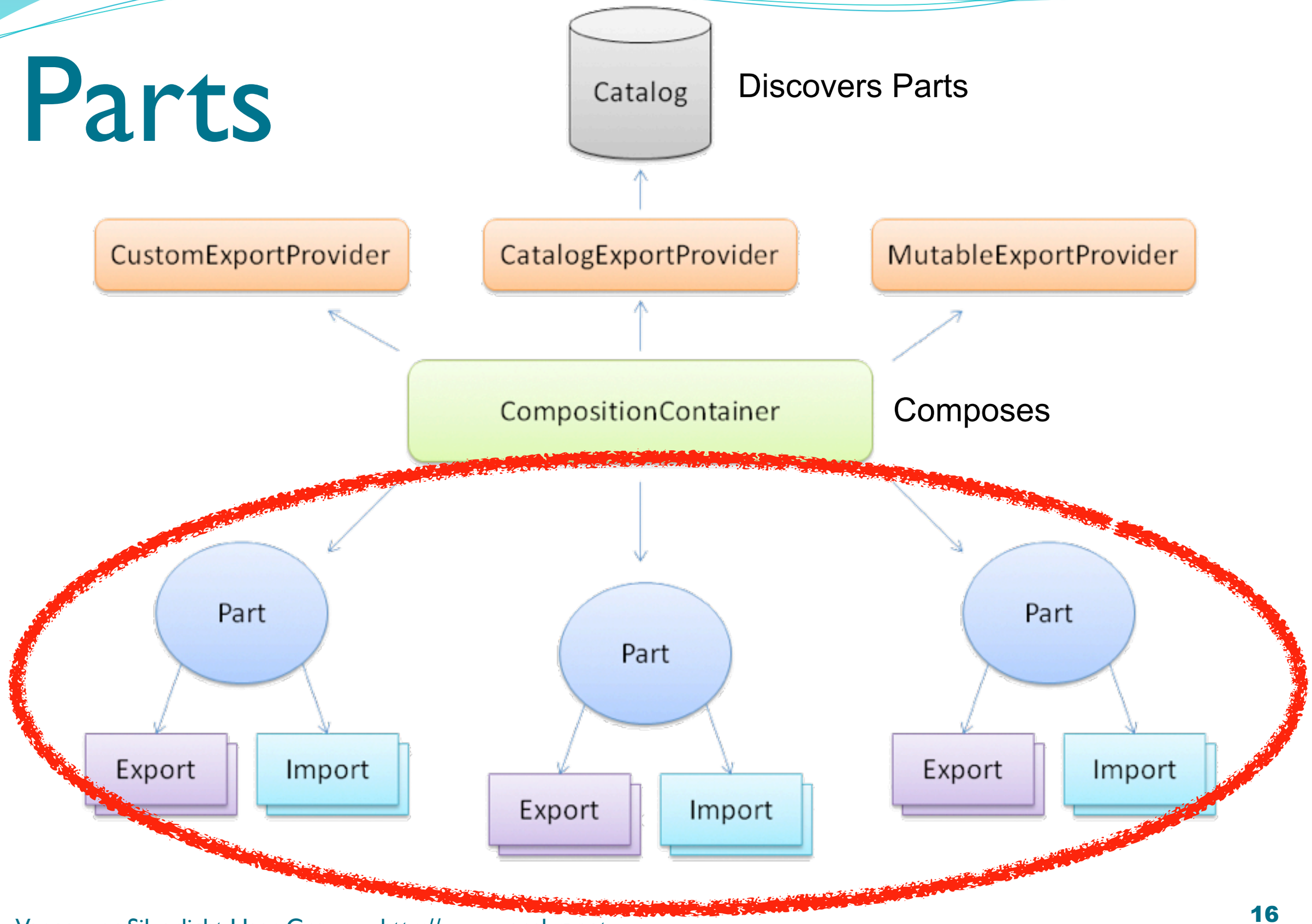

Catalog
Discovers Parts
CustomExportProvider
CatalogExportProvider
MutableExportProvider
CompositionContainer
Composes
Part
Part
Part
Export
Import
Export
Import
Export
Import

Composition Container

- Performs composition for an object using a **single** catalog
- Can hold references to objects
- AssemblyCatalog Example:

```
    private void ComposeObject(object toCompose)
    {
       // Create Catalog:
      AssemblyCatalog catalog = new AssemblyCatalog
(Assembly.GetExecutingAssembly());
      // Create Container:
      var container = new CompositionContainer(catalog);
      // Perform Composition:
      container.ComposeParts(toCompose);
    }
```

Parts


Catalog
Discovers Parts
CustomExportProvider
CatalogExportProvider
MutableExportProvider
CompositionContainer
Composes
Part
Part
Part
Export
Import
Export
Import
Export
Import

Parts

- While catalogs & containers are types in themselves, a part is declared through attributes:
  - System.ComponentModel.Composition.**Export**Attribute
  - System.ComponentModel.Composition.**Import**Attribute
- Anything can be a part, if decorated with attribute
- Parts can have Metadata, which describe the part

- For Later:
  - Metadata is available without having to *instantiate* the object that the part represents (Lazy<T,M>, ExportFactory<T,M>)

Export / Import of Parts

- Contracts can specified, default contract is value type
  - String Contract (eg. Timeout): recommended for simple values
  - Type Contracts (eg. IConfiguration): recommended for objects
    - requires implementation of contract
    - converted to string contract internally

```
[Export(typeof(IConfiguration)]
public class Configuration : IConfiguration]
  {
    [Export("Timeout")]
    public int Timeout
    {
      get { return int.Parse(ConfigurationManager.AppSettings["Timeout"]); }
    }
  }

  public class UsesTimeout
  {
    [Import("Timeout")]
    public int Timeout { get; set; }
  }
```

Import Collections

- AllowRecomposition: Senders updated as more parts discovered

```
public class Notifier
{
   [ImportMany(AllowRecomposition=true)]
   public IEnumerable<IMessageSender> Senders {get; set;}

   public void Notify(string message)
   {
     foreach(IMessageSender sender in Senders)
     {
       sender.Send(message);
     }
   }
}
```

Lazy Imports

- Import is only created when accessed
- IMessageSender will be instantiated upon request, then cached for future requests.
- Only one instance will be created per container

```
public class HttpServerHealthMonitor
{
    [Import]
    public Lazy<IMessageSender> Sender { get; set; }
}
```

Export w/ Metadata

- Metadata is browsable **before** part is instantiated
- Allows for parts to be expose values to your application without a part instance
- Metadata is declared via attributes, must be a constant value

```
public interface IMessageSender
{
    void Send(string message);
}

[Export(typeof(IMessageSender))]
[ExportMetadata("Transport", "smtp")]
[ExportMetadata("IsSecure", true)]
public class EmailSender : IMessageSender
{
}
```

Import w/ Metadata

- Interface is used, **needs to match** metadata types and names for parts to be imported
- Use Lazy<T,Metadata>[] to sort through all matching exports

```
public interface IMessageSenderCapabilities
{
    string Transport { get; }
    bool IsSecure { get; }
}

public class HttpServerHealthMor
{
    [ImportMany]
    public Lazy<IMessageSender, IMessageSenderCapabilities>[] Senders
{ get; set; }
```

```
[Export(typeof(IMessageSender))]
[ExportMetadata("Transport", "smtp")]
[ExportMetadata("IsSecure", true)]
public class EmailSender : IMessageSender {}
```

Objects & Instances

- Export Instances are stored by container, re-used unless explicitly specified
- PartCreatePolicyAttribute applied on export part:
  - NonShared: one instance of the part may exist per container
  - Shared: each request for exports of the part will be served by a new instance

```
[PartCreationPolicy(CreationPolicy.NonShared)]
[Export(typeof(IMessageSender))]
public class SmtpSender : IMessageSender
{
}
```

ExportFactory<T> Import

- ExportFactory will give you a **new instance for every composition**, as opposed to Lazy (single instance per composition.)
- Instance will never be shared
- has a sibling - ExportFactory<T,M> which uses Metadata

```
public class OrderController {

  [Import]
  public ExportFactory<OrderViewModel> OrderVMFactory {get;set;}

  public OrderViewModel CreateOrder() {
    return OrderVMFactory.CreateExport().Value;
  }
}
```

Good MEF Scenarios

- Plugin based Applications
  - **Visual Studio** uses MEF
  - Seesmic Desktop Twitter Client uses MEF

- Application that reference GPL Assemblies
  - develop open source plugins, not applications

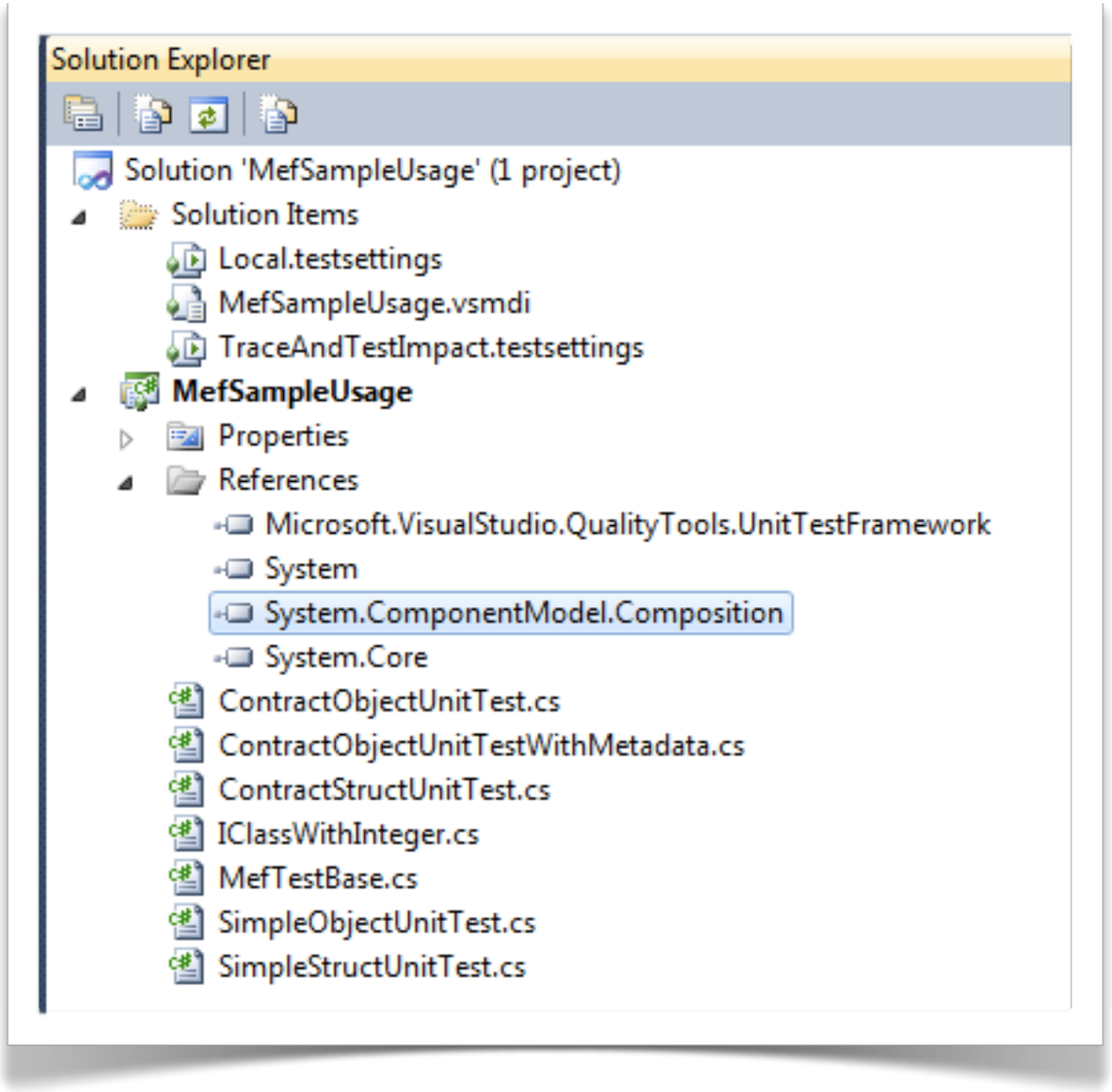
- Silverlight
  - Split your application into **multiple XAPs**, not one XAP
    - faster start time
    - Only load the modules you need, when you need them
  - Navigation uri resolution
  - Loading Views dynamically
  - ViewModel locators

Demos

Simple MEF & Silverlight-Specific XAP downloads

Simple Demo: Unit Tests

- Using Struct:
  - Simple
  - Contract
- Using Objects:
  - Simple
  - Contract
  - Lazy with Metadata



Notes: Base Class for Unit Tests

- My custom base class to encapsulate MEF for Unit Tests

```
public class MefUnitTest
{
    public MefUnitTest()
    {
        // create catalog to use current assembly
        var cat = new AssemblyCatalog(Assembly.GetExecutingAssembly());
        // create container instance
        container = new CompositionContainer(cat);
    }

    // container instance
    protected CompositionContainer container;

    protected void Compose(object toCompose)
    {
        container.ComposeParts(toCompose);
    }
}
```

Notes: Nested Classes Used

- Types used for MEF are isolated inside of unit test class
- No conflicts between types used in different unit tests

```
[TestClass]
public class SimpleStructUnitTest : MefUnitTest
{
    [TestMethod]
    public void TestMethod1()
    {
        ClassNeedingInteger c1 = new ClassNeeding
        Assert.AreEqual(0, c1.IntegerToImport);

        Compose(c1);
        Assert.AreEqual(5, c1.IntegerToImport)

    }

    public class ClassNeedingInteger
    {
        [Import]
        public int IntegerToImport { get; set; }
    }
```

MefSampleUsage.SimpleStructUnitTest.ClassNeedingInte...

ClassNeedingInteger is contained inside of unit test class

Advanced SL Demo

- Taken from Glenn's Mix10 Session:
- Demonstrating:
  - XAP Partitioning
  - Delayed Composition of XAPs
    - ie - downloading xaps

Additional Resources

- Documentation on Home page @ Codeplex:
  - mef.codeplex.com
- Silverlight TV
- Glenn Block's Blog
- multiple blogs (~~Google~~ Bing is your friend)

- Links are available on VanSlug forum page