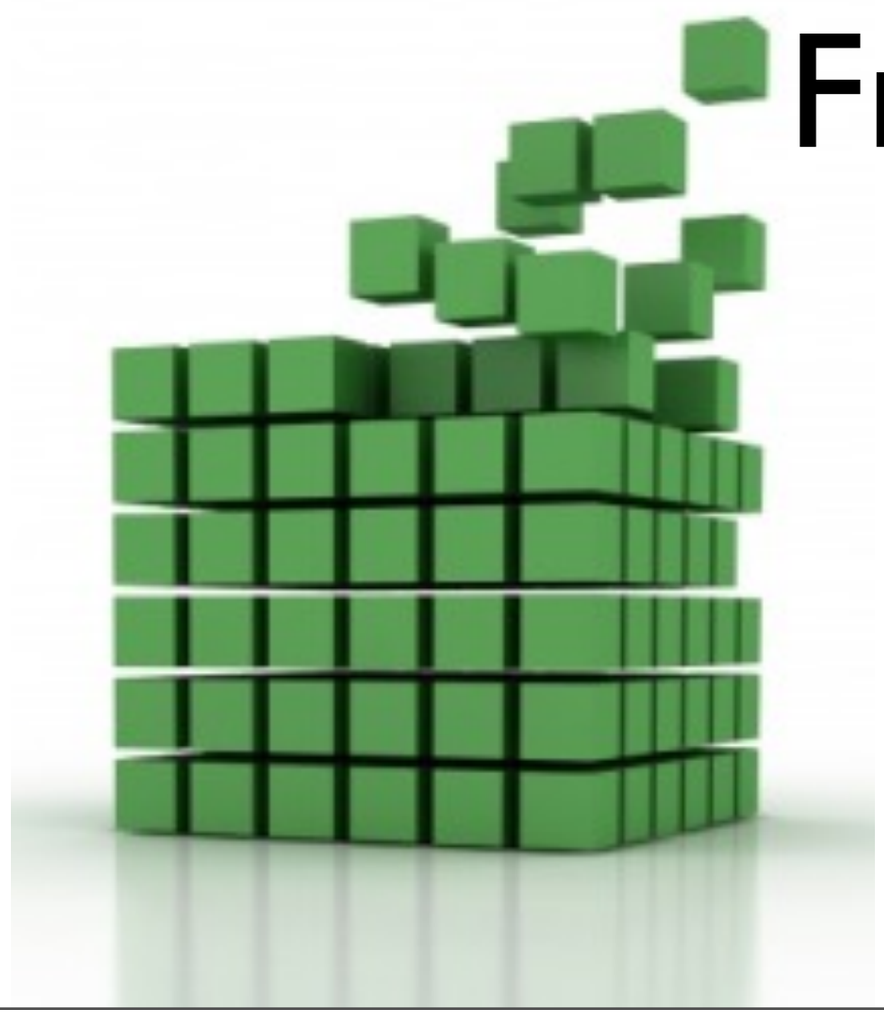
Intro to MEF

Managed Extensibility Framework



About Jeremiah Redekop

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

Outline

- What problems does MEF solve?
- How does MEF work?
- What are some good scenarios for MEF?
  - .Net
  - Silverlight
- Demo
- 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

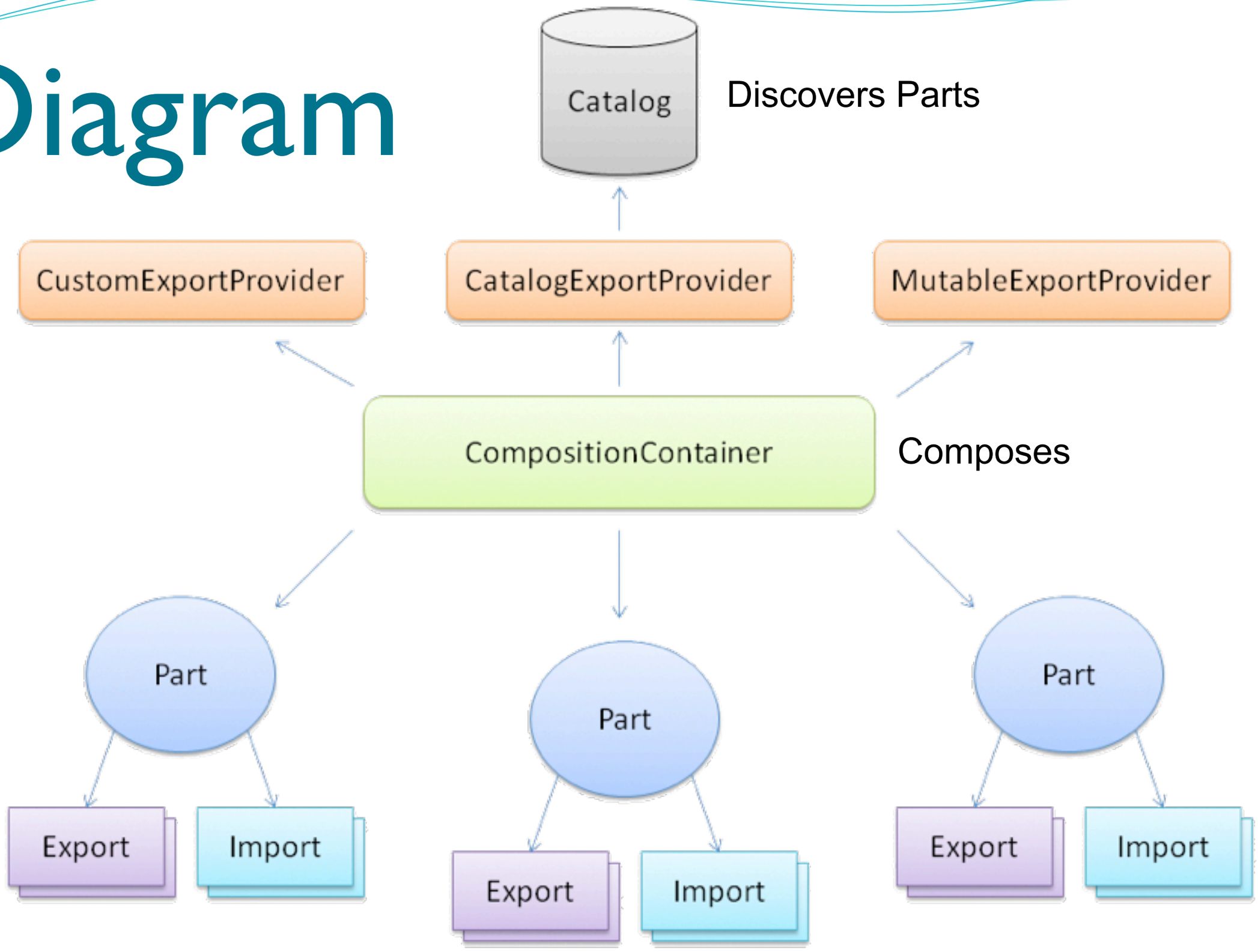
- Modules can be **individually** developed, tested, and deployed by **different individuals or teams**
- **Separation** of teams and responsibilities
- Recompile modules **individually**
- **Independent** modules
- Reduces cost of development and maintenance for long term

How does it work?

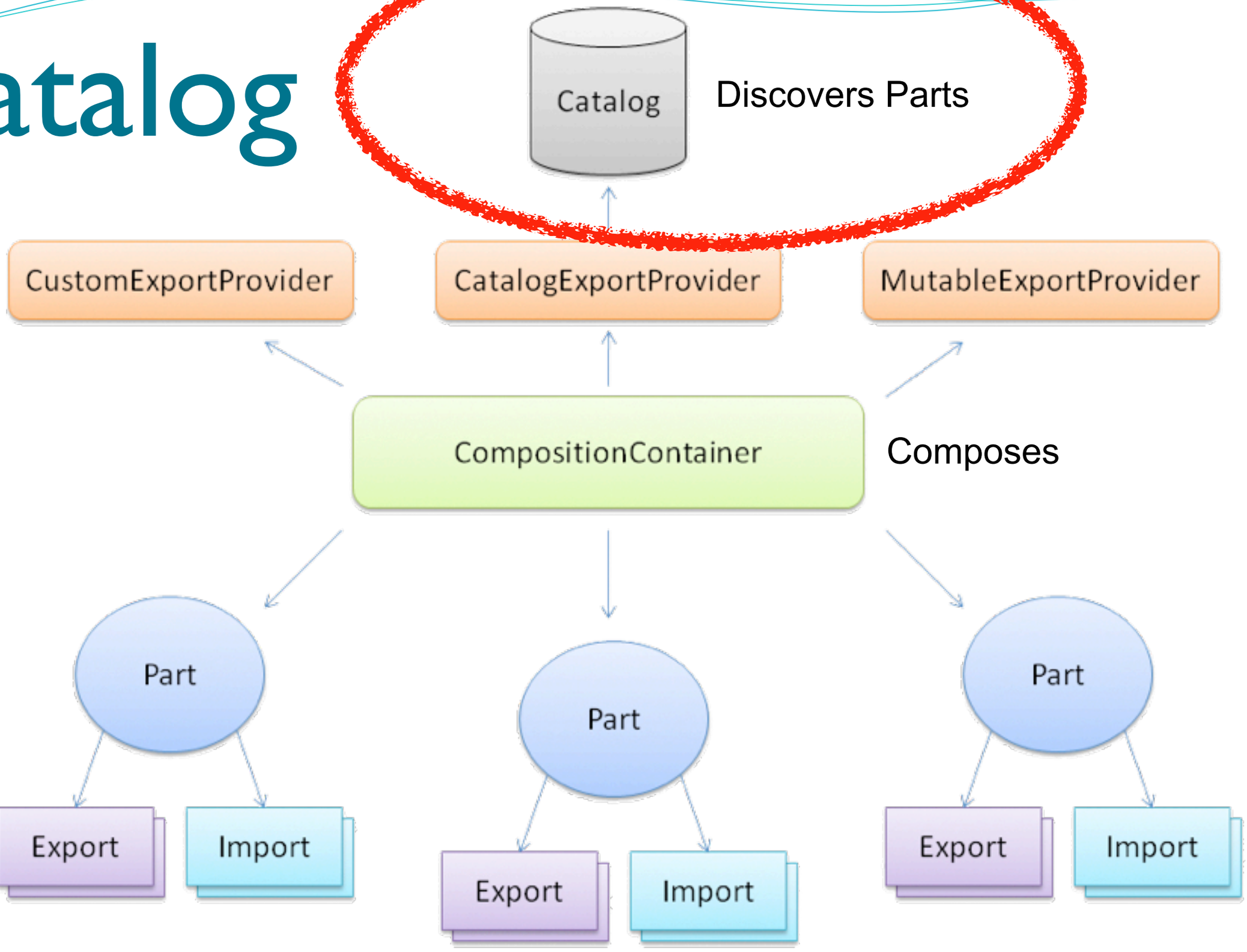
Magic!

"The good kind of Magic..."
Glenn Block,

Diagram


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

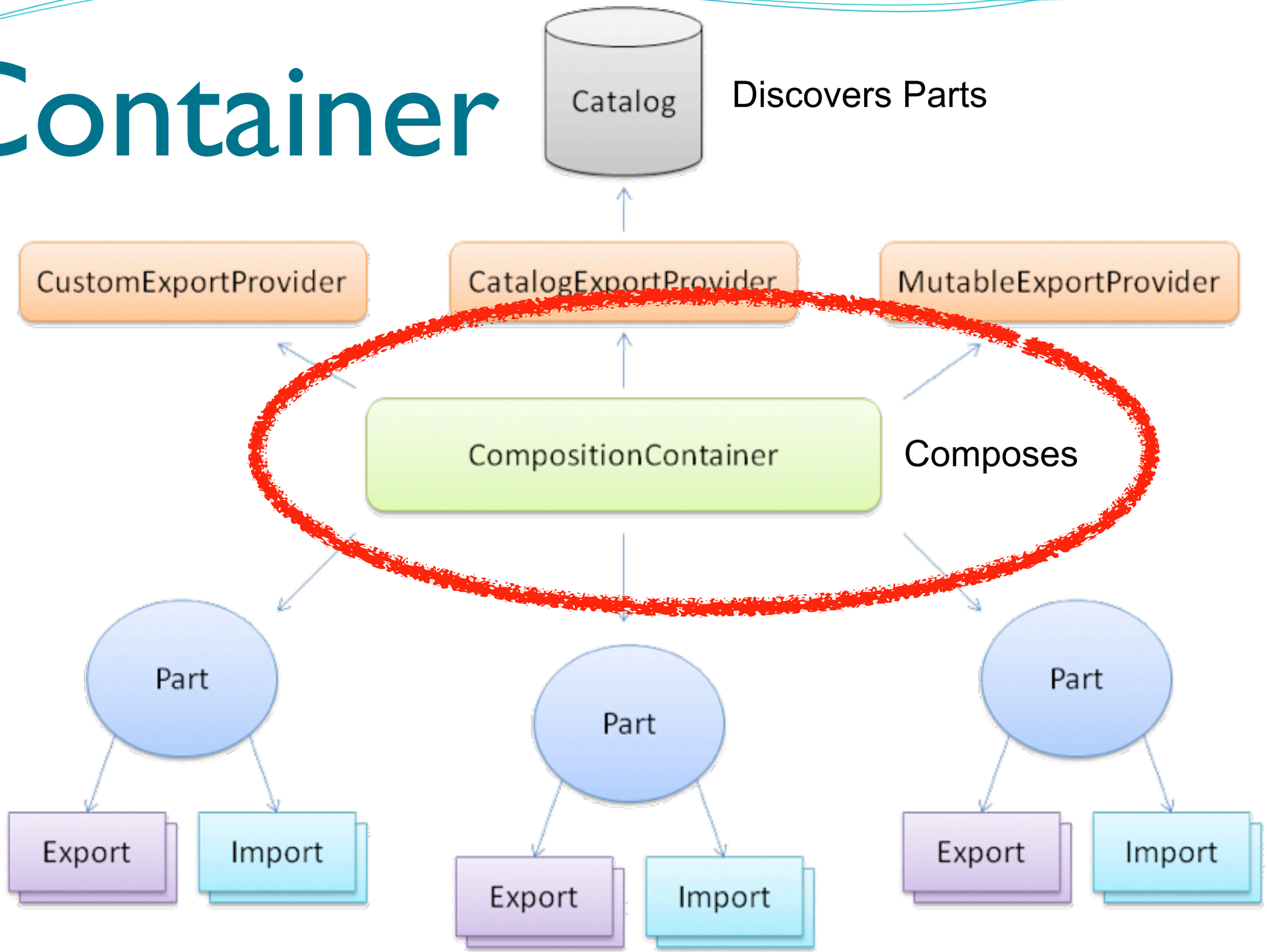
Catalog


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

Catalogs

- "Catalogs are where parts are found"
- Types of Catalogs:
  - Assembly Catalog
    - discovers exports in a given assembly
  - Aggregate Catalog
    - collection of catalogs
  - Deployment Catalog
    - uses dynamically downloaded XAPs
  - Type Catalog
    - declared with an array of Types that are used for part discovery
  - Directory Catalog (*not supported in Silverlight*)
    - discovers exports in dlls in a given directory

Container

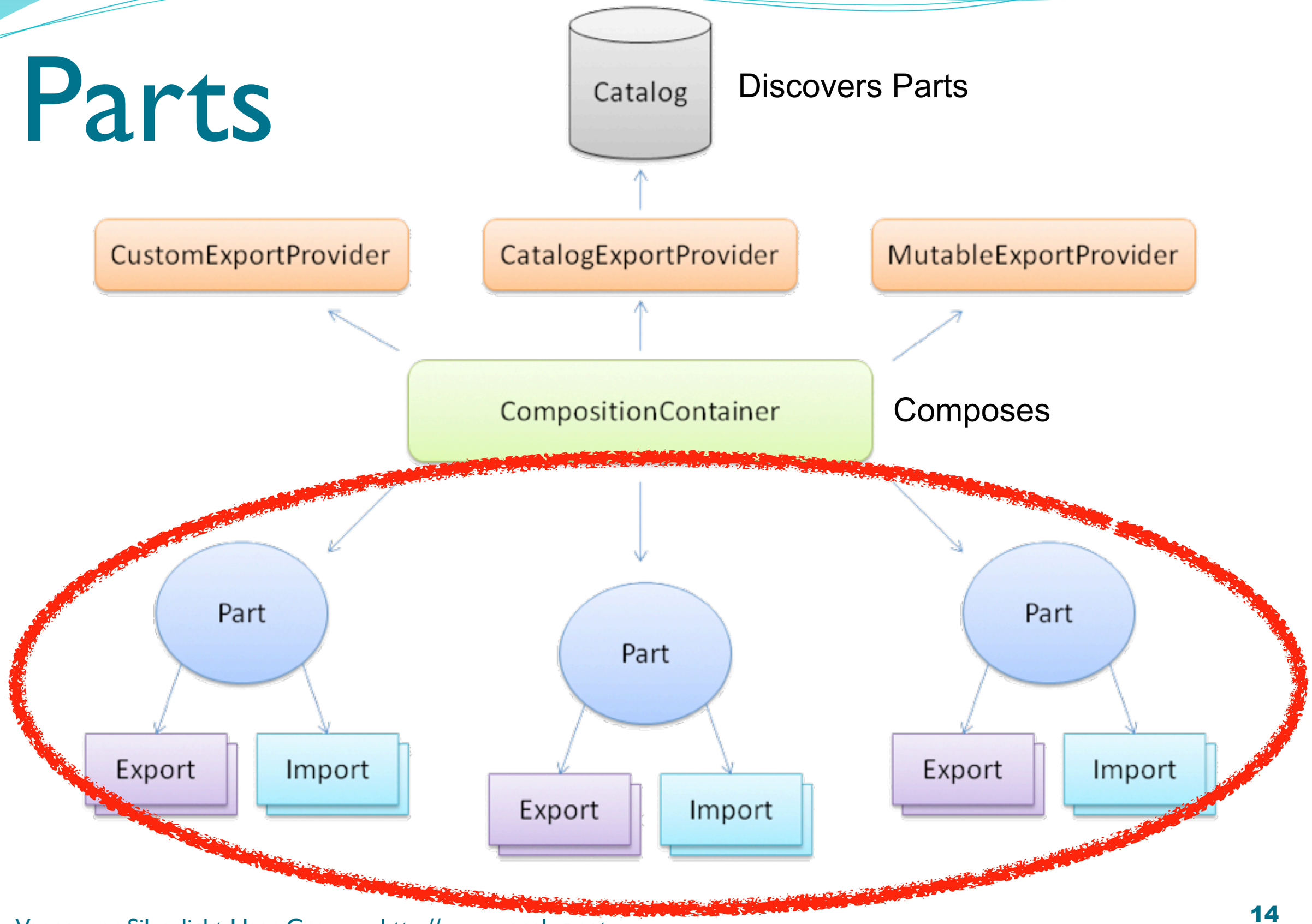

Catalog
Discovers Parts
CustomExportProvider
CatalogExportProvider
MutableExportProvider
CompositionContainer
Composes
Part
Part
Part
Export
Import
Export
Import
Export
Import

Composition Container

- Performs composition for an object using a **single** catalog
- 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

- Should be declared with Contract, which acts as a filter
  - String Contract (eg. Timeout): recommended for simple values
  - Type Contracts (eg. IConfiguration): recommended for objects
    - requires implementation of contract

```
[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

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

ExportFactory<T> Import

- ExportFactory will give you a **new instance for every request**, as opposed to Lazy (new on first request only.)

```
public class OrderController {

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

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

- has a brother, ExportFactory<T,M> which uses metadata

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 {}
```

Good MEF Scenarios

- Plugin based Application
  - Visual Studio uses MEF
  - Seesmic Desktop Twitter Client uses MEF

- Application with 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 MEF Demo

- Simple demonstration of MEF in action
- Unit test that shows MEF can compose instance
- For simplicity, AssemblyCatalog is used

Advanced SL Demo

- Taken from Glenn's Mix10 Session:
- http://live.visitmix.com/MIX10/Sessions/CL52
- Demonstrating:
  - Silverlight
  - XAP Partitioning
  - Delayed Composition of XAPs

Additional Resources

- Mef Home page on codeplex: mef.codeplex.com
- Links to SL TV
- Links to Glenn Block (Mr. MEF)'s blog