Microsofts
Managed **E**xtensibility **F**ramework
MEF

Agenda

- What is MEF
- MEF and it's siblings
  - Why and where to use MEF
- MEF Basics
- MEF Advanced

What is MEF?

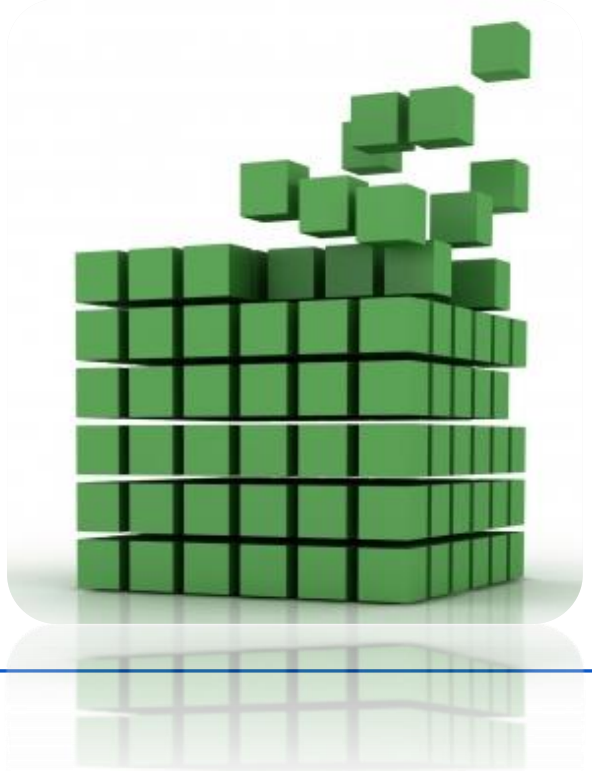
- The Managed Extensibility Framework (MEF) is a new library in the .NET Framework v. 4.0 that enables greater reuse of applications and components.
- Using MEF, .NET applications can make the shift from being statically composed to dynamically composed.

- Dynamically Composed:
  - composition takes place at runtime and composition behaviour differs depending on how it is configured.

What is “Extensibility”?

- In software engineering, **extensibility** is a systemic measure of the ability to extend a system and the level of effort required to implement the extension.
- Open/Closed Principle:

Software entities should be open for extension, but closed for modification.



MEF Hello World

1. **Create a host class**.
   In the sample below we are using a console application, so the host is the Program class.
2. Add a reference to the **System.ComponentModel.Composition** assembly.
3. Add the using statement: **using System.ComponentModel.Composition;**
4. Add a **Compose()** method which creates an instance of the container and composes the host.
5. Add a **Run()** method which calls **Compose()**;
6. In the **Main()** method instantiate the host class.
   (*Note: For an ASP.NET or WPF application the host class is instantiated by the runtime making this step unnecessary)*

```
using System.ComponentModel.Composition;
…
public class Program {
  public static void Main(string[] args) {
    Program p = new Program();
    p.Run();   }
  public void Run() {
    Compose(); }
  private void Compose() {
    var container = new CompositionContainer();
    container.ComposeParts(this); }
  }
```

MEF Hello World

7. Define one or more exports which the host will import.

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

[Export(typeof(IMessageSender))]
public class EmailSender : IMessageSender
{
  public void Send(string message)
  {
    Console.WriteLine(message);
  }
}
```

MEF Hello World

8. Add properties to the host class for each import which are decorated with the **[Import]** attribute:

```
[Import]
public IMessageSender MessageSender { get; set; }
```

9. Add parts to the container.

   In the Compose() method manually add each Composable Part by using the ComposeParts() extension method, OR add to the container using an **AssemblyCatalog**.

```
private void Compose() {
  var catalog = new AssemblyCatalog(Assembly.GetExecutingAssembly());
  var container = new CompositionContainer(catalog);
  container.ComposeParts(this);
}
```

MEF AND IT'S SIBLINGS

MEF is a new library in .NET that enables greater reuse of applications and components.

But is that new?

For the Nth Time, the CLR Has Its First Plugin Model

- **Managed Extensibility Framework** is the fourth extensibility framework to be released by Microsoft.
  - Though like all the previous times, Microsoft is claiming that it is the first.
- The three other extensibility frameworks for the .NET runtime developed by Microsoft itself:
  - **Unity** (the Enterprise Library Unity Application Block)
  - The **Composite Application Library** aka **PRISM**
  - The **CLR Add-In Model**, also known as the MAF or **System.Add-In**
- So which to use?

Unity Application Block

- It isn't a real Extensibility Framework.
  - It's just an IoC-container for Dependency Injection.
- But it **can** be used to build your own Extensibility Framework.
- Unity is NOT a part of the .Net framework.
  - It is developed by the Pattern &Practices group at MS.
  - It must be downloaded from: http://msdn.microsoft.com/en-us/library/dd203101.aspx
  - Community site: http://unity.codeplex.com/

PRISM aka Composite Application Library

- Is only for WPF and Silverlight Applications
- Architectural Goals:
  - Create a complex application from modules that can be built, assembled, and, optionally, deployed by independent teams using WPF or Silverlight.
  - Minimize cross-team dependencies and allow teams to specialize in different areas, such as user interface (UI) design, business logic implementation, and infrastructure code development.
  - Use an architecture that promotes reusability across independent teams.
  - Increase the quality of applications by abstracting common services that are available to all the teams.
  - Incrementally integrate new capabilities.

  Even single-person projects experience benefits in creating more testable and maintainable applications using the modular approach

- PRISM is NOT a part of the .Net framework
- Download: http://msdn.microsoft.com/en-us/library/cc707819.aspx

System.Add-in

- Is a great technology for addressing issues around versioning resilience, isolation and recoverability.
- Allows you to host different components in separate app domains, thus allowing those addins to have different versions of assemblies which they reference which all run in the same process.
- Manages automatically unload app-domains that are no longer used thus reclaim memory.
- **System.Addin has a bunch of security infrastructure (addins run in a sandbox) to ensure that components that are loaded do not have unauthorized access to data in the rest of the app.**
- System.AddIn allows your app to gracefully recover whenever one of the addins crashes (this is due to isolation).
- **BUT System.Add-In is very complicated to use!!!**
  - And with poor performance.
- System.Add-in is a part of the .Net framework from v. 3.5
- CodePlex site: http://www.codeplex.com/Wikipage?ProjectName=clraddins

MEF's Advantages

- Composes deep object hierarchies of components
- Abstracts components from static dependencies.
- Can allow components to be lazily loaded and lazily instantiated.
- **Provides a cataloging mechanism and rich metadata for components to allow them to be dynamically discovered.**
- Can compose components from various programming models, it is not bound to static types.
- MEF is a part of the .Net framework from v. 4.0
- CodePlex site: http://mef.codeplex.com/

Confused?

- One possible explanation for this is that the Add-In Model just doesn't work for most people
  - It is an embarrassingly complex framework
  - and code generation tools such as the **System.Addin Pipeline Builder** are needed just to get started.

- The confusion over the different technologies suggests that there are some disputes within Microsoft.
  - The Program Manager Nicholas Blumhardt has said that interoperability between the two projects (Add-In and MEF) would be considered after .NET 4 is released.
  - But the only known work in progress is integration of PRISM and MEF.

What problems does MEF solve?

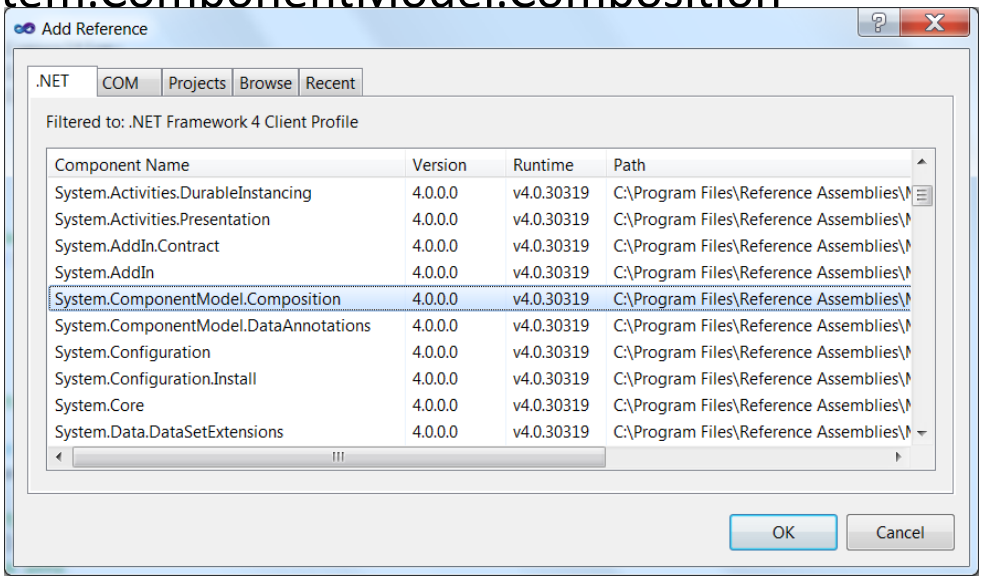
- MEF provides a standard way for the host application to expose itself and consume external extensions.
- Extensions, by their nature, can be reused amongst different applications.
  - However, an extension is very often implemented in a way that is application-specific.
- Extensions themselves can depend on one another and MEF will make sure they are wired together in the correct order.
- MEF offers a set of discovery approaches for your application to locate and load available extensions.
- MEF allows tagging extensions with additional metadata which facilitates rich querying and filtering

MEF
Basics

MEF-aware applications are built of parts

Prepare for MEF usage

- To use MEF every project must:
  1. Add a reference to System.ComponentModel.Composition



  2. Add the following using declarations:

```
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
```

And sometimes also:

```
using System.ComponentModel.Composition.Primitives;
using System.Reflection;
```

The 5 elements of MEF

- To understand MEF and utilize it in your application development you need to understand the 5 basic elements:
  1. Contract
  2. Import
  3. Export
  4. Catalog
  5. Composition Container

Contract

- In MEF a contract is a string
  - **that is used to match imports with exports**
- Composable Parts do not directly depend on one another, instead they "depend" on a contract.
- If no contract is specified, MEF will implicitly use the fully qualified name of the type as the contract.
- **If a type is passed, it will use the fully qualified type name.**

```
namespace MEFSample
{
  [Export]
  public class Exporter {...}

  [Export(typeof(Exporter))]
  public class Exporter1 {...}

  [Export("MEFSample.Exporter")]
  public class Exporter2 {...}
}
```

Recommended style

The contract

Import

- Import is used to indicate services that you require from others
  - Also known as dependencies
- I.e. you use import to mark where you want dependency injection to happen.
- MEF supports field, property and constructor injection.
  - Property

```
class Demo
{
  [Import]
  public IMyInterface Prop { get; set; }
}
```

  - Constructor

```
class Demo2
{
  [ImportingConstructor]
  public Program(IMyInterface parameter)
    {
      ...
    }
}
```

The contract

ImportMany

- You can import collections with the ImportMany attribute.
  - This means if there are 3 exports of IMessageSender available in the container, they will be pushed in to the Senders property during compositon.

```
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);
      }
    }
 }
```

Export

- Exports are contracts a part offers to other parts.
- For a Composable Part to export itself, simply decorate the Composable Part with an [Export] attribute.

```
[Export(typeof(IMyInterface)]
public class MyClass : IMyInterface
{ ... }
```

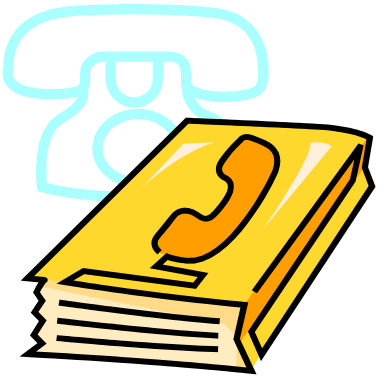
The contract

- You can also export properties
  - and methods (as delegates)

```
public class Configuration
{
  [Export("Timeout")]
  public int Timeout
  {
    get { return int.Parse(Con.Settings["Timeout"]); }
  }
}
```

Catalogs

- A Catalog provide the Composition Container with a list of known exports.



Think of a **Phonebook**…

```
{
   var catalog = new AssemblyCatalog(Assembly.GetExecutingAssembly());
   …
}
```

Build in Catalog Types

TypeCatalog

You add the types directly to the catalog yourself.

AssemblyCatalog

Will go through all types within an assembly and find all types that are decorated with the Export attribute and return those as parts that can partake in composition.

DirectoryCatalog

Will go through each assembly (only *.dll) in a given directory, and find all types that are decorated with the Export attribute and return those as parts that can partake in composition.

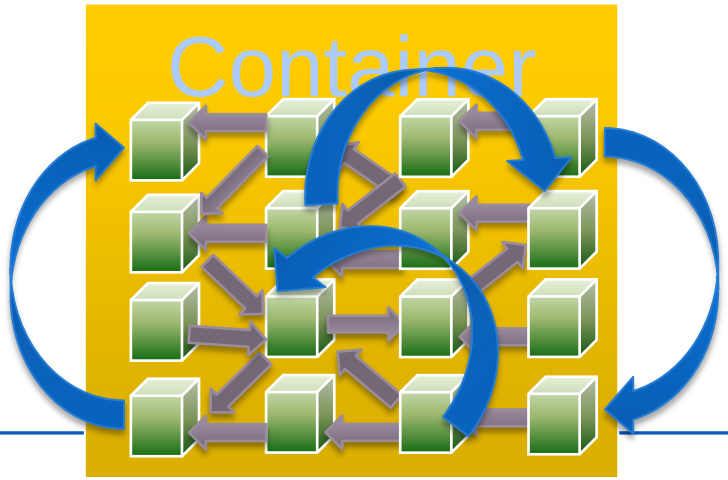
AggregatingCatalog

Aggregates different catalogs together. These could be catalogs of any type and any different combination.

Composition Container

- The Composition Container is the Match Maker.
- An application will “ask” the composition container for an object with a specific contract (of a given type),
  - the container will then build the entire object graph and return an object that exports the specified contract.

```
var catalog = new DirectoryCatalog(@".\Extensions");
var Container = new CompositionContainer(catalog);
// Call composeparts to connect the Import with Export.
Container.ComposeParts(this)
```



Advanced MEF

Lifetime

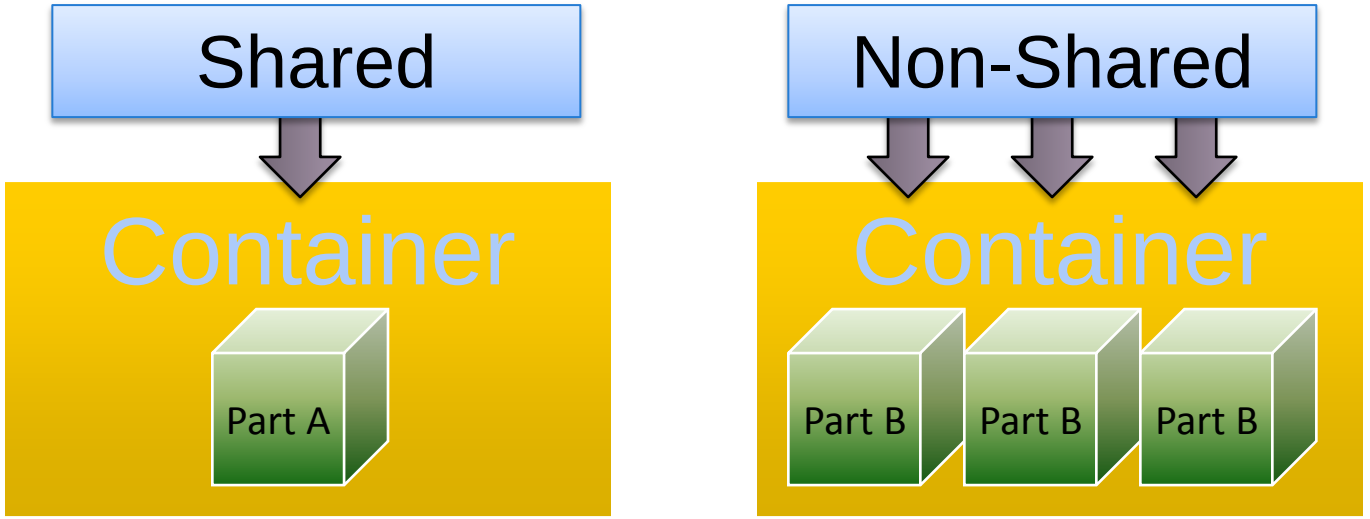
MEF also provides parts the capability of specifying what their lifetime behavior should be

On Export

```
[CompositionOptions(CreationPolicy=CreationPolicy.NonShared)]
[CompositionOptions(CreationPolicy=CreationPolicy.Shared)]
```

On Import

```
[Import(RequiredCreationPolicy=CreationPolicy.NonShared)]
[Import(RequiredCreationPolicy=CreationPolicy.Shared)]
```



Being Lazy with MEF

- The Lazy<T> class is a new class in .NET 4.0 that allow you to support lazy initialization.
  - I.e. you can use Lazy<T> to defer the creation of a large or resource-intensive object, till you really need it.
  - (see http://msdn.microsoft.com/en-us/library/dd997286.aspx)

- If you want to use lazy with MEF you can do it like this:

```
[ImportMany(typeof(IAnimal))]
IEnumerable<Lazy<IAnimal>> Animals { get; set; }
```

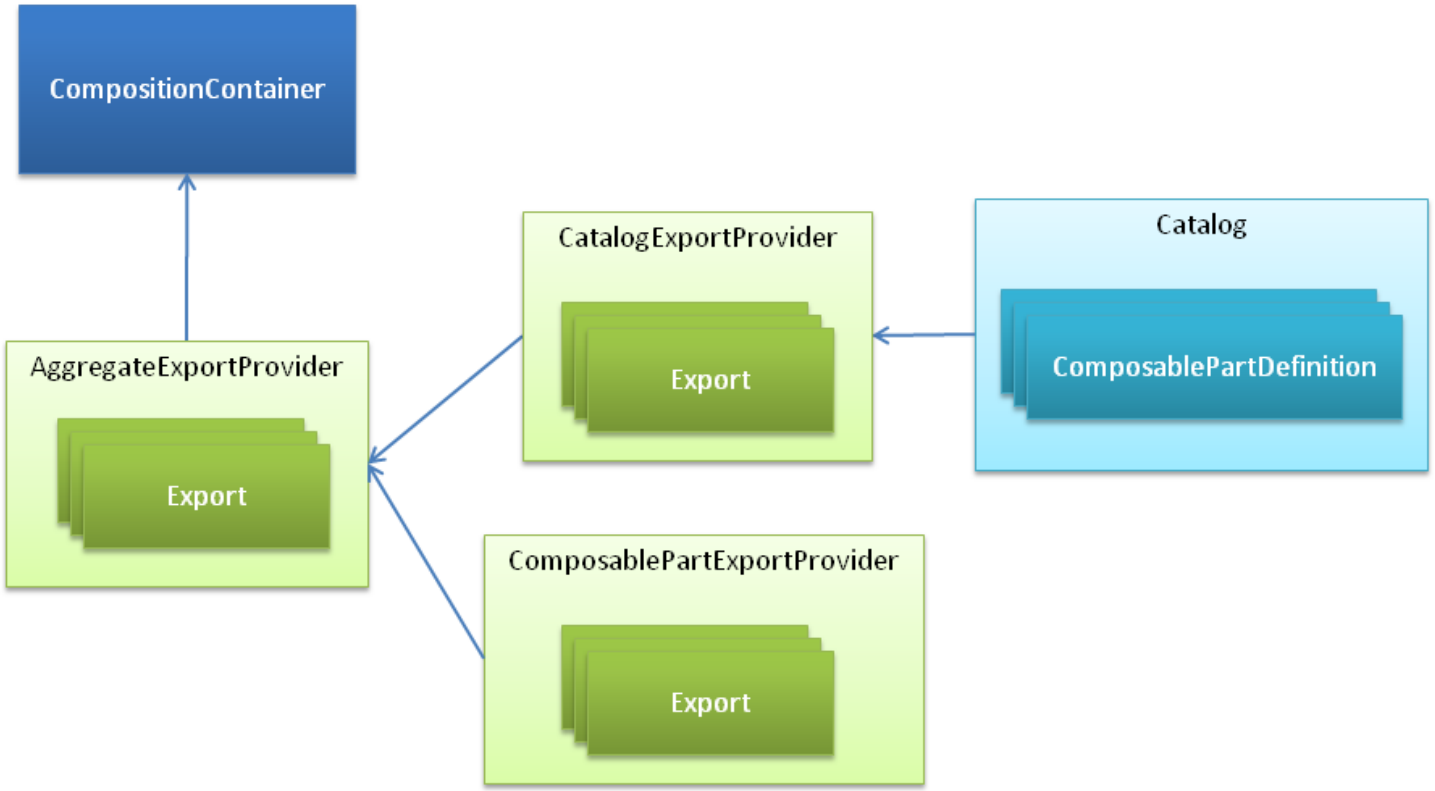
Dealing with Metadata when MEF goes Lazy

- MEF allows you to export Metadata along with the types you export.
- For example, assume that you want to specify whether your exported animals eat meat or not, so that some one can decide what to give them as food.
- One way is to create your own ExportAttribute derived class and add metadata capabilities to this class with the MetadataAttribute
  - Se example on the next slide

```
 //Our custom metadata attribute to exp animals
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
class ExportAnimal : ExportAttribute
  {
      //Pass the contract type to the base
      public ExportAnimal() : base(typeof(IAnimal)) { }
      //Additional metadata info
      public bool EatMeat { get; set; }
  }
//Concrete animal 1
[ExportAnimal(EatMeat = true)]
class Lion : IAnimal
{
   public Lion() { Console.WriteLine("Grr.. Lion got created");
      }
   public void Eat() { Console.WriteLine(
      "Grr.. Lion eating meat");
      }
}
// Use the metadata
if (animal.Metadata.EatMeat)
   animal.Value.Eat();
```

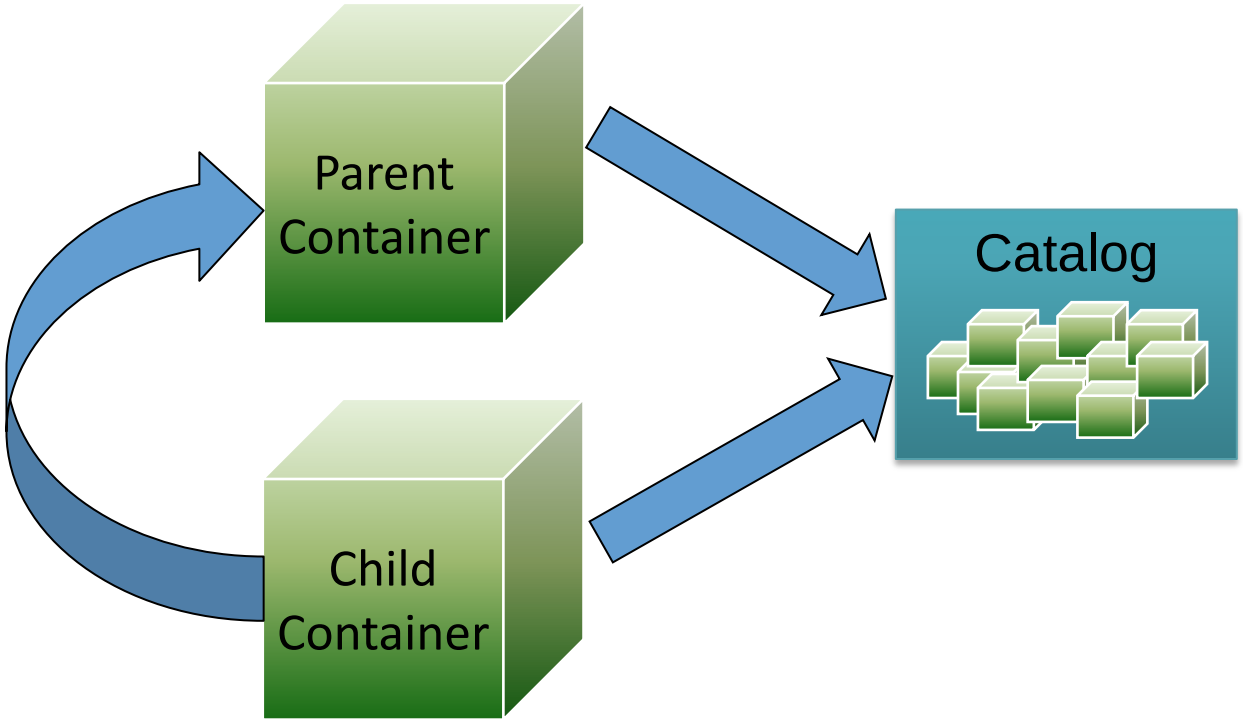
Architecture Overview

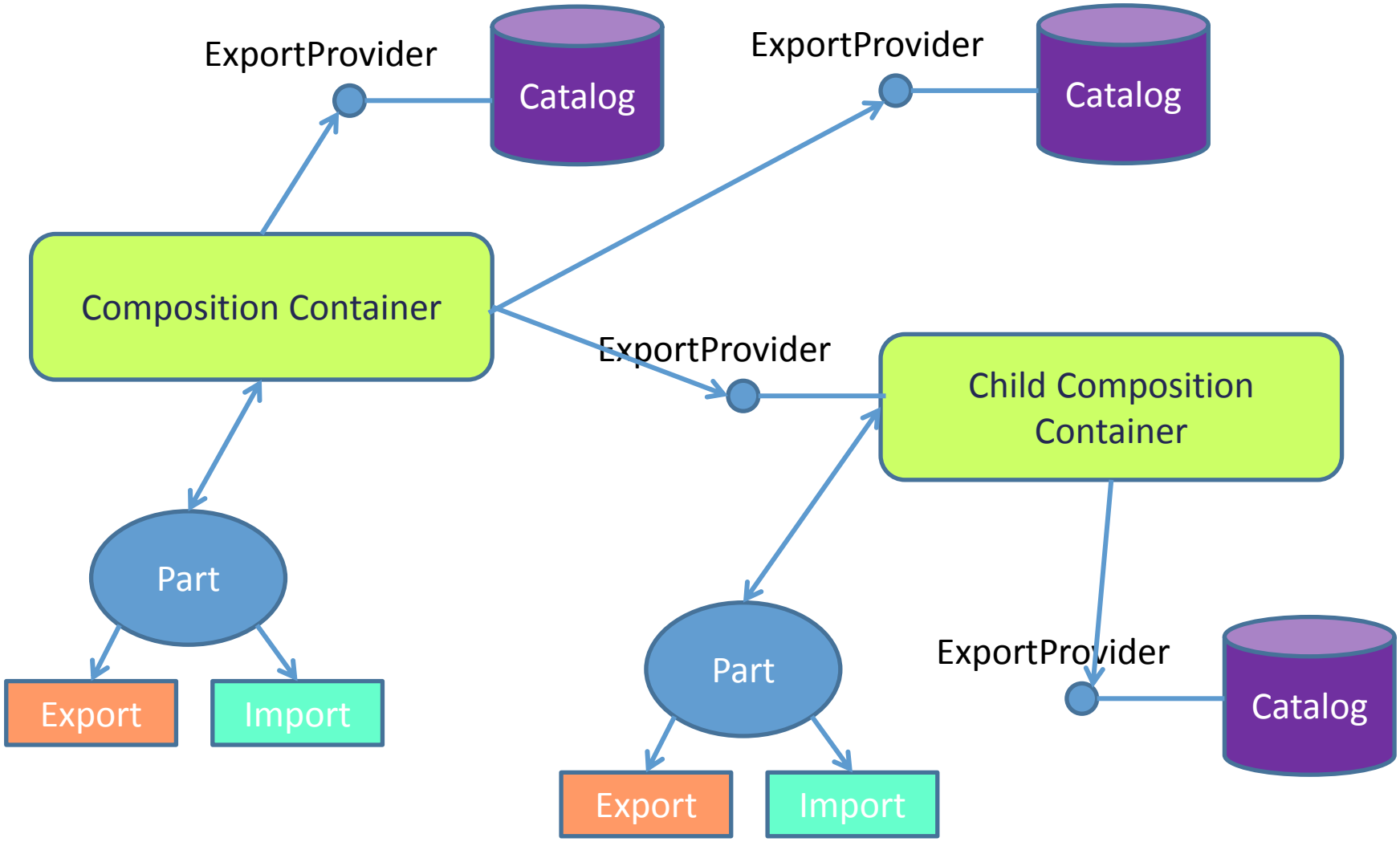
- The container is more of a coordinator and builder of a ExportProvider topology than anything else.
- The set of ExportProvider instances in use by a container instance is chained, so they can query each other for exports when satisfying dependencies.
- You can implement a custom ExportProvider to expose exports from any source – for example, WCF, an IoC Container, Remoting.



Scoping

- If you need to isolate parts within a sandbox, you can do so within a child container.
- This allows composition to happen properly within the sandbox itself, but won't impact composition done in the parent container.




ExportProvider
Catalog
ExportProvider
Catalog
Composition Container
ExportProvider
Child Composition Container
Part
Export
Import
Part
Export
Import
ExportProvider
Catalog

Limitations

When not to use MEF

MEF as DI-IoC

- MEF may seem to be very similar to your favorite IoC container.
  - But the devil is in the details.
- Here are a few details which should help you in your decision and set proper expectations about using MEF.

Centralized Configuration

- If you want centralized configuration of plain old CLR objects (POCOs) then MEF is not the right choice.
  - MEF does have TypeCatalogs which allow you pick and choose which Types you have in the container,
  - But those types still have to be attributed, thus the configuration mechanism is configuring which MEF parts show up.
  - This is because MEF's attributed model is really optimized around discovery of a set of unknown components vs configuring a pre-defined set.
- MEF v.2 do not require use of attributes so use of POCOs is supported in .Net v. 4.5.

MEF doesn't support:

- AOP (Interception) like injecting logging, or automatic transaction management.
- Open generics support
  (register IRepository<> which can handle all IRepository<T> dependencies).
- Custom lifetime support such as per-thread or per-session. MEF includes only two lifetime models, shared and non-shared.
  - To get more complex lifetimes requires manually setting up container hierarchies. It is doable, but requires work.
- If you need parameterized construction, that is creating a part where specific imports are passed in at the time of construction.

New in the .NET Framework 4.5

- Support for generic types.
- Convention-based programming model that enables you to create parts based on naming conventions rather than attributes.
- Multiple scopes.
- A subset of MEF that you can use when you create Windows Store apps.
  - This subset is available as a downloadable package from the NuGet Gallery. To install the package, open your project in Visual Studio 2012, choose Manage NuGet Packages from the Project menu, and search online for the **Microsoft.Composition** package.

```
PM> Install-Package Microsoft.Composition
```

MEF for *Windows Store apps* is in a new namespace!

References and Links

- MEF V.2
  http://mef.codeplex.com/

- MEF and MVVM
  http://mefedmvvm.codeplex.com/

- Inversion of Control with MEF
  http://www.informit.com/articles/article.aspx?p=1635818

- Lazy MEF
  http://amazedsaint.blogspot.com/2010/06/mef-or-managed-extensibility-framework.html#