Microsoft

Dependency Injection

Crystal Tenn
Crystal.Tenn@microsoft.com

What is Dependency Injection (DI)?

S
O
L
I
D

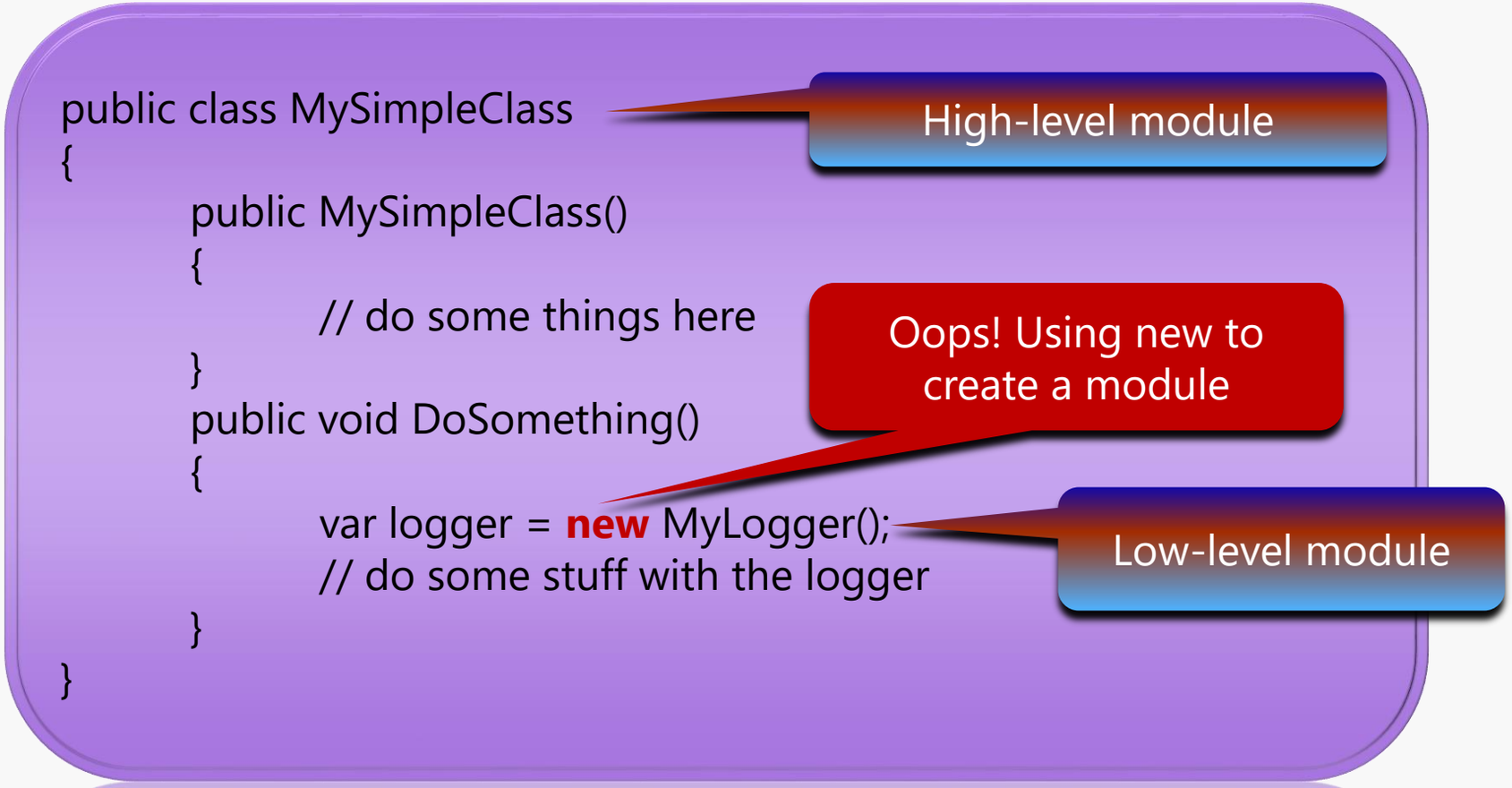
Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

In other words…

- Do not create new objects inside your modules!!!

Violation of the Dependency Inversion Principle



3 Types of DI

- There are at least three ways an object can receive a reference to an external module:

- constructor injection: the dependencies are provided through a class constructor.
- setter injection: the client exposes a setter method that the injector uses to inject the dependency.
- interface injection: the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency.

3 Types of DI

- constructor injection:

```
// Constructor
Client(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

- setter injection:

```
// Setter method
public void setService(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

3 Types of DI

- interface injection:

```
// Service setter interface.
public interface ServiceSetter {
    public void setService(Service service);
}

// Client class
public class Client implements ServiceSetter {
    // Internal reference to the service used by this client.
    private Service service;

    // Set the service that this client is to use.
    @Override
    public void setService(Service service) {
        this.service = service;
    }
}
```

Tight Coupling

- Component *responsible* for instantiating dependent classes
  - Concrete classes "*hard-coded*" in class
  - Resistant to change/difficult to test
  - Change to dependent class can directly affect component
  - Change to dependent class requires recompiling and testing component
  - Difficult to quickly *swap-out* dependent classes for testing and such
- Considered *liability* in your design

How to use DI?

What's Wrong with this Component?

```
public class StoreManagerController : Controller
{
        private AlbumService _albumService;
        private ArtistService _artistService;
        private GenreService _genreService;

        public StoreManagerController()
        {
                _albumService = new AlbumService();
                _genreService = new GenreService();
        _artistService = new ArtistService();
        }
```

Contains 3 dependencies

Has direct control over creating dependencies

Component communicates with concrete classes

Relationship said to be "Tightly Coupled"

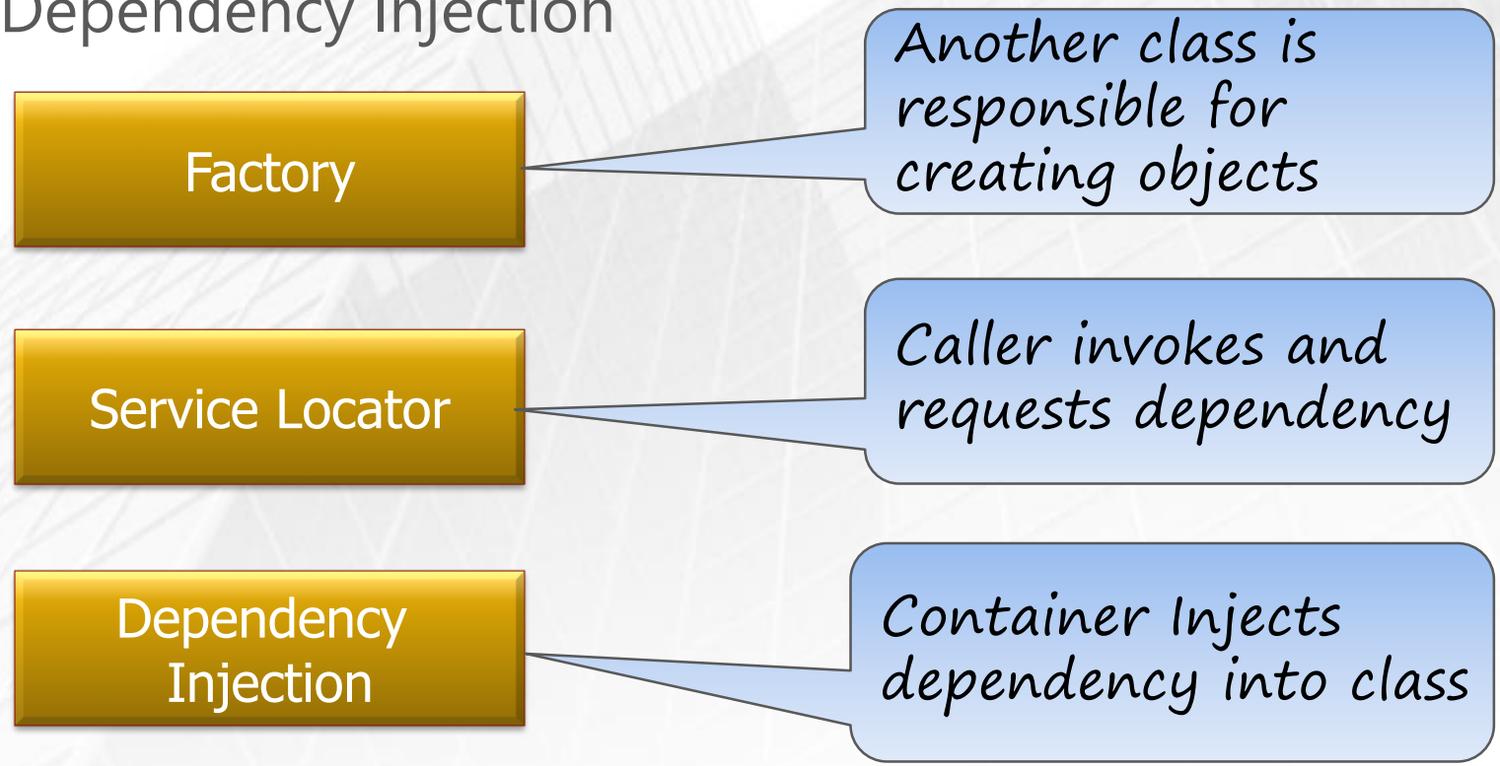
Reduce coupling by removing dependencies

```
public class StoreManagerController : Controller
{
        public StoreManagerController()
        {
                AlbumService _albumService = ???????
                GenreService _genreService = ???????
        ArtistService _artistService = ???????
        }
}
```

How do these get filled in?

Patterns for removing dependencies

- Three approaches:
  - Factory
  - Service Locator
  - Dependency Injection



Reducing Coupling – Option 1

```
public class StoreManagerController : Controller
{
    private readonly AlbumService _albumService;
    private readonly ArtistService _artistService;
    private readonly GenreService _genreService;

    public StoreManagerController()
    {
        _albumService = FactoryOrServiceLocator.CreateAlbumService();
        _genreService = FactoryOrServiceLocator.CreateGenreService();
        _artistService =  FactoryOrServiceLocator.CreateArtistService();
    }
}
```

The factory or service locator knows how to create the instance

Reducing Coupling – Option 2

- Move responsibility for instantiating dependency classes from component to consuming code

Component receives dependencies from calling class via constructor

```
public class StoreManagerController : Controller
{
        private readonly AlbumService _albumService;
        private readonly ArtistService _artistService;
        private readonly GenreService _genreService;

        public StoreManagerController(AlbumService albumService,
                                      GenreService genreService,
                                      ArtistService albumService)
        {
                _albumService = albumService;
                _genreService = genreService;
        _artistService = artistService;
        }
```

Constructor Injection

- Doing so *inverts control*
- Dependency class instantiation moved outside of component and performed by the consuming code
- Component relieved of responsibility of instantiating dependency classes
- *Two problems* still persist:
  - Calling class now more complex (select, instantiate and inject dependent classes)
  - Component still tightly-coupled to concrete classes

Embrace abstractions

- Define dependencies as interfaces
  - Constructor arguments (or properties)
- Calling class passes the instances
  - Use DI container to do this automatically
- Unit test pass in Mocks
  - A test version of the class

Reduce Coupling Further

- Add *abstraction layer* between component and dependencies by implementing *interfaces*
- Component now unaware of concrete class

```
public class StoreManagerController : Controller
{
        private readonly IAlbumService _albumService;
        private readlonly IArtistService _artistService;
        private readonly IGenreService _genreService;

        public StoreManagerController(IAlbumService albumService,
                                      IGenreService genreService,
                                      IArtistService albumService)
        {
                _albumService = albumService;
                _genreService = genreService;
        _artistService = artistService;
        }
```

Interface references

Communicates with concrete class via an interface

Refactoring to Interfaces

- Component dependent on interfaces
  - Not aware of concrete class
  - Communicates with interface
  - Talks to any reference that implements interface
- Inject different implementations of dependent class without modifying component
  - Facilitates testing
  - Pass fake implementations for unit testing

Loose Coupling

- Implement Interfaces to communicate across layers -- *enforces loose coupling*
- A layer should not expose internal details on which another layer could depend
- Controller, Services, Repositories all loosely coupled with interface references

Controller has dependency on Service - interface-based

Service has dependency on Repository - interface-based

Repository has dependency on DbContext

- One Layer can Mock or Fake another layer to isolate functionality, enabling testability across each layer

Compare the main DI frameworks Performance

Compare the main DI frameworks

- Mef
- Mef2
- Unity
- Ninject
- Castle Windsor
- Autofac
- StructureMap

Performance comparisons:

- https://github.com/danielpalme/IocPerformance
- http://www.palmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison

Compare the main DI frameworks

Microsoft released three versions of MEF using two only unique names: MEF and MEF2

- System.ComponentModel.Composition.* MEF in .NET 4 (typically called just MEF), no support for CompositionScopes, ExportFactories, etc
- System.ComponentModel.Composition.* MEF2 in .NET 4.5 (sometimes called MEF2 or MEF), support for composition scopes, ExportFactories
- System.Composition.* from independent package Microsoft.Composition lightweight version of MEF typically called MEF2

- The benchmark site refers to lightweight MEF2 System.Compostion.* from Microsoft.Composition package.

Explantions

First value: Time of single-threaded execution in [ms]

Second value: Time of multi-threaded execution in [ms]

*: Benchmark was stopped after 1 minute and result is extrapolated.

- Singleton: Objects with is singleton lifetime are resolved
- Transient: Objects with is transient lifetime are resolved
- Combined: Objects with two dependencies (singleton and transient lifetime) are resolved
- Complex: Objects with several nested dependencies are resolved

Basic Features

Container	Singleton	Transient	Combined	Complex
Autofac 4.6.1	781 616	715 556	1933 1947	6248 6452
Mef 4.0.0.0	22679 11820	37640 25052	57462 68730*	112712* 131716*
Mef2 1.0.32.0	309 217	267 174	363 241	693 411
Ninject 3.3.0	2673 1831	8121 6143	23986 16122	69556* 50795
StructureMap 4.5.2	1183 656	1306 800	3471 2036	8933 5270
Unity 4.0.1	2517 1375	3761 1962	10161 5372	27963 16013
Windsor 4.1.0	459 289	1772 1050	6018 3601	19319 10972

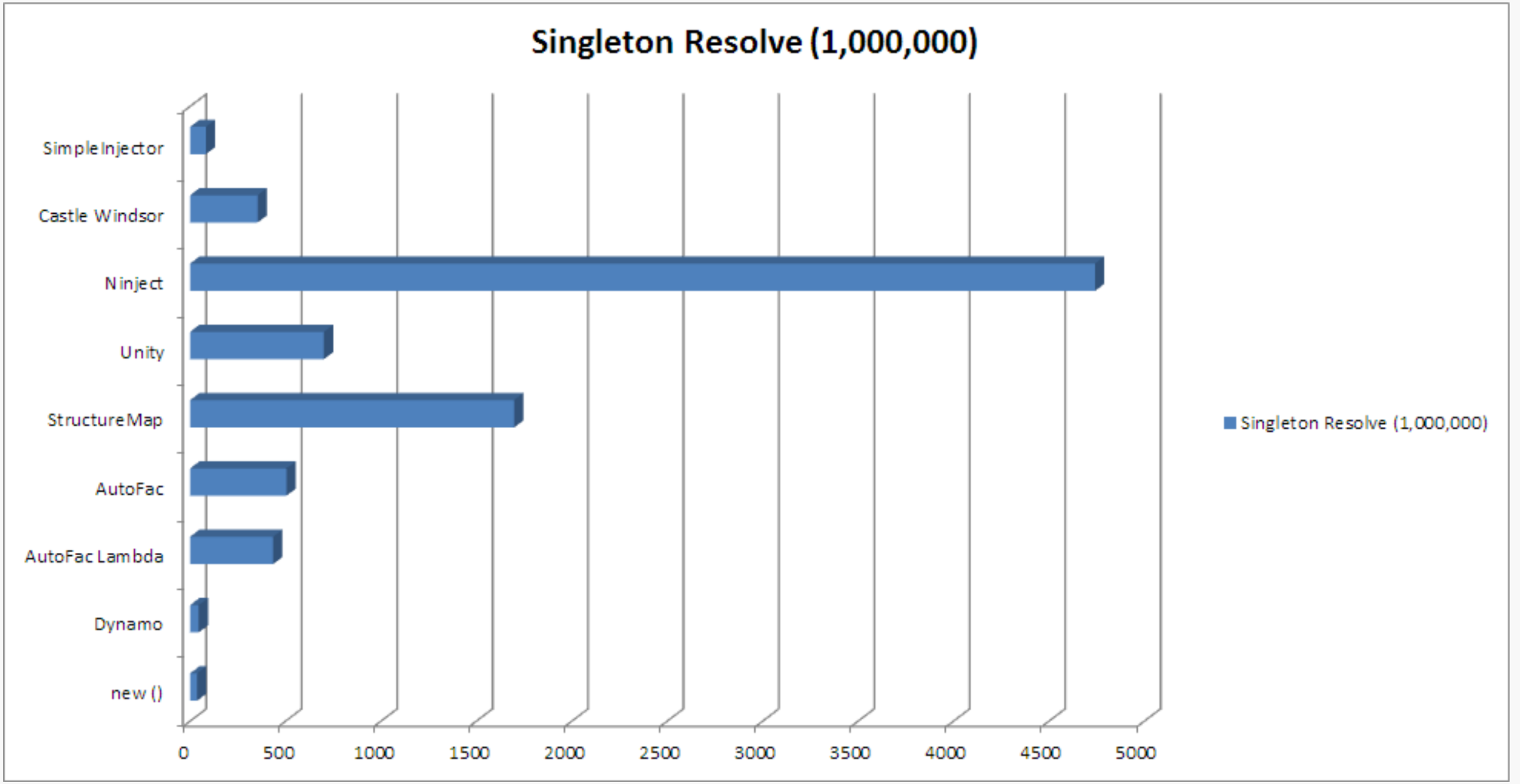
Mef2, Autofac, StructureMap, and Castle Windsor and are the fastest DI frameworks, though it depends on what type of objects.

Mef, Unity, and Ninject are the slowest.

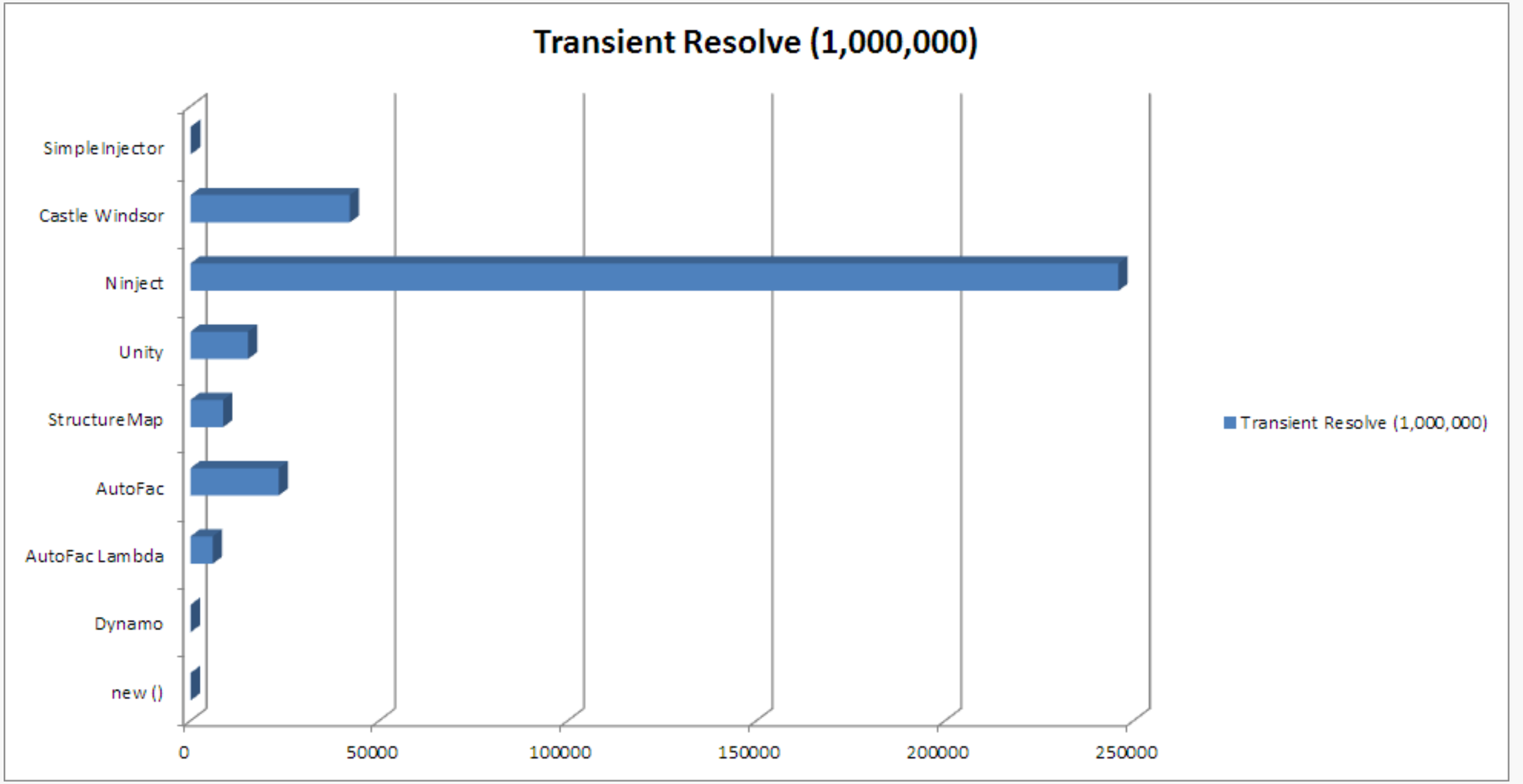
Compare the main DI frameworks

- https://github.com/danielpalme/IocPerformance

- http://www.palmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison



https://cardinalcore.co.uk/2015/01/28/ioc-battle-in-2015-results-using-ninject-think-again/



https://cardinalcore.co.uk/2015/01/28/ioc-battle-in-2015-results-using-ninject-think-again/

Microsoft

Mef

Crystal Tenn
Crystal.Tenn@microsoft.com

Managed Extensibility Framework

What is it?

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

MEF Basics…

An Application is built of *parts*.

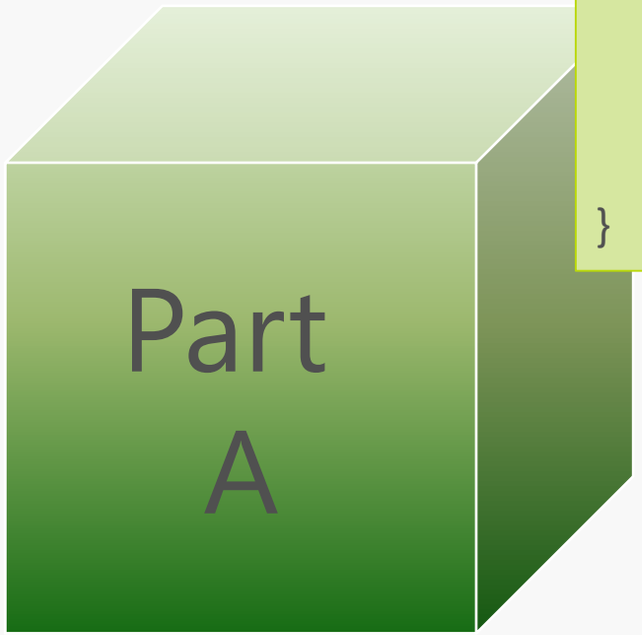
MEF Basics…

Export it.

Import it.

Compose it.

Part, enter stage left…

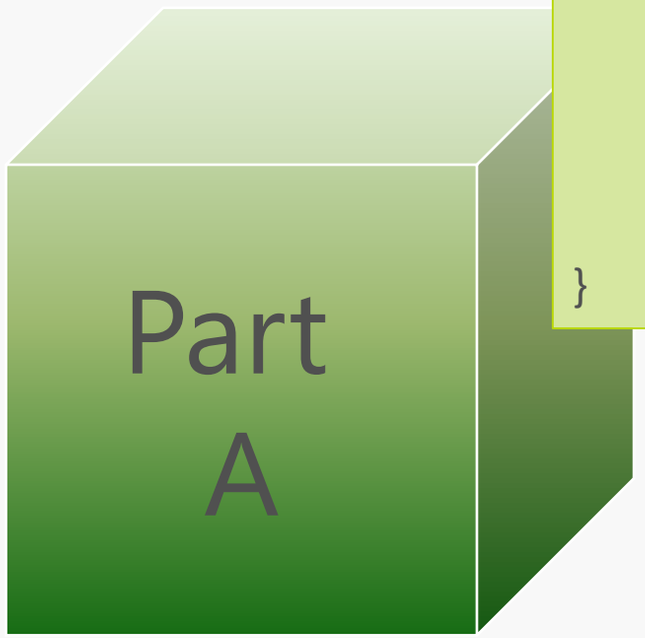



```
public class SimpleMortgageCalculator : IMortgageCalculator
{
        public ILogger Logger { get; set; }

        public float Calculate()
        {
                Logger.Log("Calculating Mortgage");

                return ...;
        }
}
```

Export

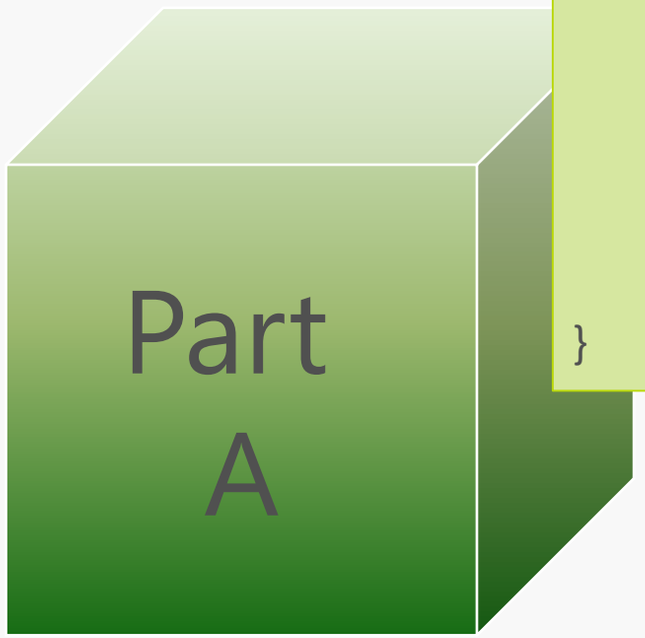



```
[Export(typeof(IMortgageCalculator))]
public class SimpleMortgageCalculator : IMortgageCalculator
{
        public ILogger Logger { get; set; }

        public float Calculate()
        {
                Logger.Log("Calculating Mortgage");

                return ...;
        }
}
```

Importit...



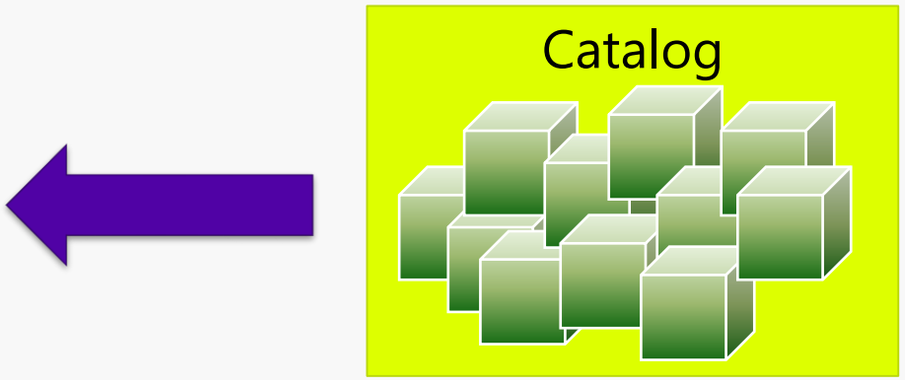

```
[Export(typeof(IMortgageCalculator))]
public class SimpleMortgageCalculator : IMortgageCalculator
{
        [Import(typeof(ILogger))]
        public ILogger Logger { get; set; }

        public float Calculate()
        {
                Logger.Log("Calculating Mortgage");

                return ...;
        }
}
```

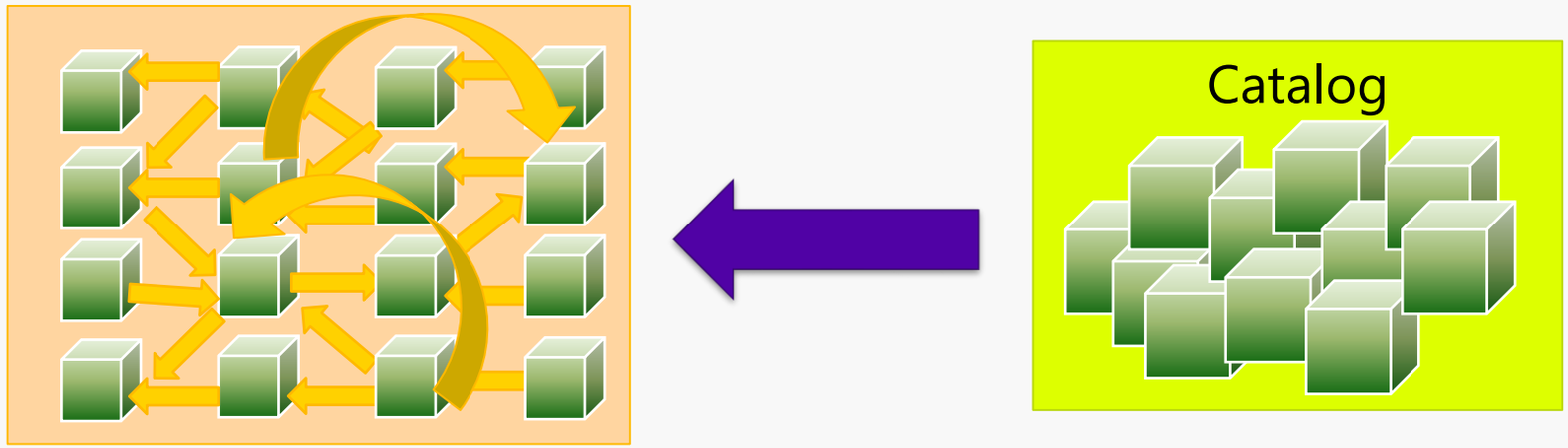
Compose it.

Catalogs provide the parts.



Compose it.

Container is the matchmaker.



Catalog Categories

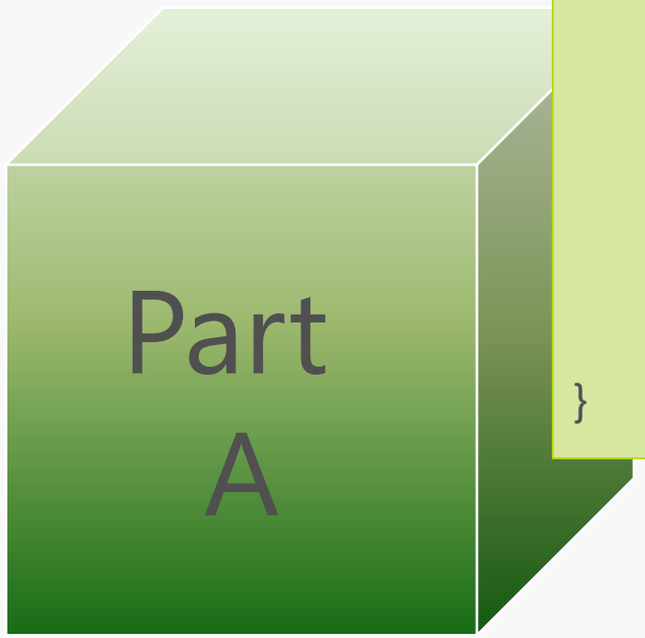
Aggregating Catalog

Directory Catalog

Assembly Catalog

Type Catalog

Metadata…



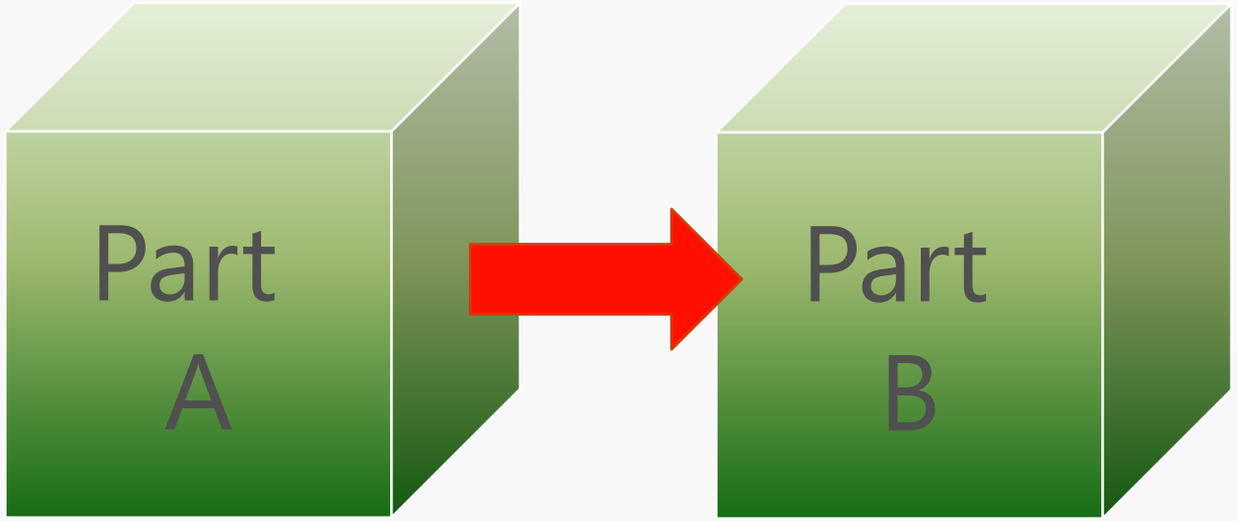

```
[Export(typeof(IMortgageCalculator))]
[ExportMetadata(“Calculation”, “Simple”)]
[ExportMetadata(“Tax Aware”, null)]
public class SimpleMortgageCalculator : IMortgageCalculator
{
        public ILogger Logger { get; set; }

        public float Calculate()
        {
                Logger.Log("Calculating Mortgage");

                return ...;
        }
}
```

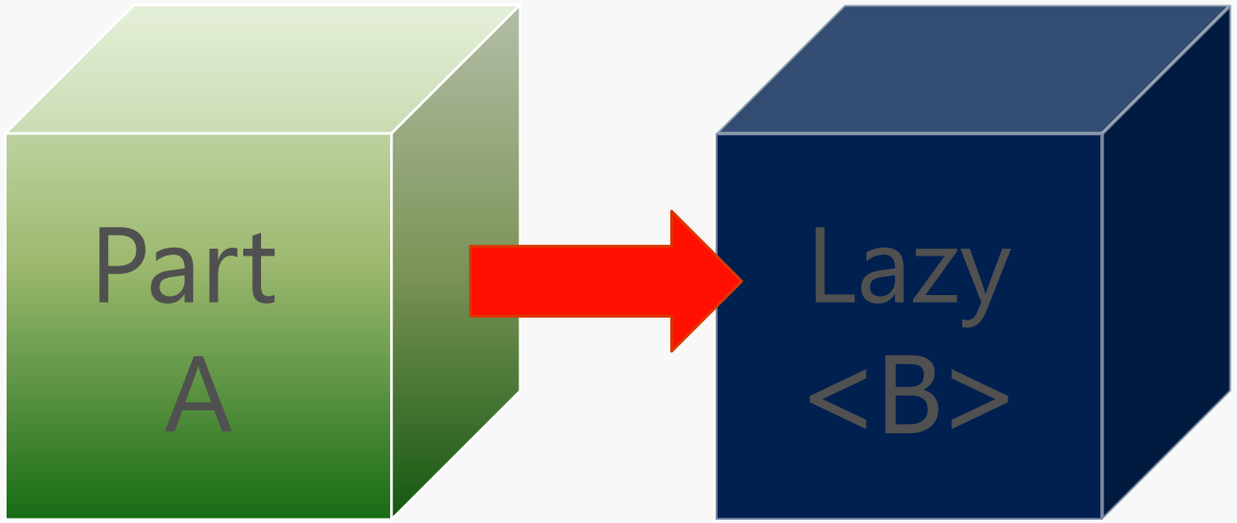
Parts can be lazy…

```
[Import(typeof(ILogger))]
public ILogger Logger { get; set; }
```



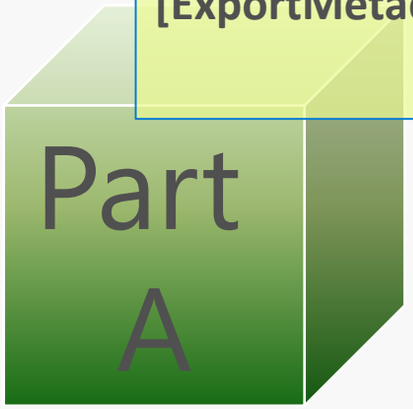
Parts can be lazy…

```
[Import(typeof(ILogger))]
public ~~ILogger~~Lazy<ILogger>Logger { get; set; }
```



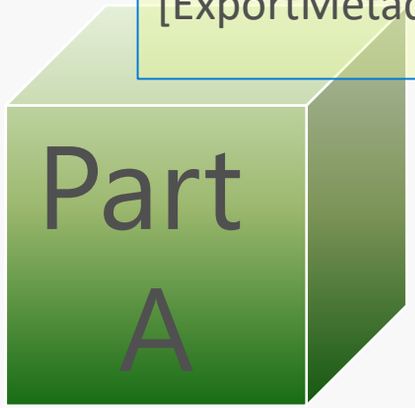
The slippery slope…

```
[Export(typeof(IMortgageCalculator))]
[ExportMetadata("Calculation", "Simple")]
[ExportMetadata("Tax Aware", null)]
[ExportMetadata("This", "foo")]
[ExportMetadata("That", "bar")]
```



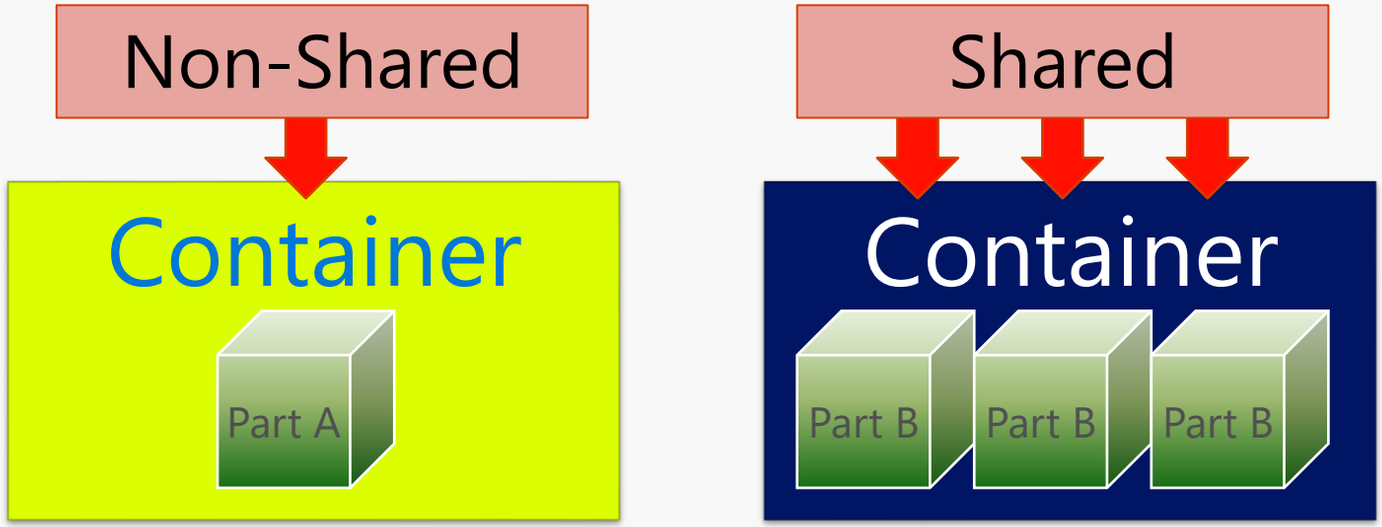
The slippery slope… solved

```
[Export(typeof(IMortgageCalculator))]
[ExportMetadata(“Calculation”, “Simple”)]
[ExportMetadata(“Tax Aware”, null)]
[ExportMetadata(“This”, “foo”)]
[ExportMetadata(“That”, “bar”)]
```

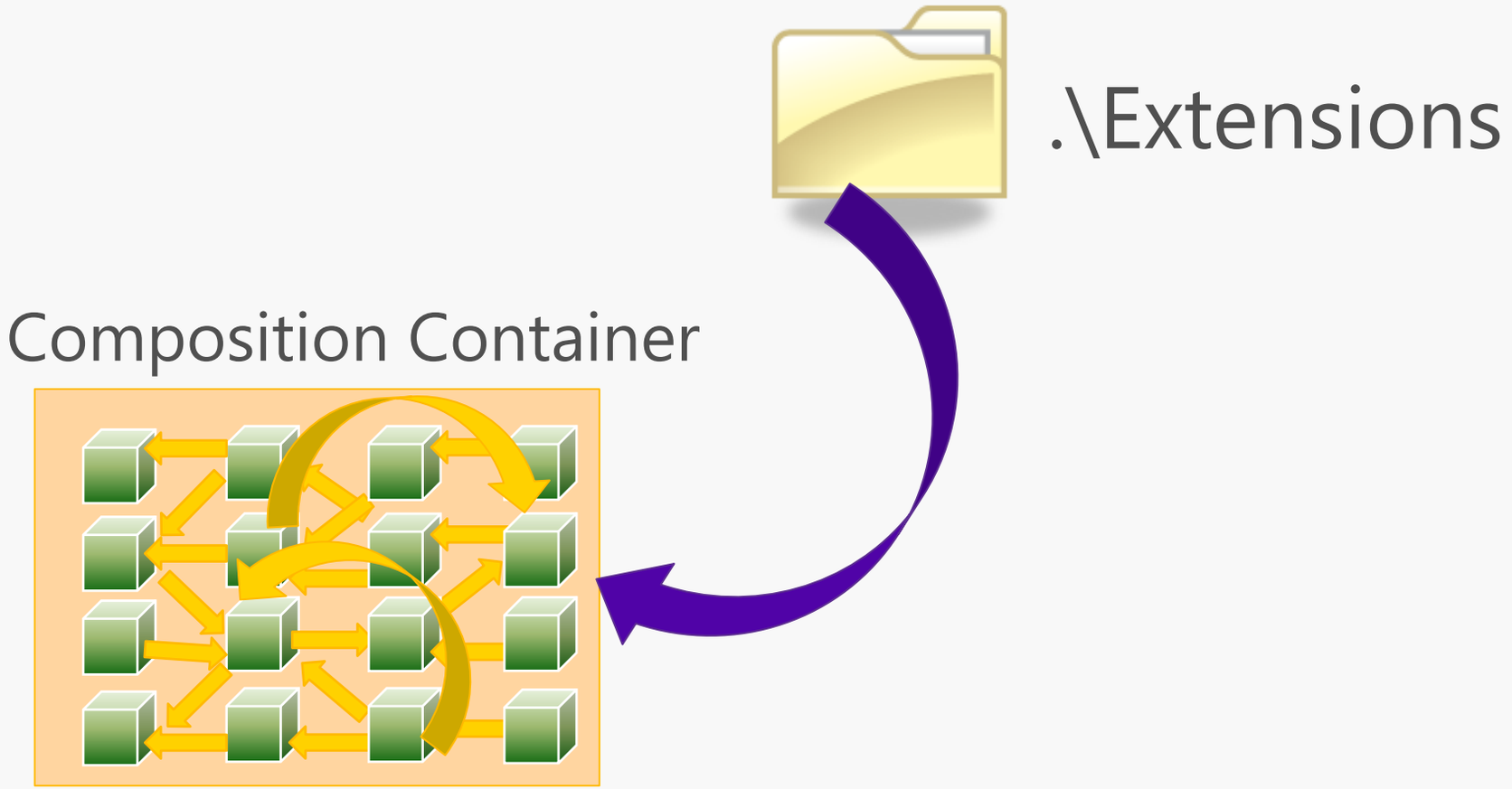



```
[Export(typeof(IMortgageCalculator))]
[CalcCapabilities(
        Mode=Complexity.Simple,
        TaxAware=true,
        This=“foo”,
        That=“bar”)]
```

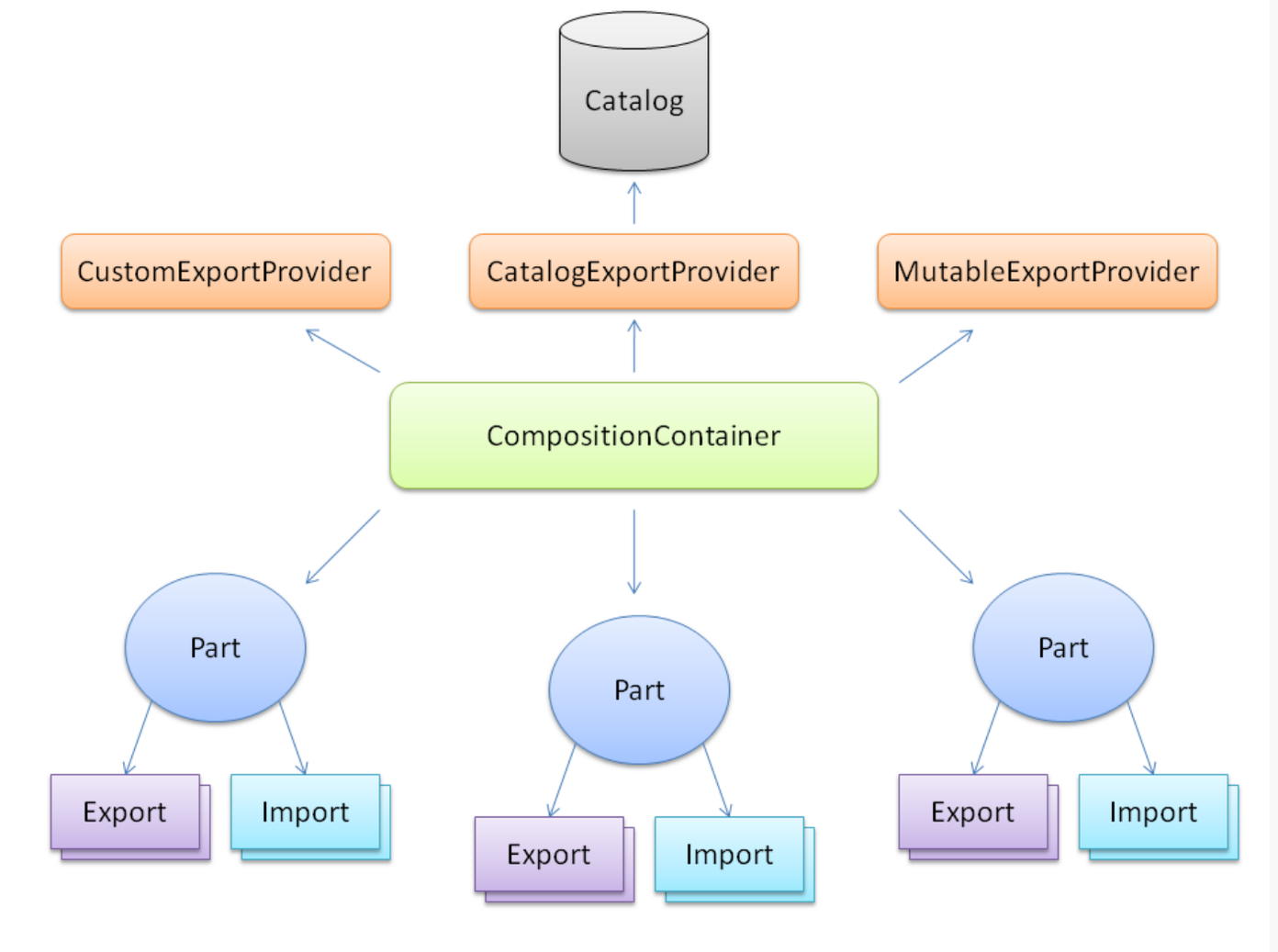
Lifetime



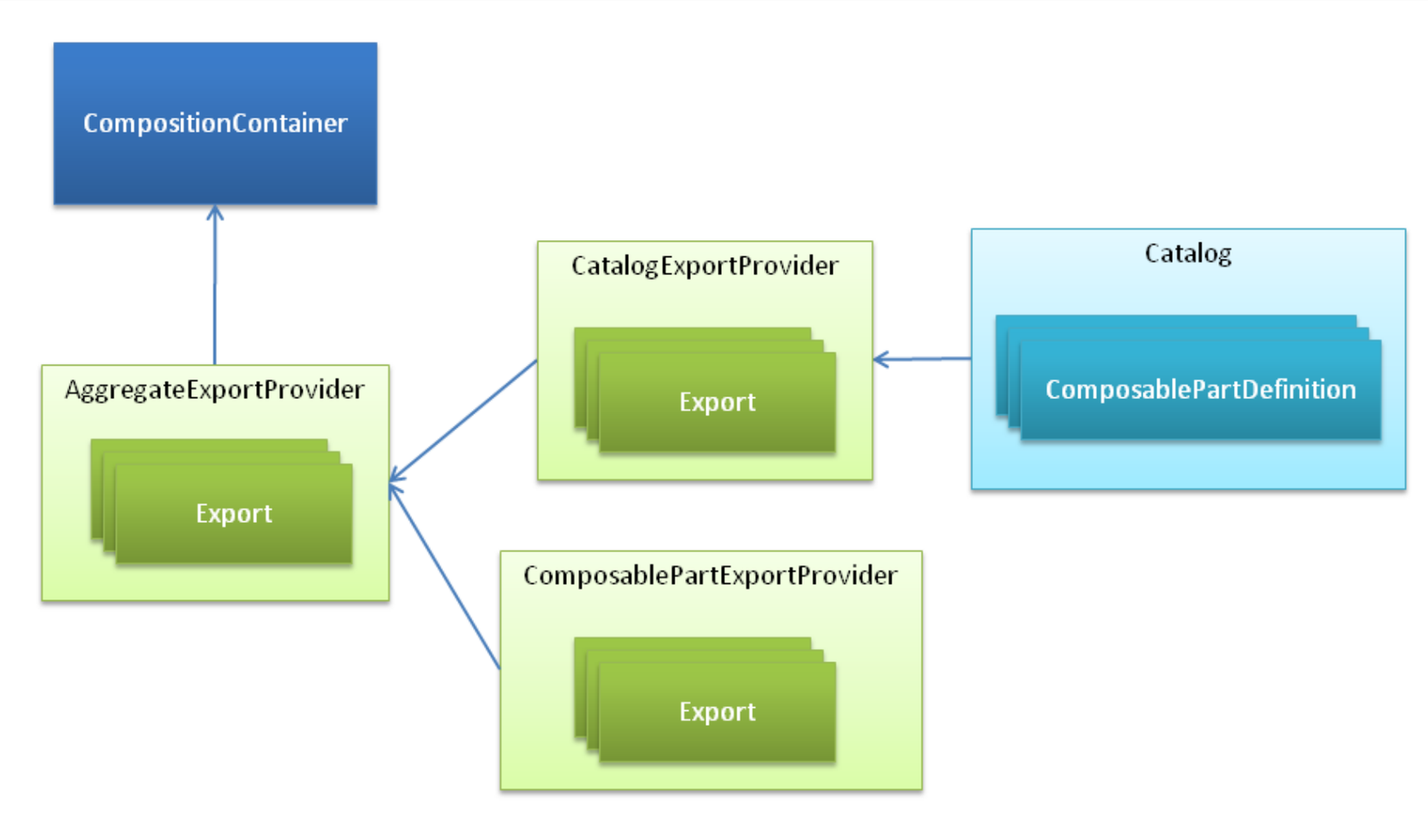
Dependencies



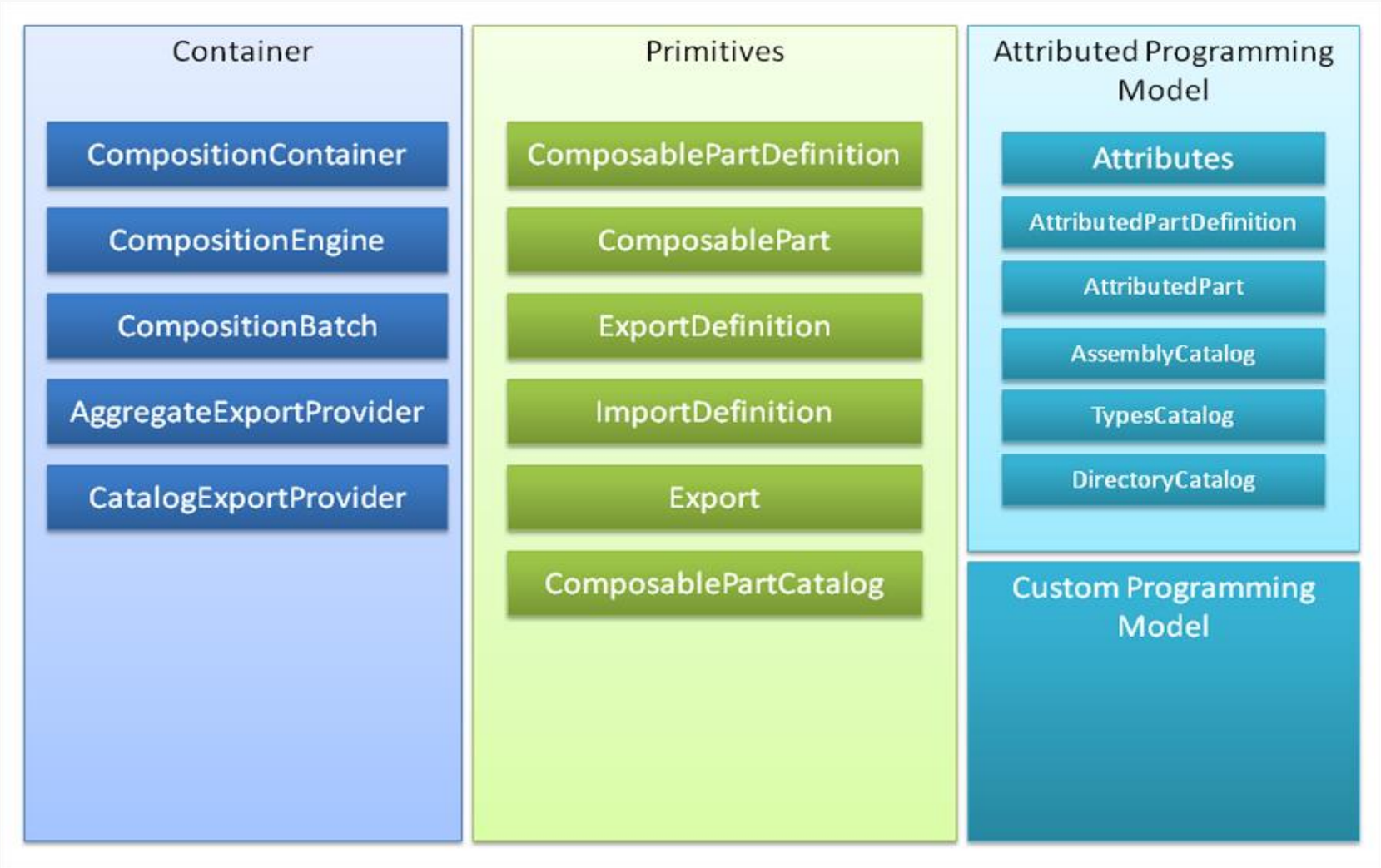
MEF Architecture



MEF Architecture



MEF Container



Microsoft

Unity

Crystal Tenn
Crystal.Tenn@microsoft.com

Dependency Injection Lifecycle

- Register: tell the container how to instantiate an object
- Resolve: instantiate the object
- Dispose: object becomes available for garbage collection

Register

```
var container = new UnityContainer();
container.RegisterType<IMyInterface, MyClass>();
```

Resolve

```
var service = container.Resolve<MyController>();
```

Simple Code Example of DI

- A asdasdasdasdafor running large-scale parallel and high-performance computing (HPC) applications efficiently in the cloud.

- Azure Batch schedules compute-intensive work to run on a managed collection of virtual machines, and can automatically scale compute resources to meet the needs of your jobs.

When NOT to use DI.

When NOT to use DI

- Small projects
  - It can be overkill for simple small projects and add complexity for no good reason
- Functional programs (as opposed to object oriented)
- Some legacy applications or pre-existing ones with a difficult architecture
  - If you try to implement DI's specific way of layering and decoupling onto an existing old application it could cause problems if the original app was not build with inversion of control in mind.
- If you have many junior developers on the team, it could pose an issue and extra confusion to them. Consider taking time out to teach the junior developers how to use this if they will work on a project with DI.

Labs on Dependency Injection Frameworks

Please complete all 3 labs

- 04a DI with Mef
- 04b DI with Mef2
- 04c DI with Unity

Microsoft

Thank you!