



# Microsoft Unity

**Succinctly**

by Ricardo Peres

# Microsoft Unity Succinctly

---

By

Ricardo Peres

Foreword by Daniel Jebaraj



Copyright © 2014 by Syncfusion Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

## **I** mportant licensing information. Please read.

This book is available for free download from [www.syncfusion.com](http://www.syncfusion.com) on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed for reading only if obtained from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

**Technical Reviewer:** Pedro Gomes

**Copy Editor:** Suzanne Kattau

**Acquisitions Coordinator:** Hillary Bowling, marketing coordinator, Syncfusion, Inc.

**Proofreader:** Darren West, content producer, Syncfusion, Inc.

# Table of Contents

<b>The Story behind the <i>Succinctly</i> Series of Books .....</b>	<b>8</b>
<b>About the Author.....</b>	<b>10</b>
<b>Introduction .....</b>	<b>11</b>
<b>Chapter 1 Installing Unity.....</b>	<b>13</b>
Introduction .....	13
NuGet Package.....	13
Installation Package.....	13
Source Code .....	14
<b>Chapter 2 Inversion of Control (IoC) .....</b>	<b>15</b>
Introduction .....	15
Setup.....	15
Domain Model .....	16
Registration by Code .....	17
Instance vs. Type.....	17
Generic Types.....	19
Creation Factories.....	19
Base Types .....	20
Registration by XML Configuration .....	20
XML Configuration Section .....	20
Generic Types.....	21
Type Aliases and Assembly Discovery.....	22
IntelliSense .....	23
Loading XML Configuration .....	24
Registration by Convention.....	24

Component Resolution .....	25
Resolving an Instance .....	25
Resolving Multiple Instances .....	26
Resolving with Overrides .....	26
Deferred Resolution .....	28
Testing If a Registration Exists .....	28
Lifetime Managers .....	29
Child Containers .....	34
Common Service Locator .....	35
<b>Chapter 3 Dependency Injection .....</b>	<b>37</b>
Introduction .....	37
Injection Types .....	37
Auto Injection .....	37
Configuring Injection by Attributes .....	38
Constructor Injection .....	38
Property Injection .....	39
Method Injection .....	39
Configuring Injection by Code .....	40
Constructor Injection .....	41
Property Injection .....	42
Method Injection .....	42
Factory Injection .....	43
Putting It All Together .....	44
Configuring Injection by XML .....	44
Constructor Injection .....	44
Property Injection .....	45
Method Injection .....	46

Explicit Injection .....	46
<b>Chapter 4 Aspect-Oriented Programming (AOP).....</b>	<b>48</b>
Introduction .....	48
Interception Kinds and Techniques .....	48
Adding Interfaces .....	50
Call Handlers .....	50
Interception Behaviors .....	53
Configuring Interception by Attributes.....	55
Configuring Interception by Code .....	57
Applying Interception .....	57
Interception Behaviors .....	57
Call Handlers .....	58
Adding Interfaces to Interception Proxies.....	59
Configuring Interception by XML .....	62
Applying Interception .....	62
Interception Behaviors .....	62
Call Handlers .....	63
Adding Interfaces to Interception Proxies.....	64
Applying Interception in an Existing Instance .....	64
Included Call Handlers.....	65
<b>Chapter 5 Extending Unity .....</b>	<b>66</b>
Introduction .....	66
Lifetime Managers .....	66
Convention Classes .....	69
Extensions .....	71
Call Handlers .....	75
Injection Values.....	79

Injection Factories in XML Configuration .....	86
Policy Rules .....	87
<b>Chapter 6 Other Application Programming Interfaces (APIs) .....</b>	<b>90</b>
Introduction .....	90
ASP.NET Web Forms .....	90
Injecting Dependencies .....	90
Disposing of Components.....	91
ASP.NET MVC.....	92
Resolving Components.....	92
Injecting Controllers .....	94
Registration.....	95
Injecting Action Method Parameters.....	95
Disposing of Components.....	96
ASP.NET Web API .....	96
Component Resolution .....	96
Dependency Injection in Controllers .....	99
Registration.....	101
Dependency Injection in Action Methods .....	101
Entity Framework .....	103
Component Resolution .....	103
Registration.....	105
<b>Chapter 7 Putting It All Together .....</b>	<b>106</b>
<b>Chapter 8 Getting Help .....</b>	<b>107</b>
<b>Additional References .....</b>	<b>108</b>

# The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President  
Syncfusion, Inc.

**S**taying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

## Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

## The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

## The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.



## Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

## Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click” or “turn the moon to cheese!”

## Let us know what you think

If you have any topics of interest, thoughts or feedback, please feel free to send them to us at [succinctly-series@syncfusion.com](mailto:succinctly-series@syncfusion.com).

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



## About the Author

Ricardo Peres is a Portuguese developer who has been working with .NET since 2001. He's a technology enthusiast and has worked in many areas, from games to enterprise applications. His main interests nowadays are enterprise application integration and web technologies. For the last 10 years, he has worked for [Critical Software](#), a multinational company based in Portugal. He keeps a blog on technical subjects entitled "[Development With A Dot](#)" and can be followed on Twitter [@riperes75](#).

Ricardo Peres also authored *Entity Framework Code First Succinctly* and *NHibernate Succinctly* for the Succinctly series.

# Introduction

This is a book about Microsoft Unity, Microsoft's implementation of an Inversion of Control (IoC), Dependency Injection (DI), and Aspect-Oriented Programming (AOP) container and library. Unity is part of the Enterprise Library, a collection of Application Blocks containing features not found in the core .NET framework but which are usually quite useful when writing enterprise applications.

So, now that we've heard the buzzwords, what exactly is Unity used for? Well, Unity helps us in writing decoupled applications by adhering to the design principles known as [SOLID \(Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion\)](#), specifically, the **Liskov Substitution Principle** that states:

*“Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.”*

And the **Dependency Inversion Principle**:

*“One should depend upon abstractions. Do not depend upon concretions.”*

What these two principles tell us is that we should expose classes containing business logic through interfaces or abstract base classes that define their contract, and write our programs to this contract. This way, we are shielded from any implementation changes that may occur, and it makes it possible to change an implementation for another altogether. We are, therefore, decoupling concrete implementations from their contracts and minimizing dependencies between code which, in the end, makes programs easier to maintain and understand.

How exactly does Unity achieve this? Well, we register our contracts for those parts of our business layer that are more likely to change or to have different implementations, and we map them to concrete implementations. In the code, we ask Unity for an implementation of these contracts and don't care where that implementation comes from or what exactly it is (I am oversimplifying, of course, but I think you get the picture).

Other than that, Unity also helps us manage complex component dependencies. Imagine that you have a class that should be initialized with a property of some type. This type needs to receive in its constructor a number of parameters and, after its creation, needs to have an initialization method called upon it. The constructor parameters are themselves also of complex types and so on. See what I mean? Things can get really complicated. Unity allows us to define rules for creating each of these components and for injecting them in the proper places so that everything is ready to work.

Finally, Unity also helps in applying cross-cutting behavior to our code. This is not actual business logic but helper functionality that allows us to just write code to do the core tasks that it is meant to do, and have things such as access control, logging, exception handling, etc., applied automatically.

Hopefully, throughout this book you will understand how Unity helps us to do all of this. In the end, I will talk about how Unity integrates with modern Microsoft stack technologies.

# Chapter 1 Installing Unity

## Introduction

Unity does not require the installation of any software on your machine; just add the proper assemblies as references to your project and you're done. These assemblies can be obtained in one of two ways:

- [NuGet](#) package (preferred)
- Installation package
- Cloning the source code repository and building it ourselves

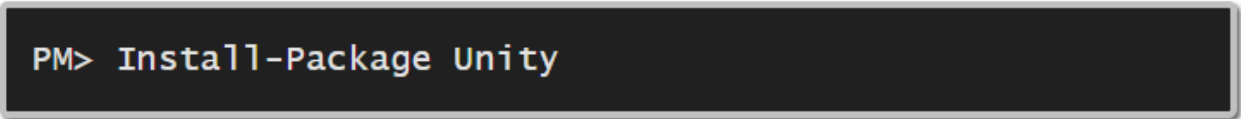
In this book we will be using Unity version 3.5, the latest version available when the book was written. This version requires .NET 4.5, which means we will need Visual Studio 2012 or 2013 (the Express editions will do).

Since Unity was rewritten using the Portable Class Library (PCL), the following platforms are supported:

- Windows Store Apps 8.0 and 8.1
- Windows Phone Silverlight 8.0 and 8.1
- .NET 4.5+
- Xamarin/Mono

## NuGet Package

Inside Visual Studio, in a .NET project, just issue the following command in the **Package Manager Console**:



```
PM> Install-Package Unity
```

*Figure 1: Installing Unity as a NuGet package*

Visual Studio will only add the Unity assembly to the startup project in your solution. If you want, you can add them to any other projects by selecting **Manage NuGet Packages for Solution**.

## Installation Package

Get the installer for Unity, which also includes its source code and quick start code, from Microsoft's [web site](#).

## Source Code

If you prefer to have a look at the source code, you are welcome to clone its repository. You can do so using Git:

```
git clone https://git01.codeplex.com/unity
```

After you have the source code in your local disk, you can open the Visual Studio solution (**Unity.sln**), compile everything, and then add the generated assemblies as references to your project.

If you wish, you can browse the current version of the source code online; just point your browser to [CodePlex](http://unity.codeplex.com), Microsoft's free open-source project hosting site.

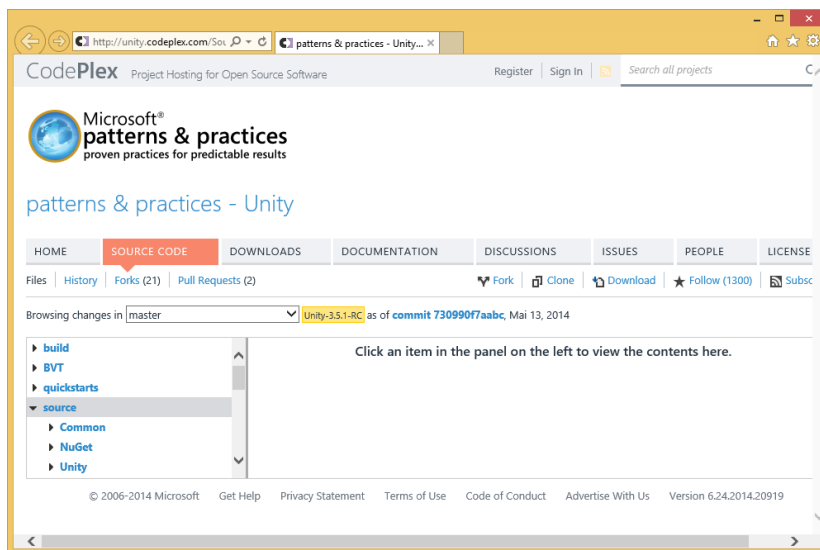


Figure 2: Browsing the Unity repository online

# Chapter 2 Inversion of Control (IoC)

## Introduction

Inversion of Control (IoC) is a software design pattern in which the developer does not explicitly specify the concrete class that will be instantiated; instead, the developer asks for an abstract base class or an interface that defines the functionality he or she wants. This increases decoupling by allowing the actual concrete class to be changed at a later stage or be dynamically selected based on some criteria. You are not tied to a particular implementation but, instead, rely on your system to provide you a proper one (you have the control without actually needing to care what it is).

So, instead of:

```
var logger = new ConsoleLogger();
```

You will have:

```
var logger = unity.Resolve<ILogger>();
```

Of course, you have to tell Unity how to do that. The following sections talk about precisely that.

## Setup

First things first: You need to have an instance of the Unity container in your code. Save it in an instance or static field in your bootstrap class:

```
internal static readonly IUnityContainer unity = new UnityContainer();
```

That's all it takes. The [IUnityContainer](#) interface defines the contract of the Unity container, which is itself implemented by the [UnityContainer](#) class. Do change the visibility to match your requirements but try to keep it as restricted as possible. You will, most likely, need only a single instance of the Unity container. Later on we'll see a pattern for accessing Unity in an indirect, more decoupled way (



**Tip:** *Don't forget that `Dispose` is only called for some lifetime managers.*

Since `IUnityContainer` implements `IDisposable`, it is a good principle to always create child containers in a **using** block so that they are properly disposed of when no longer needed.

A child container exposes the same interface as the parent—`IUnityContainer`—which has a `Parent` property but does not keep track of its children.

A typical scenario in which you would use a child container would be in a web application request. We'll look at that when we talk later about Unity's integration with Introduction

A number of technologies in the Microsoft stack use IoC. In fact, its number seems to be growing every time a new version or technology is launched. It is generally possible to plug in an IoC container of choice; in our case, let's see how to use Unity.

ASP.NET Web Forms.

Common Service Locator).

It is important to keep in mind that the Unity container, depending on its configuration (more on this in section Lifetime Managers) may dispose of the registered types automatically; for this to happen, you must explicitly dispose of it. In this case, it might make more sense to have something like:

```
using (var unity = new UnityContainer())  
{  
    //program goes here  
}
```

## Domain Model

For the purpose of our conversation, we will use a simple class model:

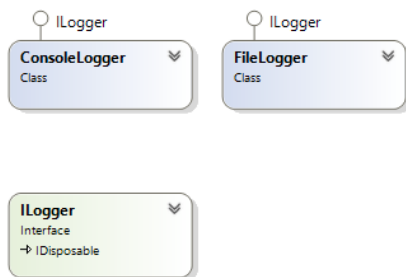


Figure 3: Logger class model



Here we have an **ILogger** interface that defines our logging contract and two concrete implementations: **ConsoleLogger**, which logs to the console (would you know?) and **FileLogger**, which logs to a file in the disk. Let's forget about the actual implementation of these classes; it's not really important.

## Registration by Code

### Instance vs. Type

We need to tell Unity which concrete classes (**components**) we want to register and through which interfaces or abstract base classes they can be accessed (**keys**). Remember, the whole point of IoC is not knowing beforehand which concrete class is to be returned. The concept is that we register a concrete class under one of its base types or implementing interfaces. This is a requirement; the component class must be convertible to its key. If you are to register a **FileLogger**, you need to use as its key one of its base classes (such as **Object**, which is not really useful) or one of the interfaces that it implements (in this case, **ILogger**, which is what we want but could also be [IDisposable](#), which is implemented by **ILogger**).

The [IUnityContainer](#) interface exposes some methods that allow us to register dynamic (untyped) components by code and there are also some extension methods for strongly-typed components that wrap the former in the [UnityContainerExtensions](#) class. The generic extension method allows for writing compile-time safe code and should generally be preferred. In the end, it all comes down to two methods:

- [RegisterInstance](#): Takes an existing instance of a class and makes it available through a base class or an interface that the class exposes.
- [RegisterType](#): Associates some base type (class or interface) with a concrete type.

Some of these methods take a name but those who don't just assume a **null** name. It is possible to have a type registered several times but with different names:

```
unity.RegisterType<ILogger, ConsoleLogger>("Console");  
unity.RegisterType<ILogger, FileLogger>("File");  
unity.RegisterInstance<ILogger>(new ConsoleLogger());
```

What we have here is multiple strong-typed registrations for the **ILogger** type:

- One that maps type **ILogger** to type **ConsoleLogger**, with a name of **Console**,
- Another that maps **ILogger** to **FileLogger**, with a name of **File** (how predictable),
- And finally, a mapping of **ILogger** to a concrete instance of a **ConsoleLogger**, without a name.

We can also do that dynamically:

```
unity.RegisterType(typeof(ILogger), typeof(ConsoleLogger), "Console");
unity.RegisterType(typeof(ILogger), typeof(FileLogger), "File");
unity.RegisterInstance(typeof(ILogger), new ConsoleLogger());
```



**Note:** Remember that a registration is composed of a base type (class or interface) and a name. Every subsequent registration of a type with the same name—or lack of it—overwrites the previous one.



**Tip:** By default, only types with public, parameterless constructors can be used with `RegisterType`. Later on we'll see how to go around this limitation by having Unity inject dependencies automatically.

Of course, it is also possible to register the same concrete type or instance several times for different keys. The type just needs to inherit from or implement all of them:

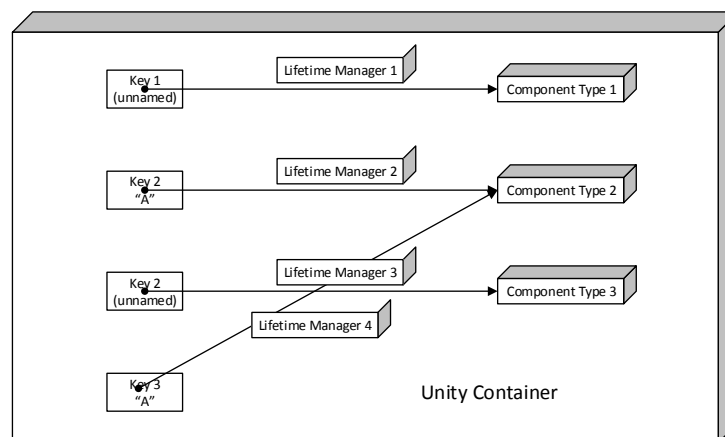


Figure 4: Type registrations inside Unity

Why would you use [RegisterInstance](#) instead of [RegisterType](#)? Well, imagine that you want to store an object that you obtained somewhere else, maybe with complex configuration applied upon it, or your class does not expose a public, parameterless constructor. In these cases, you would need to register the instance that you have because, out of the box, Unity wouldn't know how to build it.



**Tip:** Objects registered by `RegisterInstance` are singletons and those registered by `RegisterType` are created by default every time they are requested.



**Note:** Unity always registers itself automatically under the `IUnityContainer` key, with no name.



**Note:** Try to keep all of your registrations in the same place (bootstrap method) as this makes them easier to find and change.

## Generic Types

But what about open (generic) types? Very easy:

```
public interface IGeneric<T>
{
    T Member { get; set; }
}

public class Generic<T> : IGeneric<T>
{
    public T Member { get; set; }
}

unity.RegisterType(typeof(IGeneric<>), typeof(Generic<>));
```

Unity lets you resolve an open type but you have to explicitly specify its generic parameter.

## Creation Factories

A slightly more advanced registration option consists of using a delegate to build the class instance. We register a construction delegate as this:

```
unity.RegisterType<ILogger>("Factory", new InjectionFactory(
    u => new ConsoleLogger() { File = "out.log" } //or use your factory method
));
```

[InjectionFactory](#) is one kind of injection member, which will be covered in more detail in the Chapter 3 Dependency Injection chapter.

## Base Types

Just for fun, we can also register base types—strings, integers, Booleans, etc.—as instances but not as types:

```
unity.RegisterInstance<string>("One", "1");  
unity.RegisterInstance<string>("Two", "2");  
unity.RegisterInstance<int>("One", 1);  
unity.RegisterInstance<int>("Two", 2);
```

I will leave it to you to decide whether or not this is useful.

## Registration by XML Configuration

### XML Configuration Section

We just saw how to register mappings by code but it is certainly possible—and sometimes highly desirable—to do so by XML configuration, either through the **App.config** or the **Web.config** depending on your application type.



**Note:** If you use Visual Studio 2012, you can download the Enterprise Library Configuration Console from [the Microsoft Enterprise Library Download Center](#). This helps you configure all Enterprise Library modules in the application configuration file through a user interface.

So, let's declare a Unity configuration section in our configuration file and add a Unity entry:

```
<?xml version="1.0" encoding="utf-8"?>  
  
<configuration>  
  <configSections>  
    <section name="unity" type="Microsoft.Practices.Unity.Configuration.  
UnityConfigurationSection, Microsoft.Practices.Unity.Configuration"/>  
  </configSections>  
  
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
```

```

<container>

    <register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.
ConsoleLogger, Succinctly" name="Console"/>

    <register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.
FileLogger, Succinctly" name="File"/>

</container>

</unity>

</configuration>

```

In a nutshell, inside the [unity](#) section you add a [container](#) element and, inside of it, you add a number of [register](#) declarations. A [register](#) section expects the following attributes:

- **type** (required): The key to the registration, either a base type or an interface.
- **mapTo** (required): The component (concrete class) type.
- **name** (optional): An optional name to give to the registration; the default is **null**.



**Tip: Do use full assembly qualified type names in your XML configuration so that Unity can find the types.**

These are the exact same declarations as in the code example, with the obvious exception of instance registrations. There is no easy way to register an instance in XML since an instance is something that exists in memory.

## Generic Types

If you want to register a generic type in XML:

```

<container>

    <register type="Succinctly.IGeneric`1, Succinctly"

        mapTo="Succinctly.Generic`1, Succinctly" name="Console"/>

</container>

```

This uses the same syntax that .NET expects so make sure you use it properly.

## Type Aliases and Assembly Discovery

You can create type aliases, which can be valuable for avoiding long type names. Here's how it goes:

```
<unity>

<alias type="Succinctly.IGeneric`1, Succinctly" alias="GenericInterface"/>

<alias type="Succinctly.Generic`1, Succinctly" alias="GenericImplementation"/>

</unity>
```

If you want to avoid specifying assembly qualified type names, you can add assembly names to a list that Unity uses for type discovery:

```
<unity>

  <assembly name="Succinctly"/>

</unity>
```

And the same goes for namespaces:

```
<unity>

  <namespace name="Succinctly"/>

</unity>
```

Using an alias simplifies the configuration:

```
<unity>

  <register type="GenericInterface" mapTo="GenericImplementation"
name="Console"/>

</unity>
```

## IntelliSense

In order to make your life easier, you can add IntelliSense to the XML configuration. In Visual Studio, open up your configuration file and click on the ellipsis (...) next to **Schemas** in the Properties window:

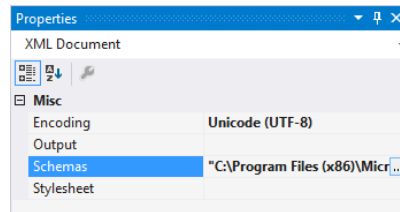


Figure 5: XML schema configuration

You will be prompted with the schema configuration dialog. Click on **Add**, navigate to the **packages\Unity.3.5.1404.0** folder below the solution folder, and select **UnityConfiguration30.xsd**:

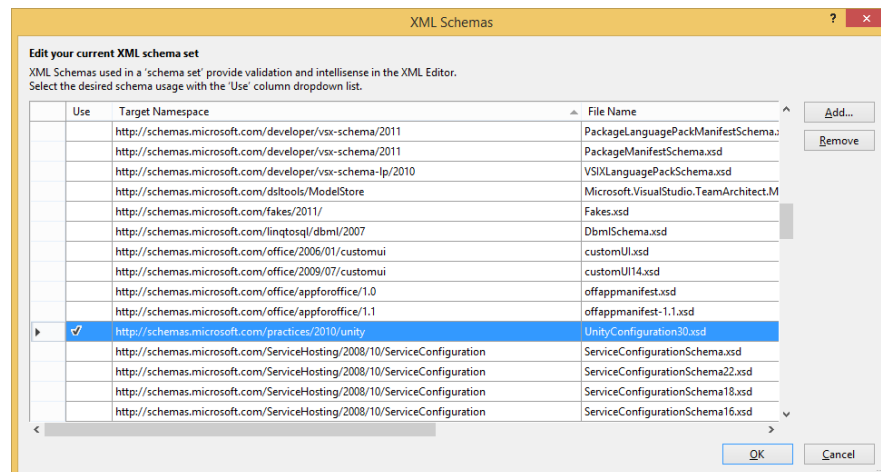


Figure 6: XML schemas

This file contains the [XML Schema Definition \(XSD\)](#) for the Unity configuration elements. After following this step, you now have IntelliSense inside of the Unity section of the configuration file:

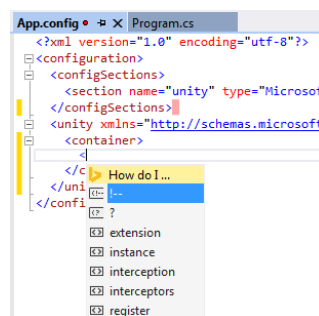


Figure 7: XML IntelliSense

## Loading XML Configuration

Before you can use XML configuration, you need to call the [LoadConfiguration](#) extension method from [UnityContainerExtensions](#):

```
unity.LoadConfiguration();
```



**Tip: Don't forget to call `LoadConfiguration()` otherwise your XML registration will not be found.**

## Registration by Convention

While the previous techniques give us a lot of power, they do force us to register all entries manually, which can be a lot of work. Fortunately, Unity has the ability to register types automatically by applying conventions. This ability comes in the form of extension methods in the class [UnityContainerRegistrationByConventionExtensions](#). Here is one example of a call to the [RegisterTypes](#) method:

```
unity.RegisterTypes
(
    AllClasses.FromLoadedAssemblies()
        .Where(x => (x.IsPublic) && (x.GetInterfaces().Any()) && (!x.IsAbstract)
            && (x.IsClass)),
    WithMappings.FromAllInterfacesInSameAssembly,
    type => (unity.Registrations
        .Select(x => x.RegisteredType)
        .Any(r => type.GetInterfaces()
            .Contains(r))) ? WithName.TypeName(type) : WithName.Default(type)
);
```

This needs a bit of explaining:

- The first argument to [RegisterTypes](#) is the list of types to register. In this case, we are passing a list of public, non-interface, non-abstract classes from the currently loaded assemblies ([AllClasses.FromLoadedAssemblies](#)) that implement at least one interface. Classes can also be obtained from a list of assemblies ([AllClasses.FromAssemblies](#)) or from the file system ([AllClasses.FromAssembliesInBasePath](#)).



- The second argument is a delegate that returns, for each registered type, its mapping key. In this case, all interfaces that the class exposes, declared in the same assembly as the class ([WithMappings.FromAllInterfacesInSameAssembly](#)). Another possibility is all of the type's interfaces ([WithMappings.FromAllInterfaces](#)) or the type's single interface that matches the type's name ([WithMappings.FromMatchingInterface](#)).
- The third parameter is how the registration shall be named. Here we are doing something clever: if there is already a registration for the same key, then use the full registered class name as the name ([WithName.TypeName](#)); otherwise, use **null** ([WithName.Default](#)). This example first creates a default, unnamed registration for the first interface and class found, and then all other classes are registered under their full name.
- An optional fourth parameter, if present, must contain a delegate that determines the lifetime manager for each of the created instances. Common values are [WithLifetime.ContainerControlled](#) for singletons, [WithLifetime.Transient](#) for always creating new instances, and [WithLifetime.PerThread](#) for creating an instance per thread. More on this as well as a full list of options in the section Lifetime Managers.

## Component Resolution

### Resolving an Instance

Once we have our registrations, we can ask for an instance (resolve) of a component. In Unity, this is achieved through the [Resolve](#) method or one of the similar extension methods in [UnityContainerExtensions](#). Some examples of the strongly typed (extension) versions are:

```
var logger1 = unity.Resolve<ILogger>();           //ConsoleLogger
var logger2 = unity.Resolve<ILogger>("File");     //FileLogger
var logger3 = unity.Resolve<ILogger>("Console"); //ConsoleLogger
```

And some examples of the dynamic (not strongly typed) ones:

```
var logger1 = unity.Resolve(typeof(ILogger)) as ILogger;
//a ConsoleLogger
var logger2 = unity.Resolve(typeof(ILogger), "File") as ILogger;
var logger3 = unity.Resolve(typeof(ILogger), "Console") as ILogger;
```



**Tip:** If the registration you are trying to obtain does not exist, Unity will throw an exception.

## Resolving Multiple Instances

If we have multiple named registrations for the same type, we can retrieve them all at once:

```
var loggers1 = unity.ResolveAll<ILogger>();           //ConsoleLogger and FileLogger
var loggers2 = unity.ResolveAll(typeof(ILogger)) as IEnumerable<ILogger>;
```



**Tip:** Only named registrations are returned by `ResolveAll`.



**Note:** Unity resolves types regardless of the registration method.

The same unnamed `ILogger` instance, registered through [RegisterInstance](#), will always be retrieved whenever [Resolve](#) is called without a name. This is because we registered an instance, not a type. For entries registered with [RegisterType](#), new instances of the concrete classes are instantiated and returned every time. We discuss this behavior further in the following section Lifetime Managers.

## Resolving with Overrides

It may happen that you want to resolve a component with different parameters (constructor or method arguments, property values) than the ones with which it was registered. I'm not saying that you should do this as this requires knowledge of how the component was registered which, in a way, defies the purpose of IoC. But, nevertheless, Unity allows you to do this.

The [Resolve](#) method takes an optional array of [InjectionMember](#) instances. Two of them ([ParameterOverride](#) and [PropertyOverride](#)) can be used to specify alternative values for constructor or method parameters or properties that were configured at registration time. For example, if you configured an [InjectionProperty](#) (see Property Injection for an explanation on this) to supply some value for some property, now you can change it to have a different value. This requires knowledge of how the component was registered because you can only supply alternative values for those members that were configured.

Here's an example, with an overridden property value:

```
public class FileLogger : ILogger
{
    public string Filename { get; set; }
}

//registration with injected property
```

```

unity.RegisterType<ILogger, FileLogger>(new InjectionProperty("Filename",
"log.txt"));

//resolution with overridden property

unity.Resolve<ILogger>(new PropertyOverride("Filename", "output.log"));

```

And another one for constructor parameter overriding:

```

public class FileLogger : ILogger
{
    public FileLogger(string filename)
    {
        this.Filename = filename;
    }

    public string Filename { get; set; }
}

//registration with injected constructor
unity.RegisterType<ILogger, FileLogger>(new InjectionConstructor("log.txt"));

//resolution with overridden constructor parameter

unity.Resolve<ILogger>(new ParameterOverride("output.log"));

```

By default, overridden members get applied throughout all of the resolution hierarchy. For example, the class you asked for could have injected properties of other classes and some of these classes could have compatible injected members as well. But you can also cause the overriding to only occur in a specific type instead of all types that have compatible members, by means of [OnType](#) and [OnType<T>](#):

```

//resolution with overridden property but only on instances of FileLogger

unity.Resolve<ILogger>(new PropertyOverride("Filename", "output.log")
.OnType<FileLogger>()); //OnType(typeof(FileLogger))

```

Three things to keep in mind:

- If using a lifetime manager that keeps existing instances, constructor overriding will not work since the object was already instantiated.
- Overriding will only work for members configured at registration time.
- Try to define the target for your member overriding by specifying [OnType](#) or [OnType<T>](#).



**Note:** Try to avoid using resolution overrides as they are not a good idea.

## Deferred Resolution

Now, it may happen that we wish to have a reference from a component at hand but may not need to use it immediately (if at all). If that is the case, we can obtain lazy loading references in the form of a [Lazy<T>](#) or [Func<T>](#). This is called deferred resolution:

```
var lazyLogger = unity.Resolve<Lazy<ILogger>>();  
var loggerDelegate = unity.Resolve<Func<ILogger>>();
```

The actual component is only instantiated when we access its value:

```
var logger1 = lazyLogger.Value; //lazyLogger.IsValueCreated = true  
var logger2 = loggerDelegate();
```



**Tip:** You don't have to register a type with *Lazy<T>* or *Func<T>*, **Unity does it for you.**

## Testing If a Registration Exists

If you want to test whether or not a given registration exists, you have two options. You can either look at the [Registrations](#) property or use one of the [IsRegistered](#) extension methods in [UnityContainerExtensions](#). Both of these methods look internally at the [Registrations](#) collection. We can also do so explicitly, as in this example where we retrieve all registered **ILogger** concrete types and their registration names:

```
var loggerTypes = unity  
    .Registrations  
    .Where(x => x.RegisteredType == typeof(ILogger))  
    .ToDictionary(x => x.Name ?? String.Empty, x => x.MappedToType);
```

Because the resolve methods throw exceptions if the registration is not found, it is sometimes useful to use a helper method such as this:

```
public static T TryResolve<T>(this IUnityContainer unity, String name = null)
{
    return ((T) (TryResolve(unity, typeof(T), name) ?? default(T)));
}

public static Object TryResolve(this IUnityContainer unity, Type type,
String name = null)
{
    try
    {
        return (unity.Resolve(type, name));
    }
    catch
    {
        return (null);
    }
}
```

## Lifetime Managers

We have seen that [RegisterInstance](#) and [RegisterType](#) behave differently. By default, [RegisterInstance](#) always causes the same registered instance to be returned in all subsequent calls to [Resolve](#) while [RegisterType](#) does exactly the opposite as a new instance is always returned. This behavior comes from the default lifetime manager used by each method. If you look at the documentation for each of the **Register\*** methods, you will see that they take an optional [LifetimeManager](#) instance, the base class for all lifetime managers.

A lifetime manager handles how an instance of a registered type is created and all registrations have an associated lifetime manager or **null**. A **null** lifetime manager will always cause a new instance to be created every time the **Resolve** method is called. Unity includes a number of lifetime managers, described in the following table:

Table 1: Lifetime managers

Lifetime Manager	Alias (XML/conventional configuration)	Purpose
<a href="#">ContainerControlledLifetimeManager</a>	<b>singleton/</b> <a href="#">WithLifetime.ContainerControlled</a>	Unity creates a single instance of the type (a singleton) and always returns it. When the container is disposed, if the registered type implements <a href="#">IDisposable</a> , Unity also disposes of it.
<a href="#">ExternallyControlledLifetimeManager</a>	<b>external/</b> <a href="#">WithLifetime.ExternallyControlled</a>	Registers an instance created elsewhere (not by Unity). Does not dispose of the registered instances. <a href="#">WithLifetime.ExternallyControlled</a> , in conventional configuration.
<a href="#">HierarchicalLifetimeManager</a>	<b>hierarchical/</b> <a href="#">WithLifetime.Hierarchical</a>	Identical to <a href="#">ContainerControlledLifetimeManager</a> but, in child containers, each gets its own instance. Disposes of created instances when the container is disposed.
<a href="#">PerRequestLifetimeManager</a>	N/A	<p>In web applications, Unity looks for an instance of the type in the <a href="#">HttpContext.Items</a> collection and, if not found, creates a new instance and stores it there. Only disposes of created instances if module <a href="#">UnityPerRequestHttpModule</a> is enabled (discussed in Introduction)</p> <p>A number of technologies in the Microsoft stack use IoC. In fact, its number seems to be growing every time a new version or technology is launched. It is generally possible to plug in an IoC container of choice; in our case, let's see how to use Unity.</p>

		ASP.NET Web Forms). Requires NuGet package <a href="#">Unity.Mvc</a> .
<a href="#">PerResolveLifetimeManager</a>	<b>perresolve/</b> <a href="#">WithLifetime.PerResolve</a>	During a resolve operation, if an instance of a type is needed multiple times, only creates one instance and reuses it throughout the operation. In different resolves, always creates new instances. Disposes of created instances when the container is disposed.
<a href="#">PerThreadLifetimeManager</a>	<b>perthread/</b> <a href="#">WithLifetime.PerThread</a>	Creates a new instance per thread and always returns that same instance to the same thread. Disposes of created instances when the container is disposed.
<a href="#">TransientLifetimeManager</a>	<b>transient/</b> <a href="#">WithLifetime.Transient</a>	Always creates and returns a new instance of the registered type. Disposes of created instances when the container is disposed. Identical to passing <b>null</b> . The default for <a href="#">RegisterType</a> .

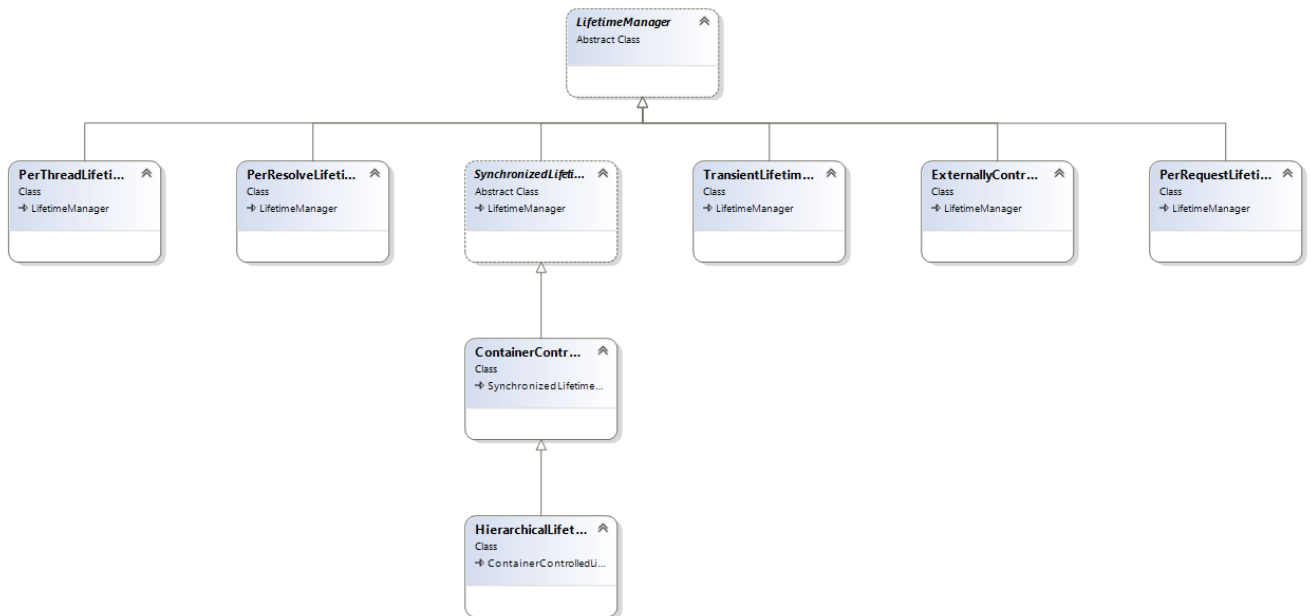


Figure 8: Lifetime manager class hierarchy



**Note: Singleton and transient are type aliases. See Type Aliases.**

So, we have three kinds of lifetime managers:

- First, one that never creates instances: [ExternallyControlledLifetimeManager](#) and [HierarchicalLifetimeManager](#).
- Second, one that may or may not create an instance (depending on whether or not one already exists) for its context: [ContainerControlledLifetimeManager](#), [PerRequestLifetimeManager](#), [PerResolveLifetimeManager](#), and [PerThreadLifetimeManager](#).
- And third, one that always creates new instances: [TransientLifetimeManager](#).



**Tip: The only meaningful lifetime managers that can be passed to `RegisterInstance` are `ContainerControlledLifetimeManager` (the default) or `ExternallyControlledLifetimeManager`. The difference between the two is that the latter does not dispose of the registered instance when the container is disposed.**



**Tip: To use `PerRequestLifetimeManager`, you will need to add a reference to the [Unity.Mvc](#) NuGet package.**

Remember the **Singleton** design pattern? Well, in the age of IoC containers, you might as well consider it an anti-pattern. You no longer have to implement your classes in a funny way whenever you want to have a singleton, nor do you have to watch out for all possible alternative ways developers can use to fool your code. Just configure a component to have a lifetime of singleton and ask Unity for it. That's it. If you want to change this behavior later on, it is just a matter of configuring it otherwise.

An example of registering a different instance of a type per calling thread, in configuration by code, would be:

```
unity.RegisterType<ILogger, ConsoleLogger>(new PerThreadLifetimeManager());
```

And in XML configuration, notice how we can use either the lifetime manager's alias or its full class name:

```
<?xml version="1.0" encoding="utf-8"?>

<configuration>

  <configSections>

    <section name="unity" type="Microsoft.Practices.Unity.Configuration.
UnityConfigurationSection, Microsoft.Practices.Unity.Configuration"/>
```



```

</configSections>

<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <container>
    <register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.
ConsoleLogger, Succinctly" name="Console">
      <lifetime
type="Microsoft.Practices.Unity.ContainerControlledLifetimeManager,
Microsoft.Practices.Unity"/>
    </register>
    <register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.
FileLogger, Succinctly" name="File">
      <lifetime type="transient"/>
    </register>
  </container>
</unity>
</configuration>

```



**Tip:** If you are not using an alias, make sure you use the assembly qualified name of the lifetime manager class.

Finally, in conventional configuration, we can also specify the lifetime manager for each registration. For that, we need to pass a delegate as the fourth parameter to [RegisterTypes](#). This delegate receives a type as its only argument and returns a [LifetimeManager](#) instance. The class [WithLifetime](#) has some predefined methods for the most usual lifetime managers and there's also [Custom<T>](#), which can be used to return a custom one ([PooledLifetimeManager](#) is defined in chapter Chapter 5 Extending Unity):

```

unity.RegisterTypes
(
  AllClasses.FromLoadedAssemblies()
    .Where(x => (x.IsPublic == true) && (x.GetInterfaces().Any() == true)

```

```

        && (x.IsAbstract == false) && (x.IsClass == true)),
    WithMappings.FromAllInterfacesInSameAssembly,
    type => (unity.Registrations.Select(x => x.RegisteredType)
        .Any(r => type.GetInterfaces().Contains(r) == true) == true) ?
    WithName.TypeName(type) : WithName.Default(type),
    WithLifetime.Custom<PooledLifetimeManager>()
);

```



**Tip:** *Lifetime manager instances cannot be shared by different registrations and the Register\* methods will throw an exception if you try.*

## Child Containers

Creating a child container can help in managing instances in a broader scope than that of the Unity container. A child container has these three characteristics:

- All registrations from the parent are accessible in a child container.
- Registrations originating in the child container are not accessible in its parent.
- Disposable instances created by a child container are disposed of when the child container is itself disposed.

We create a child container by calling [CreateChildContainer](#):

```

using (var child = unity.CreateChildContainer())
{
    child.RegisterType<ILogger, ConsoleLogger>("DisposableChild",
        new ContainerControlledLifetimeManager());
    var logger = child.Resolve<ILogger>("DisposableChild");
}    //logger.Dispose is called

```



**Tip:** *Don't forget that Dispose is only called for some lifetime managers.*

Since [IUnityContainer](#) implements [IDisposable](#), it is a good principle to always create child containers in a **using** block so that they are properly disposed of when no longer needed.

A child container exposes the same interface as the parent—[IUnityContainer](#)—which has a [Parent](#) property but does not keep track of its children.

A typical scenario in which you would use a child container would be in a web application request. We'll look at that when we talk later about Unity's integration with Introduction

A number of technologies in the Microsoft stack use IoC. In fact, its number seems to be growing every time a new version or technology is launched. It is generally possible to plug in an IoC container of choice; in our case, let's see how to use Unity.

ASP.NET Web Forms.

## Common Service Locator

A number of IoC containers exist in the .NET world including [Unity](#), [Autofac](#), [Ninject](#), [Spring.NET](#), [StructureMap](#), and [Castle Windsor](#). Although each one has specific functionality, when it comes to resolving a registered component, they all have a similar interface consisting of two basic operations:

- “Get me an instance of type XYZ identified by key ABC”.
- “Get me all instances of type XYZ”.

Having that in mind, the development community, together with Microsoft, defined and implemented a common interface to which all IoC containers can comply called the Common Service Locator.

The Common Service Locator is an implementation of the **Service Locator** pattern maintained by Microsoft and hosted [here at CodePlex](#). If you added Unity as a NuGet package, you might have noticed that the Common Service Locator package came along as well.

It serves as a central repository that abstracts whatever IoC container you want to use. With the Common Service Locator, you don't have to expose Unity as a shared field.

The Common Service Locator developers have implemented adapters for some of the most popular IoC containers, all of which are available at the Common Service Locator site. Other vendors have implemented adapters for their own products. In the case of Unity, all it takes to set it up is:

```
ServiceLocator.SetLocatorProvider(() => new UnityServiceLocator(unity));
```

After that, instead of directly referencing Unity, you can instead reference the Common Service Locator.

So, where you had:

```
var logger = unity.Resolve<ILogger>();
```

You can now have:

```
var logger = ServiceLocator.Current.GetInstance<ILogger>();
```

And that's it. This way, you can even replace Unity with another IoC container and your code won't even notice it.

For the record, the interface exposed by the Common Service Locator is [IServiceLocator](#):

```
public interface IServiceLocator : IServiceProvider
{
    IEnumerable<TService> GetAllInstances<TService>();
    IEnumerable<object> GetAllInstances(Type serviceType);
    TService GetInstance<TService>();
    TService GetInstance<TService>(string key);
    object GetInstance(Type serviceType);
    object GetInstance(Type serviceType, string key);
}
```

As you can see, only component resolution methods are supported, not registration. There are strongly typed generic methods as well as dynamic methods that return plain objects. Calls to these methods will be directed to the proper implementation; in our case, the Unity container instance. So anything that applies to Unity also applies to the Common Service Locator, such as an exception being thrown in case a registration is not found.



**Note:** *If you ever want to go from the Common Service Locator back to Unity, you just have to retrieve the unnamed `IUnityContainer` instance.*

One final note: The [IServiceLocator](#) inherits from [IServiceProvider](#), an interface used since the early days of .NET to offer a kind of poor man's IoC. It is still used by the Visual Studio designer and by some APIs such as [Runtime Caching](#) and [WCF Data Services](#).



**Note:** *Some people think that the Service Locator pattern is actually a bad thing. I understand the arguments but don't quite agree with them. For in-depth coverage on this, read the blog post "[Service Locator is an Anti-Pattern](#)" by Mark Seemann.*

# Chapter 3 Dependency Injection

## Introduction

A topic close to IoC is Dependency Injection (DI). We call DI to the ability of an IoC container to automatically set properties and method parameters in newly created component instances, as necessary and recursively. Unity fully supports DI in its core, integrated with IoC. When you ask for an instance, Unity injects all its dependencies automatically but you can also ask it to inject dependencies explicitly. Let's see how we do that.

## Injection Types

Unity supports the following five kinds of dependency injection:

- **Automatic injection:** When Unity is asked to create a component that takes parameters in its constructor, it automatically obtains values for them when invoking the constructor.
- **Constructor injection:** Unity will pass a registered component as a parameter to a constructor, therefore allowing the usage of constructors with parameters.
- **Property injection:** Unity sets values for properties from registered components.
- **Method injection:** Unity calls initialization methods automatically.
- **Injection factory:** Unity calls a custom delegate that builds things for us.

All of these injections, except factory injection, are performed automatically when Unity resolves and creates instances of components.

## Auto Injection

If Unity is asked to instantiate a class that has a constructor with parameters, and Unity has unnamed registered components of the same types, then Unity automatically retrieves these components and passes them to the constructor of the class to build. For instance:

```
public class SomeClass
{
    public SomeClass() { }

    public SomeClass(ILogger logger) { }
}
```

Even if Unity finds a parameterless constructor and one that takes parameters, it will always choose the one with parameters—as long as it is public and Unity can find registered components that match its types. In this case, it will pass an **ILogger** instance, which is obtained exactly as if we would be calling [Resolve](#), meaning, this instance can be a singleton or a newly created one depending on the lifetime manager associated with it.



**Note:** In case you are wondering, *Lazy<T>* and *Func<T>* also work here.

## Configuring Injection by Attributes

### Constructor Injection

We can explicitly control some aspects of the injection process. First, we can select the injection constructor by applying an [InjectionConstructorAttribute](#) attribute to an existing public constructor:

```
public class SomeClass
{
    public SomeClass() { }

    [InjectionConstructor]
    public SomeClass(ILogger logger, IUnityContainer unity) { }

    public SomeClass(ILogger logger) { }
}
```



**Tip:** Do not add *InjectionConstructorAttribute* attributes to more than one constructor because it will confuse Unity and it won't know what constructor to use.

If we have a component registered as multiple types with different names, we can select which to inject by applying a [DependencyAttribute](#) attribute with a name to the constructor argument:

```
public class SomeClass
{
```

```

public SomeClass() { }

public SomeClass([Dependency("Console")] ILogger logger) { }
}

```

## Property Injection

If we want a property to be injected as well, it needs to have a setter and it needs to have applied a [DependencyAttribute](#) attribute, with or without a name:

```

public class SomeClass
{
    [Dependency("File")]
    public ILogger File { get; set; }

    [Dependency("Console")]
    public ILogger Console { get; set; }
}

```

You can override a registration by applying a [PropertyOverride](#) declaration. See Resolving with Overrides.

## Method Injection

Finally, method injection: We tell Unity to execute a method after the instance is built. For that purpose, we apply the [InjectionMethodAttribute](#) attribute to a public method as in the following example:

```

public class SomeClass
{
    [InjectionMethod]
    public void Initialize() { }
}

```

```
}
```

Of course, you can also apply [InjectionMethodAttribute](#) to a method with parameters if you want them to be injected by Unity:

```
public class SomeClass
{
    [InjectionMethod]
    public void Initialize(ILogger logger) { }
}
```

And if you want to use a registration with a particular name as a method parameter:

```
public class SomeClass
{
    [InjectionMethod]
    public void Initialize([Dependency("Console")] ILogger logger) { }
}
```



**Tip:** The *InjectionConstructorAttribute*, *DependencyAttribute* and *InjectionMethodAttribute* **attributes can only be applied to public members.**



**Tip:** The *DependencyAttribute* and *InjectionMethodAttribute* **attributes can only be applied to the registered type (concrete class), not the key (interface or abstract base class); otherwise they have no effect.**

## Configuring Injection by Code

Other than by using attributes, it is possible to specify dependencies entirely by code. This is good so that you can apply injections to types that you don't control or when you don't want to add references to injection attributes that have no meaning to the classes being injected.

Code injection is configured at registration type; the [RegisterType](#) method takes additional parameters in the form of an optional array of [InjectionMember](#) instances. The included subclasses of [InjectionMember](#) (itself an abstract class) are:



- [InjectionConstructor](#): Specifies which constructor and values to use for its parameters, which may include Unity resolutions.
- [InjectionProperty](#): The properties to set from Unity registrations.
- [InjectionMethod](#): What initialization methods to invoke plus their parameters.
- [InjectionFactory](#): How to build the component.

## Constructor Injection

Here are some examples. First, here's how to supply base type parameters for a constructor:

```
public class SomeClass
{
    public SomeClass(string key, int num) { }
}

unity.RegisterType<SomeClass, SomeClass>("Arguments",
    new InjectionConstructor("Some Name", 10));
```

Next, here's a constructor with resolved parameters (types registered in Unity):

```
public class SomeClass
{
    public SomeClass(ILogger logger) { }
}

unity.RegisterType<SomeClass, SomeClass>(
    new InjectionConstructor(new ResolvedParameter<ILogger>("Console"));
```

And you can even mix both plain values and resolved types, like in this example where we are injecting a string and a resolved **ILogger** named "Console":

```
public class SomeClass
{
    public SomeClass(string key, ILogger logger) { }
}
```

```
unity.RegisterType<SomeClass, SomeClass>(
    new InjectionConstructor("Some Name",
    new ResolvedParameter<ILogger>("Console"));
```

So, you pass parameters to [InjectionConstructor](#), which must either be static values or instances of an [InjectionParameterValue](#)-derived class (most often [ResolvedParameter](#) or [ResolvedParameter<T>](#)). The number of values and their order and types dictate which constructor shall be called. [ResolvedParameter](#) takes two parameters, one for the type and the other for the optional name, which can be `null` for default registrations. [ResolvedParameter<T>](#) only requires the name because the type is inferred from the generic parameter.

## Property Injection

Of course, you can also configure which properties shall have their values injected. In the next example, we inject an `ILogger` named "Console" to the `Logger` property:

```
public class SomeClass
{
    public ILogger Logger { get; set; }
}

unity.RegisterType<SomeClass, SomeClass>(new InjectionProperty("Logger",
    new ResolvedParameter<ILogger>("Console"));
```

And we can also inject plain values as well:

```
public class FileLogger : ILogger
{
    public string Name { get; set; }
}

unity.RegisterType<ILogger, FileLogger>(new InjectionProperty("Name",
    "output.log"));
```

## Method Injection

Code configuration also allows us to configure method injection and its eventual parameters, as in this example:

```
public class SomeClass
{
    public void Initialize(ILogger logger) { }
}

unity.RegisterType<SomeClass, SomeClass>(
    new InjectionMethod("Initialize", new ResolvedParameter<ILogger>("Console")));
```

Or with no parameters at all:

```
public class SomeClass
{
    public void Initialize() { }
}

unity.RegisterType<SomeClass, SomeClass>(new InjectionMethod("Initialize"));
```



**Tip:** All constraints from mapping by attributes apply: All members must be public and injected properties must have getters and setters.

## Factory Injection

Finally, we can also specify exactly how our component will be built. This will be the most complete solution to creating instances since you can write your own code for that purpose. We use a special injection member called [InjectionFactory](#):

```
public class SomeClass
{
    public SomeClass(string key) { }

    public ILogger Logger { get; set; }

    public void Initialize() { }
}

unity.RegisterType<SomeClass, SomeClass>(
```

```
new InjectionFactory((u) => { var x = new SomeClass("Some Name"){ Logger =  
new FileLogger("output.log") }; x.Initialize(); return (x); }));
```

[InjectionFactory](#) takes a delegate that receives a single parameter of type [IUnityContainer](#) and returns any kind of object. Of course, you should return an instance of the registered component type or one that inherits from it.



***Tip: When using `InjectionFactory`, all of the other injection techniques—constructor, property, and method—do not apply; it's all up to you.***

## Putting It All Together

One final example on how we can mix constructor, property, and method injection:

```
public class SomeClass  
{  
    public SomeClass(string key) { }  
  
    public ILogger Logger { get; set; }  
  
    public void Initialize() { }  
}  
  
unity.RegisterType<SomeClass, SomeClass>(  
    new InjectionConstructor("Some Name"),  
    new InjectionMethod("Initialize"),  
    new InjectionProperty("Logger", new ResolvedParameter<ILogger>("Console")));
```

## Configuring Injection by XML

### Constructor Injection

Constructor, property, and method injections can be configured in XML. An example for setting a simple string value in a constructor is achieved by means of the [constructor](#) element:

```

<register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.FileLogger,
Succinctly" name="FileLogger">
    <lifetime type="singleton" />
    <constructor>
        <param name="filename" value="output.log"/>
    </constructor>
</register>

```

One [param](#) element must exist for each of the constructor's parameters. The [param](#) takes the following attributes:

- **name** (required): The name of the constructor parameter.
- **value** (optional): A base type that can be converted from a string representation. If not present, then **dependencyType** must be supplied.
- **dependencyType** (optional): A type name to resolve from Unity. It must be specified unless **value** is.
- **dependencyName** (optional): A registration name to be taken together with **dependencyType** for resolving a component from Unity.

## Property Injection

If we want to set a property from a Unity registration of type **ILogger** and name "File", we use a [property](#):

```

<register type="Succinctly.SomeClass, Succinctly" mapTo="Succinctly.SomeClass,
Succinctly">
    <lifetime type="transient" />
    <property name="Logger" dependencyName="File"
        dependencyType="Succinctly.ILogger, Succinctly" />
</register>

```

The [property](#) element takes the same parameters as [param](#) of [constructor](#) so there's no point in discussing it here.

## Method Injection

Finally, calling a method as part of the initialization process—the [method](#) element:

```
<register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.FileLogger,
Succinctly" name="FileLogger">
    <lifetime type="singleton" />
    <method name="Initialize" >
        <param name="filename" value="output.log"/>
    </method>
</register>
```

Again, its parameters are exactly the same as those of the [constructor](#) element.



**Note:** *There is not an out-of-the-box way to use `InjectionFactory` in XML but the chapter entitled “Chapter 5 Extending Unity” presents a simple workaround.*



**Tip:** *Don’t forget to call `LoadConfiguration()` otherwise the XML configuration will be ignored by Unity.*

## Explicit Injection

If we have an instance of a component that wasn’t obtained through Unity, or if its dependency configuration has somehow changed, we can ask Unity to inject all dependencies explicitly. For that purpose, we have the [BuildUp](#) method from [IUnityContainer](#) or the [BuildUp<T>](#) generic extension method from [UnityContainerExtensions](#).

[BuildUp](#) takes an instance and a type. Here are two examples, one with automatic type inference and the other with explicit type specification:

```
unity.BuildUp(instance); //same as unity.BuildUp(instance.GetType(), instance)
unity.BuildUp(typeof(IMyService), svc);
```

The reason why you can specify a type is because different base types can have different dependency configurations defined upon them.

You can call [BuildUp](#) multiple times and the result should always be the same as long as the injection rules don't change; all injection methods are called and all injection properties are set.

# Chapter 4 Aspect-Oriented Programming (AOP)

## Introduction

Aspect-Oriented Programming (AOP) is about the ability to inject cross-cutting behavior (aspects) in existing code so as to do useful yet somewhat lateral operations transparently. This is called, in Unity terms, **interception**. Some examples include:

- Logging all method invocations, including parameters.
- Controlling access to a method based on the identity of the current user.
- Catching exceptions and doing something with them.
- Caching the results of a “heavy” operation so that in subsequent executions results come from the cache.

Other examples exist, of course. Aspects are applied to target code in a totally transparent way; that is, you just call your methods in the usual way and interception occurs magically. For now, let’s understand how Unity helps us applying aspects.

## Interception Kinds and Techniques

Interception in .NET can be implemented in one of the following five ways:

- **Virtual method interception:** Dynamically generates a subclass that overrides the virtual methods of the base class so as to add new behavior or suppress the existing one.
- **Interface interception:** Generates a proxy that will sit behind an interface and its concrete implementation to which all of the interface’s operations will be diverged.
- **Transparent proxy interception:** Generates a transparent or Remoting proxy to an interface or to an instance of a [MarshalByRefObject](#); this is what [Windows Communication Foundation \(WCF\)](#) proxies use.
- **Context bound interception:** Leverages the [ContextBoundObject](#) class and its associated framework. Since this is very specific and not very usual (because it requires using a base class of type [ContextBoundObject](#)), it is not supported by Unity and will not be covered.
- **Code weaving:** This is probably the most powerful mechanism but it is also a complex one since it involves changing the code on an assembly—in memory as well as in the compiled output file. Unity also does not support it.

Some of these techniques can be applied to existing instances (**instance interception**) or to Unity-created ones (**type interception**). In any case, they only apply to instance, not static methods. The following table lists just that:

*Table 2: Interception kinds and targets*



Interception	Kind	Target	What Can Be Intercepted
Virtual method	Type	Non-sealed classes with virtual methods.	All virtual methods, properties, and event add and remove handlers.
Interface	Instance	Existing classes implementing an interface.	All interface methods, properties, and event add and remove handlers.
Transparent proxy	Instance	Existing classes implementing an interface or inheriting from <a href="#">MarshalByRefObject</a> .	All interface methods, properties, and event add and remove handlers (in the case of interfaces) or all virtual methods, properties, and event add and remove handlers (for <a href="#">MarshalByRefObject</a> -derived classes).

Interception, if enabled, is applied automatically. But before we can use it, we need to configure what interceptor to use per registration. Unity includes the following interceptors, which cover the most typical scenarios:

- [VirtualMethodInterceptor](#) ([IInstanceInterceptor](#))
- [InterfaceInterceptor](#) ([IInstanceInterceptor](#))
- [TransparentProxyInterceptor](#) ([ITypeInterceptor](#))

The base interfaces that specify the interception kinds, [IInstanceInterceptor](#) and [ITypeInterceptor](#), expose a small number of methods. Worthy of note are two methods that both interfaces inherit from [IInterceptor](#):

- [CanIntercept](#): Checks if the interceptor can intercept a given type, passed as parameter.
- [GetInterceptableMethods](#): Returns the list of interceptable methods for a given key-component pair of types, which may include property getters and setters, and event add and remove handlers.

Depending on the registration key, we need to set the proper interceptor; this is the one that will build our interception proxy for the target instance or type. Before we look at how to configure it, let's understand how interception is implemented. But, even before that, let's get `EnterpriseLibrary.PolicyInjection`, a required package from NuGet. This is where all interception-related classes are defined and is a prerequisite for this chapter.

```
PM> Install-Package EnterpriseLibrary.PolicyInjection
```

Figure 9: Installing Policy Injection as a NuGet package

## Adding Interfaces

We can also add one or more interfaces to the generated proxy. In this case, we will need to implement—either in a behavior or in a call handler (see the next sections)—all of its methods and properties. We shall see an example later on.

## Call Handlers

Interception means that all calls to a method, property, event add, or remove handler are diverged to custom code. In it, we decide what to do before or after (or instead of) the call to the original code.

In Unity, the atomic unit by which we can perform interception is the [ICallHandler](#) interface. Its interface is very simple:

```
public interface ICallHandler
{
    IMethodReturn Invoke(IMethodInvocation input, GetNextHandlerDelegate getNext);
    int Order { get; set; }
}
```

The [Invoke](#) method is the one to where the action actually goes. It receives the execution context—the instance where the method invocation was made and any method parameters—as well as a delegate for the next handler.

The [Order](#) property tells Unity how to sort all call handlers that are applied to the same target.

Let's see an example that measures the time that it took a method to execute:

```
public class MeasureCallHandler : ICallHandler
{
    public IMethodReturn Invoke(IMethodInvocation input,
    GetNextHandlerDelegate getNext)
    {
```

```

var watch = Stopwatch.StartNew();

var result = getNext()(input, getNext);

var time = watch.ElapsedMilliseconds;

var logger = ServiceLocator.Current.GetInstance<ILogger>();

logger.Log(String.Format(
    "Method {0} took {1} milliseconds to complete", input.MethodBase, time));

return (result);
}

public int Order { get; set; }
}

```

The call to `getNext()(input, getNext)` is normally always present in a call handler, and it tells Unity to execute the target method or property that the call handler is intercepting. Its result is a value suitable for returning from [Invoke](#) and contains the actual returned value from the intercepted method if it is not **void**.

The [MethodBase](#) property contains a reference to the intercepted method. If you want to see if this method is a property getter or setter, you can use:

```

if (input.MethodBase.IsSpecialName && (
    input.MethodBase.Name.StartsWith("get_") || input.MethodBase.Name.StartsWith("set_")
))) { }

```

Or an event handler add or remove method:

```

if (input.MethodBase.IsSpecialName && (
    input.MethodBase.Name.StartsWith("add_") || input.MethodBase.Name
        .StartsWith("remove_"))) { }

```

[Target](#) is the targeted object instance that contains the intercepted method.

[Arguments](#) contains a collection of all method parameters, if any.

[InvocationContext](#) is a general-purpose dictionary indexed by string and storing objects, where call handlers targeting the same method or property can share data.

After calling the base implementation, we get a result from which we can obtain the actual returned value, if the method was not **void** ([ReturnValue](#)) and did not return an exception or the exception that was thrown ([Exception](#)).

If we want to return a value without actually executing the intercepted method, we can use the [CreateMethodReturn](#) method. In its simplest form, it just takes a value to return for non-**void** methods which, of course, must match the method's expected return type:

```
public class NullCallHandler : ICallHandler
{
    public IMethodReturn Invoke(IMethodInvocation input,
    GetNextHandlerDelegate getNext)
    {
        return (input.CreateMethodReturn(null));
    }

    public int Order { get; set; }
}
```

And, if we want to return an exception instead, we call [CreateExceptionMethodReturn](#):

```
public class ExceptionCallHandler : ICallHandler
{
    public IMethodReturn Invoke(IMethodInvocation input,
    GetNextHandlerDelegate getNext)
    {
        return (input.CreateExceptionMethodReturn(new NotImplementedException()));
    }

    public int Order { get; set; }
}
```

If we have two call handlers configured for the same method, we can pass values from the first to the second:

```

public class MessagingCallHandler : ICallHandler
{
    public IMethodReturn Invoke(IMethodInvocation input,
    GetNextHandlerDelegate getNext)
    {
        if (input.InvocationContext.ContainsKey("message") == true)
        {
            //someone got here first...
        }
        else
        {
            input.InvocationContext["message"] = "I was here first";
        }
        return (getNext()(input, getNext));
    }
    public int Order { get; set; }
}

```

## Interception Behaviors

A call handler only applies to a single interception target (a method or property). Unity offers another construction for intercepting all calls to all interceptable members of a class, called an interception behavior.

An interception behavior is a class that implements [IInterceptionBehavior](#). Again, it is a simple interface although slightly more complex than [ICallHandler](#). Its contract is this:

```

public interface IInterceptionBehavior

```

```

{
    IEnumerable<Type> GetRequiredInterfaces();

    IMethodReturn Invoke(IMethodInvocation input,
        GetNextInterceptionBehaviorDelegate getNext);

    bool WillExecute { get; }
}

```

Method [GetRequiredInterfaces](#) is a way developers can use to force the behavior to only run if the target class implements one or more specific interfaces. In most cases, we just return an empty collection.

[Invoke](#) is exactly the same as in [ICallHandler](#) so there's no point in discussing it again.

Property [WillExecute](#) is used to tell Unity if the behavior should be applied or not. If it returns **false**, method [Invoke](#) will not run.



***Note: Since `WillExecute` receives no context, it is hard to use. Most of the time we just return `true`.***



***Note: The difference between call handlers and behaviors is that behaviors intercept all interceptable methods in a class whereas call handlers intercept just one.***

A simple interception behavior:

```

public class MyInterceptionBehavior : IInterceptionBehavior
{
    IEnumerable<Type> IInterceptionBehavior.GetRequiredInterfaces()
    {
        return (Type.EmptyTypes);
    }
}

```

```

IMethodReturn IInterceptionBehavior.Invoke(IMethodInvocation input,
GetNextInterceptionBehaviorDelegate getNext)
{
    //before base call
    Console.WriteLine("About to call method {0}", input.MethodBase);
    var result = getNext().Invoke(input, getNext);
    if (result.Exception != null)
    {
        Console.WriteLine("Method {0} returned {1}", input.MethodBase,
result.ReturnValue);
    }
    else
    {
        Console.WriteLine("Method {0} threw {1}", input.MethodBase,
result.Exception);
    }
    //after base call
    return (result);
}

bool IInterceptionBehavior.WillExecute { get { return (true); } }
}

```

## Configuring Interception by Attributes

We configure interception on an interceptable property or method by applying a [HandlerAttribute](#) that returns an instance of an [ICallHandler](#). For example, imagine you have an implementation of [ICallHandler](#) called **CachingHandler**, you could write a host attribute for it as this:

```

[Serializable]
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public class CachingHandlerAttribute : HandlerAttribute
{
    public CachingHandlerAttribute(int hours, int minutes, int seconds)
    {
        this.Duration = new TimeSpan(hours, minutes, seconds);
    }

    public TimeSpan Duration { get; public set; }

    public override ICallHandler CreateHandler()
    {
        return (new CachingHandler(this.Duration) { Order = this.Order });
    }
}

```

After we have our attribute, it's just a matter of applying it to a method. When Unity resolves the containing class, it will recognize it:

```

[CachingHandler(1, 0, 0)]           //1 hour, 0 minutes, 0 seconds
void DoLongOperation(int someParameter, int anotherParameter);

```

See chapter Chapter 5 Extending Unity for a full implementation.



**Note:** *Interception attributes can be applied to either the key or the component type.*



# Configuring Interception by Code

## Applying Interception

First, we need to tell Unity to use the [Interception](#) extension:

```
unity.AddNewExtension<Interception>().Configure<Interception>();
```

We have two basic options:

- Register an injection behavior
- Register a call handler

## Interception Behaviors

Interception behaviors are injected at registration time using the [RegisterType](#) method, like in the following example in which we add a **MyInterceptionBehavior** to the **FileLogger**:

```
unity.RegisterType<ILogger, FileLogger>(  
    new Interceptor<InterfaceInterceptor>(),  
    new InterceptionBehavior<MyInterceptionBehavior>());
```



**Note:** You can add several interception behaviors; all will be applied in sequence.

Notice how we define the interceptor to use ([InterfaceInterceptor](#)); it is also possible to configure it as the default for a type:

```
unity.Configure<Interception>().Configure<Interception>()  
    .SetDefaultInterceptorFor<ILogger>(new InterfaceInterceptor());
```

And this is it. Every time you ask for a **FileLogger** instance, you will get a proxy that includes the **MyInterceptionBehavior**.

## Call Handlers

Call handlers are a bit trickier to apply because we need to define to which member or members they will apply. For that, we add call handler types ([AddCallHandler](#) or [AddCallHandler<T>](#)) or instances ([AddCallHandler](#)) to a policy ([AddPolicy](#)) and a set of rules ([AddMatchingRule](#)):

```
unity.Configure<Interception>().Configure<Interception>()
    .AddPolicy("Add MeasureCallHandler to FileLogger.Log")
    .AddMatchingRule(new TypeMatchingRule(typeof(FileLogger)))
    .AddMatchingRule(new MemberNameMatchingRule("Log"))
    .AddCallHandler<MeasureCallHandler>();
```



**Tip:** You should add a call handler instance instead of its type if the call handler requires special configuration (like setting its execution order or if it doesn't have a public, parameterless constructor).

This example has two rules:

- A [TypeMatchingRule](#): Matches when the member is declared in the given type.
- A [MemberNameMatchingRule](#): Matches when the given member name is the same as the one supplied in the constructor.



**Tip:** The call handler is only applied if all rules match.



**Tip:** You can add any number of call handlers; they will be processed and sorted by their [Order](#) property.

A rule must implement [IMatchingRule](#), which exposes a very simple interface:

```
public interface IMatchingRule
{
    bool Matches(MethodBase member);
}
```

There are several rules included out of the box:

- [AssemblyMatchingRule](#): Matches the assembly name of the given member.

- [AttributeDrivenPolicyMatchingRule](#): Checks to see if the member (or type containing that member) has any [HandlerAttribute](#).
- [CustomAttributeMatchingRule](#): Checks to see if the member tested has an arbitrary attribute applied.
- [MethodSignatureMatchingRule](#): Matches methods with the given names and method signature.
- [NamespaceMatchingRule](#): Matches members in a given namespace. You can specify either a single namespace (e.g., System.Data) or a namespace root (e.g., System.Data.\* to match types in that namespace or below).
- [ParameterTypeMatchingRule](#): Matches methods that have any parameters of the given types.
- [PropertyMatchingRule](#): Matches properties by name. You can match the getter, setter, or both.
- [ReturnTypeMatchingRule](#): Checks to see if a member has a specified type.
- [TagAttributeMatchingRule](#): Checks a member for the presence of the [TagAttribute](#) on the method, property, or class, and that it matches a specific tag.

You can easily implement your own rules; see an example in Policy Rules.



**Note:** All rules that take names accept regular expressions.

## Adding Interfaces to Interception Proxies

When we register a type for interception, we can add additional interfaces that Unity will add to the generated proxy. The injection member classes used for this purpose are [AdditionalInterface](#) and [AdditionalInterface<T>](#). Here is one example of its usage:

```
unity.RegisterType<ILogger, FileLogger>(
    new Interceptor<InterfaceInterceptor>(),
    new InterceptionBehavior<DisposableBehavior>(),
    new AdditionalInterface<IDisposable>()
);
```

This tells Unity to add interface [IDisposable](#) to all proxies to **FileLogger** and to apply a **DisposableBehavior**. The result of calling [Resolve](#) will then implement the [IDisposable](#) interface and the [Dispose](#) method is available for invocation. Of course, because this method was not implemented by the **FileLogger** class, it is up to the **DisposableBehavior** to make sure that, if the method is called, something happens.

If you add some interface to a class, you have to implement all of the interface's methods and properties. A common example is adding support for [INotifyPropertyChanged](#), which is a very useful interface that is somewhat cumbersome to implement when there are many properties. Let's see how we can do it automatically with an interception behavior.

First, the registration; this is where we add the [INotifyPropertyChanged](#), set the interceptor ([VirtualMethodInterceptor](#) in this case), and set the interceptor class:

```
unity.RegisterType<MyComponent, MyComponent>(
    new Interceptor<VirtualMethodInterceptor>(),
    new InterceptionBehavior<NotifyPropertyChangedInterceptionBehavior>(),
    new AdditionalInterface<INotifyPropertyChanged>()
);
```

The NotifyPropertyChangedInterceptionBehavior class:

```
public class NotifyPropertyChangedInterceptionBehavior : IInterceptionBehavior
{
    private event PropertyChangedEventHandler handlers = delegate { };

    public IEnumerable<Type> GetRequiredInterfaces()
    {
        yield return (typeof(INotifyPropertyChanged));
    }

    public IMethodReturn Invoke(IMethodInvocation input,
        GetNextInterceptionBehaviorDelegate getNext)
    {
        var result = null as IMethodReturn;

        if (input.MethodBase.IsSpecialName == true)
        {
            if (input.MethodBase.Name.StartsWith("set_") == true)
```

```

        {
            result = getNext()(input, getNext());

            this.handlers(input.Target, new PropertyChangedEventArgs
(input.MethodBase.Name.Substring(4)));
        }

        else if (input.MethodBase.Name == "add_PropertyChanged")
        {

this.handlers += (input.Arguments[0] as PropertyChangedEventHandler);

            result = input.CreateMethodReturn(null, input.Arguments);
        }

        else if (input.MethodBase.Name == "remove_PropertyChanged")
        {

            this.handlers -
= (input.Arguments[0] as PropertyChangedEventHandler);

            result = input.CreateMethodReturn(null, input.Arguments);
        }
    }

    return (result);
}

public Boolean WillExecute { get { return(true); } }
}

```

And a sample usage:

```

var component = unity.Resolve<Component>();

var npc = component as INotifyPropertyChanged;

npc.PropertyChanged += delegate(Object source, PropertyChangedEventArgs args)

```

```
{
    //raised
};

component.Property = "Some Value"; //raise PropertyChanged event
```

## Configuring Interception by XML

### Applying Interception

When using XML configuration, we usually register the Unity extension ([Interception](#)) for applying the interception functionality and a section extension ([InterceptionConfigurationExtension](#)) that allows the configuration of it in XML:

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <sectionExtension type="Microsoft.Practices.Unity.InterceptionExtension
.Configuration.InterceptionConfigurationExtension, Microsoft.Practices.Unity.Interception.Configuration" />

    <container>
        <extension type="Microsoft.Practices.Unity.InterceptionExtension
.Interception, Microsoft.Practices.Unity.Interception" />

    </container>
</unity>
```

### Interception Behaviors

Behaviors are configured inside the [register](#) declaration, in an [interceptionBehavior](#) element:

```
<register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.
ConsoleLogger, Succinctly">
    <interceptor type="InterfaceInterceptor"/>
    <interceptionBehavior type="Succinctly.MyBehavior, Succinctly"/>
```

```
</register>
```

The [InterceptionBehavior](#) allows three attributes:

- **name** (optional): A name that, if supplied, causes Unity to resolve the behavior instead of just creating a new instance.
- **type** (required): The behavior type (a class that implements [IInterceptionBehavior](#)).
- **isDefaultForType** (optional): If **true**, the behavior will be applied to all registrations of this type; the default is **false**.

## Call Handlers

Call handlers are, as you know, slightly more complicated because of the matching rules:

```
<container>

  <register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.FileLogger,
Succinctly">

    <interceptor type="InterfaceInterceptor"/>

  </register>

  <interception>

    <policy name="Add MeasureCallHandler to FileLogger.Log">

      <matchingRule name="type" type="TypeMatchingRule">

        <constructor>

          <param name="typeName" value="Succinctly.FileLogger" />

        </constructor>

      </matchingRule>

      <matchingRule name="type" type="MemberNameMatchingRule">

        <constructor>

          <param name="nameToMatch" value="Log" />

        </constructor>

      </matchingRule>

    </policy>

  </interception>

</container>
```

```

        <callHandler name="MeasureCallHandler" type="Succinctly
.MeasureCallHandler, Succinctly" />

    </policy>

</interception>

</container>

```

I believe that the new sections [interceptors](#), [default](#), [policy](#), [matchingRule](#), and [callHandler](#) are self-explanatory so there's really nothing to add here. Do notice the [interceptor](#) declaration inside [register](#); this is required so that Unity knows what interceptor to use.

## Adding Interfaces to Interception Proxies

This is all it takes to add a new element, [addInterface](#), to the [register](#) declaration:

```

<register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.
ConsoleLogger, Succinctly">

    <interceptor type="InterfaceInterceptor"/>

    <addInterface type="System.IDisposable, mscorlib"/>

</register>

```

## Applying Interception in an Existing Instance

An alternative approach to resolving an intercepted component from Unity is to generate a runtime proxy through an interface to an existing instance. In the process, we can add a number of interception behaviors and even interfaces. We do that by calling one of the [ThroughProxy](#) overloads:

```

var logger = new FileLogger("output.log");

var loggerProxy = Intercept.ThroughProxy<ILogger>(logger,

    new InterfaceInterceptor(),

    new AddInterface<IDisposable>(),

    new IInterceptionBehavior[] { new MyInterceptionBehavior() });

```



```
var disposableLogger = loggerProxy as IDisposable;
```

Do note that this only works through interfaces and that the existing instance is left untouched.

## Included Call Handlers

The Enterprise Library—of which Unity is a part—includes a number of call handlers that you can use in your code:

*Table 3: Included call handlers*

Call Handler	Purpose
<a href="#">Logging Handler</a>	Logging Method Invocation and Property Access  <a href="#">LogCallHandler</a> Microsoft.Practices.EnterpriseLibrary.Logging.dll
<a href="#">Exception Handling</a>	Handling Exceptions in a Structured Manner  <a href="#">ExceptionCallHandler</a> Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.dll
<a href="#">Validation Handler</a>	Validating Parameter Values  <a href="#">ValidationCallHandler</a> Microsoft.Practices.EnterpriseLibrary.Validation.dll
<a href="#">Authorization Handler</a>	Authorizing Method and Property Requests  <a href="#">AuthorizationCallHandler</a> Microsoft.Practices.EnterpriseLibrary.Security.dll
<a href="#">Performance Counter Handler</a>	Measuring Target Method Performance  <a href="#">PerformanceCounterCallHandler</a> Microsoft.Practices.EnterpriseLibrary.PolicyInjection.dll



**Note:** Some of these call handlers require a specific installation procedure. Don't forget to read their documentation.

# Chapter 5 Extending Unity

## Introduction

Unity is highly extensible. This chapter talks about some of the various aspects by which it can be extended, and provides some examples of useful functionality that is not included out of the box.

## Lifetime Managers

It is certainly possible to create new lifetime managers. One example might be a pooled lifetime manager that creates up to N instances of a type and then returns them in a round robin fashion. Let's see how we could implement one. Enter class **PooledLifetimeManager**:

```

public sealed class PooledLifetimeManager : LifetimeManager, IDisposable
{
    private const Int32 DefaultPoolSize = 10;
    private readonly List<Object> pool;
    private Int32 index = -1;

    public PooledLifetimeManager(Int32 poolSize)
    {
        this.pool = new List<Object>(poolSize);
    }

    public PooledLifetimeManager() : this(DefaultPoolSize)
    {
    }

    public override Object GetValue()
    {
        if (this.pool.Count < this.pool.Capacity)
        {
            return (null);
        }
        else
        {
            if (++this.index == this.pool.Capacity)
            {

```

```

        this.index = 0;

    }

    return (this.pool[this.index]);

}

}

public override void SetValue(Object newValue)
{
    this.pool.Add(newValue);
}

public override void RemoveValue()
{
    this.Dispose();
}

public void Dispose()
{
    foreach (var disposable in this.pool.OfType<IDisposable>())
    {
        disposable.Dispose();
    }

    this.pool.Clear();

    this.index = -1;
}
}

```

This class will return up to 10 new instances by default; after that, [Resolve](#) will return one of the existing ones in round robin order. To the [Resolve](#) method, the lifetime manager does not matter; it is totally transparent.

When implementing a lifetime manager, you need to know that its methods are called in this order:

1. [GetValue](#): Returns an existing instance from the lifetime manager store, if it exists. If it doesn't exist or if one should be created, return **null**.
2. [SetValue](#): Allows the lifetime manager to store a new instance that was created by Unity, if [GetValue](#) returned **null**.
3. [RemoveValue](#): Removes the created instance from the lifetime manager store; never called by Unity but can be called explicitly.
4. [IDisposable.Dispose](#): Only called if and when the Unity container is disposed of. Has the responsibility to dispose of all the instances implementing [IDisposable](#) that have been created by the lifetime manager.

## Convention Classes

For reusability purposes, Unity allows us to have conventions classes that wrap these rules. These classes need to inherit from [RegistrationConvention](#) and one class with the exact same behavior is:

```
public class InterfaceToClassConvention : RegistrationConvention
{
    private readonly IUnityContainer unity;

    private readonly IEnumerable<Type> types;

    public InterfaceToClassConvention(IUnityContainer unity, params
Assembly [] assemblies) : this(unity, assemblies
.SelectMany(a => a.GetExportedTypes()).ToArray())
    {
        this.unity = unity;
    }

    public InterfaceToClassConvention(IUnityContainer unity, params Type[] types)
```

```

{
    this.unity = unity;

    this.types = types ?? Enumerable.Empty<Type>();
}

public override Func<Type, IEnumerable<Type>> GetFromTypes()
{
    return (WithMappings.FromAllInterfacesInSameAssembly);
}

public override Func<Type, IEnumerable<InjectionMember>>
GetInjectionMembers()
{
    return (x => Enumerable.Empty<InjectionMember>());
}

public override Func<Type, LifetimeManager> GetLifetimeManager()
{
    return (WithLifetime.None);
}

public override Func<Type, String> GetName()
{
    return (type => (this.unity.Registrations.Select(x => x.RegisteredType)
.Any(r => type.GetInterfaces().Contains(r))) ? WithName.TypeName(type) :
WithName.Default(type));
}

```

```

public override IEnumerable<Type> GetTypes()
{
    return (this.types.Where(x => (x.IsPublic) &&
(x.GetInterfaces().Any()) && (!x.IsAbstract) && (x.IsClass))));
}
}

```

And you would use it as:

```

unity.RegisterTypes(new InterfaceToClassConvention(unity,
Assembly.GetExecutingAssembly()));

```

It is a nice feature to have at hand.

## Extensions

Unity allows us to register extensions to augment its basic functionality. These extensions can be registered either by code or by XML configuration. One such registration might be one that sets up interception and then goes through all registered types (current and future) and sets its interceptors accordingly (see chapter Chapter 4 Aspect-Oriented Programming):

```

public class DefaultInterception : Interception
{
    private static readonly IInterceptor [] interceptors = typeof(IInterceptor)
        .Assembly
        .GetExportedTypes()
        .Where
        (
            type =>
                (typeof(IInterceptor).IsAssignableFrom(type) == true) &&

```

```

        (type.IsAbstract == false) &&
        (type.IsInterface == false)
    )
    .Select(type => Activator.CreateInstance(type) as IInterceptor)
    .ToArray();

protected override void Initialize()
{
    base.Initialize();

    var configSource = ConfigurationSourceFactory.Create();
    var section = configSource.GetSection(
PolicyInjectionSettings.SectionName) as PolicyInjectionSettings;

    if (section != null)
    {
        PolicyInjectionSettings.ConfigureContainer(this.Container,
configSource);
    }

    foreach (var registration in this.Container.Registrations)
    {
        if (registration.RegisteredType.Assembly !=
typeof(IUnityContainer).Assembly)
        {
            this.SetInterceptorFor(registration.RegisteredType,
registration.MappedToType, registration.Name);

```



```

    }

}

this.Context.Registering += delegate(Object sender, RegisterEventArgs e)
{
    this.SetInterceptorFor(e.TypeFrom ?? e.TypeTo, e.TypeTo, e.Name);
};

this.Context.RegisteringInstance += delegate(Object sender,
RegisterInstanceEventArgs e)
{
    this.SetInterceptorFor(e.RegisteredType, e.Instance.GetType(),
e.Name);
};
}

private void SetInterceptorFor(Type typeFrom, Type typeTo, String name)
{
    foreach (var interceptor in interceptors)
    {
        if ((interceptor.CanIntercept(typeFrom))
&& (interceptor.GetInterceptableMethods(typeFrom, typeTo).Any()))
        {
            if (interceptor is IInstanceInterceptor)
            {
                this.Container.Configure<Interception>().SetInterceptorFor(

```

```

typeFrom, name, interceptor as IInstanceInterceptor);
    }
    else if (interceptor is ITypeInterceptor)
    {
        this.Container.Configure<Interception>().SetInterceptorFor(
typeFrom, name, interceptor as ITypeInterceptor);
    }
    break;
}
}
}
}
}
}
}

```



**Tip:** You will need to add a reference to the *System.Configuration* assembly and also the NuGet package *EnterpriseLibrary.PolicyInjection*. See [Chapter 4 Aspect-Oriented Programming](#).

An extension for Unity must inherit from [UnityContainerExtension](#); in our example, we are instead inheriting from one subclass that Unity offers for configuring interception (AOP) policies, [Interception](#). Here we create a static list of all interceptors available, go through all registered types ([Registrations](#) collection), and set up an interceptor for each registration that can intercept its key type. At the same time, we hook up the [RegisteringInstance](#) and [Registering](#) events, and we do the same for every new registration that may exist.

We configure this extension in code as this:

```
unity.AddNewExtension<DefaultInterception>();
```

And in XML:

```

<container>
    <extension type="Succinctly.DefaultInterception, Succinctly" />
</container>

```



**Tip:** Again, don't forget to call `LoadConfiguration()` otherwise the extension will not be loaded.

If you need access to your extension instance, which may have been defined in XML or in code, you can get a reference to it by calling the [Configure](#) method:

```
var defaultInterceptionExtension = unity.Configure<DefaultInterception>();
```

## Call Handlers

Imagine we want to implement a caching call handler so that all methods that have it applied to (the first time they are called) will cache their result for some time. All subsequent calls in that period of time will not cause the method to execute but, instead, will return the cached value.

A simple implementation, making use of .NET 4's extensible caching providers, follows as a marker attribute to be applied to cacheable methods that is also a cache handler implementation:

```
[Serializable]
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public class CachingHandlerAttribute : HandlerAttribute, ICallHandler
{
    private readonly Guid KeyGuid = new Guid("ECFD1B0F-0CBA-4AA1-89A0-179B636381CA");

    public CachingHandlerAttribute(int hours, int minutes, int seconds)
    {
        this.Duration = new TimeSpan(hours, minutes, seconds);
    }

    public TimeSpan Duration { get; private set; }
```

```

public override ICallHandler CreateHandler(IUnityContainer ignored)
{
    return (this);
}

IMethodReturn ICallHandler.Invoke(IMethodInvocation input,
GetNextHandlerDelegate getNext)
{
    if (this.TargetMethodReturnsVoid(input) == true)
    {
        return (getNext()(input, getNext));
    }

    var inputs = new Object [ input.Inputs.Count ];
    for (var i = 0; i < inputs.Length; ++i)
    {
        inputs [ i ] = input.Inputs [ i ];
    }

    var cacheKey = this.CreateCacheKey(input.MethodBase, inputs);
    var cache = MemoryCache.Default;
    var cachedResult = (Object []) cache.Get(cacheKey);
    if (cachedResult == null)
    {
        var realReturn = getNext()(input, getNext);
        if (realReturn.Exception == null)
        {
            this.AddToCache(cacheKey, realReturn.ReturnValue);
        }
    }
}

```

```

    }

    return (realReturn);

}

var cachedReturn = input.CreateMethodReturn(cachedResult [ 0 ],
input.Arguments);

return (cachedReturn);
}

private Boolean TargetMethodReturnsVoid(IMethodInvocation input)
{
    var targetMethod = input.MethodBase as MethodInfo;

    return ((targetMethod != null) &&
(targetMethod.ReturnType == typeof(void)));
}

private void AddToCache(String key, Object value)
{
    var cache = MemoryCache.Default;

    var cacheValue = new Object [] { value };

    cache.Add(key, cacheValue, DateTime.Now + this.Duration);
}

private String CreateCacheKey(MethodBase method,
params Object [] inputs)
{

```

```

var sb = new StringBuilder();

sb.AppendFormat("{0}:", Process.GetCurrentProcess().Id);

sb.AppendFormat("{0}:", KeyGuid);

if (method.DeclaringType != null)
{
    sb.Append(method.DeclaringType.FullName);
}

sb.Append(':');

sb.Append(method);

if (inputs != null)
{
    foreach (var input in inputs)
    {
        sb.Append(':');

        if (input != null)
        {
            sb.Append(input.GetHashCode().ToString());
        }
    }
}

return (sb.ToString());
}
}

```



**Tip:** You will need to add a reference to assembly *System.Runtime.Caching*.

This attribute can be applied to any interceptable method (see Table 2: Interception kinds and targets) with a return type other than `void`. When the method is called the first time, the call handler generates a key from the method name and signature, plus the hash code of all of its parameters and the current process identifier. The intercepted method is called and its result is stored under the generated cache key for a period of time. In all subsequent calls, the call handler checks the cache for an entry with the generated key and, in case it is present, returns the cached value immediately. The cache provider in use is the one returned from [MemoryCache.Default](#), which means the cache will be stored in memory. By all means, you can switch to a different one; for example, the ASP.NET cache.

Here's how we would apply the caching handler attribute:

```
[CachingHandler(1, 0, 0)]           //1 hour, 0 minutes, 0 seconds
void DoLongOperation(int someParameter, int anotherParameter);
```



**Note:** *In a real-life case, you would probably want to separate the `ICallHandler` implementation from the `HandlerAttribute` for better decoupling and manageability.*

## Injection Values

In chapter Chapter 3 Dependency Injection, we saw how we can configure dependencies so that our components have constructor and method parameters and properties injected automatically upon them. However, the techniques that were shown only allow two types of injected values:

- Instance values specified explicitly
- Components resolved from Unity

This is a big limitation; fortunately, Unity offers some extension hooks for more advanced options. As a sample, let's implement a way to inject values coming from the [appSettings](#) configuration section.

We start by defining a custom section extension, which we will use in XML configuration (if you don't plan to use XML, just skip it):

```
public class AppSettingsParameterInjectionElementExtension : SectionExtension
{
    public override void AddExtensions(SectionExtensionContext context)
```

```

{
    context.AddElement<AppSettingsParameterValueElement>("appSettings");
}
}

```

Then, the [ValueElement](#) implementation; it's a bit more complex since it has to support property, method, and constructor injection, and also convert the value read from the [appSettings](#) into the target type:

```

public class AppSettingsParameterValueElement : ValueElement,
IDependencyResolverPolicy
{
    private Object CreateInstance(Type parameterType)
    {
        var configurationValue = ConfigurationManager.AppSettings
[this.AppSettingsKey] as Object;

        if (parameterType != typeof(String))
        {
            var typeConverter = this.GetTypeConverter(parameterType);

            if (typeConverter != null)
            {
                configurationValue = typeConverter.
ConvertFromInvariantString(configurationValue as String);
            }
        }

        return (configurationValue);
    }
}

```



```

private TypeConverter GetTypeConverter(Type parameterType)
{
    if (String.IsNullOrEmpty(this.TypeConverterTypeName) == false)
    {
        return (Activator.CreateInstance(TypeResolver.ResolveType(
this.TypeConverterTypeName)) as TypeConverter);
    }
    else
    {
        return (TypeDescriptor.GetConverter(parameterType));
    }
}

public override InjectionParameterValue GetInjectionParameterValue(
IUnityContainer container, Type parameterType)
{
    var value = this.CreateInstance(parameterType);
    return (new InjectionParameter(parameterType, value));
}

Object IDependencyResolverPolicy.Resolve(IBuilderContext context)
{
    Type parameterType = null;

    if (context.CurrentOperation is ResolvingPropertyValueOperation)
    {

```

```

        var op = context.CurrentOperation as ResolvingPropertyValueOperation;

        var prop = op.TypeBeingConstructed.GetProperty(op.PropertyName);

        parameterType = prop.PropertyType;
    }

    else if (context.CurrentOperation is ConstructorArgumentResolveOperation)
    {
        var op = context.CurrentOperation as
ConstructorArgumentResolveOperation;

        var args = op.ConstructorSignature.Split('(')[1].Split(' ')[0];

        var types = args.Split(',').

Select(a => Type.GetType(a.Split(' ')[0])).ToArray();

        var ctor = op.TypeBeingConstructed.GetConstructor(types);

        parameterType = ctor.GetParameters()

.Single(p => p.Name == op.ParameterName).ParameterType;
    }

    else if (context.CurrentOperation is MethodArgumentResolveOperation)
    {
        var op = context.CurrentOperation as MethodArgumentResolveOperation;

        var methodName = op.MethodSignature.Split('(')[0].Split(' ')[1];

        var args = op.MethodSignature.Split('(')[1].Split(' ')[0];

        var types = args.Split(',')

.Select(a => Type.GetType(a.Split(' ')[0])).ToArray();

        var method = op.TypeBeingConstructed.GetMethod(methodName, types);

        parameterType = method.GetParameters()

.Single(p => p.Name == op.ParameterName).ParameterType;
    }

```

```

    }

    return (this.CreateInstance(parameterType));
}

[ConfigurationProperty("appSettingsKey", IsRequired = true)]
public String AppSettingsKey
{
    get
    {
        return (base["appSettingsKey"] as String);
    }
    set
    {
        base["appSettingsKey"] = value;
    }
}
}

```

Finally, the application attribute, which you will only need if you want to use attribute-based configuration:

```

[Serializable]
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Property, AllowMultiple = false, Inherited = true)]
public class AppSettingsAttribute : DependencyResolutionAttribute
{
    public AppSettingsAttribute(String appSettingsKey)
    {

```

```

        this.AppSettingsKey = appSettingsKey;
    }

    public String TypeConverterTypeName { get; set; }

    public String AppSettingsKey { get; private set; }

    public override IDependencyResolverPolicy CreateResolver(Type typeToResolve)
    {
        return (new AppSettingsParameterValueElement() { AppSettingsKey =
this.AppSettingsKey, TypeConverterTypeName = this.TypeConverterTypeName });
    }
}

```

That's it. Now we can apply our value injector in XML:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <section name="unity" type="Microsoft.Practices.Unity.Configuration.
UnityConfigurationSection, Microsoft.Practices.Unity.Configuration"/>
    </configSections>
    <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
        <sectionExtension
type="Succinctly.AppSettingsParameterInjectionElementExtension, Succinctly" />
        <container>
            <register type="Succinctly.Ilogger, Succinctly" mapTo="Succinctly.
FileLogger, Succinctly" name="File">

```

```

        <property name="Filename">
            <appSettings appSettingsKey="LoggerFilename"/>
        </property>
    </register>
</container>
</unity>
<appSettings>
    <add key="LoggerFilename" value="Log.txt" />
</appSettings>
</configuration>

```

Or through an attribute applied to an injected property:

```

public class FileLogger : ILogger
{
    [AppSettings("LoggerFilename")]
    public String Filename { get; set; }

    public void Log(String message) { }
}

```

Or constructor parameter:

```

public class FileLogger : ILogger
{
    public FileLogger ([AppSettings("LoggerFilename")] String filename)
    {
        this.Filename = filename;
    }
}

```

```

    }

    public String Filename { get; set; }

    public void Log(String message) { }
}

```

Or even to an injection method parameter:

```

public class FileLogger : ILogger
{
    [InjectionMethod]
    public void Initialize([AppSettings("LoggerFilename")] String filename)
    {
        this.Filename = filename;
    }

    public String Filename { get; set; }

    public void Log(String message) { }
}

```

## Injection Factories in XML Configuration

[Chris Tavares](#), one of the leading developers of Unity, has published [in his Bitbucket repository](#) a sample that shows how to implement a section extension for defining an injection factory method in XML. This injection factory is a way to create object instances by calling a static method of a class.

I advise you to look at Chris's code. Just as an appetizer, you can configure it as this:

```

<?xml version="1.0" encoding="utf-8"?>

<configuration>

  <configSections>

    <section name="unity" type="Microsoft.Practices.Unity.Configuration.
UnityConfigurationSection, Microsoft.Practices.Unity.Configuration"/>

  </configSections>

  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

    <sectionExtension type="Unity.FactoryConfig.FactoryConfigExtension,
Unity.FactoryConfig"/>

    <container>

      <register type="Succinctly.ILogger, Succinctly" mapTo="Succinctly.
ConsoleLogger, Succinctly">

        <factory type="Succinctly.LoggerFactory, Succinctly"
method="Create" />

      </register>

    </container>

  </unity>

</configuration>

```

I think it is easy to understand: There should be a **LoggerFactory** class that exposes a static method called **Create** which returns an **ILogger** instance. Instances of the **ILogger** type, when resolved by Unity, will be built using this method.

## Policy Rules

The included rules for member matching cover quite a few scenarios but one thing they do not have is logical groupings: **OR** and **AND**. Fortunately, this is very easy to achieve. We can do this by rolling our own classes that implement [IMatchingRule](#):

```

public sealed class AndRule : IMatchingRule

```

```

{
    private readonly IEnumerable<IMatchingRule> rules;

    public AndRule(IEnumerable<IMatchingRule> rules)
    {
        this.rules = rules;
    }

    public AndRule(params IMatchingRule[] rules)
    {
        this.rules = rules;
    }

    public Boolean Matches(MethodBase member)
    {
        foreach (var rule in this.rules)
        {
            if (rule.Matches(member) == false)
            {
                return (false);
            }
        }
        return (true);
    }
}

public sealed class OrRule : IMatchingRule

```



```

{
    private readonly IEnumerable<IMatchingRule> rules;

    public OrRule(IEnumerable<IMatchingRule> rules)
    {
        this.rules = rules;
    }

    public OrRule(params IMatchingRule [] rules)
    {
        this.rules = rules;
    }

    public Boolean Matches(MethodBase member)
    {
        foreach (var rule in this.rules)
        {
            if (rule.Matches(member) == true)
            {
                return (true);
            }
        }
        return (false);
    }
}

```

# Chapter 6 Other Application Programming Interfaces (APIs)

## Introduction

A number of technologies in the Microsoft stack use IoC. In fact, its number seems to be growing every time a new version or technology is launched. It is generally possible to plug in an IoC container of choice; in our case, let's see how to use Unity.

## ASP.NET Web Forms

### Injecting Dependencies

The ASP.NET Web Forms framework doesn't allow the retrieval of pages or controls from an IoC container. It does, however, allow injecting dependencies on a page after it was created. One way to do that is by rolling our own implementation of [IHandlerFactory](#), the interface that is responsible for creating handlers (such as pages), and adding the dependency injection behavior.

Our page handler factory could be:

```
public class UnityHandlerFactory : PageHandlerFactory
{
    public override IHttpHandler GetHandler(HttpContext context, string requestType,
        string virtualPath, string path)
    {
        var handler = base.GetHandler(context, requestType, virtualPath, path);
        var unity = ServiceLocator.Current.GetInstance<IUnityContainer>();
        unity.BuildUp(handler.GetType(), handler);
        return (handler);
    }
}
```

We inherit from [PageHandlerFactory](#) because it knows better than we do about how to build pages. After it does that, we call [BuildUp](#) to inject any required dependencies. We need to register the page handler factory on the **Web.config** file as the default factory for all **.aspx** requests; it should go in the [handlers](#) section of [system.webServer](#):

```
<system.webServer>

  <handlers>

    <add path="*.aspx" type="Succinctly.UnityHandlerFactory, Succinctly"
verb="*" name="UnityHandlerFactory"/>

  </handlers>

</system.webServer>
```

For injecting dependencies in controls, we should call [BuildUp](#) explicitly upon them, since there is no hook that allows us to do that automatically.

## Disposing of Components

When we have components that use the [PerRequestLifetimeManager](#), these components don't get disposed of automatically at the end of the request unless we enable the module [UnityPerRequestHttpModule](#). This module is part of the [Unity bootstrapper for ASP.NET MVC](#) package but don't mind the name: it can be useful for Web Forms projects as well. What it does is, at the [EndRequest](#) event, it checks all of the components that were created by Unity and implements [IDisposable](#) and calls [Dispose](#) upon them. Register [UnityPerRequestHttpModule](#) in the [modules](#) section of [system.webServer](#):

```
<system.webServer>

  <modules>

    <add name="UnityPerRequestHttpModule"
type="Microsoft.Practices.Unity.Mvc.UnityPerRequestHttpModule,
Microsoft.Practices.Unity.Mvc"/>

  </modules>

</system.webServer>
```



**Tip:** An alternative approach to using [UnityPerRequestHttpModule](#) is to create a child container and store it in `HttpContext.Items` and dispose of it in `EndRequest`.

# ASP.NET MVC

## Resolving Components

ASP.NET MVC's dependency resolution interface is [IDependencyResolver](#):

```
public interface IDependencyResolver
{
    object GetService(Type serviceType);

    IEnumerable<object> GetServices(Type serviceType);
}
```

The methods it exposes should be very familiar to us; the only differences to Unity/Common Service Locator is that they don't have strong-typed versions and they don't accept named registrations. A simple implementation is:

```
public class UnityDependencyResolver : IDependencyResolver
{
    private readonly IUnityContainer unity;

    public UnityDependencyResolver(IUnityContainer unity)
    {
        this.unity = unity;
    }

    public object GetService(Type serviceType)
    {
        try
        {

```

```

        var svc = this.unity.Resolve(serviceType);

        return (svc);
    }

    catch
    {
        return (null);
    }
}

public IEnumerable<object> GetServices(Type serviceType)
{
    try
    {
        var svcs = this.unity.ResolveAll(serviceType);

        return (svcs);
    }

    catch
    {
        return (Enumerable.Empty<object>());
    }
}
}

```



**Tip:** Note that, unlike Unity, the *IDependencyResolver* **resolution methods cannot throw**.

## Injecting Controllers

Then, when we register a controller factory, we implement our own by inheriting from [DefaultControllerFactory](#) and we pass it our Unity instance:

```
public class UnityControllerFactory : DefaultControllerFactory
{
    private readonly IUnityContainer unity;

    public UnityControllerFactory(IUnityContainer unity)
    {
        this.unity = unity;
    }

    public override IController CreateController(RequestContext requestContext,
string controllerName)
    {
        var controller = unity.Resolve<IController>(controllerName);

        if (controller == null)
        {
            controller = base.CreateController(requestContext, controllerName);
        }

        if (controller != null)
        {
            unity.BuildUp(controller.GetType(), controller);
        }
    }
}
```

```

        return (controller);
    }
}

```

This controller factory first tries to resolve a registered controller from Unity and then, if one is not found, falls back to the default creation method. If a controller is found, it tries to inject all of its dependencies.

## Registration

We should register our dependency resolver in the **Global.asax** code-behind class; for example, in **Application\_Start**:

```

DependencyResolver.SetResolver(new UnityDependencyResolver(unity));
ControllerBuilder.Current.SetControllerFactory(new UnityControllerFactory(unity));

```

Of course, for it to find a controller, you have to register one. You can do that explicitly or by using a convention (see Registration by Convention). An explicit registration is as follows:

```

unity.RegisterInstance<IController>("Home", new HomeController());

```

## Injecting Action Method Parameters

Besides returning the full controller instance, we can inject parameters in action methods automatically. First, let's start by defining a custom model binder ([IModelBinder](#) implementation):

```

public sealed class ServiceLocatorModelBinder : IModelBinder
{
    public Object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        return (ServiceLocator.Current.GetInstance(bindingContext.ModelType));
    }
}

```

```
}
```

We need to set our model binder on the parameter we wish to inject. We do this by applying a [ModelBinderAttribute](#); no additional registration is required:

```
public ActionResult Index(  
    [ModelBinder(typeof(ServiceLocatorModelBinder))] ILogger logger)  
{  
    return this.View();  
}
```

## Disposing of Components

To dispose of registered instances, you should also register the [UnityPerRequestHttpModule](#) module. See this discussion in [For injecting dependencies in controls](#), we should call BuildUp explicitly upon them, since there is no hook that allows us to do that automatically.

Disposing of Components.

# ASP.NET Web API

## Component Resolution

ASP.NET Web API also has an extensible dependency resolution interface named [IDependencyResolver](#) but with a slightly more complex interface (which we can see flattened here):

```
public interface IDependencyResolver : IDependencyScope, IDisposable  
{  
    object GetService(Type serviceType);  
  
    IEnumerable<object> GetServices(Type serviceType);  
}
```



```
IDependencyScope BeginScope();

void Dispose();
}
```

I suggest the following implementation:

```
public class UnityDependencyResolver : IDependencyResolver
{
    private readonly IUnityContainer unity;
    private readonly IDependencyResolver baseResolver;
    private readonly LinkedList<IUnityContainer> children =
new LinkedList<IUnityContainer>();

    public UnityDependencyResolver(IUnityContainer unity,
IDependencyResolver baseResolver)
    {
        this.unity = unity;
        this.baseResolver = baseResolver;
    }

    public IDependencyScope BeginScope()
    {
        var child = this.unity.CreateChildContainer();
        this.children.AddLast(child);
        return (new UnityDependencyResolver(child));
    }
}
```

```
public object GetService(Type serviceType)
{
    try
    {
        var svc = this.unity.Resolve(serviceType);
        return (svc);
    }
    catch
    {
        return (null);
    }
}

public IEnumerable<object> GetServices(Type serviceType)
{
    try
    {
        var svcs = this.unity.ResolveAll(serviceType);
        return (svcs);
    }
    catch
    {
        return (Enumerable.Empty<object>());
    }
}
```

```

public void Dispose()
{
    foreach (var child in this.children)
    {
        child.Dispose();
    }

    this.children.Clear();
}
}

```



**Tip:** Again note that, unlike Unity, the *IDependencyResolver* resolution methods cannot throw.

Three things to keep in mind:

- If a type cannot be resolved, it delegates its resolution to the base resolver.
- [BeginScope](#) starts a new child container; this is for defining a new isolated scope, and it keeps a reference to it.
- [Dispose](#) disposes of all of the child containers but not of the actual Unity instance because it is probably disposed of somewhere else.

## Dependency Injection in Controllers

For dependency injection in controllers, we should also implement our own [IHttpControllerActivator](#), which is the interface responsible for creating controller instances. This ensures that, even if it's not Unity that creates an instance of a controller, its dependencies are still injected properly. An example could be:

```

public class UnityControllerActivator : IHttpControllerActivator
{
    private readonly IUnityContainer unity;

    private readonly IHttpControllerActivator baseActivator;

    public UnityControllerActivator(IUnityContainer unity,

```

```

IHttpControllerActivator baseActivator)
{
    this.unity = unity;
    this.baseActivator = baseActivator;
}

public IHttpController Create(HttpRequestMessage request,
HttpControllerDescriptor controllerDescriptor, Type controllerType)
{
    var controller = GlobalConfiguration.Configuration.DependencyResolver
.GetService(controllerType) as IHttpController;

    if (controller == null)
    {
        controller = this.baseActivator.Create(request, controllerDescriptor,
controllerType);

        if (controller != null)
        {
            this.unity.BuildUp(controller.GetType(), controller);
        }
    }
    return (controller);
}
}

```

## Registration

All registration goes in **Global.asax.cs**, method **Application\_Start**:

```
GlobalConfiguration.Configuration.DependencyResolver = new UnityDependencyResolver(
    unity);

GlobalConfiguration.Configuration.Services.Replace(typeof(IHttpControllerActivator)
    , new UnityControllerActivator(unity, GlobalConfiguration.Configuration.Services
    .GetHttpControllerActivator()));
```

## Dependency Injection in Action Methods

For action methods, it is equally as easy to inject resolved values. Let's define a **DependencyResolverParameterBindingAttribute** by inheriting from [ParameterBindingAttribute](#):

```
[Serializable]

[AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false)]

public sealed class DependencyResolverParameterBindingAttribute :
    ParameterBindingAttribute
{
    public override HttpParameterBinding GetBinding(
        HttpParameterDescriptor parameter)
    {
        return (new DependencyResolverParameterBinding(parameter));
    }
}
```

And an associated parameter binder class ([HttpParameterBinding](#)):

```
public sealed class DependencyResolverParameterBinding : HttpParameterBinding
{
    public DependencyResolverParameterBinding(HttpParameterDescriptor descriptor)
```

```

: base(descriptor) { }

public override Task ExecuteBindingAsync(ModelMetadataProvider metadataProvider,
HttpActionContext actionContext, CancellationToken cancellationToken)
{
    if (actionContext.ControllerContext.Configuration.DependencyResolver != null)
    {
        actionContext.ActionArguments[this.Descriptor.ParameterName] =
actionContext.ControllerContext.Configuration.DependencyResolver.GetService
(this.Descriptor.ParameterType);
    }
    return(Task.FromResult(0));
}
}

```

With these, it is easy to say that an action method's parameter should come from Unity:

```

[HttpGet]
[Route("api/home/index")]
public IQueryable<string> Index(
[DependencyResolverParameterBinding] ILogger logger)
{
    return (new [] { "a", "b", "c" }.AsQueryable());
}

```

The **logger** parameter is automatically injected by MVC.

# Entity Framework

## Component Resolution

Entity Framework 6 brought along extensibility in the form of a dependency resolution API. It defines an interface for resolving components but only contains simple implementations of it ([ExecutionStrategyResolver<T>](#), [SingletonDependencyResolver<T>](#), and [TransactionHandlerResolver](#)). Instead, Entity Framework allows developers to plug in their own IoC of choice by implementing a simple adapter.

The dependency resolution interface is [IDbDependencyResolver](#) and it is a very simple one, only two methods:

```
public interface IDbDependencyResolver
{
    object GetService(Type type, object key);
    IEnumerable<object> GetServices(Type type, object key);
}
```

As you can see, these methods match closely to Unity's own component resolution ones. Therefore, a Unity adapter for Entity Framework's dependency resolution interface could be:

```
public class UnityDbDependencyResolver : IDbDependencyResolver
{
    private readonly IUnityContainer unity;

    public UnityDbDependencyResolver(IUnityContainer unity)
    {
        this.unity = unity;
    }

    public object GetService(Type type, object key)
```

```

{
    try
    {
        var component = this.unity.Resolve(type, (key ?? "").ToString());
        return (component);
    }
    catch
    {
        return (null);
    }
}

public IEnumerable<object> GetServices(Type type, object key)
{
    try
    {
        var components = this.unity.ResolveAll(type);
        return (components);
    }
    catch
    {
        return (Enumerable.Empty<object>());
    }
}
}

```



**Tip:** Again, the *IDbDependencyResolver* **resolution method cannot throw**.



## Registration

Like I said, Entity Framework already contains a private implementation of this interface and it supports a dependency resolution pipeline. You register your adapter in a [DbConfiguration](#) class with the same prefix as your [DbContext](#) so that Entity Framework can find it automatically, or you add a [DbConfigurationTypeAttribute](#) to it by calling [AddDefaultResolver](#). This ensures that your resolver will be the first to be used:

```
public class MyContextConfiguration : DbConfiguration
{
    public MyContextConfiguration()
    {
        this.AddDefaultResolver(new UnityDbDependencyResolver(unity));
    }
}
```

## Chapter 7 Putting It All Together

Microsoft Unity is a very interesting framework. Not only does Microsoft Unity support a good number of features out of the box, it is also extremely extensible—and we’ve barely scratched the surface of it.

By looking at its source code, you can learn how it works and this knowledge will help you make better use of it. There is also a vast community of developers using Unity; you will most likely be able to find help should you run into trouble.

IoC is definitely a hot topic nowadays and all modern APIs seem to support it to some extent.

Because of its simplicity, you can use Unity virtually anywhere—from Windows and web applications to services and even containers such as SharePoint.

I particularly like Unity’s implementation of AOP as it is very simple and powerful and, since it does not rely on post-compilation steps, it’s much easier and less prone to errors when you implement it.

## Chapter 8 Getting Help

The primary source for information on Unity should be the [Discussions](#) page on the [Unity home page at CodePlex](#). Before you ask a question or ask for a new functionality, make sure nobody has already done so first.

Then there is the [Issues](#) page. On this page you can see bug reports, track their progress, and maybe even raise your own issues.

The Class Library reference is available on Microsoft's [web site](#).

The [patterns & practices](#) group has two e-books freely available on their web site: [Dependency Injection with Unity](#) only covers version 2 but the [Developer's Guide to Microsoft Unity](#) already covers version 3. You can also get [Dependency Injection with Unity](#) in hard copy from your favorite store.

Microsoft published some sample quick start code on their [web site](#). Have a look at the samples and see if they can help.

[Stack Overflow](#) keeps track of questions related to Unity under the tag "[unity](#)".

I myself [blog](#) about Unity occasionally and all posts are tagged "[unity](#)" as well.

# Additional References

Click on the following links for more information on each topic:

[ASP.NET MVC 4 Dependency Injection](#)

[Aspect-Oriented Programming](#)

[Common Service Locator](#)

[Dependency Injection](#)

[Dependency Injection for Web API Controllers](#)

[\*Dependency Injection with Unity\*](#)

[\*Developer's Guide to Microsoft Unity\*](#)

[Development With A Dot blog about Unity](#)

[EF Dependency Resolution](#)

[Inversion of Control](#)

[Guidance to Enable Unity in a SharePoint App](#)

[Microsoft Unity 3](#)

[Patterns & practices – Enterprise Library](#)

[Patterns & practices – Unity](#)

[Service locator pattern](#)

[SOLID \(Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion\)](#)

[Stack Overflow – Unity](#)

[Unity Bootstrapper for ASP.NET MVC](#)

[Unity Container](#)

[UnityFactoryConfig](#)

[Unity.WebAPI](#)

[Unity.WebForms](#)

[WebApiContrib.IoC.Unity](#)