Contents

[Requirements to follow the code 2](#_Toc250472174)

[New up an object using the container 2](#_Toc250472175)

[Constructor Injection 3](#_Toc250472176)

[Property Injection 4](#_Toc250472177)

[Method Injection 4](#_Toc250472178)

[Service Location Pattern (Injecting the Container) 5](#_Toc250472179)

[Registering types 6](#_Toc250472180)

[Late Binding 6](#_Toc250472181)

[Managing Object Lifetimes 7](#_Toc250472182)

[Registering singleton instances 7](#_Toc250472183)

[Registering named instances 8](#_Toc250472184)

[Singleton Pattern 8](#_Toc250472185)

[Component Management 11](#_Toc250472186)

[Working with “Lambdas” 14](#_Toc250472187)

# Requirements to follow the code

Create a console app in Visual Studio 2008 / 2010 and add the following references. You can download the assemblies under: <http://www.codeplex.com/unity> The current release used in this document is Unity 1.2

Unity 1.2 ships with the following assemblies:

Microsoft.Practices.ObjectBuilder2.dll

Microsoft.Practices.Unity.Configuration.dll

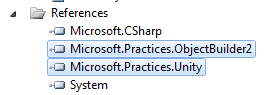
Microsoft.Practices.Unity.dll

Microsoft.Practices.Unity.Interception.Configuration.dll

Microsoft.Practices.Unity.Interception.dll

Microsoft.Practices.Unity.StaticFactory.dll

For the first part of this document you will only need the Unity and ObjectBuilder2 assemblies.



# New up an object using the container

Create a sample class

public class View

{

}

Normally we can create an instance of this class using the “new” keyword.

var view = new View();

Let’s try to new up with Unity. Initially you will need an instance of a UnityContainer.

IUnityContainer container = new UnityContainer();

We can use the “Resolve” method to get an instance of the “View” class.

var view = container.Resolve<View>();

# Constructor Injection

What if the “View” class has a constructor?

public class View

{

public View(ViewModel viewModel)

{

}

}

public class ViewModel

{

}

Well as long as the constructor argument is a concrete implementation the container will resolve it automatically. Isn’t this cool? The code fragment above would still work. OK, but what if the “View” class has more than one constructors?

public class View

{

public View()

{

}

public View(ViewModel viewModel)

{

}

public View(ViewModel viewModel, string name)

{//greedier c'tor

}

}

public class ViewModel

{

}

The container will reflect over all constructors and choose the greedier one. So in the case above the container will fail to resolve a “View” instance since it has no change to satisfy any of the constructor overloads of the “string” class. If we would like to point out to the container which constructor to use then we simply decorate that constructor with the “InjectionConstructor” attribute.

public class View

{

public View()

{

}

[InjectionConstructor]

public View(ViewModel viewModel)

{

}

public View(ViewModel viewModel, string name)

{//greedier c'tor

}

}

public class ViewModel

{

}

# Property Injection

Another way to initialize objects is using properties. Example:

var view = new View { ViewModel = new ViewModel() };

The container can also initialize properties and this is called “Property Injection”. You achieve this by creating a set property of the type you like to inject and decorating this property with the “Dependency” attribute.

public class View

{

[Dependency]

public ViewModel ViewModel { private get; set; }

}

public class ViewModel

{

}

There is no need for any changes on the client code; the initial fragment would still run.

IUnityContainer container = new UnityContainer();

var view = container.Resolve<View>();

Note that property injection occurs right after constructor injection and that the order of properties that are injected in case of multiple properties is for you non deterministic. The container uses reflection to iterate over these properties but a word of warning: don’t count on the order.

# Method Injection

Another form of injection is method injection. This is very useful in case you like to auto-execute a one-time-off method without showing this method to the public interface of the class. The only thing that you need to do is decorate a parameter less method with the “InjectionMethod” attribute. The container will first inject constructors, then properties and finally methods. Once again the order of the execution of injection methods is determined by reflection, so better not count on this.

public class View

{

[InjectionMethod]

public void Init()

{

}

}

Method injection comes very handy when designing extensibility patterns as we will see later on this document.

# Service Location Pattern (Injecting the Container)

Something that you will soon need once you start with Dependency Injection is to inject the container instance in your class.

public class View

{

public IUnityContainer \_Container;

[Dependency]

public IUnityContainer Container

{

set { \_Container = value; }

}

}

Likewise you can inject the container using constructor injection. Note that the container instance is being auto registered to its self. Beware!!! Do not inject the container instance in order to pull from that instance other instances. If you do this then you fall back into a simple service location pattern and you don’t inject dependencies anymore. The following fragment should be avoided.

//bad practice, please avoid

public class View

{

ViewModel1 \_model1;

ViewModel2 \_model2;

public View(IUnityContainer container)

{

\_model1 = container.Resolve<ViewModel1>();

\_model2 = container.Resolve<ViewModel2>();

}

}

public class ViewModel1

{

}

public class ViewModel2

{

}

On the other hand, injecting the container is pretty handy for registering things into the container, creating child containers and so on. We will see more details on these techniques later in this document.

# Registering types

When the type you need to resolve is abstract, or if you would like to resolve an interface then the container doesn’t know how to new up the type. In order to provide this information to the container you need to register (mapp) the abstract type or interface to a concrete implementation.

IUnityContainer container = new UnityContainer();

container.RegisterType<IViewModel, ViewModel>();

var view = container.Resolve<View>();

The rest is the same as before.

public class View

{

public View(IViewModel viewModel)

{

}

}

public class ViewModel : IViewModel

{

}

public interface IViewModel

{

}

# Late Binding

So far we have only covered cases were the types that we were resolving were known. What if the type is not known? This is a very usual case; you are in scope with the interface but not with the implementation. The solution is late binding using reflection of course and the container RegisterType method provides a non generic overload for this.

IUnityContainer container = new UnityContainer();

Type type = Type.GetType("DeadDevsSociety.UnityIntro.ViewModel,UnityIntro");

container.RegisterType(typeof(IViewModel), type);

var view = container.Resolve<View>();

The string you provide to the Type.GetType method is comma delimited; first part is the fully qualified name of the type and the second part is the assembly name.

No question that late binding has a performance penalty but it provides a very powerful way of starting with a component management framework.

# Managing Object Lifetimes

# Registering singleton instances

The container can also be used to register instances. Out of-the-box Unity can register an instance per type (singleton mode) or per type and string (named instance mode).

IUnityContainer container = new UnityContainer();

var viewModel = new ViewModel { Context = new[] { "a", "b" } };

container.RegisterInstance<ViewModel>(viewModel);

var view = container.Resolve<View>();

public class View

{

public View(ViewModel viewModel)

{

Console.WriteLine(viewModel.Context[0]);//should be a

Console.WriteLine(viewModel.Context[1]);//should be b

}

}

public class ViewModel

{

public string[] Context

{

get;

set;

}

}

# Registering named instances

IUnityContainer container = new UnityContainer();

var viewModel1 = new ViewModel { Context = new[] { "a", "b" } };

var viewModel2 = new ViewModel { Context = new[] { "c", "d" } };

container.RegisterInstance<ViewModel>("one", viewModel1);

container.RegisterInstance<ViewModel>("two", viewModel2);

var view = container.Resolve<View>();

public class View

{

public View([Dependency("one")]ViewModel viewModel)

{

Console.WriteLine(viewModel.Context[0]);//should be a

Console.WriteLine(viewModel.Context[1]);//should be b

}

public ViewModel \_ViewModel;

[Dependency("two")]

public ViewModel ViewModel

{

set

{

Console.WriteLine(value.Context[0]);//should be c

Console.WriteLine(value.Context[1]);//should be d

\_ViewModel = value;

}

}

}

public class ViewModel

{

public string[] Context

{

get;

set;

}

}

# Singleton Pattern

Along with object lifetimes come singleton implementations. Let’s see how a hardcoded singleton looks like.

public class ViewModel : IViewModel

{

private readonly IViewModel \_Instance = new ViewModel();

private ViewModel()

{

}

public static IViewModel Instance

{

get

{

return \_Instance;

}

}

}

public interface IViewModel

{

}

There are several things that I don’t like about the code fragment above:

1. You need to “pollute” your object with a cross-cutting concern such as the singleton implementation. What if a requirement comes up that for some reason the ViewModel should not be a singleton?
2. As client you need to access the instance statically via a property: ViewModel.Instance
3. Once you hard code singletons then it becomes a necessity to further hard code other cross-cutting concerns such as late binding implementations.
4. The traditional singleton implementation is not test friendly.

We can move the singleton implementation concern to the container. The “new ContainerControlledLifetimeManager()” registers the instance as singleton.

IUnityContainer container = new UnityContainer();

container.RegisterType<IViewModel, ViewModel>(new ContainerControlledLifetimeManager());

var view = container.Resolve<View>();

//no need to implement anything for the singleton

public class View

{

public View(IViewModel viewModel)

{

}

}

public class ViewModel : IViewModel

{

}

public interface IViewModel

{

}

# Component Management

A typical system has many dlls with many types that need to be registered by the container. As we have seen before late binding gives an easy good way of registering types into the container using reflection. However we should not forget that overusing reflection comes with a high price:

* Not easy to maintain all these strings, with correct fully qualified type names and correct assembly names.
* No type safety at compile time; all mistakes you’ll experience as runtime errors.
* The performance hit using reflection for each type is too expensive.

An easy way to overcome this is by creating classes that inject the container and then execute via method injection a registration method. Inside this method you “early” register your types in the container. During start-up “bootstrapping” you simply have to “late” resolve these classes.

IUnityContainer container = new UnityContainer();

var type = Type.GetType("DeadDevsSociety.UnityIntro.ContainerConfigurator,UnityIntro");

container.Resolve(type);

container.Resolve<View>();

public class ContainerConfigurator

{

[Dependency]

public IUnityContainer Container { private get; set; }

[InjectionMethod]

public void Register()

{

Container.RegisterType<IViewModel, ViewModel>();

}

}

public class View

{

public View(IViewModel viewModel)

{

}

}

public class ViewModel : IViewModel

{

}

public interface IViewModel

{

}

You can invent different ways of resolving the ContainerConfigurator class were all the “early” registrations are done. One nice technique is to use ConfigurationSections.

So our container configurator derives from the ConfigurationSection and implements an interface that has a configure metjod. Inside this Configure method we do all the registrations.

public interface IContainerConfigurator

{

void Configure(IUnityContainer container);

}

public class ContainerConfigurator : ConfigurationSection, IContainerConfigurator

{

public void Configure(IUnityContainer container)

{

container.RegisterType<IViewModel, ViewModel>();

}

}

We create an instance of the Container configurator by calling the GetSection method of the ConfigurationManager. A reference to the System.Configuration.Dll is required.

IUnityContainer container = new UnityContainer();

IContainerConfigurator section =

(IContainerConfigurator)ConfigurationManager.GetSection("ContainerConfigurator");

section.Configure(container);

container.Resolve<View>();

The section needs to be registered of course inside the App.Config file

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

<configuration>

<configSections>

<section name="ContainerConfigurator"

type="DeadDevsSociety.UnityIntro.ContainerConfigurator,UnityIntro"/>

</configSections>

</configuration>

The disadvantage of this technique is that you need to have a reference to the System.Configuration.Dll on all projects that have a “ContainerConfigurator”. In addition you need to have the “IContainerConfigurator” interface in a common assembly visible to all other assemblies. The advantage over this technique is that you can now inject further configuration elements into the container since the “ConfigurationSection” is already there.

public interface IContainerConfigurator

{

void Configure(IUnityContainer container);

}

public class ContainerConfigurator : ConfigurationSection, IContainerConfigurator

{

public void Configure(IUnityContainer container)

{

container.RegisterType<IViewModel, ViewModel>();

}

[ConfigurationProperty("connString")]

public string ConnString

{

get { return (string)this["connString"]; }

}

}

Likewise the App.Config file

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

<configuration>

<configSections>

<section name="ContainerConfigurator"

type="DeadDevsSociety.UnityIntro.ContainerConfigurator,UnityIntro"/>

</configSections>

<ContainerConfigurator connString="someconnstring" />

</configuration>

# Working with “Lambdas”