

An Introduction to Dependency Inversion

Originally posted by AdrianK, 1-Nov-2010 at: [http://www.morphological.geek.nz/blogs/viewpost/Peruse Muse Infuse/An Introduction to Dependency Inversion.aspx](http://www.morphological.geek.nz/blogs/viewpost/Peruse+Muse+Infuse/An+Introduction+to+Dependency+Inversion.aspx)

Tagged under: Architecture, Web Development, Patterns

An Introduction to Dependency Inversion

This article is for anyone (mainly developers, and specifically .Net developers) who is unfamiliar with Dependency Inversion or design patterns in general.

The basic rationale behind the [Dependency Inversion Principle](#)^[1] (DIP, or simply DI) is very simple; and like most other simple design principles it is suitable in many different situations. Once you understand it you'll wonder how you managed without it.

The key benefit it brings is control over dependencies – by removing them (or perhaps more correctly – by “inverting” them), and this isn't just limited to “code” – you'll find this also opens up new opportunities regarding how you can structure and run the actual development process.

Implementing DI and is fairly straight forward and will often involve other design patterns, such as the [Factory pattern](#)^[2] and [Lazy Load pattern](#)^[3]; and embodies several other principles including the Stable Abstractions Principle and Interface Segregation Principle.

I'll be using acronyms where I can, here's a quick glossary for them.

- BL: Business Logic, sometimes referred to as the Business Logic Layer.
- DA: Data Access, often referred to as the DAL or Data Access Layer.
- DI: can refer to either Dependency Inversion or Dependency Injection – both of which (at least for this article) essentially mean the same thing. You may also hear of it referred to as “Inversion of Control”.
- ISP: [Interface Segregation Principle](#)^[4].
- SAP: [Stable Abstractions Principle](#)^[5].
- SDP: [Stable Dependencies Principle](#)^[6].

I've also put together a working reference solution [available here](#)^[7] (zip file, 77Kb).

Getting Started – What to Avoid

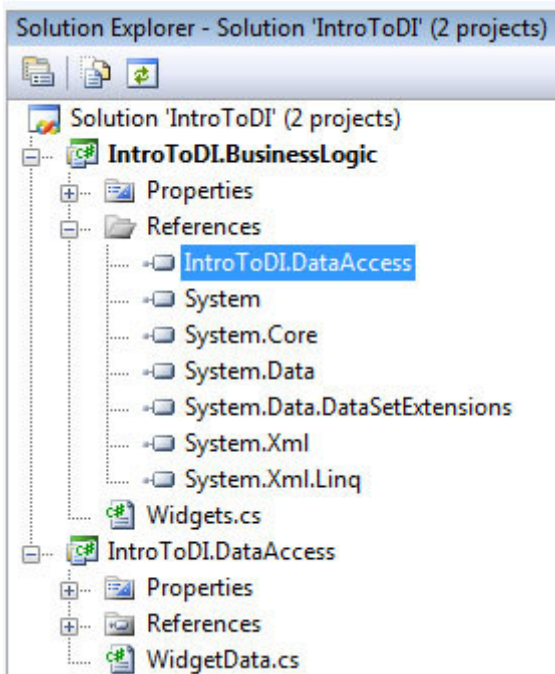
Typically, if you're not using DI you'll be directly referencing your other projects, which means they'll be dependencies; in other words they are tightly coupled and without them your code won't work (or perhaps even compile).

Perhaps the most common example is when you have some Business Logic (BL) which directly calls the underlying Data Access (DA). For example, you might have a method (in a class) in your business logic that looks like this:

```
using IntroToDI.DataAccess;
```

```
namespace IntroToDI.BusinessLogic
{
    public class Widgets
    {
        public static string GetAllWidgets()
        {
            // Calling IntroToDI.DataAccess.WidgetData.GetAll() directly:
            return WidgetData.GetAll();
        }
    }
}
```

In this example I've been good and separated my BL and DA into separate projects (which will translate into separate physical (deployable) assemblies; so currently my project browser looks like this:



Why is this bad?

No doubt our DA code will be talking to SQL Server, and obviously because our DA is built for SQL Server it won't run unless it has a SQL Server database to work with. As I mentioned above, our BL won't run unless it has an instance of the DA to work with – and so we have a dependency tree where our BL is effectively tied to SQL Server.

So what are we going to do when we need to upgrade to SQL Server 2011? What are we going to do if there's a technology change (or any kind of change) and we need to integrate with a different database platform altogether?

What we are going to do is panic. The bigger our application is the bigger our refactoring task is going to be – actually it won't just be refactoring it'll be a major re-write. And everyone knows how unpopular re-writing major applications are with project managers and business stakeholders.

Instead, we'll use DI to essentially remove the dependency tree and save our skins.

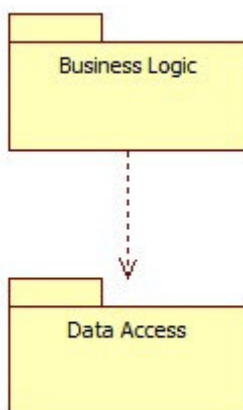
Dependency Inversion 101

The [Dependency Inversion Principle](#)^[8] states that:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

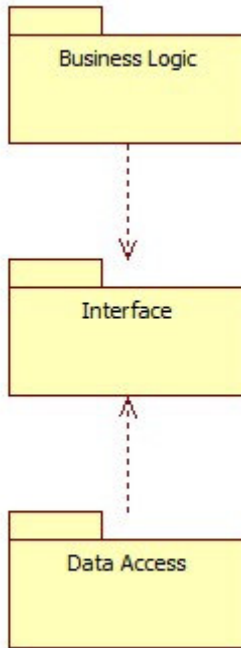
The second point's the one to remember; the idea is that it's infinitely better to tie yourself to something that isn't going to change on you when you least want it to. We'll look at this more closely when we discuss the Stable Abstractions and Stable Dependencies Principles.

If you were to represent the previous (bad) example using UML you'd end up with something like this:



Here we have a package which represents the Business Logic project / assembly, and another for the Data Access. The line connecting them is a Dependency connector; it's saying that the Business Logic package depends on the Data Access package.

In comparison, DI looks like this:



We now have a new package called Interface, and both the BL and DA depend on that instead of each other.

The Principles of Being Stable: Stable Abstractions and Stable Dependencies

So what are the Stable Abstractions and Stable Dependencies Principles (SAP & SDP)? To quote [Ward Cunningham](#)^[9], SDP is:

"The dependencies between packages should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable than it is. (<http://www.objectmentor.com/resources/articles/stability.pdf>^[10])"

Where as SAP is:

Packages that are maximally stable should be maximally abstract. Unstable packages should be concrete. The abstractness of a package should be in proportion to its stability.

(<http://www.objectmentor.com/resources/articles/stability.pdf>^[11])

The concept is that if we're going to depend on something it needs to be as stable as possible. 'Real' things (like database platforms) do change; so rather than depend on something 'real' it's much better to depend on something abstract; also, an interface in itself (with no implementation) has little reason to change - if it does it's likely that you'd be wanting to change your implementations anyway.

When we say "abstract" we specifically mean an interface or some other Object Orientated construct that we have defined and have control over, and this is right at the heart of DI.

In the UML diagram above, the Interfaces package is where these abstractions go.

So what does an Interface look like in C#?

For starters, in this context we're not talking about a "User Interface". To create an interface in .Net you use the (wait for it) "[interface](#)^[12]" keyword .

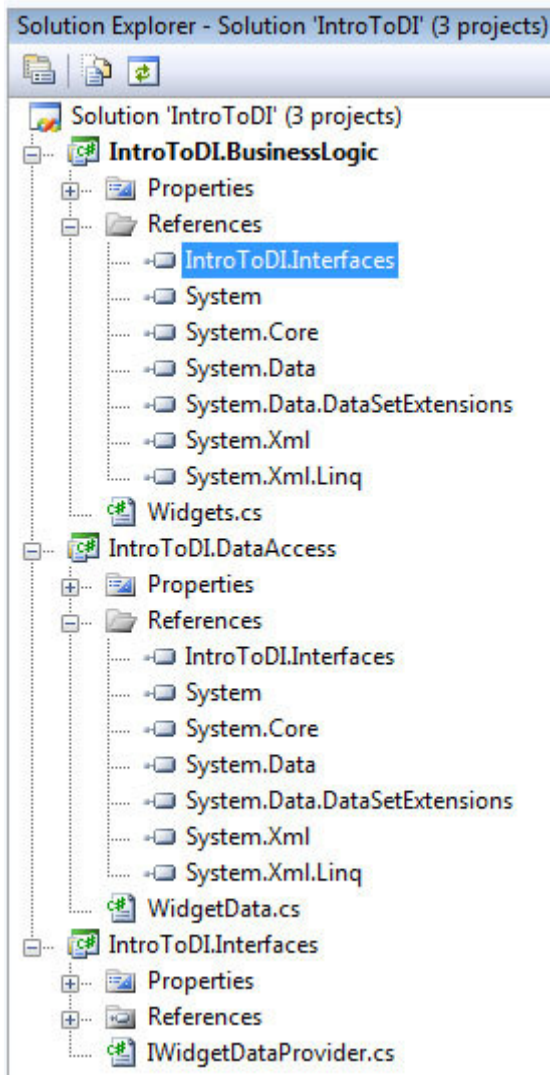
Here's possibly the simplest interface you'll ever see, it's also the interface we'll use to separate the BL and DA in our Widget example.

```
namespace IntroToDI.Interfaces
{
    public interface IWidgetDataProvider
    {
        string GetAll();
    }
}
```

Like most things, interfaces have a lot of depth to them (which I'm not going to cover here), so check out the [Interfaces \(C# Programming Guide\)](#)^[13] on MSDN for more information; but while we're here, here are a few basics to get you started.

- Interfaces define a "contract" (which .Net enforces).
- An interface can only contain the signatures of methods, properties, events or indexers.
- The implementation of the members is done in the class or struct that implements the interface.
- Scope is not defined on interface members, and the "contract" is only satisfied if the implementation implements all the member signatures defined – and they are declared as public.
- Interface members (including implementation) cannot be static.

The first step in correcting the Widget example is to remove the BL reference to the DA; then, have both the BL and DA reference the interface project which we've called IWidgetDataProvider.



Then we need to remove the using statement in the BL Widgets class which references the DA WidgetData class.

You might also be wondering why I've put the interfaces in a separate project (and assembly)? It comes back to SDP. The important part is that it'll end up in a separate assembly (and most of the time "project == assembly" in terms of how code is organised and deployed). By keeping the interfaces in a separate assembly no-one who depends (references) on them needs to worry about any other dependencies (the assumption is that the interfaces assembly is totally devoid of as many references as possible. So as it's less likely to change it's therefore safer to depend on.

So far I've been deliberately vague and ambiguous with how I've named things in the Widget example, but with the IWidgetDataProvider I've started to improve this.

- The popular convention is to prefix your interfaces with the letter "I" – just as Microsoft themselves do (e.g.: the IDisposable interface).
- I have Widget in the name – it's a good idea to have multiple interfaces defined, and defining them by business object isn't a bad place to start. See the Interface Segregation Principle (ISP) for more information.
- I've used "DataProvider" because this interface is all about providing data access.

Generally speaking, the term I use when referring to interface implementations is "provider", because the it "provides" the implementation the interface demands.

Remember, these interfaces are our abstraction layer – so don't pollute them with implementation detail like "IWidgetSQLServerDataProvider".

Restoring Compilation

Great – we now have a solution which is [loosely coupled](#)^[14] but doesn't work; in fact it doesn't even compile – because in our BL Widgets class we're still trying to call `WidgetData.GetAll()`.

The next task is to remove calls against specific implementations and instead call our abstractions – specifically the interface. The following code will do that; our system still won't work but it will compile:

```
using IntroToDI.Interfaces;

namespace IntroToDI.BusinessLogic
{
    public class Widgets
    {
        private static IWidgetDataProvider _IWidgetDataProvider;

        public static string GetAllWidgets()
        {
            return _IWidgetDataProvider.GetAll();
        }
    }
}
```

The key line here is the last significant one:

```
return _IWidgetDataProvider.GetAll();
```

Even though we don't have a working system we do have one that will at least compile, and at this point it's worth mentioning a really valuable benefit.

Having defined the interface I can now develop the BL without the DA project, in fact I don't need any physical data access implementation because I'm only talking to the interface. The only time an actual implementation is needed is during runtime.

Likewise someone else can develop the data access code without having the BL project present. For them to do this they'll first need to ensure the data access class inherits the appropriate interface (in this case `IWidgetDataProvider`).

True, it's not practical to do any real development without something to call and run the DA, and without having any DA implementation for the BL to call – but this is a blessing and is easy to get around.

What this means is that:

- We can split the project up and have different team members working on the different parts in relative isolation – as long as the interface doesn't change (and even then that's a process we can manage).
- The BL developers don't have to wait for the DA developers to finish writing their code and inserting data.

- The DA developers don't need to wait for the BL (and maybe UI) developers to write their code in order to call and test the DA components.

The Factory Pattern

This is where things really start to come together. A crucial part of any DI implementation is where contract providers are identified, instantiated and returned for use.

The purpose of the Factory pattern is to create "objects" (in the wider sense of the word), often applying rules in the process. Using a Factory allows you to encapsulate (sometimes complex) code into a single place for re-use, making the rest of your code more elegant and simpler to maintain.

For the DI implementation I'm going to present to you, we need to do two key things:

- Build a way by which we can specify which implementation is to be used.
- Create a mechanism that will instantiate the implementation.

Our Factory will perform the second task, but how we design our Factory will be driven by how we want to specify implementations; this in turn is driven by our overall design and architectural drivers.

In the context of the Widget example we can comfortably use configuration to indentify the implementation(s) to be used at runtime; at this stage we will be using DI to load a data access provider, and as this isn't something that we'll need to change very often configuration will be fine.

Instantiating Providers

I tend to call the Factory which instantiates providers the "ProviderLoader". We'll need to dynamically instantiate the providers at runtime - because the rest of the system isn't referencing any of them. Reflection is the way we get around this, specifically a neat bit of magic called `Activator.CreateInstance()`.

As stated by Microsoft:

- `Activator` contains methods to create types of objects locally or remotely, or obtain references to existing remote objects. This class cannot be inherited.
- `CreateInstance(Type)` creates an instance of the specified type using that type's default constructor.

`Activator` has several methods that can be used to create instances, and `CreateInstance` has several overloads. `CreateInstance(Type)` simply requires you to pass in the type you'd like to instantiate.

Getting the type is easily accomplished by calling `Type.GetType(fullTypeName fullTypeName)`, where "fullTypeName" is a string containing the name of the type we want to create. There's a specific convention that needs to be followed: "[type name], [assembly name]".

To do this in code (specifically for the `WidgetData` class):

```
Type t = Type.GetType("IntroToDI.DataAccess.WidgetData,IntroToDI.DataAccess",
true);
return Activator.CreateInstance(t);
```

Please note:

- When specifying the name of the type you need to include its full namespace.
- The code shown here would normally reside in a Factory method (hence the use of "return").
- Obviously we'd normally pass in a variable rather than hard-coding "...WidgetData..." into the GetType() call. Instantiating objects via Activator.CreateInstance isn't without its traps: try to instantiate something when one of its dependants is missing and your system will explode spectacularly.

As a minimum, I would implement a Provider Loader Factory like this:

```
public static object CreateInstance(string fullTypeName)
{
    Type t;
    try
    {
        t = Type.GetType(fullTypeName, true);
    }
    catch (System.Exception ex)
    {
        throw new ProviderLoaderException(string.Format("t = Type.GetType
(fullTypeName, true) failed. fullTypeName = {0}", fullTypeName), ex);
    }

    try
    {
        return Activator.CreateInstance(t);
    }
    catch (System.Exception ex)
    {
        throw new ProviderLoaderException(string.Format("Activator.CreateInstance(t)
failed. fullTypeName = {0}", fullTypeName), ex);
    }
}
```

This specifically includes:

- Calling GetType() and Activator.CreateInstance() separately. Sure you can do it in one line but maintenance and debugging gets messy.
- Wrap each in a specific try / catch block, preferably throwing a custom exception (I use one called "ProviderLoaderException").
- When you throw an exception ensure you include the full name of the type you were hoping to create. Another trap to watch out for is (to use the Widget example) when both you BL and DA are compiled against different versions of the same common dependency.

Specifying Providers

As mentioned previously, using configuration is often appropriate (via AppSettings); the trick here is to effectively manage the AppSetting keys and Provider identifiers. I suggest following these simple steps:

- Devise a naming convention that is easy to understand; I strongly recommend using a URI based approach – you don't have to use the namespaces in your application (although it might make sense if your system is large or complex).
- Define your AppSetting keys as constants in a common part of your system (and use them).

How it all hangs together is as follows; first the configuration:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="IntroToDI.IWidgetDataProvider"
value="IntroToDI.DataAccess.WidgetData,IntroToDI.DataAccess"/>
  </appSettings>
</configuration>
```

Then, define the appSetting key in your code as a constant.

```
namespace IntroToDI.Common.Constants
{
  public class ProviderKeys
  {
    public static string WidgetDataProvider = "IntroToDI.IWidgetDataProvider";
  }
}
```

Finally, use the constant when requesting an AppSetting from the configuration. In this example I'm passing that constant into my provider Factory.

```
iWidgetDataProvider = ProviderLoader.CreateInstance
(ProviderKeys.WidgetDataProvider) as IWidgetDataProvider;
```

Here we can see the provider factory itself, including a call to the ConfigurationManager to actually obtain the configuration value (which is then passed into GetType()).

```
using System;

namespace IntroToDI.Common
{
  public class ProviderLoader
  {
    public static object CreateInstance(string fullTypeNameConfigKey)
    {
```

```

        string fullTypeName =
System.Configuration.ConfigurationManager.AppSettings[fullTypeNameConfigKey];

        Type t = Type.GetType(fullTypeName, true);
        return Activator.CreateInstance(t);
    }
}
}

```

Please note that this code is “cut-down” to the bare minimum needed to demonstrate DI, you’d want to include appropriate validation and error handling for a production grade system.

Consuming a Provider

This is the final piece of the puzzle. The Factory method used in the Widgets example (ProviderLoader.CreateInstance) returns an instance of an object – not a specific provider, so before using it we need to cast it to the interface:

```

myWidgetDataProvider = ProviderLoader.CreateInstance
(ProviderKeys.WidgetDataProvider) as IWidgetDataProvider;

```

In practice, I recommend using a “Lazy Load” Pattern when calling the Factory method, for example you might access your provider via an internal property, as shown below:

```

private static IWidgetDataProvider iWidgetDataProvider = null;
internal static IWidgetDataProvider IWidgetDataProvider
{
    get
    {
        if (iWidgetDataProvider == null)
        {
            iWidgetDataProvider = ProviderLoader.CreateInstance
(ProviderKeys.WidgetDataProvider) as IWidgetDataProvider;
        }

        return iWidgetDataProvider;
    }
}

```

The IWidgetDataProvider property is a member of the BL Widgets class – whenever the Widgets class needs to access the IWidgetDataProvider it does so via this property. When called the property will return its IWidgetDataProvider instance – and only create it if it does not already exist.

Apart from personal discipline there’s nothing stopping other BL Widgets code from calling the privately declared variable iWidgetDataProvider, but this would be unwise as the calling code would first have to check its existence.

Quick Recap and Bonus Points

- Apply the Dependency Inversion Principle by abstracting out dependencies and defining them as interfaces.
- Use a Factory to instantiate the providers.

- Use Lazy Load to only create new instances of your providers only when you need them.

This will get you started on the road to Dependency Inversion nirvana, but there are still a few other specifics worth pointing out:

- If done properly you BL will run – but fail, because by default there won't be an implementation of the DA in the bin directory to be executed – so you'll have to copy it there yourself (preferably automagically).
- Be mindful where you define the interfaces; putting them in a separate assembly is the safest approach.
- You can have as many implementations in the bin as you like – only the specified one will be used.
- BL developers can use DI to implement a mock provider for testing or development purposes, particularly if the “real” provider is not available.
- Assuming your provider implementation is compiled against the same version of the interfaces as your consumer (i.e. the BL) you can deploy new versions of your providers at anytime – no recompilation of the host system (BL, UI, etc) is required.
- Using DI you can provide a default provider with your system and allow users of the system to make their own (particularly in Open Source scenarios).

In this article I've outlined how to “do” Dependency Inversion from the ground-up, which is a worthwhile exercise if you're not familiar with how DI works. There are also plenty of DI frameworks available that you may wish to check out too, and many mainstream frameworks include “sub-frameworks” that deal with DI. Scott Hanselman has written up a [List of .NET Dependency Injection Containers \(IOC\)](#)^[15] which is a good place to start if you're interested in looking into this further.

References

1. http://en.wikipedia.org/wiki/Dependency_inversion_principle
2. http://en.wikipedia.org/wiki/Factory_method_pattern
3. http://en.wikipedia.org/wiki/Lazy_loading
4. http://en.wikipedia.org/wiki/Interface_segregation_principle
5. <http://c2.com/cgi/wiki?StableAbstractionsPrinciple>
6. <http://c2.com/cgi/wiki?StableDependenciesPrinciple>
7. <http://localhost/Morpholia/FileListings/IntroToDI%20Source%20Code.zip>
8. http://en.wikipedia.org/wiki/Dependency_inversion_principle
9. http://en.wikipedia.org/wiki/Ward_Cunningham
10. <http://www.objectmentor.com/resources/articles/stability.pdf>
11. <http://www.objectmentor.com/resources/articles/stability.pdf>
12. <http://msdn.microsoft.com/en-us/library/87d83y5b.aspx>
13. <http://msdn.microsoft.com/en-us/library/ms173156.aspx>
14. http://en.wikipedia.org/wiki/Loose_coupling
15. <http://www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx>



Some rights reserved.

<http://creativecommons.org/licenses/by-nc-sa/3.0>