.net

(https://www.c-sharpcorner.com/UploadFile/4d9083/dependency-injection-using-microsoft-unity-framework/)

Dependency Injection Using Microsoft Unity Framework

This article explains Dependency Injection using Microsoft Unity.

-
- Saineshwar Bageri
- Feb 01 2015

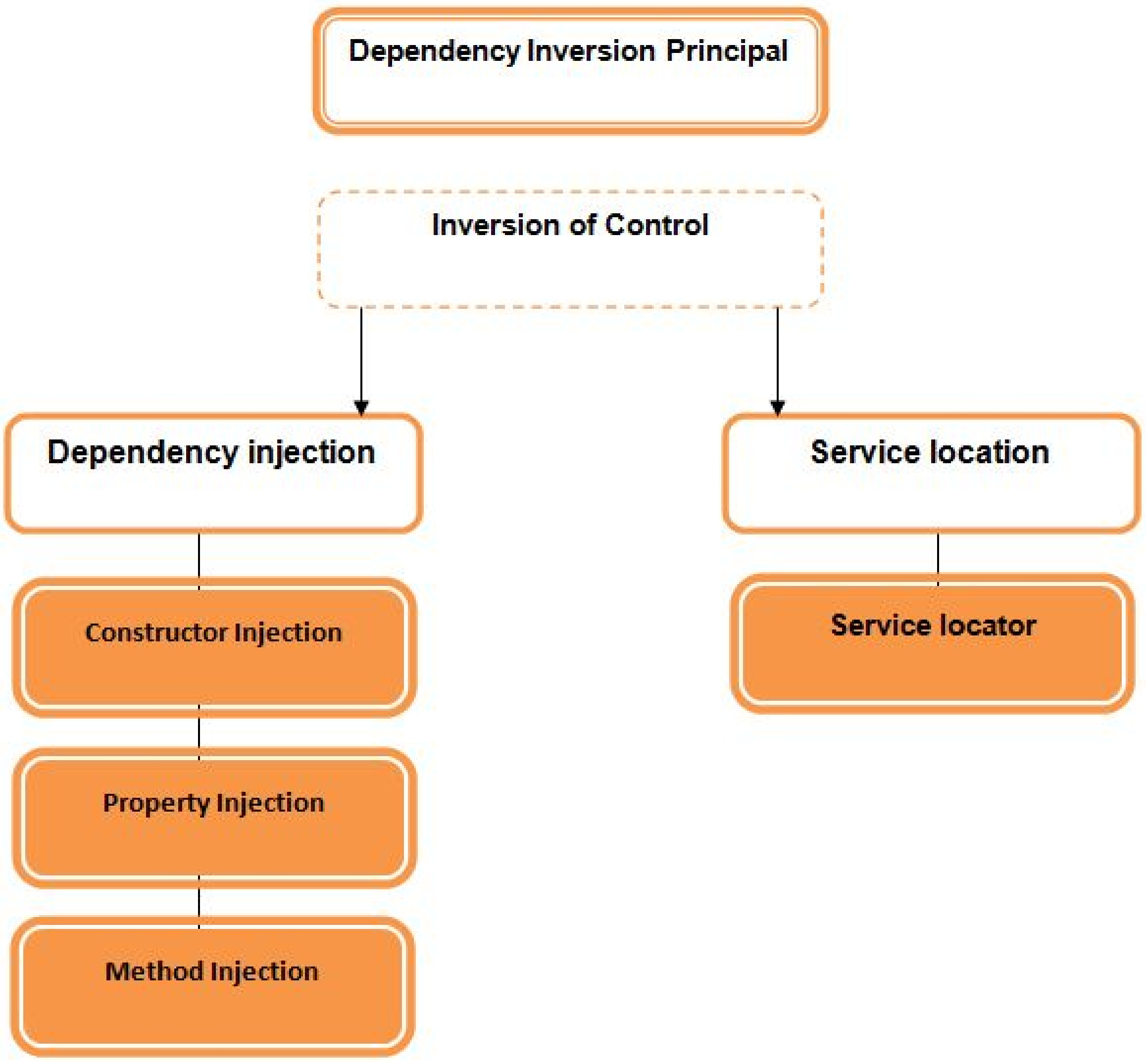
DIinject.zip

Let's have look at where it starts. It begins with the SOLID principle last Letter “D”, the Dependency Inversion Principal.

Agenda

- Dependency Inversion Principal.
- Inversion of Control.
- Dependency Injection.
- Creating a console application.
- Adding Reference to Microsoft Unity Framework.
- Adding BL class.
- Adding DL class.
- Adding Interface.
- How to configure Unity Container.
- Running and debugging an application.
- Final output.
- Dependency Injection Pros and Cons.

Flow of Dependency Inversion Principal



Dependency Inversion Principal

The Dependency Inversion principal says that components that depend on each other should interact via an abstraction, not directly with a concrete implementation.

Example: If we have a data access layer and business layer then they should not directly depend on each other, they should depend on an interface or abstract for object creation.

Advantages:

1. Using abstraction allows various components to be developed and changed independently of each other.
2. And easy for testing components.

Inversion of Control (IOC)

Inversion of Control is a design principal that promotes loosely coupled layers, components, and classes by inverting the control flow of the application.

Dependency Injection (DI)

Dependency Injection is defined as a design pattern that allows removing hard-coded dependencies from an application.

There is one major point to remember:

“Inversion of control is principal and Dependency Injection is implementation”.

Now let's start with implementing Dependency Injection using the Microsoft Unity Framework.

The Microsoft Unity Framework helps us to inject external dependencies into software components. To use it in a project we just need to add a reference for the Unity Container DLLs to our project. To download it click on this link.

There are the following three types of dependencies:

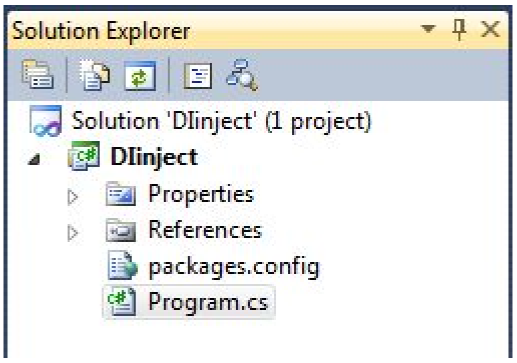
1. Constructor
2. Setter (Properties)
3. Method.

Creating a console application

I am creating a Console application named **DIinject**.

And I am showing Constructor Injection with Unity Framework (that is mostly widely used).

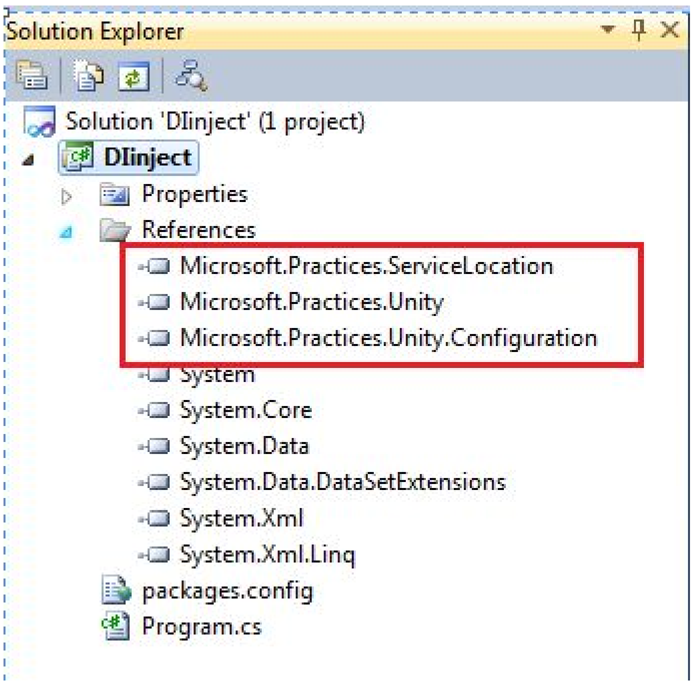
See in the following snapshot.



Adding Reference to Microsoft Unity Framework

Now I had added a reference for the Microsoft Unity Framework to my application.

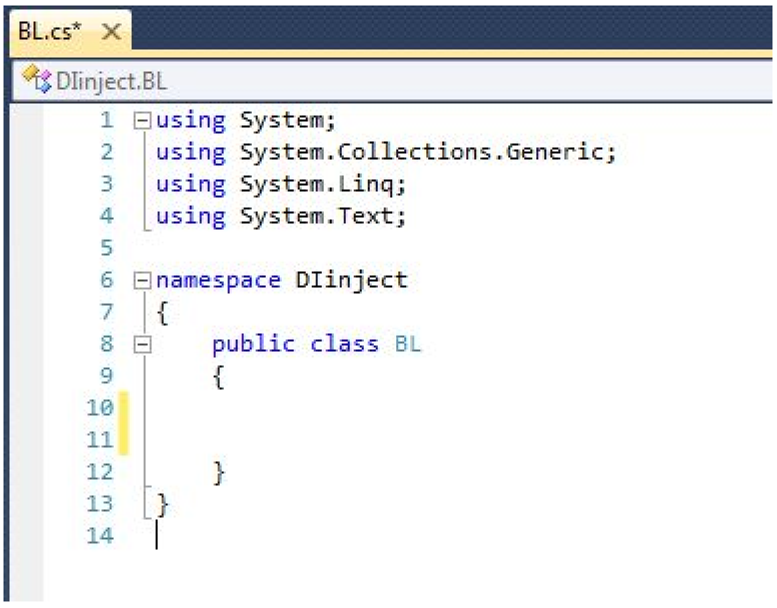
See in the following snapshot.


Solution Explorer
Solution 'DIinject' (1 project)
DIinject
Properties
References
Microsoft.Practices.ServiceLocation
Microsoft.Practices.Unity
Microsoft.Practices.Unity.Configuration
System
System.Core
System.Data
System.Data.DataSetExtensions
System.Xml
System.Xml.Linq
packages.config
Program.cs

Adding BL class

I have added **BL.cs** (a Business Logic Layer) class to the application.

See in the following snapshot.

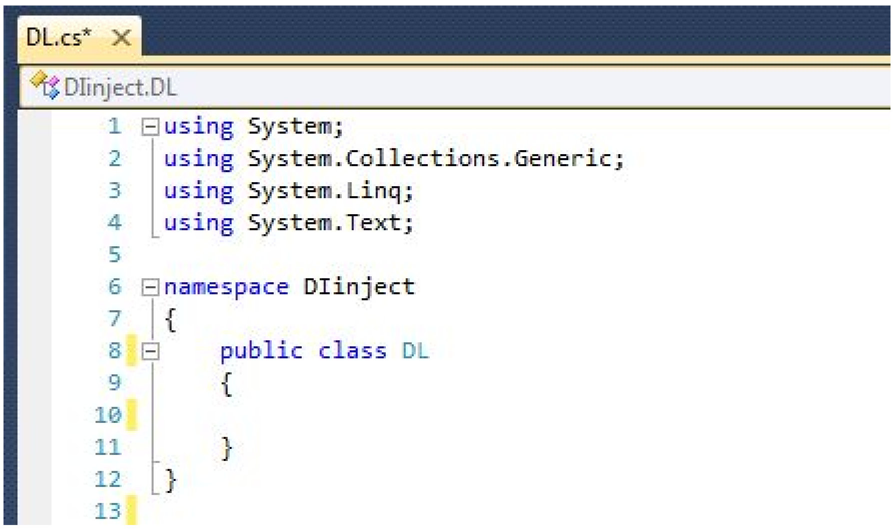


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DIinject
{
    public class BL
    {


    }
}
```

Adding DL class

After adding **BL.cs** I have added **DL class** to the application (Data Access Layer).

See in the following snapshot.



DL.cs*

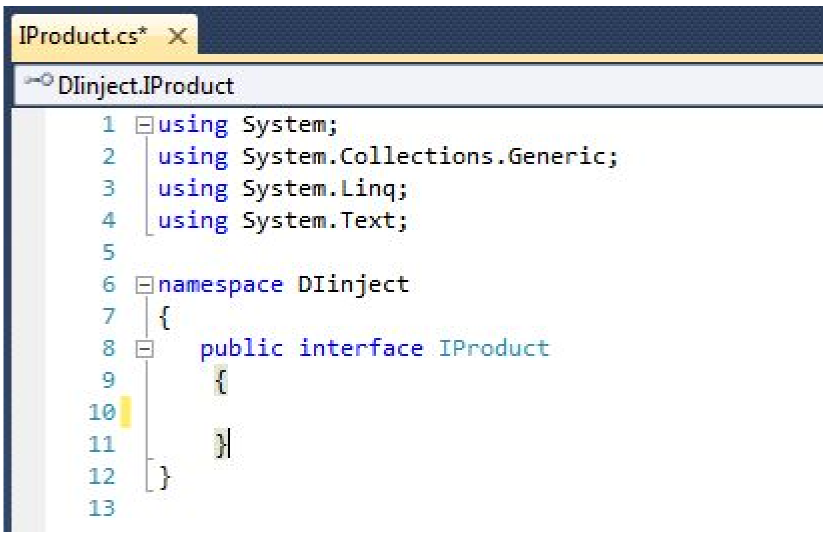
DInject.DL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DInject
{
    public class DL
    {

    }
}
```

Adding Interface

For decoupling both classes, I have added an interface with the name **IProduct**.

See in the following snapshot.



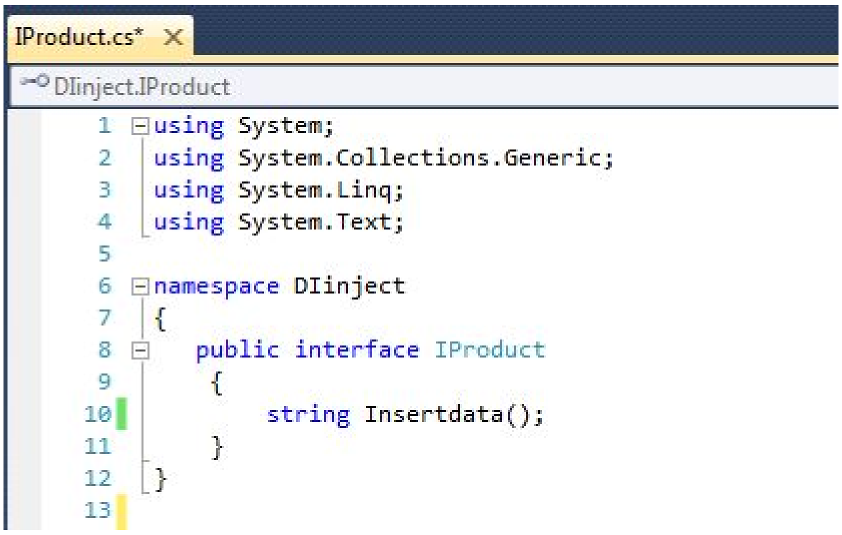
IProduct.cs*

DInject.IProduct

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DInject
{
    public interface IProduct
    {

    }
}
```

In the interface, I have declared a method with the name **Insertdata()**.

See in the following snapshot.



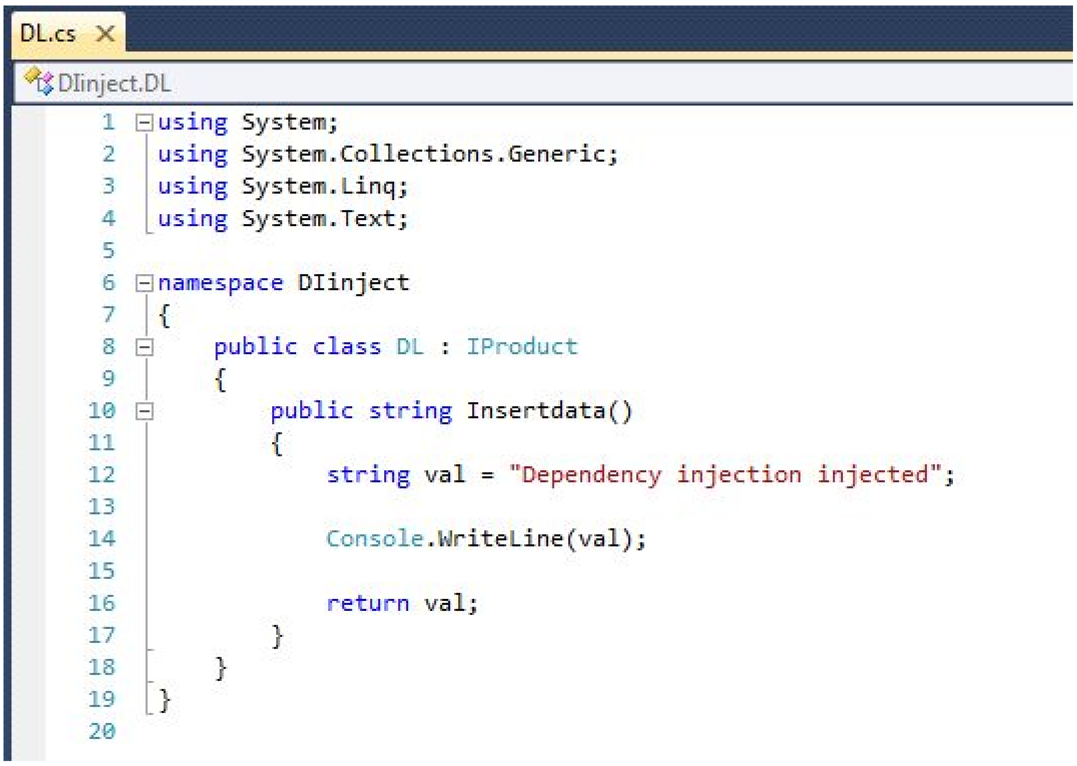
IProduct.cs*

DInject.IProduct

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DInject
{
    public interface IProduct
    {
        string Insertdata();
    }
}
```

Now the **DL** (Data access layer) will implement the interface **IProduct**.

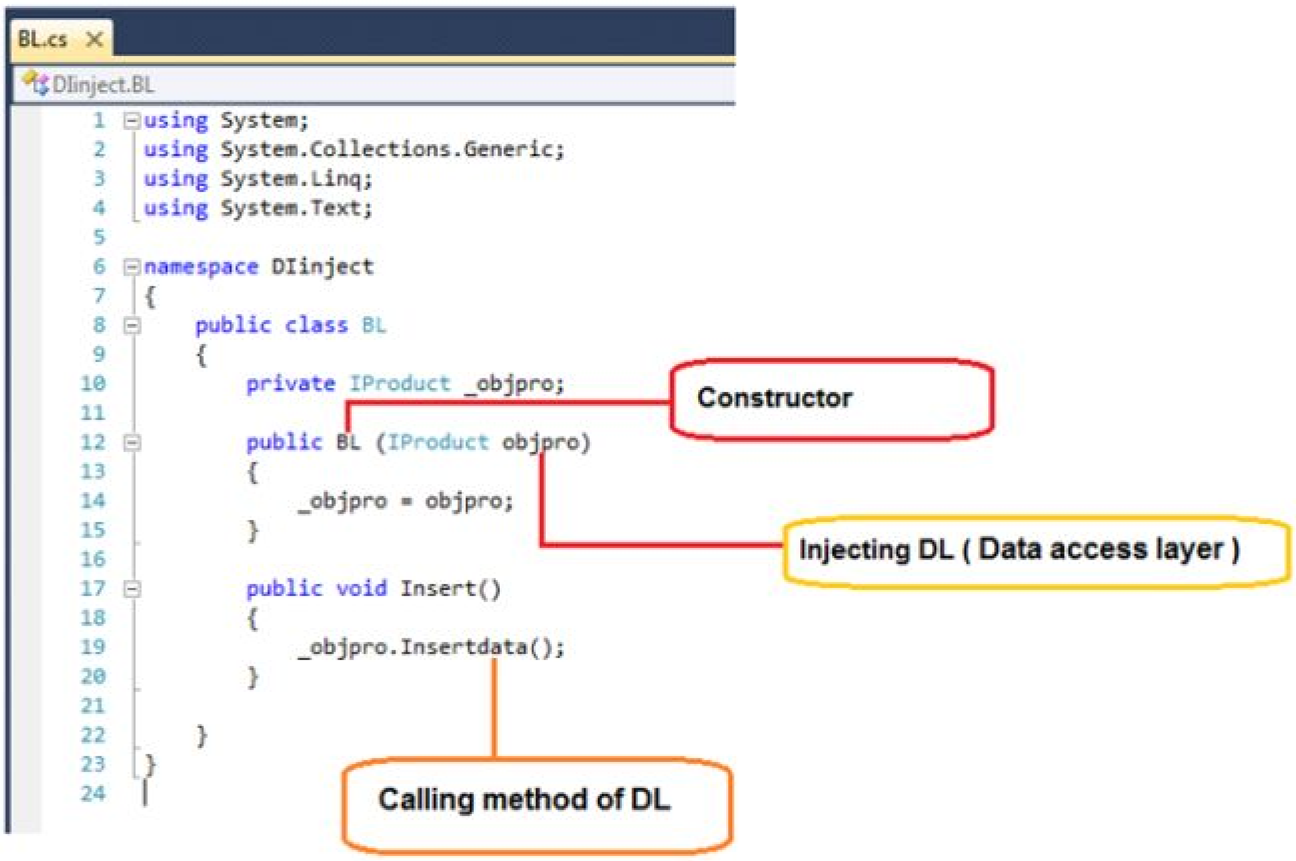
See it in the following snapshot.


DL.cs
DIinject.DL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DIinject
{
    public class DL : IProduct
    {
        public string Insertdata()
        {
            string val = "Dependency injection injected";

            Console.WriteLine(val);

            return val;
        }
    }
}
```

After implementing the interface, now in the BL (Business Logic) layer I am injecting a dependency from the Constructor.

See in the following snapshot.


BL.cs
DIinject.BL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DIinject
{
    public class BL
    {
        private IProduct _objpro;

        public BL (IProduct objpro)
        {
            _objpro = objpro;
        }

        public void Insert()
        {
            _objpro.Insertdata();
        }

    }
}
```

How to configure Unity Container

In the Program.cs add the following code to register the dependency and resolve the BL (Business Logic) layer instance as shown below. This will automatically take care of injecting the DA (Data Access) layer object into the BL (Business Logic) layer Constructor.

For that, we need to register a type and resolve an instance of the default requested type from the container.

1. Creating a Unity Container.

   ```
   /* Creating microsoft unity Container*/

   UnityContainer IU = new UnityContainer();
   ```

   Here I have created a Unity Container object with a named IU.

2. Register a type.

   ```
   /* Register a type*/

     IU.RegisterType<BL>();

     IU.RegisterType<DL>();
   ```

   Here I registered types that I am using for the injection.

   In this, I am injecting a DA (Data Access) layer into the BL (Business Logic) layer. That is why I registered both types.
3. Register a type with specific members to be injected.

   ```
   /* Register a type with specific members to be injected. */

   IU.RegisterType<IProduct, DL>();
   ```

   1. RegisterType<From , To>( );

   From: Type that will be requested.

   To: Type that will actually be returned.

   You call the **RegisterType** method to specify the registered type as an interface or object type and the target type you want to be returned in response to a query for that type. The target type must implement the interface, or inherit from the class, that you specify as the registered type.

4. Resolve. (Resolve an instance of the default requested type from the container.)

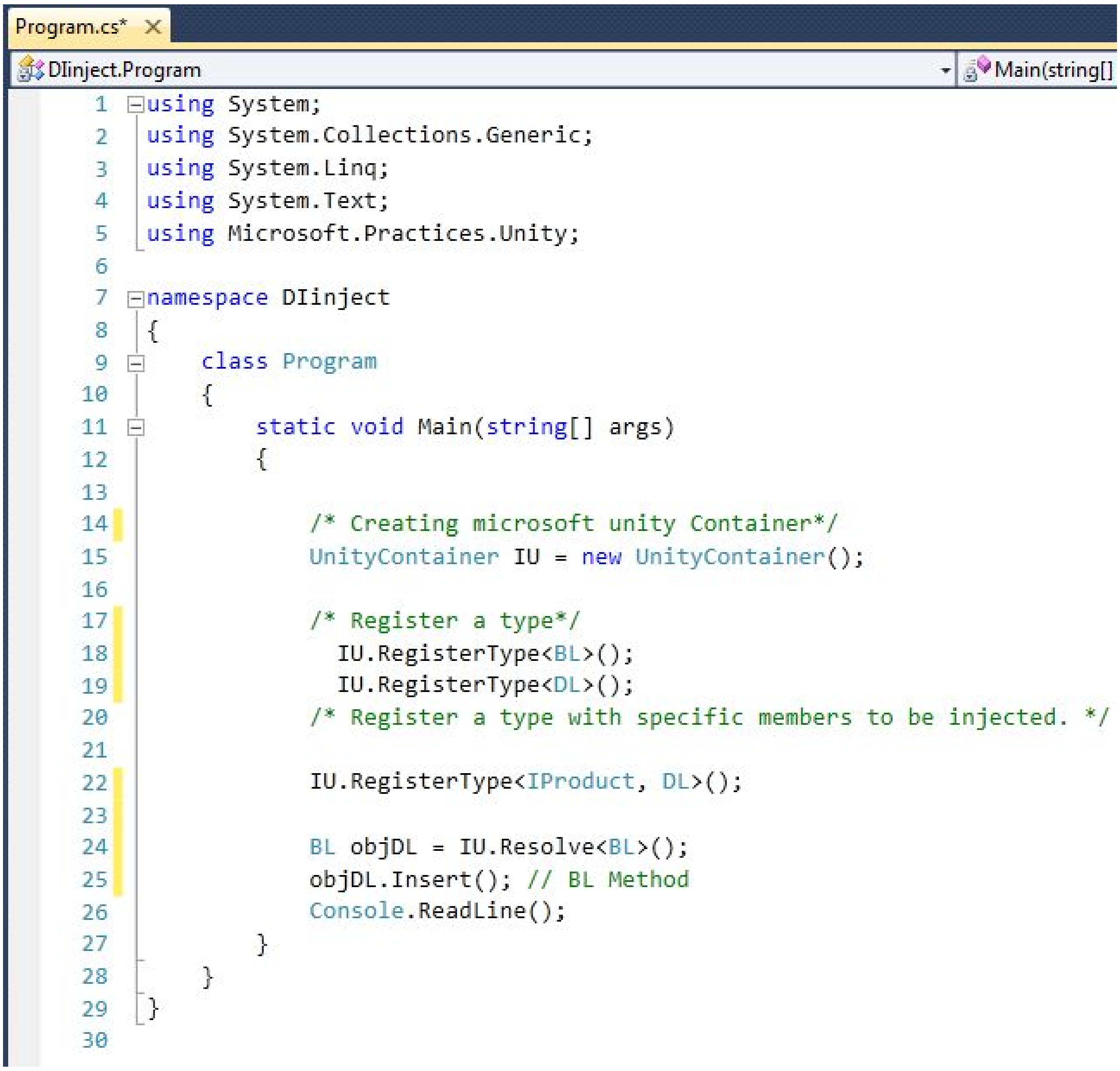
   ```
   BL objDL = IU.Resolve<BL>();
   ```

   We want to resolve the BL by injecting a DL. That is why I wrote Resolve<BL>( );.

5. Finally calling the method.

```
BL objDL = IU.Resolve<BL>();
objDL.Insert(); // BL Method
```

See in the following snapshot containing the entire Program.cs.



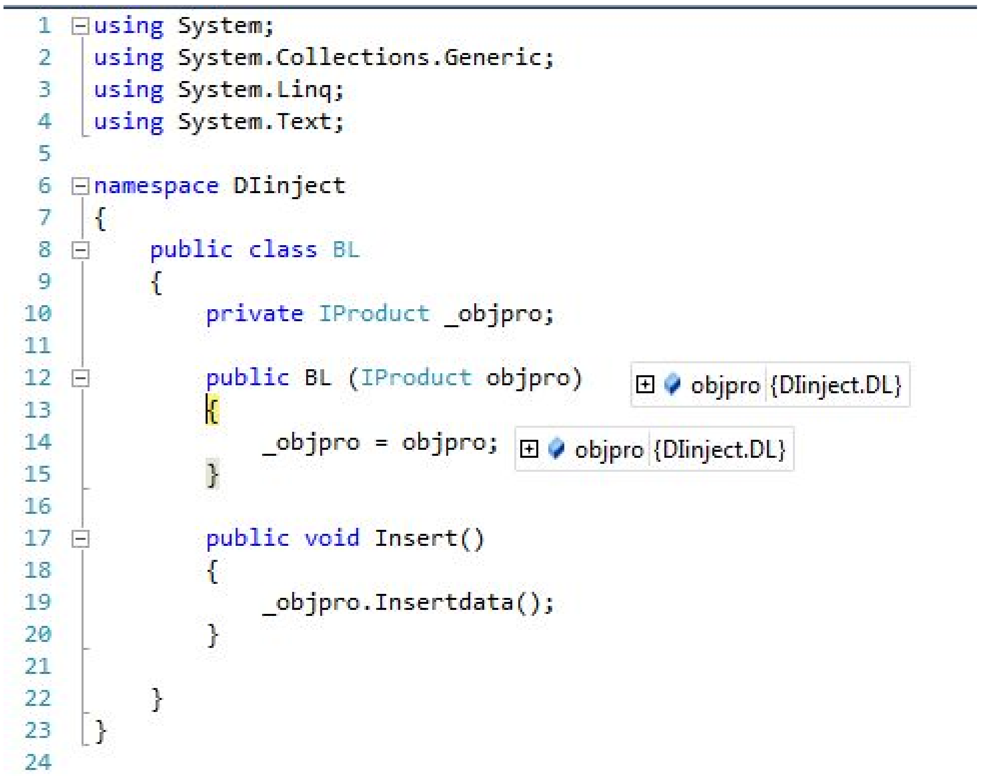
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Practices.Unity;

namespace DIinject
{
    class Program
    {
        static void Main(string[] args)
        {

            /* Creating microsoft unity Container*/
            UnityContainer IU = new UnityContainer();

            /* Register a type*/
              IU.RegisterType<BL>();
              IU.RegisterType<DL>();
            /* Register a type with specific members to be injected. */

            IU.RegisterType<IProduct, DL>();

            BL objDL = IU.Resolve<BL>();
            objDL.Insert(); // BL Method
            Console.ReadLine();
        }
    }
}

```

Running and debugging application

Now just run the application.

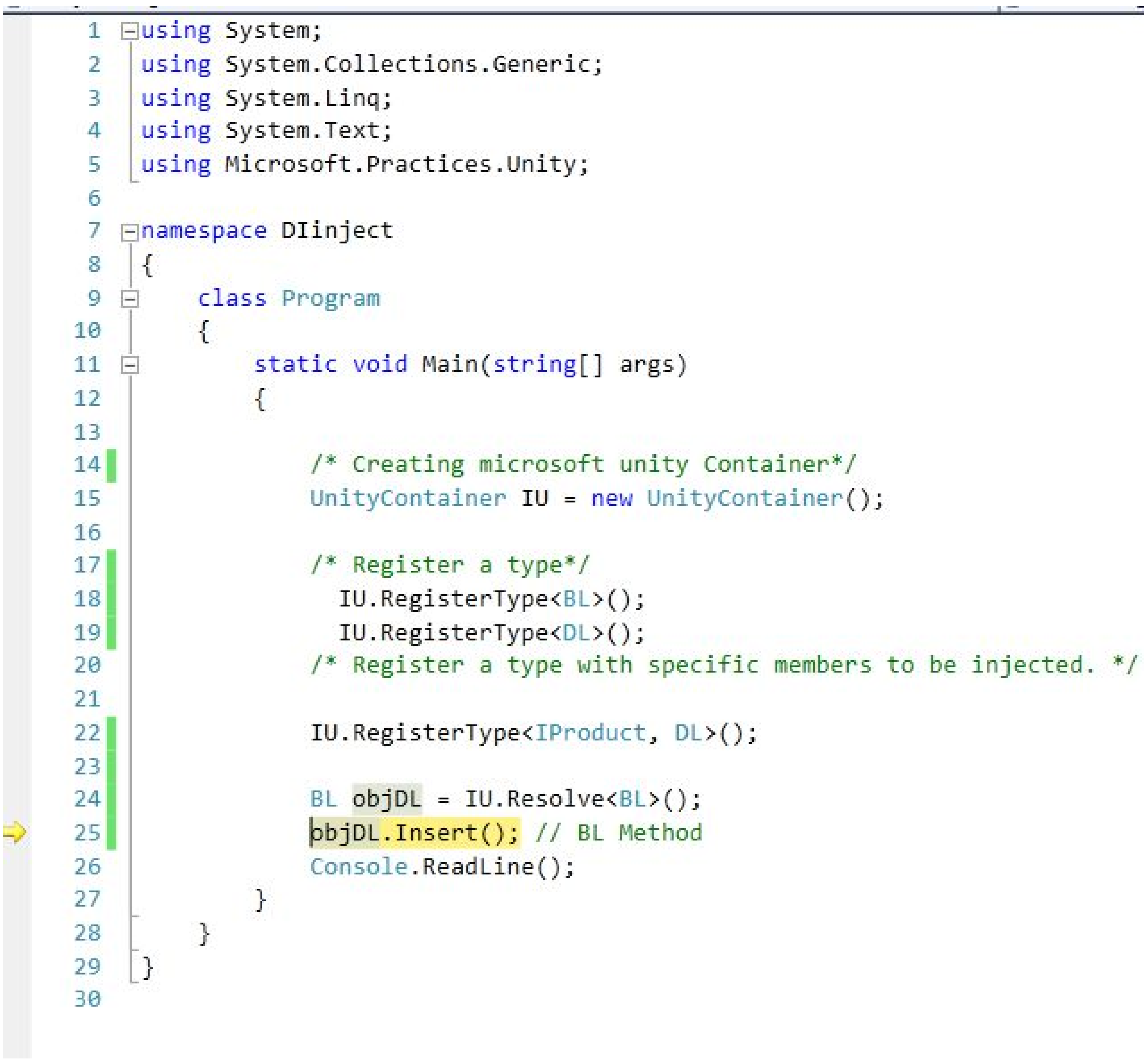
I show below a snapshot of running the BL during debugging that clearly shows the DL injecting.

See in the following snapshot the Data Access layer injecting using the constructor.



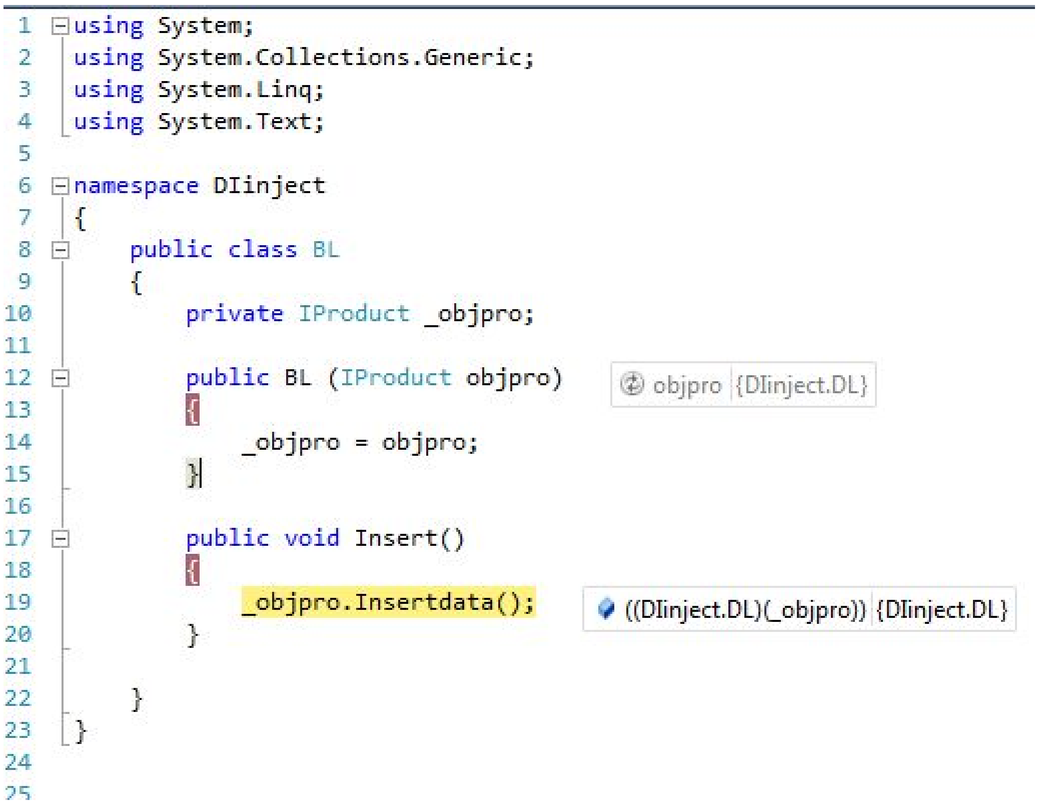
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DInject
{
    public class BL
    {
        private IProduct _objpro;

        public BL (IProduct objpro)
        {
            _objpro = objpro;
        }

        public void Insert()
        {
            _objpro.Insertdata();
        }

    }
}
```

See in the following snapshot a method of BL has been called.



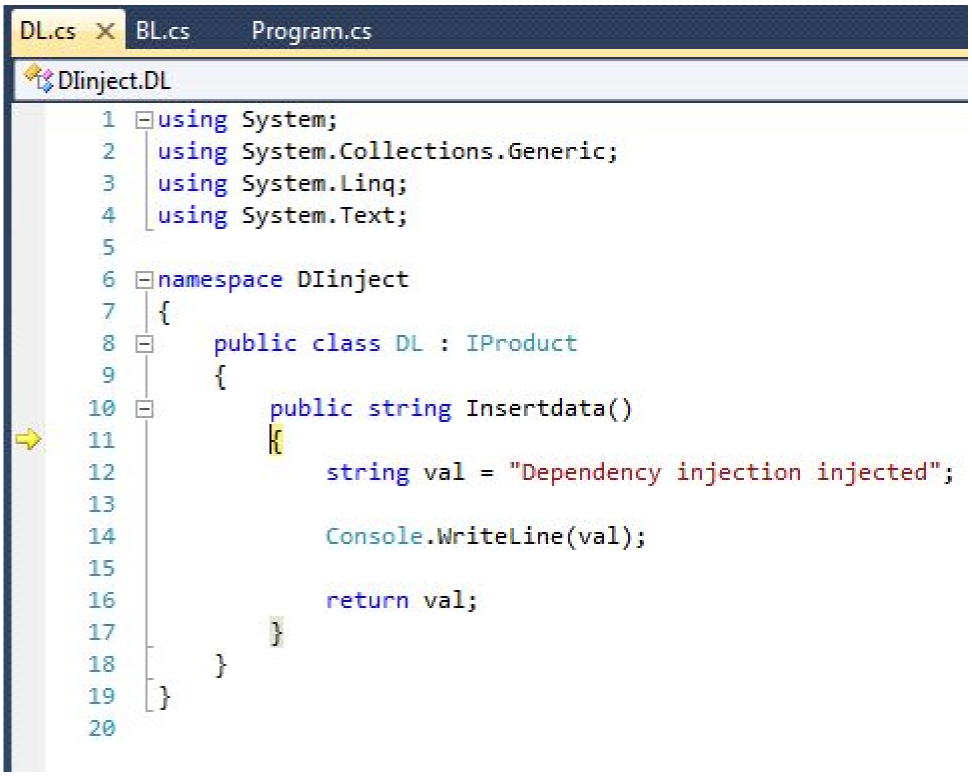
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Practices.Unity;

namespace DInject
{
    class Program
    {
        static void Main(string[] args)
        {

            /* Creating microsoft unity Container*/
            UnityContainer IU = new UnityContainer();

            /* Register a type*/
              IU.RegisterType<BL>();
              IU.RegisterType<DL>();
            /* Register a type with specific members to be injected. */

            IU.RegisterType<IProduct, DL>();

            BL objDL = IU.Resolve<BL>();
            objDL.Insert(); // BL Method
            Console.ReadLine();
        }
    }
}
```

See in the following snapshot, it is calling a method of a DA (Data Access) layer.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DInject
{
    public class BL
    {
        private IProduct _objpro;

        public BL (IProduct objpro)
        {
            _objpro = objpro;
        }

        public void Insert()
        {
            _objpro.Insertdata();
        }

    }
}
```

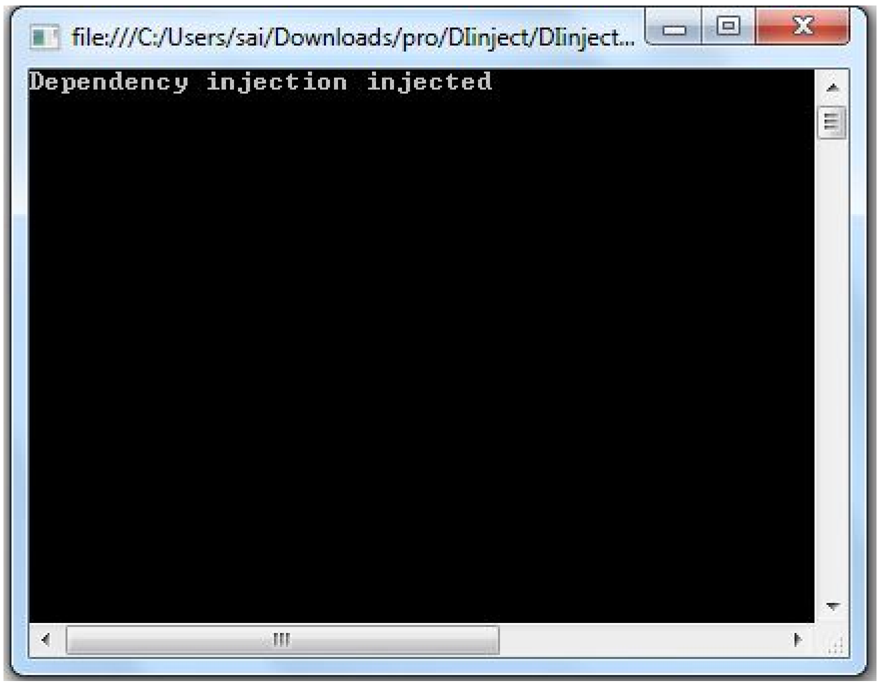
See in the following snapshot it has called the DA (Data Access) layer.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DInject
{
    public class DL : IProduct
    {
        public string Insertdata()
        {
            string val = "Dependency injection injected";

            Console.WriteLine(val);

            return val;
        }
    }
}
```

Final output

It has finally injected the dependency.



```
Dependency injection injected
```

Dependency Injection Pros and Cons

Pros

1. Loosely coupling.
2. Increase Testability.

Cons

1. Increase code complexity.
2. Complicate debugging of code.

The preceding is a simple example for developers. After this, you will find Dependency Injection to be an easy concept.