Dependency Injection Vs Inversion of Control

Dependency Injection (DI) and Inversion of Control (IoC) are two of the most important and popular architectural patterns. There is a widespread misunderstanding regarding the distinction between Dependency Injection and Inversion of Control. These two ideas are often misunderstood.

Inversion of Control is a design concept that enables the creation of dependent objects to be inverted. On the flipside, Dependency Injection, a software architectural pattern, is an implementation of the Inversion of control principle.

This article talks about both patterns with sample code to illustrate the concepts discussed.

What is Coupling in C#?

Broadly, object coupling is of two types: tight coupling and loose coupling. When an item is loosely connected with another object, the coupling may be easily changed; when the coupling is strong, the objects are not independently reusable and, therefore, challenging to use.

When the components of an application are tightly coupled, it is extremely difficult to change the components. Besides, extensibility, code reuse and testability in such applications is a challenge.

In the sections that follow, we'll examine how we can leverage DI and IOC to build systems that are loosely coupled, extensible and testable.

What is Dependency Injection (DI)?

Dependency injection removes the tight coupling between objects, allowing both the objects and the applications that utilize them to be more flexible, reusable, and testable. It makes it easier to create loosely coupled objects and their dependencies.

The core principle underlying Dependency Injection is to separate an object's implementation from the creation of objects on which it relies.

The various ways in which DI can be implemented are Constructor injection, Setter/Getter injection, Interface injection and Service locator.

Benefits and Drawbacks of Dependency Injection

The Dependency Injection pattern aids in the design and implementation of loosely linked, tested frameworks and components. Because it abstracts and isolates class dependencies, the source code becomes more testable.

The main disadvantage of dependency injection is that wiring instances together may become a headache if there are too many instances and dependencies to manage.

Read: Understanding Dependency Injection

What is Inversion of Control (IOC)?

Inversion of Control promotes loose coupling amongst the application's components. If your application's components are loosely coupled, you have greater reusability, easier maintainability and you can easily have mock objects.

Dependency Injection is a design pattern that is widely used to design and implement loosely coupled software components and frameworks. Note that while IOC is a principle or an architectural pattern, DI is a way of implementing IOC.

We can inject dependencies into our classes by simply providing them at the time of creation. When we have many classes, each with its own set of dependencies, this might become cumbersome. Here's precisely where an Inversion of Control container can help.

Benefits of IOC

The major benefit of IOC is that the classes are not dependent on each other anymore. As an example, the consumer class will not be dependent on the consumed class anymore and changes in one will not affect the other. So, IOC will make your design adaptable to changes. Another benefit is that you can isolate your code when creating unit tests. In essence, IOC promotes loose coupling and enables your classes to be more easily testable.

Implementing Inversion of Control in C#

Consider the following classes:

```
public class Employee
```

```
    {

        private Address address;

        //Other members

    }

    public class Address

    {

        public String Address1 { get; set; }

        public String Address2 { get; set; }

        public String Phone { get; set; }

        public String Zip { get; set; }

        public String City { get; set; }

        public String Country { get; set; }

    }
```

As you can see in the code example above, the Employee class contains an instance of the Address class. Now, if the Address class changes, the Employee class also need to be recompiled as it contains a reference to the Address class. The Employee class here controls the creation of the Address class and knows the type. This is an example of tight coupling between classes.

The solution to this problem is IOC. We need to invert the control – just shift the creation of the instance of the Address class to some other class. Let's say we have a Factory class that creates instances and returns them.

```
public class ObjectFactory

    {

        public static Address GetAddress()
        {

            //Some code

            return new Address();

        }

    }
```

The main principles driving IOC are that the classes aggregating other classes should not depend on direct implementation of the aggregated classes. Rather, they should depend on abstraction. In the code example we just discussed the Employee class should not depend on the Address class, rather, it should depend on an abstraction using an interface or an abstract class to get the instance of the Address class.

Another key point to bear in mind is that abstraction should not be reliant on details; rather, details should be reliant on abstractions. Note that we can implement IOC in various ways.

Read: .NET Dependency Injection Frameworks

Using a Constructor

If we are to inject the dependency using constructor, we can simply have a constructor that takes a reference of the Address class as parameter as shown in the code example below:

```
public class Employee

    {

        private Address _address;

        public Employee(Address address)

        {

            this._address = address;

        }

        //Other members
    }
```

For a setter and getter injection we will need properties that would get/set values. The major drawback of this approach is that because the objects are publicly exposed, encapsulation rules are violated. Here is an example:

```
public class Employee

    {

        private Address address;

        public Address SetAddress

        {

            set

            {

                this.address = value;

            }

        }

        //Other members

    }
```

Using an Interface

When implementing interface-based DI for our example, we need an interface in place of the Address class that would be used in the parameter of the **SetAddress** method. This interface will be implemented by the Address class. We can name this interface as **IContact** as all addresses are contacts, i.e., in our example we're dealing with the contact details of an Employee.

```
public interface IContact

    {
```

```
    //Some members
}
public class Employee
{
    private IContact _address;
    //Other members
    public void SetAddress(IContact address)
    {
        this._address = address;
    }
}
public class Address : IContact
{
    public String Address1 { get; set; }
    public String Address2 { get; set; }
    public String Phone { get; set; }
    public String Zip { get; set; }
    public String City { get; set; }
    public String Country { get; set; }
}
```

Using a Service Locator

You can also implement IOC using a Service Locator. The service locator pattern leverages a strong abstraction layer to encapsulate the mechanism of acquiring a service instance. In the service locator pattern, you would typically have an object aware of how to retrieve all services that your application might need.

We will now have a **ContactsService** that has a method called **GetAddress**.

```
public class ContactsService

    {

        private static IContact address = null;

        public static IContact GetAddress ()

        {

            //Some code to locate the right Address object

           // and return it

            return address;

        }

    }

public class Employee

    {

        private IContact address =
ContactsService.GetAddressObject();

         //Other members

    }
```

Summary of Dependency Injection and Inversion Control

You can take advantage of Inversion of Control to improve code modularity, reduce code duplication, and simplify testing. Even though it is useful for building reusable libraries, it is not appropriate in all scenarios.