# A curry of Dependency Inversion Principle (DIP), Inversion of Control (IoC), Dependency Injection (DI) and IoC Container

By **Hasibul Haque**, 12 Mar 2013

★ ★ ★ ★ ★ 4.95 (64 votes)

**Download sample - 88.8 KB**

## Contents

- Introduction
- Background
- Dependency Inversion Principle (DIP)
  - Let's try to understand DIP with some example
  - Problems we will face if we do not maintain DIP
  - Benefit of maintaining DIP
- Inversion of Control (IoC)
  - What is IoC
  - IoC and DIP
  - Splitting IoC
  - Fitting altogether (DIP, IoC and DI)
  - Interface Inversion
  - Flow Inversion
  - Creation inversion
  - Types of Creation inversion
- Dependency Injection (DI)
  - What is Dependency Injection
  - Types of Dependency Injection
    - Constructor Injection
    - Setter Injection
    - Interface Injection

- Inversion of Control Container (IoC container)
  - What is IoC Container
  - What IoC container does
  - Manual Constructor Injection
  - Building own IoC Container
  - Things need to know to use IoC framework
- Using Ninject IoC Container
  - Ninject
  - Setting up the container
  - Registering dependences
  - Using Container
  - Object life cycle management using Ninject
- Conclusion

## Introduction

Welcome to my post. Here I will try to describe DIP, IoC, DI and IoC container. Most of the time beginner developer faces problem with DIP, IoC, DI and IoC Container. They mix-up all together and find difficulties to identify the difference between them and don't know why they need to use them. On the other hand lots of people using DI, IoC without knowing it and what problem it solves. There are many post, article, blog available on this topic but not all in together. Here I have tried to describe all together. I hope after reading my post reader will able to identify the difference between DIP, IoC, DI & IoC Container/framework, also know how and when to use them. I also hope after reading my post reader will able to create their own IoC container. Before going details let me describe DIP, IoC, DI and IoC container using simple sentence.

| | |
|---|---|
| Dependency Inversion Principle (DIP) | : Principle used in architecting software. |
| Inversion of Control (IoC) | : Pattern used to invert flow, dependencies and interfaces. |
| Dependency Injection (DI) | : Implementation of IoC to invert dependencies. |
| IoC Container | : Framework to do dependency injection. It Helps to map dependencies, manage object creation & lifetime. |

## Background

DIP is a Software design **principle** and IoC is a Software design **pattern**. Let's see what is Software design principle and pattern.

- **Software design principle:** Principle provides us guideline. Principle says what is right and what is wrong. It doesn't say us how to solve problem. It just gives some guideline so that we can design good software and avoid bad design. Some principles are DRY, OCP, DIP etc.
- **Software design pattern:** Pattern is a general reusable solution to a commonly occurring problem within a given context in software design. Some patterns are factory pattern, Decorator pattern etc.

Now we have everything for going details.

So principle defines good and bad. So we can say if we maintain principle during software design then it will be good design. If we don't maintain principle then we may fall in problem. Now we are only focusing Dependency Inversion Principle (DIP) which is **D** on **SOLID** principle.

## Dependency Inversion Principle (DIP)

Instead of lower level modules defining an interface that higher level module depend on, higher level modules define an interface that lower level module implement.

### Let's try to understand DIP with some example

Let try to describe above principle with simple example of portable charger or devices. Consider you have Camera, Phone and other devices. These devices use cable to connect with computer or charger. A simple jack or port is used to connect with computer or charger right? Now if you are asked who has defined the port or jack; cable or your device?

You definitely answer device. Based on device port varies. So from the story what we found port doesn't define what will be the device but device defines what will be port or jack.
So software modules are similar. High level module defines the interface and low level module implements that interface and low level module doesn't define interface like jack doesn't define the device.
Let again consider the DIP according to Bob Martins definition

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

So from the principle we can say high level module should not depend on low level module. Both depend on abstraction. Abstraction should not depend on low level.

### Let's try to understand DIP with some example

**Case 1: Dependency not inverted (High level module depend on low level interface)**

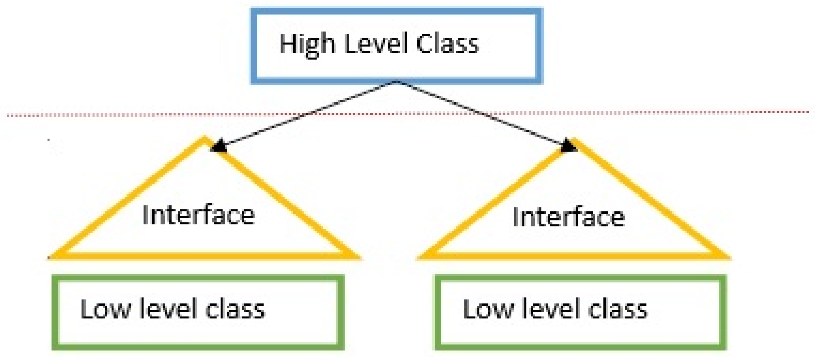


fig: When Dependency was not inverted

From the figure we found high level depends on low level interface. So high level class need to think about all the interfaces. When new low level class comes then again high level class need to change which makes complexity on maintenance and it violates open close principle.

**Case 2: Dependency Inversion**

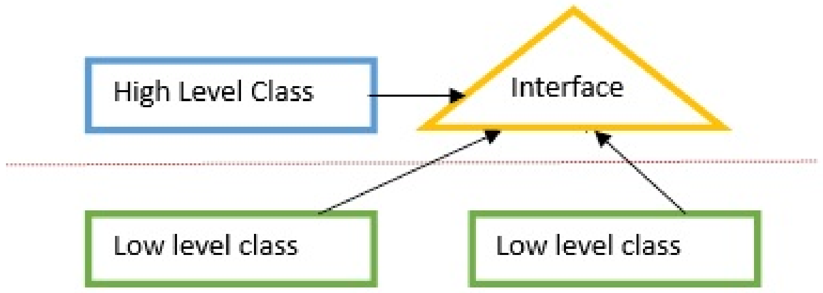


fig: Interface was defined by high level class

Now see the figure. Here higher level class defines the interface and higher level class doesn't depend on lower level class directly. Lower level classes implements interface defined by higher level class so higher level class doesn't need to change when new implementation arrived.

**Case 3: A Copy program without DIP**

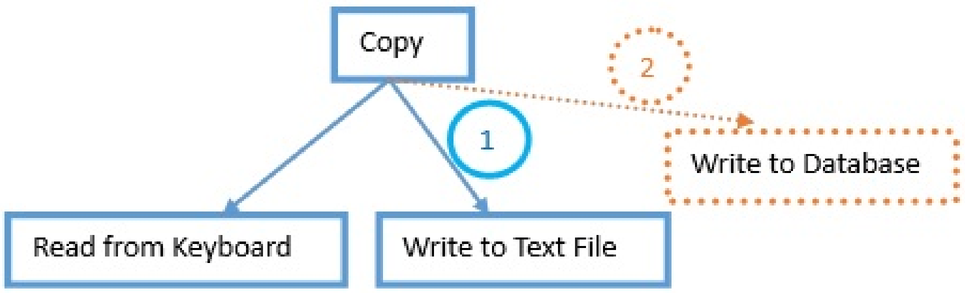


fig: Copy Program without maintaining DIP

Consider the above example.

1. First ignore doted objects. You have a copy program which is responsible for taking input from keyboard and writing to text file. Your current copy program can maintain both of them so it doesn't makes problem.
2. Now consider after certain period your boss asked you to develop your copy program so that it can store reading data to database. Then what you will do. You will again change to copy program so that it can write to database. This will increase complexity of your program.

**Case 4: Copy program with DIP**

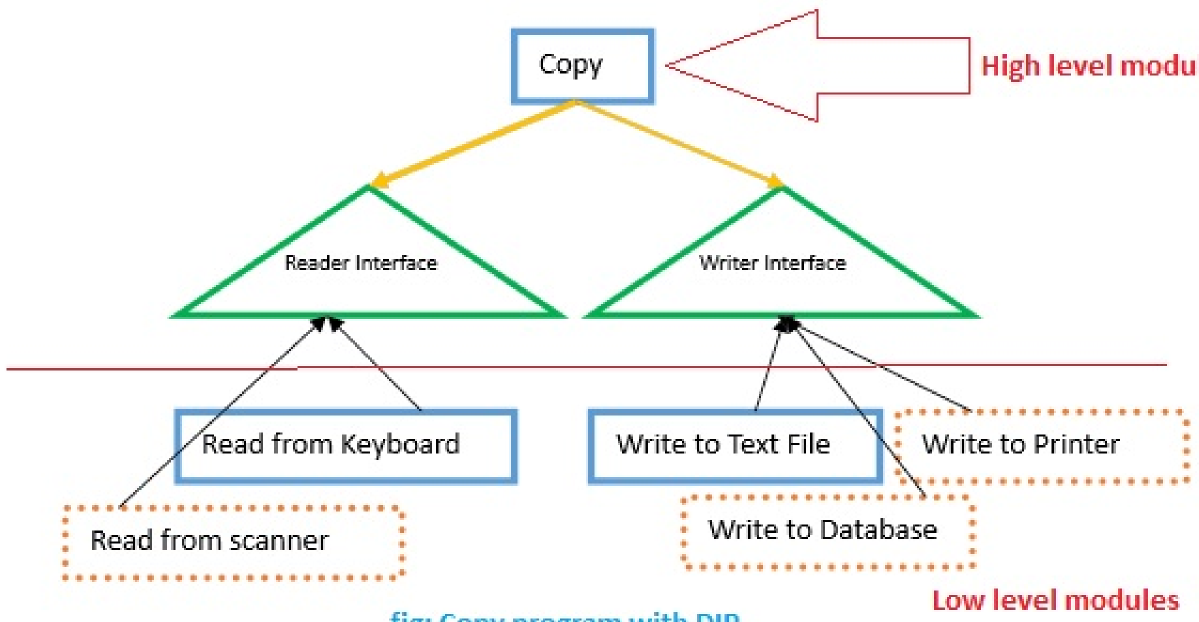


fig: Copy program with DIP

If we consider above example then we found that copy program is dependent on reader interface and writer interface defined by copy program. So it directly not dependent on lower level classes like read from scanner, write to printer etc. As a result we don't need to change copy program when new requirement will come and our copy program is independent.

I think DIP is little bit clear. Now let's see what will be the benefit for maintaining DIP and problems for not maintaining DIP. If we don't maintain software design principle then our software design will be bad design. Now consider a software design where we didn't maintain Dependency Inversion Principle (DIP.) As we are not maintaining DIP then we will fall in problem. Let see

## Problems we will face if we do not maintain DIP

- **System will rigid:** It will be difficult to change a part of the system without affection too many other parts of the system.
- **System will fragile:** When we will make a change, unexpected parts of the system will break.
- **System or component will be immobile:** It will be difficult to reuse it in another application because it cannot be disentangled from the current application. And so on……

## Benefit of maintaining DIP

First thing is we can solve above problems. That means our system will be loosely couple, independent, modular, testable and so on. As DIP is a principle it is not saying how to solve the problem. If we want to know how to solve the problem then we have to move on Inversion of Control.

## Inversion of Control (IoC)

DIP doesn't says us how to solve the problem but Inversion of Control defines some way so that we can maintain DIP. It is the thing which you can practically apply on your software development. Lots of definition available for IoC. Here I just try to give simple definition so that we can easily understand.

### What is IoC

It is basically a pattern to apply DIP. In simple IoC is Inverting the control of something by switching who control. Particular class or another module of the system will responsible for creation object from outside. Inversion of control means we are changing the control from normal way.

### IoC and DIP

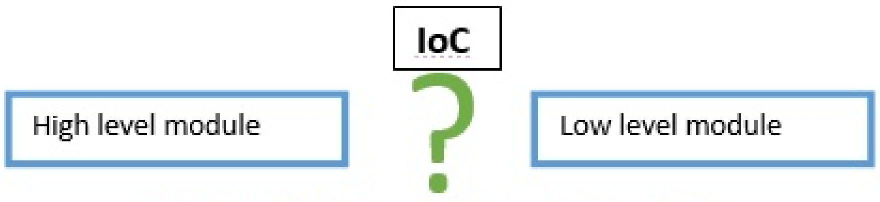


fig: What is difference between IoC and DIP

DIP says High level module should not depend on low level module and both should depend on abstraction. IoC is a way that provide abstraction. A way to change the control. IoC gives some ways to implement DIP. If you want to make independent higher level module from the lower level module then you have to invert the control so that low level module not controlling interface and creation of the object. Finally IoC gives some way to invert the control.

### Splitting IoC

We can split IoC in following ways. (Description will be provided later)

- **Interface Inversion :** Inverting interfaces.
- **Flow inversion :** Invert the flow of control and it is the foundation idea of IoC which is similar to Don't call us we will call you.
- **Creation Inversion :** This is mostly used by developer. We will use it when we go to DI and IoC container.

### Fitting altogether (DIP, IoC and DI)

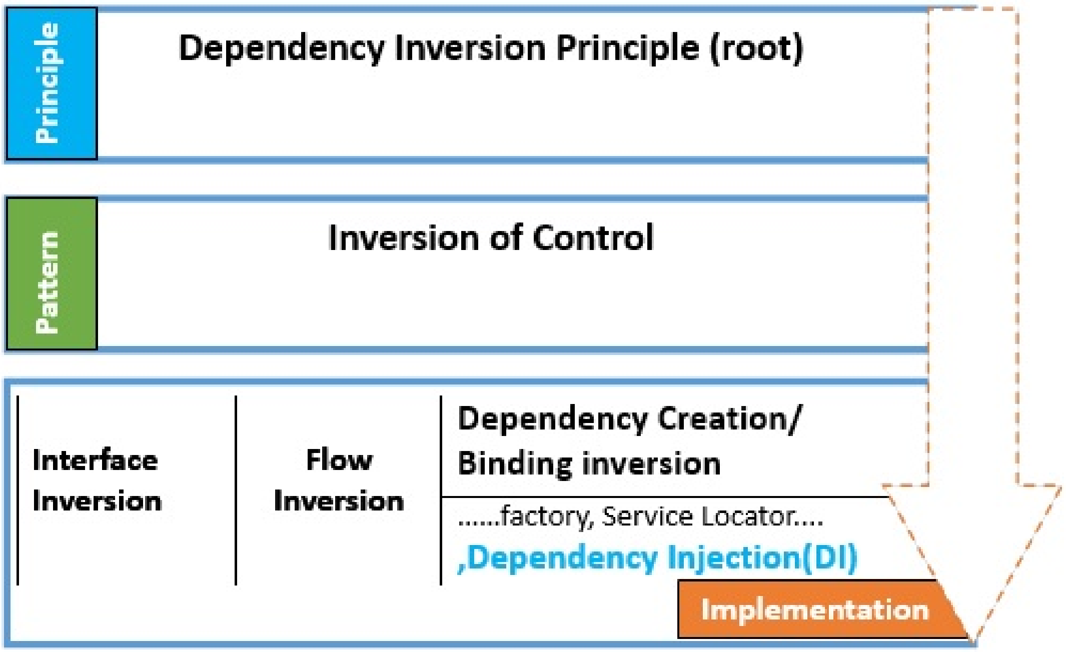
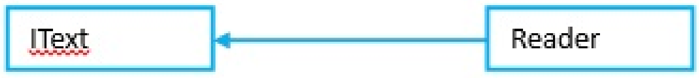

Principle
Dependency Inversion Principle (root)
Pattern
Inversion of Control
Interface Inversion
Flow Inversion
Dependency Creation/ Binding inversion
......factory, Service Locator....
,Dependency Injection(DI)
Implementation

fig: Fitting DIP, IoC and DI

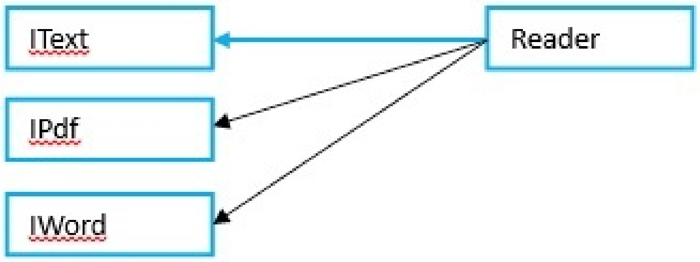
I am not making full curry here just consider above image how all together fits. This is the view how all things fits together. DI is not only the way of Dependency creation that's why I have used "…." . There are many ways to implement Dependency creation. Here I just interested to DI that's why I have shown it on figure and other are doted. At the top is DIP which is a way of designing software. It doesn't sys how to make independent module. IoC provides some way of applying DPI principle. IoC doesn't provide specific implementation. It gives some methods so that we can invert the control. If we want to invert control using Binding inversion or dependency creation then we can achieve it by implementing dependency injection (DI).

## Interface Inversion

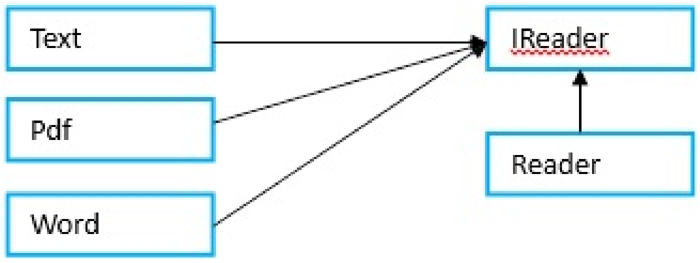
Interface inversion is the inversion of interface. Consider a Reader application. This reader application read from text file.



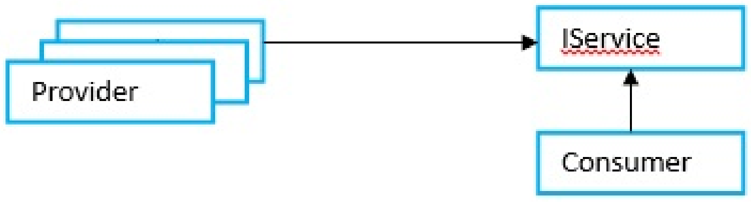
Now consider this application also read from pdf and word file



Creating too many interface doesn't mean I am implementing IoC or maintaining DPI. What is the benefit of using too many interfaces? Here we have no benefit because every time we have to change our reader class and reader class had to maintain all the interfaces. So ultimately we are not benefited. Now we have a method called interface inversion. Let invert the interfaces.



By considering above example, we remove all the interfaces from lower level classes and make single interface defined by reader class. What we did here just invert the interface. Now we have lots of benefit. We don't need to change reader class anymore and reader class is independent. Now consider above thing as a provider model.



## Flow Inversion

Let's see something about flow inversion. Flow inversion is the simple inversion which is backbone of the IoC. If you think about normal flow then you will find it is procedural.

### Normal Flow

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\hasibul>_
```

Fig: Command line program

Consider a command line program which first asks for your name then you put your name, again it will ask for your age then you will provide your age. What is going on here? You found your command line application executes one by one that means basic flow is procedural. It goes step by step.

**Inverted Flow**
Now let's try to invert the flow. Now we can think about a GUI. Consider our GUI consists of two input box one for user name and another for user age. Just consider following figure.

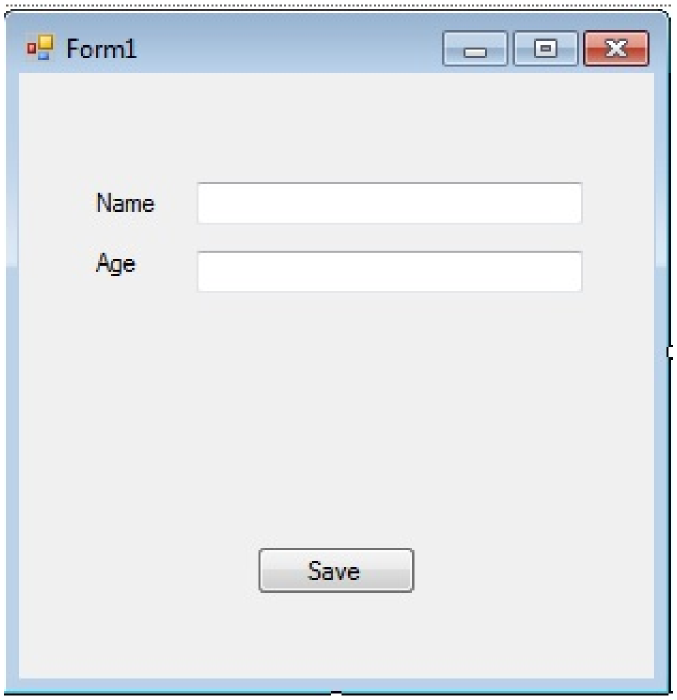


Fig: GUI Program

Now you see you can provide your name and your age without maintaining flow. If you want you can put your age first and your name last. So you don't need to depend on basic flow. On the other hand when you press save button then your program saves information. In command line program you are asking for first do this then do this but in GUI flow is controlled by user.

## Creation inversion

This is most familiar inversion of control method and we are going to use it on IoC container. Now consider following
**In normal way how we can create object:**

```
Class A
{
  YourClass yourObject = new YourClass();
}
```

So you are creating object which is dependent on another class and makes system tightly coupled. Here you are creating the object from inside your current class;
**Now consider Interface Inversion:**

```
Class A
```

```
{
    IYourInterface yourObject = new YourClass();
}
```

Even we are using interface inversion still we creating object inside the class. So creation of the object still has dependences. So what is creation inversion? What things we have to do to break the dependences? Let see
**Inverting Control/ Creation inversion:**
Creating object outside of the class they are being used in. We are making object from outside of the class so higher level class is not directly depending on lower level class. Now let's see why we need to invert the control.
Consider you have a GUI and you want to place a button on a screen. Your button has many design so your UI shows button based on UserSettings. Now what will be our code if we are doing in normal way. Lets see.

```
Button btn;
switch (UserSettings.ButtonStyle)
{
    case "Metro": btn = new MetroButton();
        break;

    case "Office2010": btn = new Office2007Button();
        break;

    case "Fancy": btn = new FancyButton();
        break;
}
```

So from the code our UI will show different style button based on user settings. Now after certain period new styles come then again you have to change UI code. If 20 style comes then we have to change 20 times. As Creation inversion says we have to create object outside of the class. Now lets try to follow that. Now consider we have a FactoryClass (we can learn more by reading factory pattern) which is responsible for provide button object based on user settings. This factory class is responsible for object creation. Now again let's see our UI class after implementing ButtonFactory class. Here I am not describing how to create Button factory class but you will find details on later.
So new UI has following code

```
Button btn = ButtonFactory.CreateButton();
```

So our button is depending on factory class and we don't need to change our UI code when a new style comes. Here object creation is done from outside of the UI class. In this way we are inverting the creation of object.

## Types of Creation inversion

**Factory Pattern:** If you check above example then you will find I have managed the creation of object from outside of the UI class. In factory pattern we do something like

```
Button btn = ButtonFactory.CreateButton();
```

**Service locator:**

```
Button btn = ServiceLocator.Create(IButtonControl);
```

In service locator pattern we are passing interface and corresponding implementation of requested interface will provided by service locator. Here I am not describing details because I am only focused to describe Dependency Injection (DI).
**Dependency Injection (DI):**
In dependency injection we pass dependency. Consider following simple code.

```
Button btn = GetButtonInstanceBasedOnUserSettings();
OurUi ourUI = new OurUi(btn);
```

Here we passed the dependencies when creation of UI. In DI the key idea is passing dependencies. DI is not only constructor injection. In next section I will describe more details on DI.
More…
There are more way to creation inversion.
So in summary any time you are taking the dependencies and binding the dependencies out of the class you are inverting the control and this is treated as Inversion of Control (IoC).

## Dependency Injection (DI)

Most of the time people mix-up DI with IoC. IoC describes different ways of inverting the control on the other hand DI inverts the Control by passing dependencies. So here I will try to describe following things

- What is Dependency Injection (DI)
- Types of DI
- Constructor Injection
- Setter Injection
- Interface Injection

### What is Dependency Injection

A type of IoC where we move the creation and binding of dependency outside of the class that depends on it. In normal object are created inside of the dependent class and bounded inside the dependent class. In DI it is done from outside of the dependent class. Let consider an example

Fig: Tiffin box with breakfast

Suppose you bring a Tiffin box in your office for breakfast. So you bring everything what you needed to breakfast. So you are managing the breakfast by Tiffin box. So this is similar to creation object inside of the class in a sense you are managing what you needed.

Fig: your breakfast will be provided like above  image

Now consider another example. In your office breakfast will be provided. You don’t need to bring the Tiffin box with you. Then what will happen. Still you can do breakfast but now it was provided by the office. This is similar to DI where required object are provided outside of the class.

## Types of Dependency Injection

### Constructor Injection

This is the most common form DI. Pass dependencies to dependent class through constructor. An injector creates the dependencies and passes the dependencies through constructor. Let consider following example. We have a `PurchaseBl` class which is responsible to perform save operation and depends on `IRepository`.

```
public class PurchaseBl
{
    private readonly IRepository _repository;

    //Constructor
    public PurchaseBl(IRepository repository)
    {
        _repository = repository;
    }

        public string SavePurchaseOrder()
    {
        return  _repository.Save();
    }
}
```

Now how we will access this `PurchaseBl` and how we will pass dependency through constructor let see:

```
//Creating dependency
IRepository dbRepository = new Repository();
//Passing dependency
PurchaseBl purchaseBl = new PurchaseBl(dbRepository);
```

From the above example we see `PurchaseBl` depends on `IRepository` and the `PurchaseBl` doesn't directly create any instance of repository. This repository will be provided outside of the class. As a result our `PurchaseBl` is independent from lower level class and it provides loose coupling.

Here Repository class implements `IRepository` which save information to Database. Now think if you need to save information to `TextFile` then you don't need to change the `PurchaseBL` class. You just have to pass another `TextRepository` class which is responsible for saving data to text file. Now let me show you if you have `TextRepository` class then how you will pass that repository.

```
IRepository textRepository = new TextRepository();
PurchaseBl purchaseBl = new PurchaseBl(textRepository);
```

So you can change dependencies without affection higher level class `PurchaseBl`.

## Setter Injection

This is another type of DI technique where we pass dependencies through setter instead of constructor. So what we have to do to setter injection:

**Create setter in dependent class:** In C# you just need to create a property. Use that property to set dependencies.

**Pass dependences through setter:** Here we can pass dependences through setter. We also create an object of class without passing dependences.

Let us see how I can create setter injection

```
public class PurchaseBl
{
   //Property
   public IRepository Repository { get; set; }

   //Here we are not creating any constructor which takes dependences
   public string SavePurchaseOrder()
   {
      return Repository.Save();
   }
}
```

Now let’s inject dependency through setter.

```
//Creating dependency
IRepository dbRepository = new Repository();
PurchaseBl purchaseBl = new PurchaseBl();
//Passing dependency through setter
purchaseBl.Repository = dbRepository;
Console.WriteLine(purchaseBl.SavePurchaseOrder());
```

It has some flexibility where we can change the dependency after creation of object and also we can create object without dependencies. There is a caution of setter injection. As we can create object without dependences then it may through exception when using dependencies without setting/injecting them.

## Interface Injection

This is not commonly in used and it is complex compare to other. Dependent class implements an interface. Interface has a method signature for setting the dependencies. Injector uses the interface to set dependency. So we can say dependent class has an implementation of interface and have a method to set dependencies instead of using setter and constructor.

Let's see an example.

1. First we have an interface class with a setter method. Let create that interface

```
public interface IDependentOnTextRepository
{
    //Method signature which is liable to inject/set dependency
    void SetDependency(IRepository repository);
}
```

2. Dependent class will implement the interface. So according to our example `PurchaseBl` will implement `IDependentOnTextRepository`.

```
public class PurchaseBl :IDependentOnTextRepository
{
    private IRepository _repository;

    public string SavePurchaseOrder()
    {
        return _repository.Save();
    }

    public void SetDependency(IRepository repository)
    {
        _repository = repository;
    }
}
```

3. Then finally how we can use `PurchaseBl` let's see:

```
//Creating dependency
IRepository dbRepository = new Repository();
PurchaseBl purchaseBl = new PurchaseBl();
//Passing dependency
```

```
((IDependentOnTextRepository)purchaseBl).SetDependency(dbRepository);
```

I think this is little bit complex compare to other and not commonly in used.

## Inversion of Control Container (IoC container)

### What is IoC Container

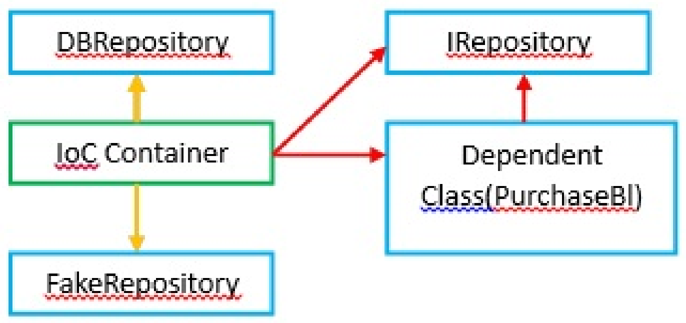
- Framework for doing dependency Injection.
- Provides a way to configure dependencies.
- Automatically resolve configured dependencies.

Still not clear let me describe another way. IoC container is a framework to create dependencies and pass them when needed. That means we don’t need to manually create dependencies like earlier examples. IoC framework automatically create objects based on request and inject them when needed. So it reduces lots of pain. You will be happy when you see all the dependencies create automatically.

### What IoC container does

1. Creates dependencies / objects
2. Pass/Inject dependencies when needed.
3. It manages object life time.
4. It also stores the mapping of classes defined by the developer. Based on this mapping your IoC container creates object.
5. And do so many works….

Lots of IoC framework (Unity, Ninject, Castle Windsor etc) are available to use. You can download them according to your need. But before using them I want to create IoC framework myself so that you can understand how IoC framework exactly works. Let’s visualize IoC Container From the figure



IoC container knows all the classes. IoC know what are dependencies and who are dependent. When dependent class request for an object then IoC provide the instance of requested class based on configuration. If Dependent class request for a class who implements `IRepository` then IoC think what dependencies (`DBRepository` or `FakeRepository`) need to provide based on configuration.

### Manual Constructor Injection

In Constructor Dependency injection section I have showed you some example there I actually did manual constructor injection. Just recall them

```
//Manually Creating dependencies
IRepository dbRepository = new Repository();
//Manually Passing dependencies
PurchaseBl purchaseBl = new PurchaseBl(dbRepository);
```

Here I am creating dependency and after creating I am passing that dependency to high level class. If my high level class dependent on several low level class then I had to create several dependencies and pass them manually. Then what IoC container do? Ans: IoC Container automatically creates the dependencies and you don't need to create them manually. I will show you how to create IoC container and how we can make them automatic and mange them from central point.

## Building own IoC Container

I know lots of IoC container available but I am showing here how we can develop our own IoC container because it helps to understand how IoC container works. I am not going show you details how IoC container create. Here I just create a simple IoC container which can resolve dependences based configuration. Let's start from scratch. Consider we have following interfaces and implementations.

```
///<summary>
///A repository interface used for Persisting
/// Persistance media will defined by implementation
///</summary>
public interface IRepository
{
    string Save();
}
```

And we have following implementation. Repository is responsible for storing data to DataBase and `TextRepository` is responsible for storing data to text file. Now consider simple implementation of `IRepository`:

```
///<summary>
/// Responsible for saving data to Database
///</summary>
public class Repository : IRepository
{
    public string Save()
    {
        return "I am saving data to Database.";
    }
}
///<summary>
/// Responsible for savaing data to text file
///</summary>
class TextRepository : IRepository
{
    public string Save()
    {
        return "I am saving data to TextFile.";
    }
}
```

Now we have `PurchaseBl` class which is dependent on `IRepository`. `PurchaseBl` is responsible for applying business logic and it uses `IRepository` to persist its information.

```
/// This is a class where we put our business logic.
/// here we have SavePurchaseOrder method which is responsible for storing data
/// But this class doesnt know where to store data. it just using implemention of IRepository
to storing data

public class PurchaseBl
{
    private readonly IRepository _repository;
    public PurchaseBl(IRepository repository)
    {
        _repository = repository;
    }

    public string SavePurchaseOrder()
    {
        return _repository.Save();
    }
}
```

So our business class `PurchaseBl` doesn't know where data will be stored. So it is not depending on lower level class. So we can change the persistence mechanism without changing our business class.

Let's use our `PurchaseBl` with manual dependency injection first.

```
public static void Main()
{
   //Creating dependency
   IRepository dbRepository = new Repository();
   //injecting dbRepository
   PurchaseBl purchaseBl = new PurchaseBl(dbRepository);
   Console.WriteLine(purchaseBl.SavePurchaseOrder());
}
```

### Output

```
I am saving data to Database.
```

Now if we want to store our data to text file without affecting our Business layer then we just have to do little change on dependency creation. Let's see:

```
public static void Main()
{
    IRepository textRepository = newTextRepository();
        //Injecting textRepository
    PurchaseBl purchaseBl = newPurchaseBl(textRepository);
    Console.WriteLine(purchaseBl.SavePurchaseOrder());
}
```

### Output

```
I am saving data to TextFile.
```

Now we think one step further and want to use IoC Container then what will be our main method looks like. Let's see:

```
public static void Main()
{
    Resolver resolver = new Resolver();//Resolver is a IoC container
    PurchaseBl purchaseBl = new PurchaseBl(resolver.ResolveRepository());
    Console.WriteLine(purchaseBl.SavePurchaseOrder());
}

public class Resolver
{
    public IRepository ResolveRepository()
    {
        return new TextRepository();
    }
}
```

But above technique is not good enough. Every time we have to create method on Resolver class which may increase complexity. Now let's think another step further. Now we want to resolve dependences like following:

```
public static void Main()
{
    Resolver resolver = new Resolver();//Resolver is a IoC container
    PurchaseBl purchaseBl = resolver.Resolve<purchasebl>();
    Console.WriteLine(purchaseBl.SavePurchaseOrder());
}
```

Now see we don't need to create dependencies IoC container automatically creates dependences and inject to `PurchaseBl` class. Now our main method is more simple compare to previous. Am I right? Now our IoC container resolve dependences by reading the type of class. Here I just showed you how to use IoC container and how to get object by passing the type only. Now let's create our one. Here I am adding my simple code with description.

```
/// <summary>
/// Our simple IoC container
/// </summary>
public class Resolver
{
   //This is dictionary used to keep mapping between classes
   private readonly Dictionary<Type, Type> _dependencyMapping = new Dictionary<Type, Type>();


   /// <summary>
   /// return requested object with requested type
   /// </summary>
   /// <typeparam name="T">Takes type which need to resolved</typeparam>
   /// <returns></returns>
   public T Resolve<T>()
   {
       // cast object to resolved type
       return (T)Resolve(typeof(T));
   }


   /// <summary>
   /// This method takes the type which need to resolve and returns an object based on
configuration
   /// </summary>
   /// <param name="typeNeedToResolve">takes type which need to resolve</param>
   /// <returns>return an object based on requested type</returns>
```

```
    private object Resolve(Type typeNeedToResolve)
    {
        Type resolvedType;
        try
        {
            //Taking resolved/return type from dictionary which was configured earlier by
Register method
            resolvedType = _dependencyMapping[typeNeedToResolve];
        }
        catch
        {
            //If no mapping found between requested type and resolved type then it will through
exception
            throw new Exception(string.Format("resolve failed for {0}",
typeNeedToResolve.FullName));
        }

        //Getting first constructor of resolved type by reflection
        var firstConstructor = resolvedType.GetConstructors().First();

        //Getting first constructor's parameter by reflection
        var constructorParameters = firstConstructor.GetParameters();

        //if no parameter found then we dont need to think about other resolved type from the
parameter
        if (!constructorParameters.Any())
            return Activator.CreateInstance(resolvedType); // returning an instance of resolved
type

        //if our resolved type has constructor then again we have to resolve that types;
        //so again we are calling our resolve method to resolve from constructor
        IList<object> parameterList = constructorParameters.Select(
          parameterToResolve => Resolve(parameterToResolve.ParameterType)).ToList();
        //invoking parameters to constructor
        return firstConstructor.Invoke(parameterList.ToArray());

    }


    /// <summary>
    /// This method is used  to store mapping between request type and return type
    /// If you request for IRepository then what implementation will be returned; you can
configure it from here by writing
    /// Registery<IRepository, TextRepository>()
    /// That means when resolver request for IRepository then TextRepository will be returned
    /// </summary>
    /// <typeparam name="TFrom">Request Type</typeparam>
    /// <typeparam name="TTo">Return Type</typeparam>
    public void Registery<TFrom, TTo>()
    {
        _dependencyMapping.Add(typeof(TFrom), typeof(TTo));
    }
}
```

So finally our main method looks following:

```
public static void Main()
{
    //registering dependences  to Container
    var resolver = new Resolver();
    resolver.Registery<IRepository, FakeRepository>();
    resolver.Registery<PurchaseBl, PurchaseBl>();
    // Resolving dependences
    var purchaseBl = resolver.Resolve<PurchaseBl>();
    Console.WriteLine(purchaseBl.SavePurchaseOrder());
}
```

### Things need to know to use IoC framework

If you think above code is complex then you don't need to understand it. You just need to know 2 things for using IoC container.

1. How to configure dependences called how to register component
2. How to resolve dependences.

For resolving object you just need to write following code:

```
//Resolving dependences
var purchaseBl = resolver.Resolve<purchasebl>;();
Console.WriteLine(purchaseBl.SavePurchaseOrder());
```

Here I am not saying anything about object life time management because as I said I will create simple IoC container so that you can understand how IoC works.

Now I want to show you how we can use another IoC container Ninject in our project.

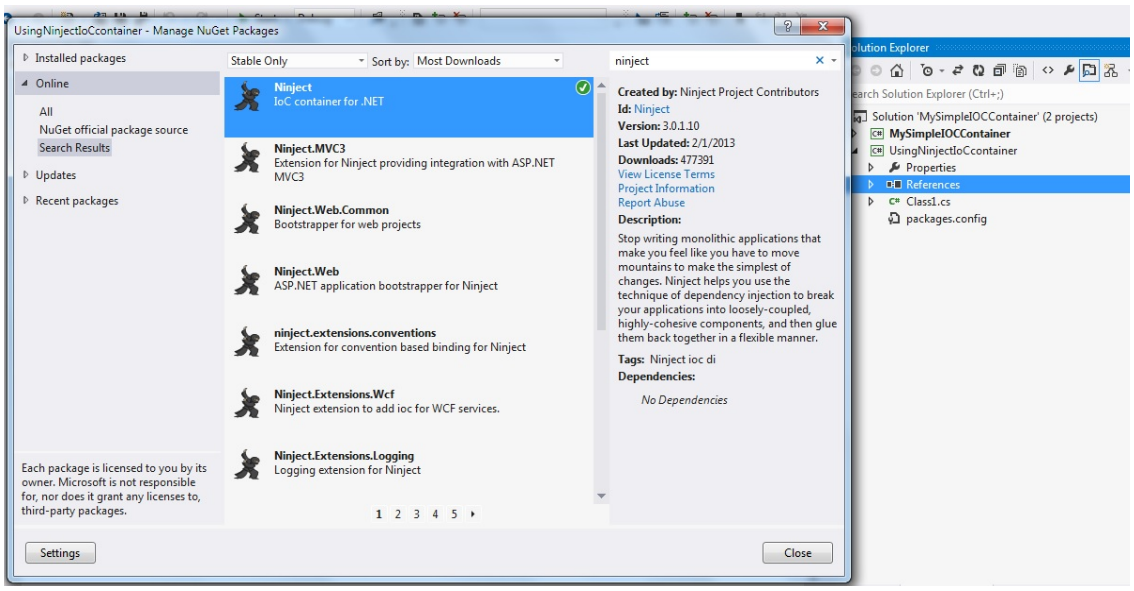
## Using Ninject IoC Container

Here I will try to cover:

- Introduction to Ninject
- Setting up the container
- Using Container
- Lifetime management and other features.

## Ninject

Ninject is a newer open source IoC Container. This is simple and extensible. You can download Ninject IoC container from the following location: http://www.ninject.org/download.html Or you can use NuGet to add ninject to your project. Let's add ninject to our project from NuGet. Just go to your project reference and right click then ManageNuGet Packages.. then search for Nintject. Then you will find something like following.


UsingNinjectIoCcontainer - Manage NuGet Packages

Installed packages
Online
All
NuGet official package source
Search Results
Updates
Recent packages

Stable Only | Sort by: Most Downloads | ninject

Ninject
IoC container for .NET

Ninject.MVC3
Extension for Ninject providing integration with ASP.NET MVC3

Ninject.Web.Common
Bootstrapper for web projects

Ninject.Web
ASP.NET application bootstrapper for Ninject

ninject.extensions.conventions
Extension for convention based binding for Ninject

Ninject.Extensions.Wcf
Ninject extension to add ioc for WCF services.

Ninject.Extensions.Logging
Logging extension for Ninject

1 2 3 4 5

Created by: Ninject Project Contributors
Id: Ninject
Version: 3.0.1.10
Last Updated: 2/1/2013
Downloads: 477391
View License Terms
Project Information
Report Abuse
Description:
Stop writing monolithic applications that make you feel like you have to move mountains to make the simplest of changes. Ninject helps you use the technique of dependency injection to break your applications into loosely-coupled, highly-cohesive components, and then glue them back together in a flexible manner.
Tags: Ninject ioc di
Dependencies:
No Dependencies

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Settings | Close

After installing Ninject you will find Ninject on project reference. For using Ninject you have to use following namespace.

```
using Ninject;
```

### Setting up the container

Ninject container uses a class named kernel. Ninject automatically register all concrete types you don't need to register them.

## Registering dependences

```
var kernel = new StandardKernel();
kernel.Bind<IRepository>().To<Repository>();
```

When `IRepository` will be requested then Repository implementation will be provided.

## Using Container

### Resolving dependences

```
//resolving dependences
//Out from kernel
PurchaseBl purchaseBl = kernel.Get<PurchaseBl>();
```

```
Console.WriteLine(purchaseBl.SavePurchaseOrder());
```

It produces the following output.
**Output**

```
I am saving data to Database.
```

If we change the bindings then

```
var kernel = new StandardKernel();
kernel.Bind<IRepository>().To<TextRepository>();
PurchaseBl purchaseBl = kernel.Get<PurchaseBl>();
Console.WriteLine(purchaseBl.SavePurchaseOrder());
```

It produces following output.
**Output**

```
I am saving data to TextFile.
```

Rebind is very useful when we need to do something over default binding. Suppose in some times we need to change default binding when new modules comes. For rebinding we can use Rebind method like following

```
kernel.Rebind<IRepository>().To<Repository>();
```

These are simple way we can use ninject. Now lets see how to manage life cycle using ninject.

## Object life cycle management using Ninject

Let modify our TextRepository class so that we can understand our object life. The following `TextRepository` is our modified class.

```
/// <summary>
/// Responsible for savaing data to text file
/// </summary>
public class TextRepository : IRepository
{
    private int _counter = 0;
    public string Save()
    {
        _counter++;
        return string.Format("I am saving data to TextFile {0}.", _counter);
    }
}
```

Let me create two instance of `PurchaseBl` like following:

```
var kernel = new StandardKernel();
kernel.Bind<IRepository>().To<TextRepository();

//resolving dependences
//Out from kernel
var purchaseBl = kernel.Get<PurchaseBl>();
Console.WriteLine(purchaseBl.SavePurchaseOrder());
```

```
var purchaseBl2 = kernel.Get<PurchaseBl>();
Console.WriteLine(purchaseBl2.SavePurchaseOrder());
```

Now we have two `PurchseOrderBl` instances `purchaseBl` and `purchaseBl2`. So the output is:

**Output:**

```
I am saving data to TextFile1.
I am saving data to TextFile1.
```

Look here both outputs showing 1 that means every time it creates new instances. Now if we want to create single instance of `TextRepository` then we have to create instance as a Singleton so that same instance share through both `purchaseBl` and `purchaseBl2`.

Let's do that:

```
kernel.Bind<IRepository>().To<TextRepository>().InSingletonScope();
```

**Output:**

```
I am saving data to TextFile1.
I am saving data to TextFile2.
```

Now first one return 1 and then second one return 2 that means single instance of `TextRepository` was shared by both. We can create our object in many scopes like Transaction Scope, Thread Scope, Singleton Scope and so on. These are the basic how we can use ninject in our project. Later I have a plan to write more details about ninject, unity and Castle Windsor.

## Conclusion

We write lots of code and after some time we face difficulties to manage and test them. This is only because of tight coupling. Tight coupling makes our system rigid. DIP, IoC and DI helps us to write loosely couple code and make independent, modular system. Here we saw lots of way to make IoC. From all the technique Creation inversion (Dependency injection with constructor injection) is very common. I think DIP, IoC, DI and IoC container are now clear to you. I just tried to present altogether here because it helps us to understand all things related to DIP. Just remember one thing without IDP you cannot create independent and pluggable system. Here I cannot describe more details because it will go large and you people feel bowering. Thanks for reading my post!! I am expecting your valuable comments, suggestion and criticism.

## Some of References

- http://martinfowler.com/articles/injection.html
- http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html
- http://www.pluralsight.com/training
- http://www.oodesign.com
- http://blog.architexa.com/2010/05/types-of-dependency-injection/
- http://www.codeproject.com/Articles/495019/Dependency-Inversion-Principle-and-the-Dependency

- and so on

## License

## About the Author

**Hasibul Haque**
Software Developer Daffodil Group
Bangladesh

Hi I am Mohammad Hasibul Haque an application developer. Now I am working as a Software Engineer(.net) at Daffodil Group. Here I am working for Business ERP, Daffodil School Management System and so on. I am dedicated to Microsoft .NET based application development.

I am very much fan of Microsoft. Always want to learn new things & new technologies..

I am Microsoft Certified Professional Developer (MCPD ASP.net .net 3.5) and also achieved Microsoft Community Contributor Award 2011 from Microsoft.

Follow on Twitter Google

## Comments and Discussions

**72 messages** have been posted for this article Visit **http://www.codeproject.com/Articles/538536/A-curry-of-Dependency-Inversion-Principle-DIP-Inve** to post and view comments on this article, or click **here** to get a print view with messages.