



Articles » Development Lifecycle » Design and Architecture » General

An Absolute Beginner's Tutorial on Dependency Inversion Principle, Inversion of Control and Dependency Injection

By **Rahul Rajat Singh**, 8 Jul 2013

★★★★★ 4.85 (75 votes)



Prize winner in Competition "Best C# article of July 2013"

Prize winner in Competition "Best overall article of July 2013"

Introduction

In this article we will talk about the Dependency Inversion Principle, Inversion of Control and Dependency Injection. We will start by looking at the dependency inversion principle. We will then see how we can use inversion of control to implement dependency inversion principle and finally we will look at what dependency injection is and how can it be implemented.

Background

Before we start talking about Dependency Injection(DI), we first need to understand the problem that DI solves. To understand the problem, we need to know two things. First dependency Inversion Principle(DIP) and second Inversion of Controls(IoC). let us start our discussion with DIP, then we will talk about IoC. once we have discussed these two, we will be in a better position to understand Dependency Injection, so we will look at dependency injection in details. Then finally we will discuss see how can we implement Dependency injection.

Dependency Inversion Principle

Dependency inversion principle is a software design principle which provides us the guidelines to write loosely coupled classes. According to the definition of Dependency inversion principle:

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

What does this definition mean? What is it trying to convey? let us try to understand the definition by looking at examples. A few years back I was involved in writing a windows service which was supposed to run on a Web server. The sole responsibility of this service was to log messages in event logs whenever there is some problem in the IIS application Pool. So what our team has done initially that we created two classes. One for monitoring the Application Pool and second to write the messages in the event log. Our classes looked like this:

```
class EventLogWriter
{
    public void Write(string message)
    {
        //Write to event log here
    }
}

class AppPoolWatcher
{
    // Handle to EventLog writer to write to the logs
    EventLogWriter writer = null;

    // This function will be called when the app pool has problem
    public void Notify(string message)
    {
        if (writer == null)
        {
            writer = new EventLogWriter();
        }
        writer.Write(message);
    }
}
```

From the first look, the above class design seems to be sufficient. It looks perfectly good code. But there is a problem in the above design. This design violates the dependency inversion principle. i.e. the high level module **AppPoolWatcher** depends on **EventLogWriter** which is a concrete class and not an abstraction. How is it a problem? Well let me tell you the next requirement we received for this service and the problem will become very clearly visible.

The next requirement we received for this service was to send email to network administrator's email ID for some specific set of error. Now, how will we do that? One idea is to create a class for sending emails and keeping its handle in the **AppPoolWatcher** but at any moment we will be using only one object either **EventLogWriter** or **EmailSender**.

The problem will get even worse when we have more actions to take selectively, like sending SMS. Then we will have to have one more class whose instance will be kept inside the **AppPoolWatcher**. The dependency inversion principle says that we need to decouple this system in such a way that the higher level modules i.e. the **AppPoolWatcher** in our case will depend on a simple abstraction and will use it. This abstraction will in turn will be mapped to some concrete class which will perform the actual operation. (Next we will see how this can be done)

Inversion of Control

Dependency inversion was a software design principle, it just states that how two modules should depend on each other. Now the question comes, how exactly we are going to do it? The answer is Inversion of control. Inversion of control is the actual mechanism using which we can make the higher level modules to depend on abstractions rather than concrete implementation of lower level modules.

So if I have to implement inversion of control in the above mentioned problem scenario, the first thing we need to do is to create an abstraction that the higher levels will depend on. So let us create an interface that will provide the abstraction to act on the notification received from **AppPoolWacther**.

```
public interface INofificationAction
{
    public void ActOnNotification(string message);
}
```

Now let us change our higher level module i.e. the **AppPoolWatcher** to use this abstraction rather than the lower level concrete class.

```
class AppPoolWatcher
{
    // Handle to EventLog writer to write to the logs
    INofificationAction action = null;

    // This function will be called when the app pool has problem
    public void Notify(string message)
    {
        if (action == null)
        {
            // Here we will map the abstraction i.e. interface to concrete class
        }
        action.ActOnNotification(message);
    }
}
```

So how will our lower level concrete class will change? how will this class conform to the abstraction i.e. we need to implement the above interface in this class:

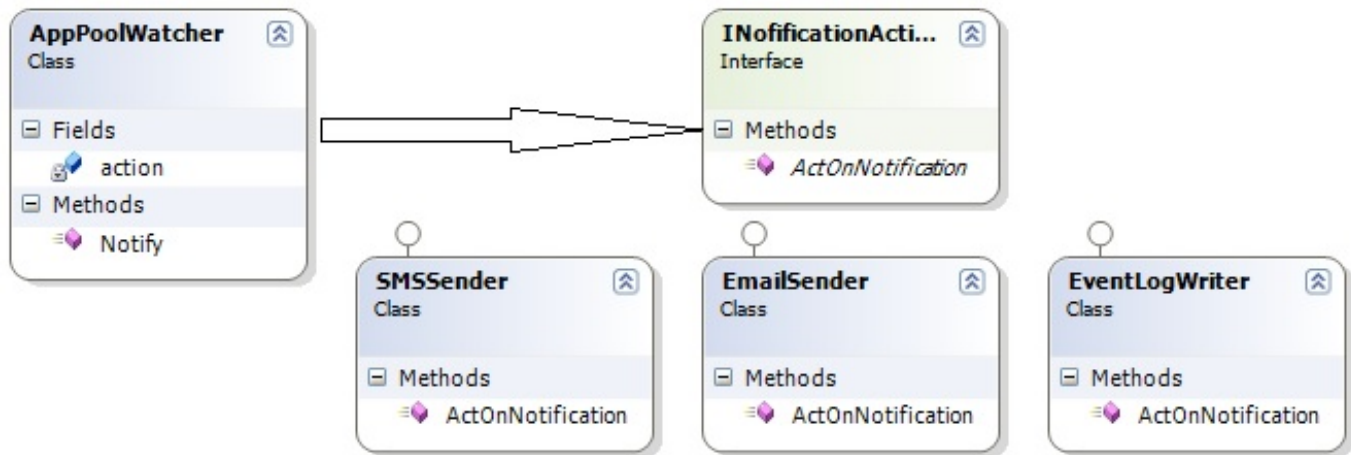
```
class EventLogWriter : INofificationAction
{
    public void ActOnNotification(string message)
    {
        // Write to event log here
    }
}
```

So now if I need to have the concrete classes for sending email and sms, these classes will also implement the same interface.

```
class EmailSender : INofificationAction
{
    public void ActOnNotification(string message)
    {
        // Send email from here
    }
}

class SMSSender : INofificationAction
{
    public void ActOnNotification(string message)
    {
        // Send SMS from here
    }
}
```

So the final class design will look like:



So what we have done here is that, we have inverted the control to conform to dependency inversion principle. Now our high level modules are dependent only on abstractions and not the lower level concrete implementations, which is exactly what dependency inversion principle states.

But there is still one missing piece. When we look at the code of our **AppPoolWatcher**, we can see that it is using the abstraction i.e. interface but where exactly are we creating the concrete type and assigning it to this abstraction. To solve this problem, we can do something like:

```

class AppPoolWatcher
{
    // Handle to EventLog writer to write to the logs
    INotificationAction action = null;

    // This function will be called when the app pool has problem
    public void Notify(string message)
    {
        if (action == null)
        {
            // Here we will map the abstraction i.e. interface to concrete class
            writer = new EventLogWriter();
        }
        action.ActOnNotification(message);
    }
}
  
```

But we are again back to where we have started. The concrete class creation is still inside the higher level class. Can we not make it totally decoupled so that even if we add new classes derived from **INotificationAction**, we don't have to change this class.

This is exactly where Dependency injection comes in picture. So its time to look at dependency injection in detail now.

Dependency Injection

Now that we know the dependency inversion principle and have seen the inversion of control methodology for implementing the dependency inversion principle, Dependency Injection is mainly for injecting the concrete implementation into a class that is using abstraction i.e. interface inside. The main idea of dependency injection is to reduce the coupling between classes and move the binding of abstraction and concrete implementation out of the dependent class.

Dependency injection can be done in three ways.

1. Constructor injection
2. Method injection
3. Property injection

Constructor Injection

In this approach we pass the object of the concrete class into the constructor of the dependent class. So what we need to do to implement this is to have a constructor in the dependent class that will take the concrete class object and assign it to the interface handle this class is using. So if we need to implement this for our **AppPoolWatcher** class:

```
class AppPoolWatcher
{
    // Handle to EventLog writer to write to the logs
    INofificationAction action = null;

    public AppPoolWatcher(INofificationAction concreteImplementation)
    {
        this.action = concreteImplementation;
    }

    // This function will be called when the app pool has problem
    public void Notify(string message)
    {
        action.ActOnNotification(message);
    }
}
```

In the above code, the constructor will take the concrete class object and bind it to the interface handle. So if we need to pass the **EventLogWriter**'s concrete implementation into this class, all we need to do is

```
EventLogWriter writer = new EventLogWriter();
AppPoolWatcher watcher = new AppPoolWatcher(writer);
watcher.Notify("Sample message to log");
```

Now if we want this class to send email or sms instead, all we need to do is to pass the object of the respective class in the **AppPoolWatcher**'s constructor. This method is useful when we know that the instance of the dependent class will use the same concrete class for its entire lifetime.

Method Injection

In constructor injection we saw that the dependent class will use the same concrete class for its entire lifetime. Now if we need to pass separate concrete class on each invocation of the method, we have to pass the dependency in the method only.

So in method injection approach we pass the object of the concrete class into the method the dependent class which is actually invoking the action. So what we need to do to implement this is to have the action function also accept an argument for the concrete class object and assign it to the interface handle this class is using and invoke the action. So if we need to implement this for our **AppPoolWatcher** class:

```
class AppPoolWatcher
{
```

```
// Handle to EventLog writer to write to the logs
INofificationAction action = null;

// This function will be called when the app pool has problem
public void Notify(INofificationAction concreteAction, string message)
{
    this.action = concreteAction;
    action.ActOnNotification(message);
}
}
```

In the above code the action method i.e. **Notify** will take the concrete class object and bind it to the interface handle. So if we need to pass the **EventLogWriter**'s concrete implementation into this class, all we need to do is

```
EventLogWriter writer = new EventLogWriter();
AppPoolWatcher watcher = new AppPoolWatcher();
watcher.Notify(writer, "Sample message to log");
```

Now if we want this class to send email or sms instead, all we need to do is to pass the object of the respective class in the **AppPoolWatcher**'s invocation method i.e. **Notify** method in the above example.

Property Injection

Now we have discussed two scenarios where in constructor injection we knew that the dependent class will use one concrete class for the entire lifetime. The second approach is to use the method injection where we can pass the concrete class object in the action method itself. But what if the responsibility of selection of concrete class and invocation of method are in separate places. In such cases we need property injection.

So in this approach we pass the object of the concrete class via a setter property that was exposed by the dependent class. So what we need to do to implement this is to have a Setter property or function in the dependent class that will take the concrete class object and assign it to the interface handle this class is using. So if we need to implement this for our **AppPoolWatcher** class:

```
class AppPoolWatcher
{
    // Handle to EventLog writer to write to the logs
    INofificationAction action = null;

    public INofificationAction Action
    {
        get
        {
            return action;
        }
        set
        {
            action = value;
        }
    }

    // This function will be called when the app pool has problem
    public void Notify(string message)
    {

```

```

        action.ActOnNotification(message);
    }
}

```

In the above code the setter of Action property will take the concrete class object and bind it to the interface handle. So if we need to pass the **EventLogWriter's** concrete implementation into this class, all we need to do is

```

EventLogWriter writer = new EventLogWriter();
AppPoolWatcher watcher = new AppPoolWatcher();
// This can be done in some class
watcher.Action = writer;

// This can be done in some other class
watcher.Notify("Sample message to log");

```

Now if we want this class to send email or sms instead, all we need to do is to pass the object of the respective class in the setter exposed by **AppPoolWatcher** class. This approach is useful when the responsibility of selecting the concrete implementation and invoking the action are done in separate places/modules.

In languages where properties are not supported, there is a separate function to set the dependency. This approach is also known as setter injection. The important thing to note in this approach is that there is a possibility that someone has created the dependent class but no one has set the concrete class dependency yet. If we try to invoke the action in such cases then we should have either some default dependency mapped to the dependent class or have some mechanism to ensure that application will behave properly.

A Note on IoC Containers

Constructor injection is the mostly used approach when it comes to implementing the dependency injection. If we need to pass different dependencies on every method call then we use method injection. Property injection is used less frequently.

All the three approaches we have discussed for dependency injection are ok if we have only one level of dependency. But what if the concrete classes are also dependent of some other abstractions. So if we have chained and nested dependencies, implementing dependency injection will become quite complicated. That is where we can use IoC containers. IoC containers will help us to map the dependencies easily when we have chained or nested dependencies.

Point of interest

In this article we have discussed about Dependency inversion principle(Which is the D part of SOLID object oriented principle). We have also discussed how inversion of control is used to implement dependency inversion and finally we have seen how dependency injection help in creating decoupled classes and how to implement dependency injection. This article has been written from beginner's perspective. I hope this has been informative.

History

- **03 July 2013:** First version.
- **09 July 2013:** Typos and few grammatical mistakes fixed.


License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author



Rahul Rajat Singh


Software Developer (Senior)
Headfitted Solutions
India 

I Started my Programming career with C++. Later got a chance to develop Windows Form applications using C#. Currently using C#, ASP.NET & ASP.NET MVC to create Information Systems, e-commerce/e-governance Portals and Data driven websites.

My interests involves Programming, Website development and Learning/Teaching subjects related to Computer Science/Information Systems. IMO, C# is the best programming language and I love working with C# and other Microsoft Technologies.

Follow on  Twitter  Google

Comments and Discussions

 **51 messages** have been posted for this article Visit <http://www.codeproject.com/Articles/615139/An-Absolute-Beginners-Tutorial-on-Dependency-Inver> to post and view comments on this article, or click [here](#) to get a print view with messages.

