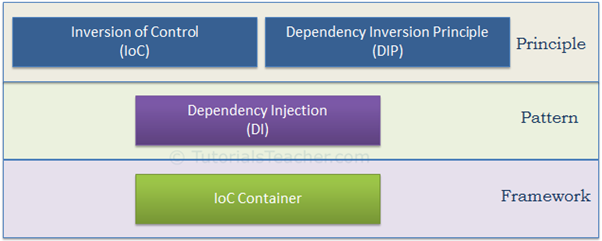
Unity Framework

Topics to be Discuss

* The terms Inversion of Control (IoC),
* Dependency Inversion Principle (DIP),
* Dependency Injection (DI),
* IoC containers



IoC and DIP are high level design principles which should be used while designing application classes. As they are principles, they recommend certain best practices but do not provide any specific implementation details. Dependency Injection (DI) is a pattern and IoC container is a framework.

Inversion Of Control(IOC)

IoC is a design principle which recommends the inversion of different kinds of controls in object-oriented design to achieve loose coupling between application classes. In this case, control refers to any additional responsibilities a class has, other than its main responsibility, such as control over the flow of an application, or control over the dependent object creation and binding (Remember SRP - Single Responsibility Principle)

Dependency Inversion of Control(DIC)

The DIP principle also helps in achieving loose coupling between classes. It is highly recommended to use DIP and IoC together in order to achieve loose coupling.

DIP suggests that high-level modules should not depend on low level modules. Both should depend on abstraction.

The DIP principle was invented by [Robert Martin](https://en.wikipedia.org/wiki/Robert_Cecil_Martin) (a.k.a. Uncle Bob). He is a founder of the SOLID principles.

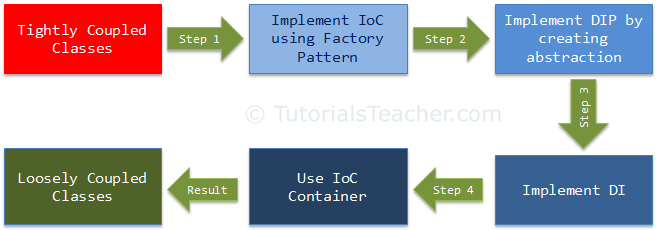
Dependency Injection

Dependency Injection (DI) is a design pattern which implements the IoC principle to invert the creation of dependent objects. We will learn about it in the [DI](https://www.tutorialsteacher.com/ioc/dependency-injection) chapter.

IOC Container

The IoC container is a framework used to manage automatic dependency injection throughout the application, so that we as programmers do not need to put more time and effort into it. There are various IoC Containers for .NET, such as [Unity](https://github.com/unitycontainer/unity), [Ninject](https://github.com/ninject" \t "_blank), [StructureMap](http://structuremap.github.io/" \t "_blank), [Autofac](https://autofac.org/" \t "_blank).

We cannot achieve loosely coupled classes by using IoC alone. Along with IoC, we also need to use DIP, DI and IoC container. The following figure illustrates how we are going to achieve loosely coupled design step by step



Inversion of Control (IoC) is a design principle (although, some people refer to it as a pattern). As the name suggests, it is used to invert different kinds of controls in object-oriented design to achieve loose coupling. Here, controls refer to any additional responsibilities a class has, other than its main responsibility. This include control over the flow of an application, and control over the flow of an object creation or dependent object creation and binding.

IoC is all about inverting the control. To explain this in layman's terms, suppose you drive a car to your work place. This means you control the car. The IoC principle suggests to invert the control, meaning that instead of driving the car yourself, you hire a cab, where another person will drive the car. Thus, this is called inversion of the control - from you to the cab driver. You don't have to drive a car yourself and you can let the driver do the driving so that you can focus on your main work.

The IoC principle helps in designing loosely coupled classes which make them testable, maintainable and extensible.

Let's understand how IoC inverts the different kinds of control.

In an object-oriented design, classes should be designed in a loosely coupled way. Loosely coupled means changes in one class should not force other classes to change, so the whole application can become maintainable and extensible. Let's understand this by using typical n-tier architecture as depicted by the following figure:

[](https://www.tutorialsteacher.com/Content/images/ioc/demo-architecture.png)

In the typical n-tier architecture, the User Interface (UI) uses Service layer to retrieve or save data. The Service layer uses the BusinessLogic class to apply business rules on the data. The BusinessLogic class depends on the DataAccess class which retrieves or saves the data to the underlying database. This is simple n-tier architecture design. Let's focus on the BusinessLogic and DataAccess classes to understand IoC.

Example:

public class CustomerBusinessLogic

{

DataAccess \_dataAccess;

public CustomerBusinessLogic()

{

\_dataAccess = new DataAccess();

}

public string GetCustomerName(int id)

{

return \_dataAccess.GetCustomerName(id);

}

}

public class DataAccess

{

public DataAccess()

{

}

public string GetCustomerName(int id) {

return "Dummy Customer Name"; // get it from DB in real app

}

}

Now, let's understand what's wrong with the above classes.

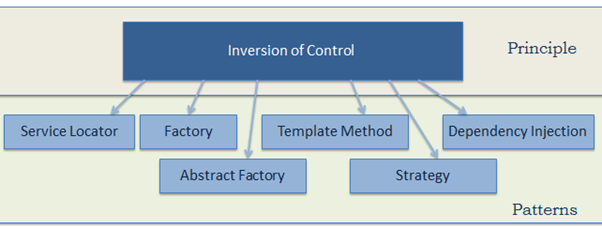
In the above example, CustomerBusinessLogic and DataAccess are tightly coupled classes because the CustomerBusinessLogic class includes the reference of the concrete DataAccess class. It also creates an object of DataAccess class and manages the lifetime of the object.

Problems in the above example classes:

1. CustomerBusinessLogic and DataAccess classes are tightly coupled classes. So, changes in the DataAccess class will lead to changes in the CustomerBusinessLogicclass. For example, if we add, remove or rename any method in the DataAccessclass then we need to change the CustomerBusinessLogic class accordingly.
2. Suppose the customer data comes from different databases or web services and, in the future, we may need to create different classes, so this will lead to changes in the CustomerBusinessLogic class.
3. The CustomerBusinessLogic class creates an object of the DataAccess class using the **new** keyword. There may be multiple classes which use the DataAccess class and create its objects. So, if you change the name of the class, then you need to find all the places in your source code where you created objects of DataAccess and make the changes throughout the code. This is repetitive code for creating objects of the same class and maintaining their dependencies.
4. Because the CustomerBusinessLogic class creates an object of the concrete DataAccess class, it cannot be tested independently (TDD). The DataAccess class cannot be replaced with a mock class.

To solve all of the above problems and get a loosely coupled design, we can use the IoC and DIP principles together. Remember, IoC is a principle, not a pattern. It just gives high-level design guidelines but does not give implementation details. You are free to implement the IoC principle the way you want.

The following pattern (but not limited) implements the IoC principle.

[](https://www.tutorialsteacher.com/Content/images/ioc/ioc-patterns.png)

Let's use the *Factory* pattern to implement IoC in the above example, as the first step towards attaining loosely coupled classes.

Use Factory Class to Retrieve Object

static clsTimeMachine \_clsTimeMachine;

static clsForcast \_clsForcast;

public clsWeatherFactory()

{

}

public clsWeatherFactory(clsTimeMachine tm, clsForcast fc)

{

\_clsTimeMachine = tm;

\_clsForcast = fc;

}

static public Iweather getData(WeatherData weatherDataType)

{

Iweather \_iweather;

if (weatherDataType.WeatherType == "TM")

{

\_iweather = \_clsTimeMachine;

}

else if (weatherDataType.WeatherType == "FC")

{

\_iweather = \_clsForcast;

}

else

{

return null;

}

return \_iweather;

}

This is a simple implementation of IoC and the first step towards achieving fully loose coupled design

How to implement the Dependency Inversion Principle as the second step to achieve loosely coupled classes.

DIP Definition

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

The second rule in DIP is "Abstractions should not depend on details. Details should depend on abstractions"

Abstraction and encapsulation are important principles of object-oriented programming. There are many different definitions from different people, but let's understand abstraction using the above example.

In English, abstraction means something which is non-concrete. In programming terms, the above CustomerBusinessLogic and DataAccess are concrete classes, meaning we can create objects of them. So, abstraction in programming means to create an interface or an abstract class which is non-concrete. This means we cannot create an object of an interface or an abstract class. As per DIP, CustomerBusinessLogic (high-level module) should not depend on the concrete DataAccess class (low-level module). Both classes should depend on abstractions, meaning both classes should depend on an interface or an abstract class.

Iweather \_iWeather;

WeatherData \_weatherData = new WeatherData();

\_weatherData.Lat = lat;//19.99;

\_weatherData.Log = log;// 73.78;

\_weatherData.DT = DateTime.Now;

\_weatherData.WeatherType = "TM";

\_iWeather = clsWeatherFactory.getData(\_weatherData);

Need to create Interface for that. So, that the Data access layes is not tightly coupled with the Business Logic Class.

public interface Iweather

{

string getData(WeatherData \_weatherData);

}

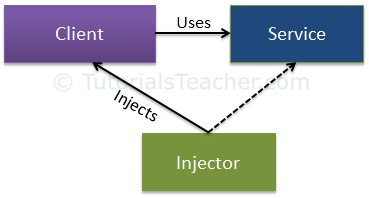
Dependency Injection

 Here, we are going to implement Dependency Injection and strategy pattern together to move the dependency object creation completely out of the class. This is our third step in making the classes completely loose coupled.

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

1. **Client Class:** The client class (dependent class) is a class which depends on the service class
2. **Service Class:** The service class (dependency) is a class that provides service to the client class.
3. **Injector Class:** The injector class injects the service class object into the client class.



Constructor Injection

As mentioned before, when we provide the dependency through the constructor, this is called a constructor injection.

Property Injection

In the property injection, the dependency is provided through a public property

Method Injection

In the method injection, dependencies are provided through methods. This method can be a class method or an interface method.

The IoC container creates an object of the specified class and also injects all the dependency objects through a constructor, a property or a method at run time and disposes it at the appropriate time. This is done so that we don't have to create and manage objects manually.

All the containers must provide easy support for the following DI lifecycle.

* **Register:** The container must know which dependency to instantiate when it encounters a particular type. This process is called registration. Basically, it must include some way to register type-mapping.
* **Resolve:** When using the IoC container, we don't need to create objects manually. The container does it for us. This is called resolution. The container must include some methods to resolve the specified type; the container creates an object of the specified type, injects the required dependencies if any and returns the object.
* **Dispose:** The container must manage the lifetime of the dependent objects. Most IoC containers include different lifetimemanagers to manage an object's lifecycle and dispose it.

Unity Framework:

Unity Container Features:

* Simplified type-mapping registration for interface type or base type.
* Supports registration of an existing instance.
* Supports code-based registration as well as design time registration.
* Automatically injects registered type at runtime through a constructor, a property or a method.
* Supports deferred resolution.
* Supports nested containers.
* Automatic disposing of instances based on lifetime managers; lifetime managers include hierarchical, per resolve, externally controlled, per request and per thread.
* Supports service location capability; this allows clients to store or cache the container.
* Supports type interception and instance interception.
* Easy to extend.

Steps to implement Unity Framework

1. Install Unity with the help of nugget Manager.
2. we need to register the type-mapping with the container, so that it can create the correct object for the given type. Use the RegisterType() method to register a type mapping. Basically, it configures which class to instantiate for which interface or base class.

UnityContainer \_unity = new UnityContainer();

\_unity.RegisterType<Iweather, clsForcast>();

1. Unity creates an object of the specified class and automatically injects the dependencies using the resolve() method.

clsProcessData \_clsProcessData = \_unity.Resolve<clsProcessData>();

By default, Resolve<T>() performs construction injection to inject dependencies and returns an object of the specified type.

public clsWeatherFactory(clsTimeMachine tm, clsForcast fc)

{

\_clsTimeMachine = tm;

\_clsForcast = fc;

}

Property injection is a type of dependency injection where dependencies are provided through properties.

Example:

public interface ICar

{

int Run();

}

public class BMW : ICar

{

private int \_miles = 0;

public int Run()

{

return ++\_miles;

}

}

public class Ford : ICar

{

private int \_miles = 0;

public int Run()

{

return ++\_miles;

}

}

public class Audi : ICar

{

private int \_miles = 0;

public int Run()

{

return ++\_miles;

}

}

public class Driver

{

public Driver()

{

}

[Dependency]

public ICar Car { get; set; }

public void RunCar()

{

Console.WriteLine("Running {0} - {1} mile ",

this.Car.GetType().Name, this.Car.Run());

}

}

Property injection in Unity container can be implemented in two ways:

1. Using the [Dependency] attribute
2. Using run-time configuration

For the property injection, we first tell the Unity container which property to inject. So, we need to decorate the dependent properties with the [Dependency] attribute.

public class Driver

{

public Driver()

{

}

[Dependency]

public ICar Car { get; set; }

public void RunCar()

{

Console.WriteLine("Running {0} - {1} mile ", this.Car.GetType().Name, this.Car.Run());

}

}

Now, we can register the ICar type and resolve it as shown below.

var container = new UnityContainer();

container.RegisterType<ICar, BMW>();

var driver = container.Resolve<Driver>();

driver.RunCar();

We can specify a name in the [Dependency("name")] attribute, which can then be used to set the property value.

public class Driver

{

public Driver()

{

}

[Dependency("LuxuryCar")]

public ICar Car { get; set; }

public void RunCar()

{

Console.WriteLine("Running {0} - {1} mile ", this.Car.GetType().Name, this.Car.Run());

}

}

We can Resolve it as follows:

var container = new UnityContainer();

container.RegisterType<ICar, BMW>();

container.RegisterType<ICar, Audi>("LuxuryCar");

var driver = container.Resolve<Driver>();

driver.RunCar();

Method Injection

In the method injection, dependencies are provided through method parameters.

Method injection in Unity can be implemented in two ways:

1. Using the [InjectionMethod] attribute
2. Using run-time configuration

For the method injection, we need to tell the Unity container which method should be used for dependency injection. So, we need to decorate a method with the [InjectionMethod] attribute as shown in the following Driver class.

public class Driver

{

private ICar \_car = null;

public Driver()

{

}

[InjectionMethod]

public void UseCar(ICar car) {

\_car = car;

}

public void RunCar()

{

Console.WriteLine("Running {0} - {1} mile ", \_car.GetType().Name, \_car.Run());

}

}

var container = new UnityContainer();

container.RegisterType<ICar, BMW>();

var driver = container.Resolve<Driver>();

driver.RunCar();

## Run-time Configuration

Unity container allows us to configure method injection with the RegisterType()method if a method is not marked with the [InjectionMethod] attribute. Pass an object of the [InjectionMethod](https://msdn.microsoft.com/en-us/library/microsoft.practices.Unity.injectionmethod.aspx" \o "InjectionMethod on MSDN" \t "_blank) class in the RegisterType() method to specify a method name and a parameter value.

Note : The [InjectionMethod](https://msdn.microsoft.com/en-us/library/microsoft.practices.Unity.injectionmethod.aspx" \o "InjectionMethod on MSDN" \t "_blank) is derived from the [InjectionMember Class](https://msdn.microsoft.com/en-us/library/microsoft.practices.Unity.injectionmember.aspx" \o "InjectionMember" \t "_blank). The InjectionMember is an abstract class which can be used to configure the injection type. There are three subclasses of InjectionMembers: InjectionConstruction to configure construction injection, InjectionProperty to configure property injection and InjectionMethod to configure method injection.

var container = new UnityContainer();

//run-time configuration

container.RegisterType<Driver>(new InjectionMethod("UseCar", new Audi()));

//to specify multiple parameters values

container.RegisterType<Driver>(new InjectionMethod("UseCar", new object[] { new Audi() }));

var driver = container.Resolve<Driver>();

driver.RunCar();