# Advanced Programming & Design Patterns (C#)

## Dependency Injection

Dependency injection is a software design pattern that implements inversion of control for resolving dependencies. A dependency is an object that can be used (a service).

The dependency injection pattern, also known as Inversion of Control, is one of the most popular design paradigms today. It facilitates the design and implementation of loosely coupled, reusable, and testable objects in your software designs by removing dependencies that often inhibit reuse. Dependency injection can help you design your applications so that the architecture links the components rather than the components linking themselves.

class EventLogger

{

public void Write(string message)

{

//Write to event log here

}

}

class Employee

{

// Handle to EventLog writer to write to the logs

EventLogger logger = null;

// This function will be called when the app pool has problem

public void Notify(string message)

{

if (logger == null)

{

logger = new EventLogger();

}

logger.Write(message);

}

}

This design violates the dependency inversion principle. i.e. the high level module Employee depends on EventLogger which is a concrete class and not an abstraction.

public class Employee

{

public Employee(ILogger logger)

{

logger.WriteToLog("New employee created");

}

}

public interface ILogger

{

void WriteToLog(string text);

}

public class EventLogger: ILogger

{

public void WriteToLog(string text)

{

Console.WriteLine(text);

}

}

public class DBLogger : ILogger

{

public void WriteToLog(string text)

{

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\n {0}\n\*\*\*\*\*\*\*\*\*\*\*", text);

}

}

Employee employee1 = new Employee(new EventLogger());

Employee employee2 = new Employee(new DBLogger());

## Types of Dependency Injection

There are three common forms of dependency injection:

1. Constructor Injection
2. Setter Injection
3. Interface-based injection

Implementing Constructor Injection  
Consider a design with two layers; a BusinessFacade layer and the BusinessLogic layer. The BusinessFacade layer of the application depends on the BusinessLogic layer to operate properly. All the business logic classes implement an IBusinessLogic interface.  
  
With constructor injection, you'd create an instance of the BusinessFacade class using its argument or parameterized constructor and pass the required BusinessLogic type to inject the dependency. The following code snippet illustrates the concept, showing the BusinessLogic and BusinessFacade classes.

interface IBusinessLogic

{

//Some code

}

class ProductBL : IBusinessLogic

{

//Some code

}

class CustomerBL : IBusinessLogic

{

//Some code

}

public class BusinessFacade

{

private IBusinessLogic businessLogic;

public BusinessFacade(IBusinessLogic businessLogic)

{

this.businessLogic = businessLogic;

}

}

You'd instantiate the BusinessLogic classes (ProductBL or CustomerBL) as shown below:

IBusinessLogic productBL = new ProductBL();

Then you can pass the appropriate type to the BusinessFacade class when you instantiate it:

BusinessFacade businessFacade = new BusinessFacade(productBL);

## Implementing Setter Injection

Setter injection uses properties to inject the dependencies, which lets you create and use resources as late as possible. It's more flexible than constructor injection because you can use it to change the dependency of one object on another without having to create a new instance of the class or making any changes to its constructor.

public class BusinessFacade

{

private IBusinessLogic businessLogic;

public IBusinessLogic BusinessLogic

{

get

{

return businessLogic;

}

set

{

businessLogic = value;

}

}

}

The following code snippet illustrates to implement setter injection using the BusinessFacade class shown above.

IBusinessLogic productBL = new ProductBL();

BusinessFacade businessFacade = new BusinessFacade();

businessFacade.BusinessLogic = productBL;

## Implementing Interface Injection

The following code shows an example in which the classes use the IBusinessLogic interface as a base contract to inject an instance of any of the business logic classes (ProductBL or CustomerBL) into the BusinessFacade class. Both the business logic classes ProductBL and CustomerBL implement the IBusinessLogic interface:

interface IBusinessLogic

{

//Some code

}

class ProductBL : IBusinessLogic

{

//Some code

}

class CustomerBL : IBusinessLogic

{

//Some code

}

class BusinessFacade

{

private IBusinessLogic businessLogic;

public void SetBLObject(IBusinessLogic businessLogic)

{

this.businessLogic = businessLogic;

}

}

IBusinessLogic businessLogic = new ProductBL();

BusinessFacade businessFacade = new BusinessFacade();

businessFacade.SetBLObject(businessLogic);

Or:

IBusinessLogic businessLogic = new CustomerBL();

BusinessFacade businessFacade = new BusinessFacade();

businessFacade.SetBLObject(businessLogic);

## Unity Framework

//// Register IGame so when dependecy is detected

//// it provides a UnitedMyanmar instance

unityContainer.RegisterType<IGame, UnitedMyanmar>();

// Make Unity resolve the interface, providing an instance

// of UnitedMyanmar class

//Type resolution

//Unity to resolve an IGame interface, the framework will check if the IGame interface is //registered, and, in possitive match, it will provide an instance of the mapped class //(in this case, UnitedMyanmar):

var game = unityContainer.Resolve<IGame>();

game.addPlayer();

game.addPlayer();

Console.WriteLine(string.Format("{0} play at {1}", game.CurrentPlayers, game.Name));

Console.ReadLine();

+++++++++++++++++++++++++++++++++++++++

// Instance a GameOrg class object through Unity

//Unity is also able to resolve types indirectly.

//IGame is registered to inject a UnitedMyanmar instance.

//gameOrg (instance from GameOrg class) got a reference to an IGame interface, which was //injected through its constructor.

var gameOrg = unityContainer.Resolve<GameOrg>();

gameOrg.AddPlayer();

gameOrg.AddPlayer();

gameOrg.Play();

Console.WriteLine(table.GameStatus());

Console.ReadLine();

+++++++++++++++++++++++++++++++++++++++

## Property injection

Inject a property when dependency is resolved

InjectionProperty injectionProperty = new InjectionProperty("Name", "Great Ball of modern myanmar.");

unityContainer.RegisterType<IGame, UnitedMyanmar>(injectionProperty);

var game = unityContainer.Resolve<IGame>();

game.addPlayer();

game.addPlayer();

Console.WriteLine(string.Format("{0} play at {1}", game.CurrentPlayers, game.Name));

Console.ReadLine();

+++++++++++++++++++++++++++++++++++++++

## Parameter injection

Another possibility that we have when we inject a dependency is making the injection just in the moment we are resolving it.

Until now we've seen the case where the default instanced game is UnitedMyanmar. However, there may be cases where we want IGame to be assigned to another game. We can ask Unity to provide us an instance of TicTacToe just injecting the constructor parameters when the dependency is resolved.

//Override the constructor parameter of GameOrg class

var gameOrg2 = unityContainer.Resolve<GameOrg>(new ParameterOverride("game", new YoungUprising()));