Dependency Injection

The dependency injection pattern, also known as Inversion of Control, is one of the most popular design paradigms today..

Tight coupling

Consider two classes, C1 and C2, where C2 is tightly coupled with C1 and requires it to operate. In this case C2 is dependent on C1, as shown below:

```
public class C1
{
    //Some code
}
public class C2
{
    C1 c1Object = new C1();
    //Some code
}
```

Dependency Injection

Dependency injection eliminates tight coupling between objects to make both the objects and applications that use them more flexible, reusable, and easier to test. It facilitates the creation of loosely coupled objects and their dependencies. The basic idea behind Dependency Injection is that you should isolate the implementation of an object from the construction of objects on which it depends. Dependency Injection is Inversion of Control Pattern where a factory object carries the responsibility for object creation and linking. The factory object ensures loose coupling between the objects and promotes seamless testability.

Advantages Dependency Injection

Loose coupling
Centralized configuration
Easily testable

Code becomes more testable because it abstracts and isolates class dependencies.

Disadvantages of Dependency Injection

Wiring instances together can become a nightmare if there are too many instances and many dependencies that need to be addressed.

Types of Dependency Injection

There are three common forms of dependency injection:
Constructor Injection
Setter Injection
Interface-based injection

Constructor injection uses parameters to inject dependencies.

Setter injection uses setters to inject dependencies.

Interface-based injection uses interface to inject dependencies.

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 parameterized constructor and pass the required BusinessLogic type to inject the dependency.

```
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);
```

Note that you can pass an instance of either BusinssLogic class to the BusinessFacade class constructor. The constructor does not accept a concrete object; instead, it accepts any class that implements the IBusinessLogic interface.

Even though it is flexible and promotes loose coupling, the major drawback of constructor injection is that **once the class is instantiated, you can no longer change the object's dependency**. Further, because you can't inherit constructors, **any derived classes call a base class constructor to apply the dependencies properly**. Fortunately, you can overcome this drawback using the setter injection technique.

Implementing Setter Injection

Setter injection 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;
        }
    }
}

IBusinessLogic productBL = new ProductBL();
BusinessFacade businessFacade = new BusinessFacade();
businessFacade.BusinessLogic = productBL;
```

The preceding code snippet uses the `BusinessLogic` property of the BusinessFacade class to set its dependency on the BusinessLogic type. The primary advantage of this design is that you can change the dependency between the BusinessFacade and the instance of BusinessLogic even after instantiating the BusinessFacade class.

Even though setter injection is a good choice, its primary drawback is that an **object with setters cannot be immutable** - and it can be **difficult to identify which dependencies are needed, and when**. You should normally choose constructor injection over setter injection unless you need to **change the dependency after instantiating an object instance**, or **cannot change constructors** and recompile.

Implementing Interface Injection

You accomplish the last type of dependency injection technique, interface injection, by using a common interface that other classes need to implement to inject dependencies.

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.

```
interface IBusinessLogic
{
    //Some code
}
class ProductBL : IBusinessLogic
{
    //Some code
}
class CustomerBL : IBusinessLogic
{
    //Some code
}

interface IBusinessFacade
{
    void SetBLObject(IBusinessLogic businessLogic);
}

class BusinessFacade : IBusinessFacade
{
    private IBusinessLogic businessLogic;
    public void SetBLObject(IBusinessLogic businessLogic)
    {
        this.businessLogic = businessLogic;
    }
}
```

The following code shows how you'd call the `SetBLObject()` method to inject a dependency for either type of BusinessLogic class:

```
IBusinessLogic product = new ProductBL();
BusinessFacade businessFacade = new BusinessFacade();
businessFacade.SetBLObject(product);
```

Or

```
IBusinessLogic customer = new CustomerBL();
BusinessFacade businessFacade = new BusinessFacade();
businessFacade.SetBLObject(customer);
```

Coding to well-defined interfaces, particularly when using the dependency injection pattern, is the key to achieving loose coupling. **By coupling an object to an interface instead of a specific implementation, you have the ability to use any implementation with minimal change and risk**.