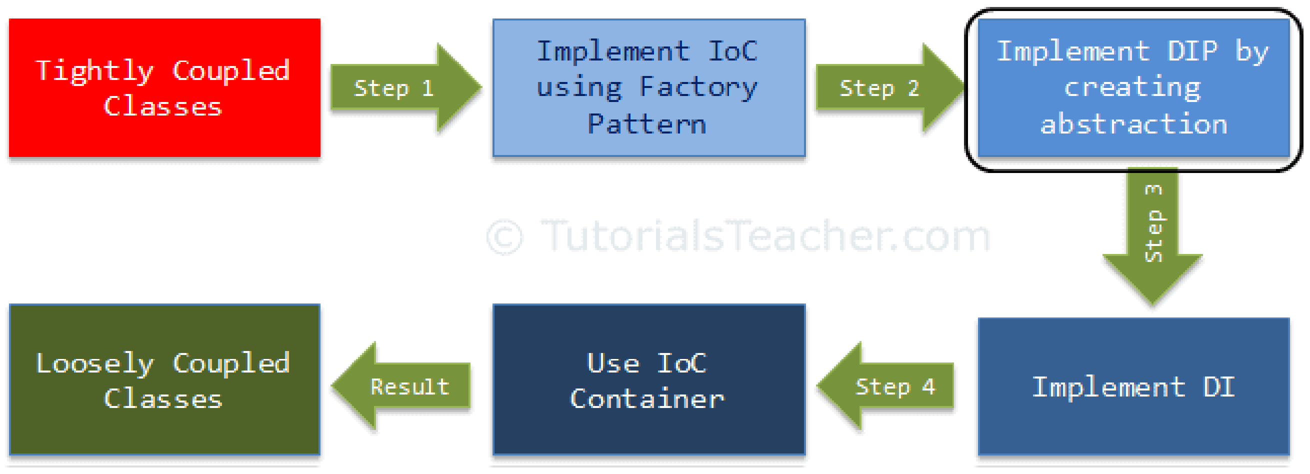
Dependency Inversion Principle

In the previous chapter, we learned about implementing the IoC principle using the Factory pattern and achieved the first level of loosely coupled design. Here, we will learn how to implement the Dependency Inversion Principle as the second step to achieve loosely coupled classes.



First, let's understand what is Dependency Inversion Principle (DIP)?

DIP is one of the SOLID object-oriented principle invented by Robert Martin (a.k.a. Uncle Bob)

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.

To understand DIP, let's take an example from the previous chapter, as shown below.

```
public class CustomerBusinessLogic
{
    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        DataAccess _dataAccess = DataAccessFactory.GetDataAccessObj();

        return _dataAccess.GetCustomerName(id);
    }
}

public class DataAccessFactory
{
    public static DataAccess GetDataAccessObj()
    {
        return new DataAccess();
    }
}

public class DataAccess
{
    public DataAccess()
    {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name"; // get it from DB in real app
    }
}
```

In the above example, we implemented the factory pattern to achieve IoC. But, the `CustomerBusinessLogic` class uses the concrete `DataAccess` class. Therefore, it is still tightly coupled, even though we have inverted the dependent object creation to the factory class.

Let's use DIP on the `CustomerBusinessLogic` and `DataAccess` classes and make them more loosely coupled.

As per the DIP definition, a high-level module should not depend on low-level modules. Both should depend on abstraction. So, first, decide which is the high-level module (class) and the low-level module. A high-level module is a module which depends on other modules. In our example, `CustomerBusinessLogic` depends on the `DataAccess` class, so `CustomerBusinessLogic` is a high-level module and `DataAccess` is a low-level module. So, as per the first rule of DIP, `CustomerBusinessLogic` should not depend on the concrete `DataAccess` class, instead both classes should depend on abstraction.

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

What is an Abstraction?

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.

Now, what should be in the interface (or in the abstract class)? As you can see, `CustomerBusinessLogic` uses the `GetCustomerName()` method of the `DataAccess` class (in real life, there will be many customer-related methods in the `DataAccess` class). So, let's declare the `GetCustomerName(int id)` method in the interface, as shown below.

```
public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}
```

Now, implement `ICustomerDataAccess` in the `CustomerDataAccess` class, as shown below (so, instead of the `DataAccess` class, let's define the new `CustomerDataAccess` class).

```
public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess()
    {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name";
    }
}
```

Now, we need to change our factory class which returns `ICustomerDataAccess` instead of the concrete `DataAccess` class, as shown below.

```
public class DataAccessFactory
{
    public static ICustomerDataAccess GetCustomerDataAccessObj()
    {
        return new CustomerDataAccess();
    }
}
```

Now, change the `CustomerBusinessLogic` class which uses `ICustomerDataAccess` instead of the concrete `DataAccess`, class as shown below.

```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

    public CustomerBusinessLogic()
    {
        _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
    }

    public string GetCustomerName(int id)
    {
        return _custDataAccess.GetCustomerName(id);
    }
}
```

Thus, we have implemented DIP in our example where a high-level module (CustomerBusinessLogic) and low-level module (CustomerDataAccess) are dependent on an abstraction (ICustomerDataAccess). Also, the abstraction (ICustomerDataAccess) does not depend on details (CustomerDataAccess), but the details depend on the abstraction.

The following is the complete DIP example discussed so far.

Example: DIP Implementation - C# Copy

```
public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess() {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name";
    }
}

public class DataAccessFactory
{
    public static ICustomerDataAccess GetCustomerDataAccessObj()
    {
        return new CustomerDataAccess();
    }
}

public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

    public CustomerBusinessLogic()
    {
        _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
    }

    public string GetCustomerName(int id)
    {
        return _custDataAccess.GetCustomerName(id);
    }
}
```

The advantages of implementing DIP in the above example is that the `CustomerBusinessLogic` and `CustomerDataAccess` classes are loosely coupled classes because `CustomerBusinessLogic` does not depend on the concrete `DataAccess` class, instead it includes a reference of the `ICustomerDataAccess` interface. So now, we can easily use another class which implements `ICustomerDataAccess` with a different implementation.

Still, we have not achieved fully loosely coupled classes because the `CustomerBusinessLogic` class includes a factory class to get the reference of `ICustomerDataAccess`. This is where the Dependency Injection pattern helps us. In the next chapter, we will learn how to use the Dependency Injection (DI) and the Strategy pattern using the above example.

‹ Previous Next ›