# Dependency injection and inversion of control

Dependency Injection (DI) is an object-oriented programming design pattern that allows us to develop loosely coupled code. DI helps in getting rid of tightly coupled software components. The purpose of DI is to make code maintainable and easy to update.  
The rule behind dependency injection is that any component (dependency) needed by another component should be passed (or ‘injected’) to the component, possibly via its constructor, so that dependency requirement is clear and evident, since is not possible to use the component if related dependencies are missing.

Let’s take this example

public class Customer

{

private DatabaseHelper helper;

public Customer()

{

helper = new DatabaseHelper();

}

...

...

}

In this case the consumer of customer class doesn’t know that internally the DatabaseHelper class is needed and he could presume that he can just reuse the class by copying it into another project and the discover that DatabaseHelper and maybe other internally created classes are required as well. Another issue of this code is that if, as example, DatabaseHelper is able to connect to a SQL server database, if in future we need to connect to a different kind of database the code inside customer could become a mess.  
A cleaner solution would be to define an interface that defines the contract between Customer and its storage and pass it into Customer’s constructor, this way, and user of Customer class will immediately know that, in order to have Customer working he need to pass it an instance of some class who implements that interface

public class Customer

{

private IStorageHelper helper;

public Customer(IStorageHelper helper)

{

this.helper = helper;

}

...

...

}

This way, if we need to use a different storage is just a matter of changing the injected DatabaseHelper and the rest will work in the same way, this also allows Customer class to be tested by passing a mocked version of the interface (more about mocking here <https://spin.atomicobject.com/2017/08/07/intro-mocking-moq/>)

# Inversion of control (IoC)

Normally, the flow of the program logic is determined by objects that are bound to one another. With the inversion of control, the flow depends on the defined abstractions to be implemented that is built up during program execution. In IoC, the code could also be linked statically during compilation to the defined interface functions, but finding the implementation of the function to execute by reading its description from external configuration instead of with a direct reference in the code itself, all this allows the creation of runtime configurable option.  
Imagine an app where the user at startup can decide whether to use SQL Server or MySql as storage service, using IoC the logic flow is determined at runtime with something like:

IDatabaseHelper helper=null;

If (userselection==sql)  
{

IDatabaseHelper=new SqlDatabaseHelper();

}  
else  
{

IDatabaseHelper=new MySqlDatabaseHelper();

}  
  
Customer c=new Customer(IDatabaseHelper);  
…  
…

Of course, things can become complicated if a component needs a lot of dependencies and each of this dependency also has its own set of dependencies and so on, instantiating a component can become a though task, that’s why normally Ioc is implemented using Ioc container that is a component who takes care of creating all required instances by taking care of instantiating any required dependency as well.  
Use of these components varies depending on the one used, but generally the use is divided into two steps

1-Definition of how different types must be instantiated

2-Ask IoC container to create an instance of a type.

And abstract example could be:

*MyIocContainer.Register<IDatabaseHelper,SqlDatabase>();*

*MyIocContainer.Register<Customer>();*

*var customer= MyIoCContainer.GetInstance<Customer>();  
…  
…*  
and the IoC container will take care of instantiating the IDataBaseInstance required by Customer and it will return a brand new instance to us.  
Of course each IoC container has a lot of configuration option to control how various instances should be created.

A more comprehensive description can be found here <https://msdn.microsoft.com/en-us/magazine/jj991965.aspx>

# Conclusion

There’s no doubt that defining object creation strategy in one place and delegate instances creation to a well-known engine makes application clean, testable and less error prone.