DEPENDENCY INJECTION

Dependency Inversion Principle

DIP provides high-level guidance to make your code loosely coupled. It says the following:

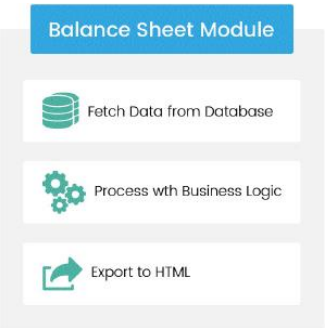
- High-level modules should not depend on low-level modules for their responsibilities. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Changes are always risky when they're made in dependent code. DIP talks about keeping a chunk of code (dependency) away from the main program to which it is not directly related.

To reduce the coupling, DIP suggests eliminating the direct dependency of low-level modules on high-level modules to perform their responsibilities. Instead, make the high-level module rely on abstraction (a contract) that forms the generic low-level behavior.

This way, the actual implementation of low-level modules can be changed without making any changes in high-level modules. This produces great flexibility and molecularity in the system. As far as any low-level implementation is bound to abstraction, high-level modules can invoke it. Let's have a look at a sample suboptimal design where we can apply DIP to improve the structure of the application.

Consider a scenario where you are designing a module that simply generates balance sheets for a local store. You are fetching data from a database, processing it with complex business logic, and exporting it into HTML format. If you design this in a procedural way, then the flow of the system would be something like the following diagram:



A single module takes care of fetching data, applying business logic to generate balance sheet data, and exporting it into HTML format. This is not the best design.

Let's separate the whole functionality into three different modules, as shown in the following diagram:

- **Fetch Database Module:** This will fetch data from a database
- **Export HTML Module:** This will export the data in HTML
- **Balance Sheet Module:** This will take data from a database module, process it, and give it to the export module to export it in HTML

In this case, the balance sheet module is a high-level module, and fetch database and export HTML are low-level modules.

Inversion of Control

IoC is a design methodology used to build a loosely coupled system in software engineering by inverting the control of flow from your main program to some other entity or framework.

Here, the control refers to any additional activities a program is handling other than its main activities, such as creating and maintaining the dependency objects, managing the application flow, and so on.

Unlike procedural programming style, where a program handles multiple unrelated things all together, IoC defines a guideline where you need to break the main program in multiple independent programs (modules) based on responsibility and arrange them in such a way that they are loosely coupled.

In our example, we break the functionality into separate modules. The missing part was how to arrange them to make them decoupled, and we will learn how IoC makes that arrangement. By inverting (changing) the control, your application becomes decoupled, testable, extensible, and maintainable.

Implementing DIP through IoC

DIP suggests that high-level modules should not depend on low-level modules. Both should depend on abstraction. IoC provides a way to achieve the abstraction between high-level and low-level modules.

Let's see how we can apply DIP through IoC on our Balance Sheet example. The fundamental design problem is that high-level modules (balance sheet) tightly depend on low-level (fetch and export data) modules.

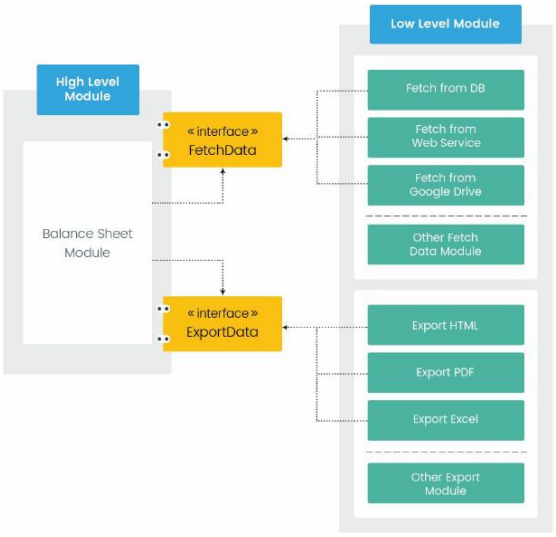
Our goal is to break this dependency. To achieve this, IoC suggests inverting the control. In IoC, inverting the control can be achieved in the following ways:

- **Inverting the interface:** Make sure the high-level module defines the interface, and low-level modules follow it.
- **Inverting object creation:** Change the creation of dependency from your main modules to some other program or framework.
- **Inverting flow:** Change the flow of application.

Inverting the interface

Inverting the interface means inverting the interaction control from low-level modules to high-level modules. Your high-level module should decide which low-level modules can interact with it, rather than keep changing itself to integrate each new low-level module.

After inverting the interface, our design would be as per the following diagram:



In this design, the balance sheet module (high-level) is interacting with fetch data and export data (low-level) modules with common interface. The very clear benefits of this design are that you can add new fetch data and export data (low-level) modules without changing anything on the balance sheet module (high-level).

As far as low-level modules are compatible with the interface, the high-level modules will be happy to work with it. With this new design, high-level modules are not dependent on low-level modules, and both are interacting through an abstraction (interface). Separating the interface from the implementation is a prerequisite to achieve DIP.

In the current directory, there are the java files representing the last diagram (FIRST APPROACH).

You may have observed that, the generateBalanceSheet() method became more straightforward. It allows us to work with additional fetch and export modules without any change. It is thanks to the mechanism of inverting the interface that makes this possible.

This design looks perfect; but still, there is one problem. If you noticed, the balance sheet module is still keeping the responsibility of creating low-level module objects (exportDataObj and fetchDataObj). In other words, object creation dependency is still with the high-level modules.

Because of this, the Balance Sheet module is not 100 percent decoupled from the lowlevel modules, even after implementing interface inversion. You will end up instantiating low-level modules with if/else blocks based on some flag, and the high-level module keeps changing for adding additional low-level modules integration.

To overcome this, you need to invert the object creation from your higher-level module to some other entity or framework. This is the second way of implementing IoC.

Inverting object creation

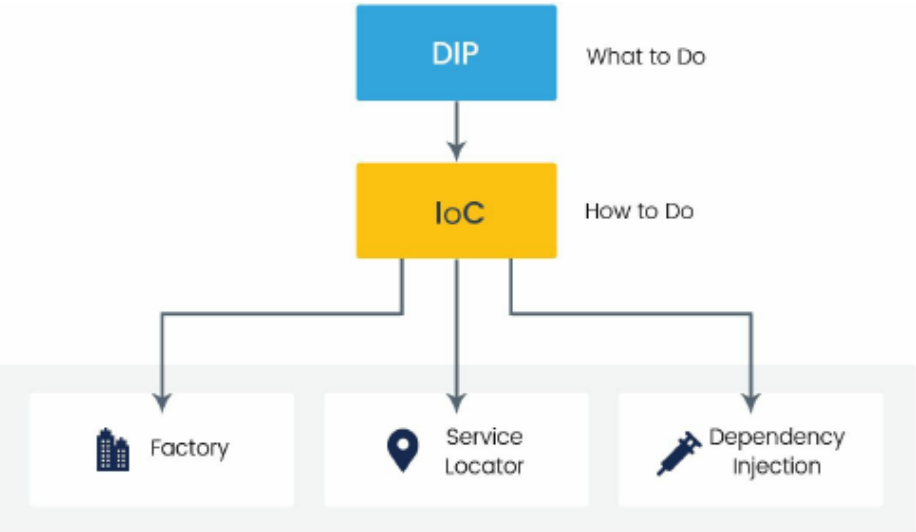
Once the abstraction between modules is set, there is no need to keep the logic of creating dependency objects in higher-level modules.

Adding two new methods in BalanceSheet.class in the package com.balanceSheet, included in the current directory (SECOND APPROACH).

Here are some quick observations:

- Objects of fetch data and export data modules are created outside the balance sheet module, and passed through configureFetchData() and configureExportData() methods
- The balance sheet module is now 100 percent decoupled from fetch data and export data modules
- For any new type of fetch and export data, no change is required in balance sheet modules

At this moment, the relation between DIP and IoC can be described as per the following diagram:



Finally, we implemented DIP through IoC and solved one of the most fundamental problems of interdependency between modules.

But hold on, something is not complete yet. We have seen that keeping the object creation away from your main module will eliminate the risk of accommodating changes and make your code decoupled. But we haven't explored how to create and pass the dependency object from outside code into your module. There are various ways of inverting object creation.

Different ways to invert object creation

We have seen how inversion of object creation helps us to decouple the modules. You can achieve the inversion of object creation with multiple design patterns as follows:

- Factory pattern
- Service locator
- Dependency injection

Dependency injection

DI is one of the ways to invert the object creation process from your module to other code or entity. The term injection refers to the process of passing the dependent object into a software component.

Since DI is one of the ways to implement IoC, it relies on abstraction to set the dependency. The client object doesn't know which class will be used to provide functionality at compile time. The dependency will be resolved at runtime. A dependent object does not directly call to the client object; instead, the client object will call a dependent object whenever required. It's similar to the Hollywood principle: Don't call us, we'll call you when we need to.

Dependency injection types

In DI, you need to set the entry point in a client object from which the dependency can be injected. Based on these entry points, DI can be implemented with the following types:

- Constructor injection
- Setter injection
- Interface injection

Constructor injection

This is the most common way to inject dependency. In this approach, you need to pass the dependent object through a public constructor of a client object. Please note that in case of construction injection, you need to pass all the dependency objects in the constructor of a client object.

Constructor injection can control the order of instantiation and consequently reduce the risk of circular dependency. All mandatory dependencies can be passed through constructor injection.

Adding a constructor in BalanceSheet.class in the package com.balanceSheet, included in the current directory (THIRD APPROACH).

All dependencies are injected from a constructor of a client object. Since constructors are called only once, it's clear that the dependency object will not be changed until the existence of a client object. If a client uses constructor injection, then extending and overriding it would be difficult sometimes.

Source: Java 9 Dependency Injection, *Write loosely coupled code with Spring 5 and Guice,* Krunal Patel, Nilang Patel. Packt, 2018, First Chapter.