**Dependency Inversion Principle**

The [Dependency Inversion Principle](https://www.oodesign.com/dependency-inversion-principle.html) is the fifth SOLID design principle represented by the last “D” and introduced by Robert C Martin. The goal of the Dependency Inversion Principle is to avoid tightly coupled code, as it easily breaks the application. The principle states that:

*“High-level modules should not depend on low-level modules. Both should depend on abstractions.”*

*“Abstractions should not depend on details. Details should depend on abstractions.”*

In other words, you need to decouple high-level and low-level classes. High-level classes usually encapsulate complex logic while low-level classes include data or utilities. Typically, most people would want to make high-level classes depend on low-level classes. However, according to the Dependency Inversion Principle, you need to invert the dependency. Otherwise, when the low-level class is replaced, the high-level class will be affected, too.

As a solution, you need to create an abstract layer for low-level classes, so that high-level classes can depend on abstraction rather than concrete implementations.

Robert C Martin also mentions that the Dependency Inversion Principle is a specific combination of the Open/Closed and Liskov Substitution Principles.

**Example of the Dependency Inversion Principle**

Now, the book store asked us to build a new feature that enables customers to put their favorite books on a shelf.

To implement the new functionality, we create a lower-level Book class and a higher-level Shelf class. The Book class will allow users to see reviews and read a sample of each book they store on their shelves. The Shelf class will let them add a book to their shelf and customize the shelf.

class Book {

void seeReviews() {...}

void readSample() {...}

}

class Shelf {

Book book;

void addBook(Book book) {...}

void customizeShelf() {...}

}

Everything looks fine, but as the high-level Shelf class depends on the low-level Book, the above code violates the Dependency Inversion Principle. This becomes clear when the store asks us to enable customers to add DVDs to their shelves, too. To fulfill the demand, we create a new DVD class:

class DVD {

void seeReviews() {...}

void watchSample() {...}

}

Now, we should modify the Shelf class so that it can accept DVDs, too. However, this would clearly break the Open/Closed Principle. The solution is to create an abstraction layer for the lower-level classes (Book and DVD). We’ll do so by introducing the Product interface that both classes will implement.

public interface Product {

void seeReviews();

void getSample();

}

class Book implements Product {

@Override

public void seeReviews() {...}

@Override

public void getSample() {...}

}

class DVD implements Product {

@Override

public void seeReviews() {...}

@Override

public void getSample() {...}

}

Now, Shelf can reference the Product interface instead of its implementations (Book and DVD). The refactored code also allows us to later introduce new product types (for instance, Magazine) that customers can put on their shelves, too.

class Shelf {

Product product;

void addProduct(Product product) {...}

void customizeShelf() {...}

}

The above code also follows the Liskov Substitution Principle, as the Product type can be substituted with both of its subtypes (Book and DVD) without breaking the program. At the same time, we have also implemented the Dependency Inversion Principle, as in the refactored code, high-level classes don’t depend on low-level classes, either.

As you can see on the left of the UML graph below, the high-level Shelf class depends on the low-level Book before the refactoring. Without applying the Dependency Inversion Principle, we should make it depend on the low-level DVD class, too. However, after the refactoring, both the high-level and low-level classes depend on the abstract Product interface (Shelf refers to it, while Book and DVD implement it).

