https://stackify.com/dependency-inversion-principle/

SOLID Design Principles Explained: Dependency Inversion Principle with Code Examples

THORBEN JANSSENMAY 7, 2018DEVELOPER TIPS, TRICKS & RESOURCES

The SOLID design principles were promoted by Robert C. Martin and are some of the best-known design principles in object-oriented software development. SOLID is a mnemonic acronym for the following five principles:

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Each of these principles can stand on its own and has the goal to improve the robustness and maintainability of object-oriented applications and software components. But they also add to each other so that applying all of them makes the implementation of each principle easier and more effective.

I explained the first four design principles in previous articles. In this one, I will focus on the Dependency Inversion Principle. It is based on the Open/Closed Principle and the Liskov Substitution Principle. You should, therefore, at least be familiar with these two principles, before you read this article.

Definition of the Dependency Inversion Principle

The general idea of this principle is as simple as it is important: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

Based on this idea, Robert C. Martin's definition of the Dependency Inversion Principle consists of two parts:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

An important detail of this definition is, that high-level **and** low-level modules depend on the abstraction. The design principle does not just change the direction of the dependency, as you might have expected when you read its name for the first time. It splits the dependency between the high-level and low-level modules by introducing an abstraction between them. So in the end, you get two dependencies:

1. the high-level module depends on the abstraction, and
2. the low-level depends on the same abstraction.

Based on other SOLID principles

This might sound more complex than it often is. If you consequently apply the Open/Closed Principle and the Liskov Substitution Principle to your code, it will also follow the Dependency Inversion Principle.

The Open/Closed Principle required a software component to be open for extension, but closed for modification. You can achieve that by introducing interfaces for which you can provide different implementations. The interface itself is closed for modification, and you can easily extend it by providing a new interface implementation.

Your implementations should follow the Liskov Substitution Principle so that you can replace them with other implementations of the same interface without breaking your application.

Let's take a look at the CoffeeMachine project in which I will apply all three of these design principles.