**Liskov Substitution Principle:**

The Liskov Substitution principle was introduced by **Barbara Liskov** in her conference keynote “Data abstraction” in 1987. A few years later, she published a paper with Jeanette Wing in which they defined the principle as:

*Let*Φ(x)*be a property provable about objects*x*of type*T*. Then*Φ(y)*should be true for objects*y*of type*S*where*S*is a subtype of*T*.*

But such a scientific definition might be necessary, but it doesn’t help a lot in our work as software developers. So, what does it mean for our code?

The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of our subclasses to behave in the same way as the objects of our superclass. We can achieve that by following a few rules, which are pretty similar to the design by contract concept defined by Bertrand Meyer.

An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass. That means we can implement less restrictive validation rules, but we are not allowed to enforce stricter ones in our subclass. Otherwise, any code that calls this method on an object of the superclass might cause an exception, if it gets called with an object of the subclass.

Similar rules apply to the return value of the method. The return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass. We can only decide to apply even stricter rules by returning a specific subclass of the defined return value, or by returning a subset of the valid return values of the superclass.

**Interface Segregation Principle:**

The Interface Segregation Principle was defined by **Robert C. Martin** while consulting for Xerox to help them build the software for their new printer systems.

He defined it as:

*“Clients should not be forced to depend upon interfaces that they do not use.”*

It’s pretty easy to violate this interface, especially if our software evolves and we have to add more and more features.

Similar to the Single Responsibility Principle, the goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.

**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, we 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 we might have expected when we 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, we get two dependencies:

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