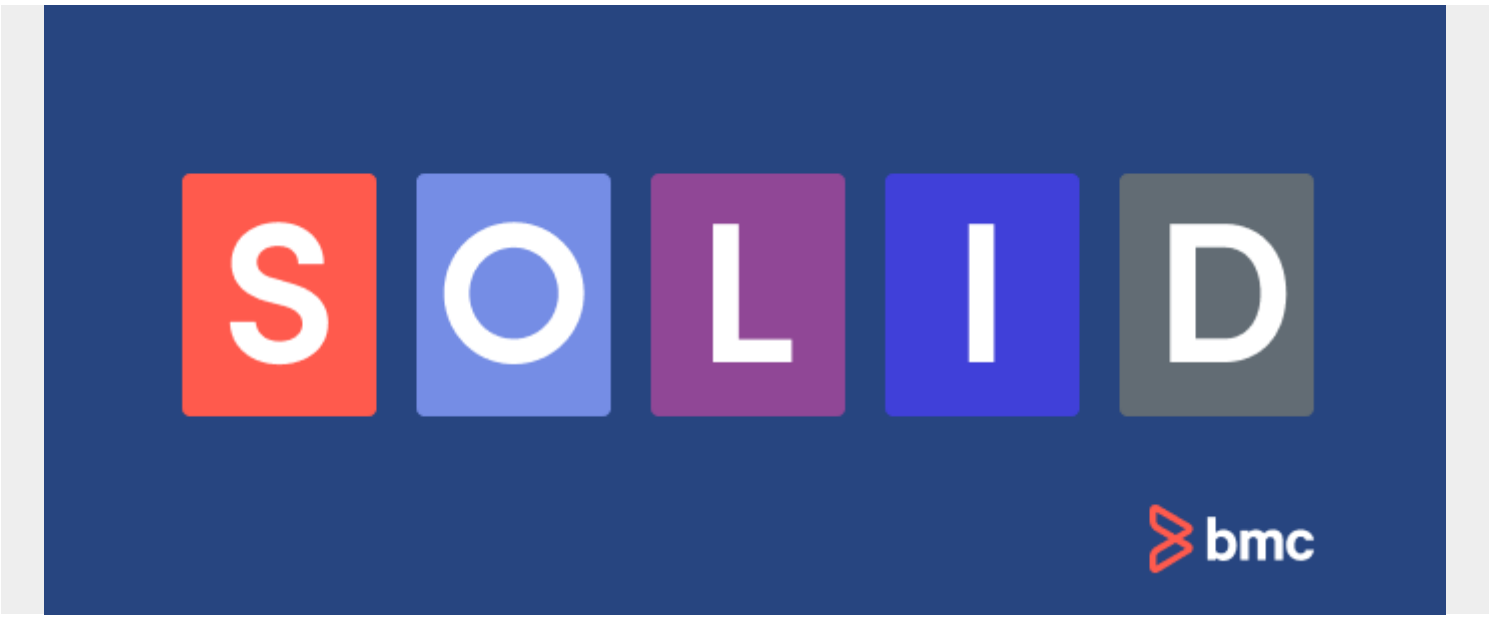
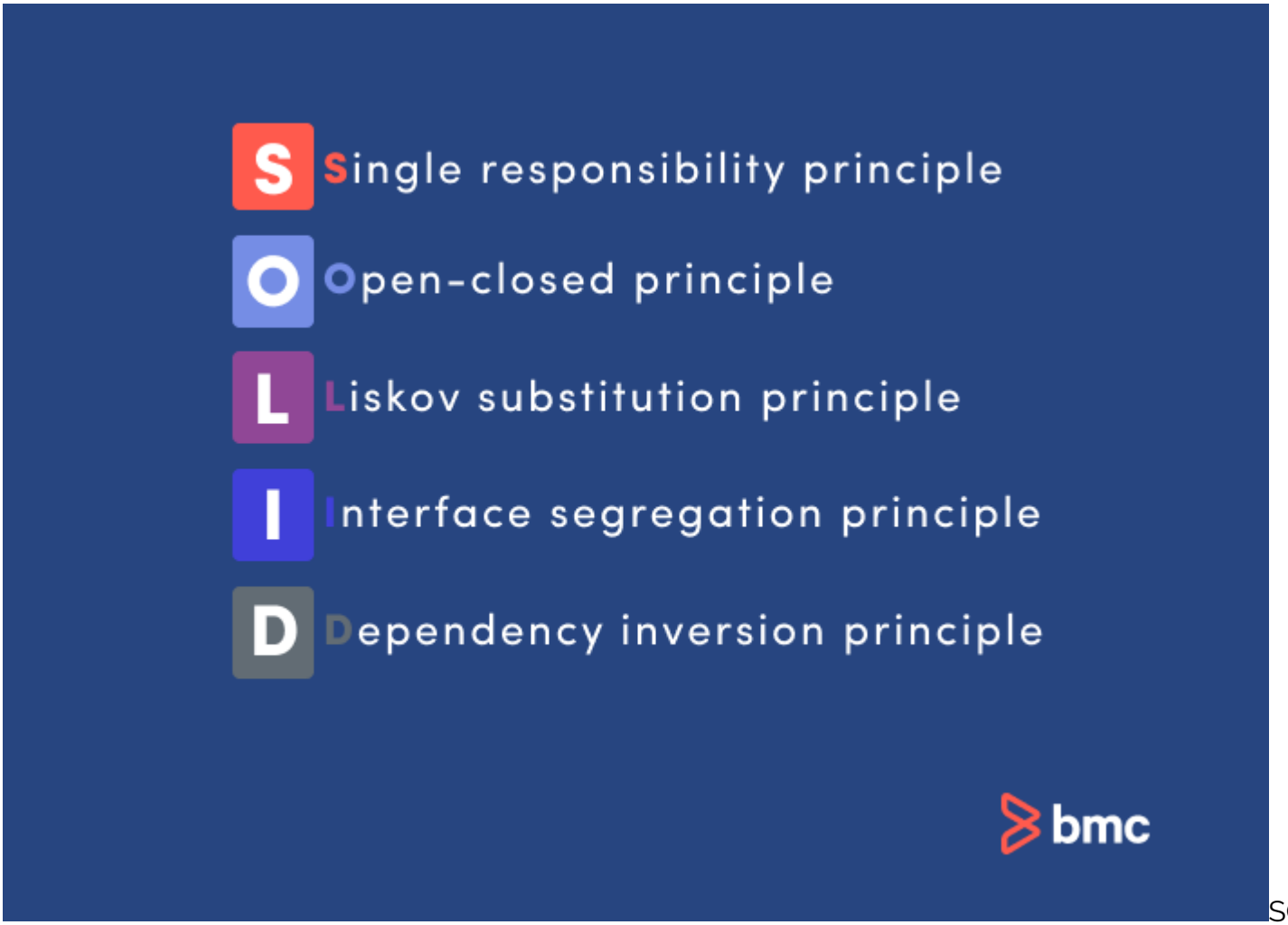
SOLID PRINCIPLES IN OBJECT ORIENTED DESIGN



SOLID programming design is a philosophy and approach to creating more robust, flexible, and scalable software systems that maintain integrity over time. As a developer, it is important to understand the right way and wrong way to write code. In this article, we will detail each of the SOLID design principles in context so that you can apply them to your work.

What are SOLID principles?



SOLID is a popular set of design principles that developers use when creating object-oriented software. SOLID is an acronym that stands for five key design principles:

- **S**ingle responsibility principle
- **O**pen-closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

All five SOLID principles are commonly used by software engineers and provide important benefits for developers.

Robert C. Martin developed the SOLID principles in his 2000 essay, "Design Principles and Design Patterns." Martin acknowledged that successful software will always need to evolve. As it changes, it has a tendency to grow increasingly complex. Without good design principles, software becomes rigid, fragile, immobile, and viscous. He developed SOLID software design principles to combat these common problems.

Why use SOLID principles in programming

SOLID principles seek to reduce dependencies so that engineers can change one area of the software without impacting others. They make it easier to understand, maintain, and extend designs. Software engineers avoid issues and build adaptive, effective, and agile software using these SOLID principles.

While the principles bring many benefits, following them may lead to longer and more complex code. That can extend the design process and make development a little more difficult. Yet, taking this extra time and effort in the present is well worth it. SOLID programming makes software so much easier to maintain, test, and extend over the long run.

SOLID principles are not a design problem cure-all. When software designers follow them correctly, however, they create code that has better readability, is easier to maintain, uses design patterns, and streamlines testing. All developers should know SOLID programming to best use these principles for creating quality output.

---

5 SOLID Design Principles

The best way to understand what SOLID principles are, is to look at them in context. Let's review SOLID software designs with one of the most commonly used programming languages. C# is a good way to demonstrate how SOLID works in action.

1. Single Responsibility Principle

The single responsibility principle (SRP) mandates that:

"A class should have one, and only one, reason to change."

When following a SOLID programming approach, recognize that each class only does one thing. Every class or module is responsible for one part of the software's functionality. More simply, each class should solve only one problem.

The single responsibility principle is a relatively basic one that most developers already use to build code. You can apply it to classes, software components, and microservices.

SRP makes it easier to test and maintain code for streamlined software implementation. It also helps to avoid unanticipated side effects of future changes.

To ensure that you're following this principle in development, consider using an automated check when building to limit class scope. This check is not foolproof for following the SRP, but goes a long way to ensuring that classes are not violating this principle.

2. Open-Closed Principle

The open-closed principle recognizes you may need to add or change something to existing, well-tested classes. Any change risks creating problems or bugs. Instead of changing the class, however, you can simply extend it. The open-closed principle states:

"You should be able to extend a class's behavior without modifying it."

Following the open-closed principle makes your code easier to maintain and revise. It requires writing code that is:

- Open for extension, meaning you can add to that class's behavior.
- Closed for modification, meaning that the source code is set and cannot be changed.

At first glance, these two criteria seem to be inherently contradictory. When you become more comfortable with it, you'll see that the open-closed principle is not as complicated as it seems. You can comply with the open-closed principle using abstractions to make sure that your class is easily extendable without having to modify it. Using inheritance or interfaces that allow polymorphic substitutions is a common way to comply with this principle.

---

3. Liskov Substitution Principle

The Liskov substitution principle is perhaps the most difficult to understand of the five SOLID design principles. The Liskov substitution principle requires that every derived class should be substitutable for its parent class. It is named for Barbara Liskov, who introduced this concept of behavioral subtyping in 1987. She explained:

"What is wanted here is something like the following substitution property: if for each object O1 of type S there is an object O2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T."

While this can be a difficult principle to internalize, it might help to think of it as an extension of the open-closed principle. The point of the Liskov substitution principle is to ensure that derived classes extend the base class without changing behavior.

Following this principle helps you avoid the unexpected consequences of changes and having to open a closed class in order to make changes.The Liskov substitution principle leads to software that you can easily extend. While it might slow down the development process, you can avoid countless time-consuming future issues during updates and extensions.

4. Interface Segregation Principle

The interface segregation principle for SOLID programming states that it is better to have many smaller interfaces than a few bigger ones:

"Make fine grained interfaces that are client-specific. Clients should not be forced to implement interfaces they do not use."

You don't want to just start with an existing interface and add new methods. Instead, build a new interface. Let your class implement multiple interfaces as needed. When you use smaller interfaces,you can prefer composition over inheritance and decoupling over coupling. According to the interface segregation principle, you should have many client-specific interfaces and avoid the temptation of having one big, general-purpose interface.

5. Dependency Inversion Principle

Use the dependency inversion principle of SOLID software design to decouple software modules. You should "depend on abstractions, not on concretions" when developing software:

"High level modules should not depend upon low level modules. Both should depend on abstractions…abstractions should not depend on details. Details should depend upon abstractions."

One way of many for complying with this SOLID principle is to use a dependency inversion pattern. Whatever method you use, following the dependency inversion principle will make your code more flexible, agile, and reusable.

Examples of other software design principles

Over decades of writing code, software developers have learned—sometimes the hard way—how to create code that is robust, easy to maintain and scale, and that functions with efficiency.

They encapsulated their hard earned lessons into software design principles for creating high-quality software. SOLID design principles are just some examples of software development best practices that support excellence in coding.

Are SOLID principles still relevant?

In the decades since Robert Martin first developed the SOLID software design principles, technology and the software industry has gone through enormous change. Yet his core ideas are every bit as valid globally today as they ever have been.

When you implement SOLID design principles you create systems that are more maintainable, scalable, testable, and reusable. You produce quality code that meets industry standards with the best chance of having a durable competitive advantage.

SOLID programming design principles can feel overwhelming at first, yet when you are in the habit of working with them regularly, you will have an innate understanding of code that complies with the principles. You will find that making SOLID code design choices makes your work easier and more efficient, with superior output.