



[New Guide] Download the 2017 Guide to Automated Testing: Improving Application Speed and Quality

[Download Guide](#) ▶

The SOLID Principles in Real Life

by Erik Dietrich MVB · Feb. 12, 16 · Java Zone

Learn how to troubleshoot and diagnose some of the most common performance issues in Java today. Brought to you in partnership with AppDynamics.

Hopefully, you're familiar with the SOLID principles, particularly if you program in object oriented languages. The wisdom contained therein (mostly) isn't *limited* to object oriented languages, but such languages were the intended target.

If you're not familiar and don't have time to read the linked Wikipedia page, SOLID is a mnemonic acronym for five principles of object oriented programming or, as I hinted, really just programming in general (except, perhaps for the Liskov Substitution Principle). These concepts have been around since at least the early 2000s and have truly stood the test of time.

What you get by following them is code that's a lot more likely to be maintainable. These design guidelines, properly followed, will tend to steer you toward writing clean code.

What I'd like to do is offer real life analogs of the principles. I'd imagine that this may make them easier to remember, but I think it can also serve to drive the points home in the first place and help encourage the "aha" moment if you haven't yet had them. And, even if you have, it never hurts to have a visual to help reinforce the concept or to explain it someone else -- even someone non-technical, potentially.





S is for Single Responsibility Principle

The single responsibility principle (SRP) asserts that a class or module should do one thing only. Now, this is kind of subjective, so the principle is reinforced with the heuristic that the class or module should have only one reason to change.

By way of counter-example, consider a class that opens a connection to the database, pulls out some table data, and writes the data to a file. This class has multiple reasons to change: adoption of a new database, modified file output format, deciding to use an ORM, etc. In terms of the SRP, we'd say that this class is doing too much.

In your day to day life, picture those "duck" vehicles you see occasionally in some lakeside towns. They're street legal and water-capable, so a duck tour affords you the unique and surreal experience of being in a 'car' that gets to the edge of the water and just keeps going. Fun, right?

And yet, you don't see a whole lot of them. There are millions of families out there that own both cars and boats, and there are very few families that buy these ducks. Do you know why? It's most likely because no one wants to be unable to drive to work because their boat rudder is broken. Ducks are fun, but they're also a great example of the pitfalls that the SRP can help you avoid.

O is for Open/Closed Principle

The Open/Closed Principle states that code entities should be open for extension, but closed for modification. To put this more concretely, you should write a class that does what it needs to flawlessly and not assuming that people should come in and change it later. It's closed for modification, but it can be extended by, for instance, inheriting from it and overriding or extending certain behaviors. An example of running afoul of the open-closed principle would be to have a switch statement somewhere that you needed to go in and add to every time you wanted to add a menu option to your application.

A great example of this in real life is sitting in your pocket in the form of a smartphone. All such phones have app stores and these app stores let you extend the base functionality of the phone. Sure, it ships with the basics: camera operation, actual calls, text messages, etc. But via the app store, you can *extend* the phone's capabilities to allow you to manage your todo list, play inane video games, and even serve as a flashlight or wireless access point.

The mechanism that allows you to do this is purely one of extension, however. It's not as though Apple, Google, and Microsoft put the OS source code up on GitHub and invite you to dive in and start building games and flashlight functionality. Rather, they make the core phone functionality *closed for modification* and they *open it to an extension*.

L is for Liskov Substitution Principle

The Liskov Substitution Principle (LSP) is the one here that is most unique to object-oriented programming. The LSP says, basically, that any child type of a parent type should be able to stand in for that parent without things blowing up.

In other words, if you have a class, `Animal`, with a `MakeNoise()` method, then any subclass of `Animal` should reasonably implement `MakeNoise()`. Cats should meow, dogs should bark, etc. What you wouldn't do is define a `MuteMouse` class that throws `IDontActuallyMakeNoiseException`. This violates the LSP, and the argument would be that this class has no business inheriting from `Animal`.

To picture this, imagine cooking yourself a stew. If you're anything like me, you'd only put things in there that were edible because you would want to eat the stew without picking through each bite, asking yourself repeatedly, "is this edible?"

I is for Interface Segregation Principle

The Interface Segregation Principle (ISP) says that you should favor many, smaller, client-specific interfaces over one larger, more monolithic interface. In short, you don't want to force clients to depend on things they don't actually need. Imagine your code consuming some big, fat interface and having to re-compile/deploy with annoying frequency because some method you don't even care about got a new signature.

To picture this in the real world, think of going down to your local corner restaurant and checking out the menu. You'll see all of the normal menu mainstays, and then something that's just called "soup of the day." Why do they do this? Because the soup changes a lot and there's no sense reprinting the menus every day. Clients that don't care about the soup needn't even be concerned, and clients that do use a different interface -- asking the server.

D is for Dependency Inversion

The Dependency Inversion Principle (DIP) encourages you to write code that depends upon abstractions rather than upon concrete details. You can recognize this in the code you read by looking for a class or method that takes something generic like `"Stream"` and performs operations on it, as opposed to instantiating a specific `Filestream` or `Stringstream` or whatever. This gives the code in question a lot more flexibility -- you can swap in anything that conforms to the `Stream` abstraction and it will still work.

To visualize this in your day to day, go down to your local store and pay for something with a credit card. The clerk doesn't examine your card and get out the "Visa Machine" after seeing that your card is a Visa. He just takes your card, whatever it is, and swipes it. Both you and the clerk depend on the credit card abstraction without worrying about specifics.

And, That's SOLID!

Hopefully, these visualizations help you. If you're always keeping SOLID in the back of your mind while writing code, you're going to make whoever maintains that code a lot happier. And, if you have an easy way to picture and remember the principles, you're a lot more likely to keep them in mind.