<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

In March of 1995 Uncle Bob wrote an article about sets of principles for OOD. “These principles expose the dependency management aspects of OOD as opposed to the conceptualization and modeling aspects. This is not to say that OO is a poor tool for conceptualization of the problem space, or that it is not a good venue for creating models. Certainly many people get value out of these aspects of OO. The principles, however, focus very tightly on dependency management.”

“Dependency Management is an issue that most of us have faced. Whenever we bring up on our screens a nasty batch of tangled legacy code, we are experiencing the results of poor dependency management. Poor dependency management leads to code that is hard to change, fragile, and non-reusable.”

The first five principles are principles of *class design*. They are:

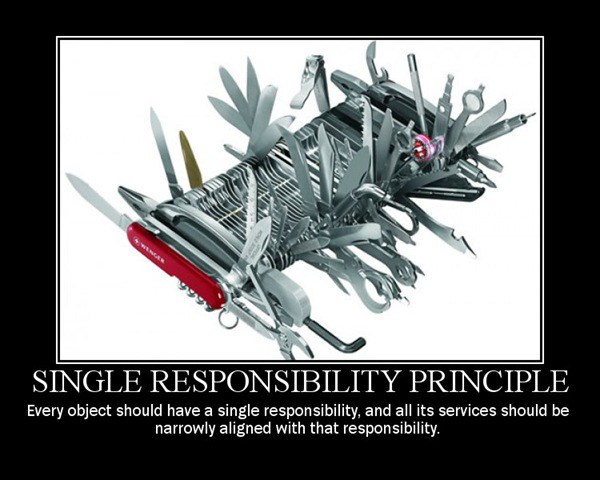
|  |  |  |
| --- | --- | --- |
| **SRP** | [The Single Responsibility Principle](https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk) | A class should have one, and only one, reason to change. |
| **OCP** | [The Open Closed Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1&hl=en) | You should be able to extend a classes behavior, without modifying it. |
| **LSP** | [The Liskov Substitution Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh&hl=en) | Derived classes must be substitutable for their base classes. |
| **ISP** | [The Interface Segregation Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi&hl=en) | Make fine grained interfaces that are client specific. |
| **DIP** | [The Dependency Inversion Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz&hl=en) | Depend on abstractions, not on concretions. |

**SRP [The Single Responsibility Principle](https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk)**

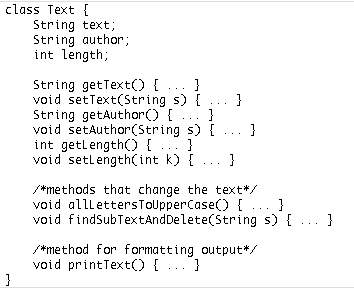
The single responsibility principle revolves around the claim that a certain code module (most often, a class) should only have responsibility over one part of the functionality provided by the software. In software engineering books, this is sometimes also defined like this: the module should only have **one reason to change**. This means that a division of concerns is performed in the program, and the methods for every concern should be completely encapsulated by a single class. Now it is obvious that this approach contributes to the high cohesion – since methods related to the same concern (same part of the functionality) will be members of the same class, and robustness – since this reduces the possibility of error.

The single responsibility principle is founded on one of the basic, general ideas of object-oriented programming – the so-called divide and conquers principle – solving a problem by solving its multiple sub-problems. This approach prevents the creation of “God objects” – objects that “know too much or do too much“.

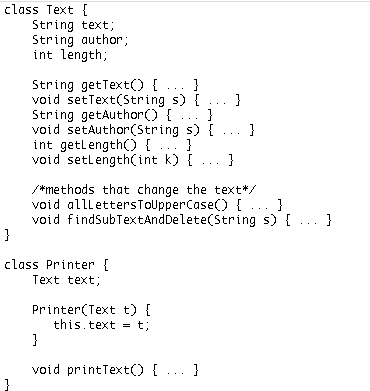
Example of some classes, but no on our code.



Bad Example



Better Example



**OCP [The Open Closed Principle](https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk)**

The Open Closed Principle represents the “O” of the five [SOLID](http://springframework.guru/solid-principles-object-oriented-programming/" \t "_blank" \o "SOLID: Principles Of Object Oriented Programming) software engineering principles to write well-designed code that is more readable, maintainable, and easier to upgrade and modify. [Bertrand Meyer](https://en.wikipedia.org/wiki/Bertrand_Meyer" \t "_blank) coined the term Open Closed Principle, which first appeared in his book [Object-Oriented Software Construction](https://www.amazon.com/Object-Oriented-Software-Construction-CD-ROM-Edition/dp/0136291554" \t "_blank" \o "Object-Oriented Software Construction), release in 1988. This was about eight years before the initial release of Java.

1. “*Open for extension* “: This means that the behavior of a software module, say a class can be extended to make it behave in new and different ways. It is important to note here that the term “*extended* ” is not limited to inheritance using the Java extend keyword. As mentioned earlier, Java did not exist at that time. What it means here is that a module should provide extension points to alter its behavior. One way is to make use of [polymorphism](http://springframework.guru/polymorphism-java/" \t "_blank" \o "Polymorphism in Java) to invoke extended behaviors of an object at run time.
2. “*Closed for modification* “: This means that the source code of such a module remains unchanged.

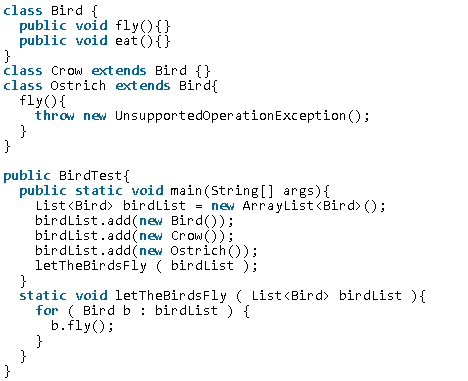
It might initially appear that the phrases are conflicting- How can we change the behavior of a module without making changes to it? The answer in Java is abstraction. You can create abstractions (Java interfaces and abstract classes) that are fixed and yet represent an unbounded group of possible behaviors through concrete subclasses.

**LSP [The Liskov Substitution Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh&hl=en)**

Liskov Substitution Principle states the following: “in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)”. Simply said, any object of some class in an object-oriented program can be replaced by an object of a child class.

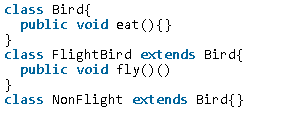
The Liskov substitution principle, written by [***Barbara Liskov***](https://en.wikipedia.org/wiki/Barbara_Liskov) in 1988, states that functions that reference base classes must be able to use objects of derived (child) classes without knowing it. It’s important for a programmer to notice that, unlike some other ***[Gang of Four](http://springframework.guru/gang-of-four-design-patterns/" \t "_blank" \o "Gang of Four Design Patterns)***principles, whose breaking might result in bad, but working code, the violation of this principle will most likely lead to buggy or difficult to maintain code.

Example



What do you think would happen when this code is executed?

This is one solution to fix the problem



**How can we identify LSP violation?**

* Derived class may require less functionality than the Base class, so some methods would be redundant.

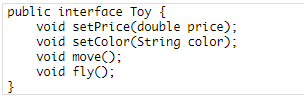
**ISP [The Interface Segregation Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi&hl=en)**

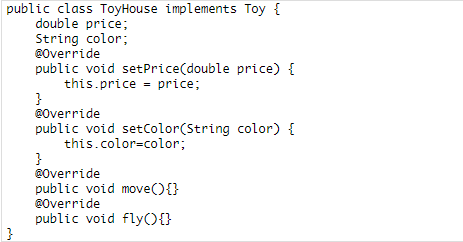
The Interface Segregation Principle represents the “I” of the five [SOLID](http://springframework.guru/solid-principles-object-oriented-programming/" \t "_blank" \o "SOLID Principles Of Object Oriented Programming) principles of object-oriented programming to write well-designed code that are more readable, maintainable, and easier to upgrade and modify. This principle was first used by Robert C. Martin while consulting for Xerox, which he mentioned in his 2002 book, [Agile Software Development: Principles, Patterns and Practices](https://amzn.to/1GJg6on" \t "_blank" \o "Agile Software Development, Principles, Patterns, and Practices). This principle states that “Clients should not be forced to depend on methods that they do not use”. Here, the term “Clients” refers to the implementing classes of an interface.

What the Interface Segregation Principle says is that your interface should not be bloated with methods that implementing classes don’t require. For such interfaces, also called “fat interfaces”, implementing classes are unnecessarily forced to provide implementations (dummy/empty) even for those methods that they don’t need. In addition, the implementing classes are subject to change when the interface changes. An addition of a method or change to a method signature requires modifying all the implementation classes even if some of them don’t use the method.

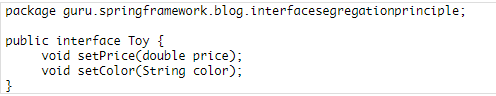
The Interface Segregation Principle advocates segregating a “fat interface” into smaller and [highly cohesive](https://en.wikipedia.org/wiki/Cohesion_%28computer_science%29)interfaces, known as “role interfaces”. Each “role interface” declares one or more methods for a specific behavior. Thus clients, instead of implementing a “fat interface”, can implement only those “role interfaces” whose methods are relevant to them.

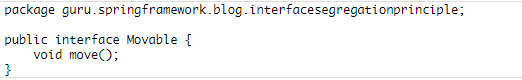
Violation Example

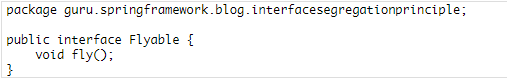


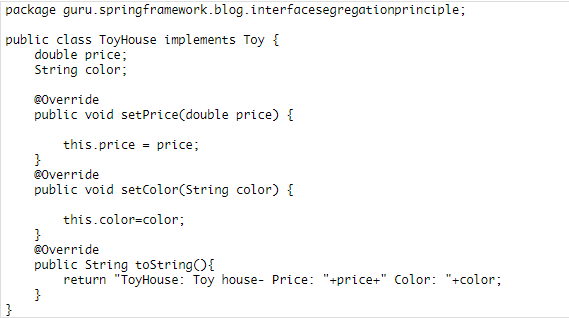


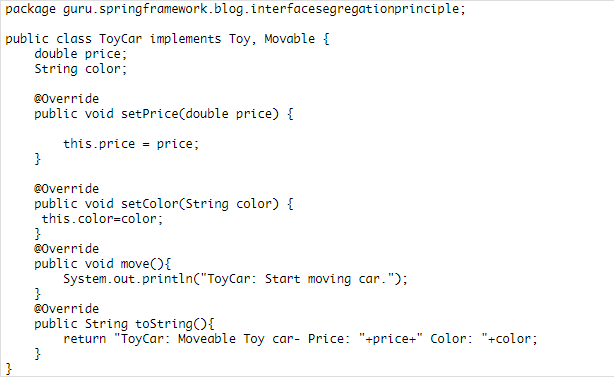
Following the interface segregation principle

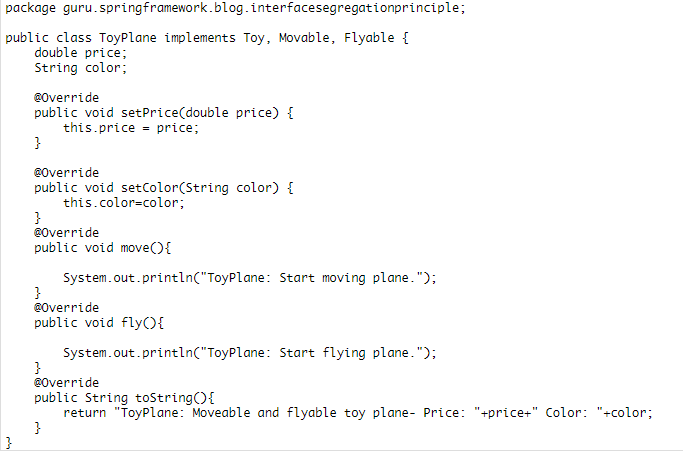












**DIP [The Dependency Inversion Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz&hl=en)**

The Dependency Inversion Principle represents the last “D” of the five [SOLID](https://springframework.guru/solid-principles-object-oriented-programming/)principles of object-oriented programming*.*[*Robert C. Martin*](http://www.objectmentor.com/omTeam/martin_r.html) first postulated the Dependency Inversion Principle and published it in 1996. The principle states:

“A. High-level modules should not depend on low-level modules. Both should depend on abstractions.B. Abstractions should not depend on details. Details should depend on abstractions.”

What Dependency Inversion Principle says is that instead of a high-level module depending on a low-level module, both should depend on an abstraction.

