1. Single-Responsibility Principle

*Each and every class in your project should have one, and only one responsibility.*

Consider this Order class in an e-commerce application.

If you lean back and think for a moment, you’ll see that the class doesn’t have a single responsibility. Rather, it does two different things — models a shopping order and takes care of its data storage. It will make sense to break the class down to two classes.

Designing your systems with classes that adhere to single responsibility principle makes your classes easier to maintain, replace, refactor and test.

2. Open-Closed Principle

*Entities in your software system should be open for extension, but closed for modification.*

To put it in other words, you need to allow others to extend the behavior of your entities, but in no way modify your source code. Let’s look at the OrderRepository class from the previous example.

Let’s assume that OrderRepository currently works with an MSSQL database. What if we want to switch to, say, an API which is provided by a third party service, which in its turn, stores the data in one of those exotic in-memory databases? An obvious solution would be to dig into the bodies of the methods in OrderRepository and simply modify them to work with the new database system. This is exactly what you shouldn’t be doing, according to the Open-Closed Principle. Even if you put the SOLID principles apart, you already have a functioning class. Why would you want to get rid of it anyway? What if you will need to use it again at some point in the future?

OK, now that directly modifying OrderRepository is out of the table, let’s think of what else can we do. What if we derive from it and redefine the methods in the child class? This is much better than direct modification, but still has its complexities. What happens when a new function or property is added to OrderRepository? Your new repository class gets cluttered with that as well. In some cases, the new properties/functions added to OrderRepository may be MSSQL-specific. You don’t want to clutter your new repository class with such baggage.

Think for a moment, what makes repository a repository? It’s just a simple service-class which works with data. So why not model this in an interface which does exactly that, without specifying any implementation details? In C# you can achieve this by defining an *interface*. There may be other tools in the language of your choice to do that.

Now, having the interface, you simply need to make your different OrderRepository implementations adhere to it.

You completely fulfill the Open-Closed Principle and your different implementations are completely decoupled. Not only that, but by programming against the IOrderRepository interface vs a concrete class, you gain additional benefits in the form of dependency injection (if needed).

3. Liskov Substitution Principle

LSP is by far the most complicated, and hard-to-grasp idea in the list of SOLID principles. I came across different definitions of the principle, and I think that the one that follows, found in [this brilliant StackOverflow thread](http://stackoverflow.com/questions/56860/what-is-the-liskov-substitution-principle), is the best.

*Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.*

The principle is best explained in a classic example of shape class hierarchy. Suppose you have to model a hierarchy of different geometric shapes. How would you model your Rectangle and Square classes? Intuitively, a square *is a* rectangle, so your intuition tells you to inherit Square from Rectangle. While, on many levels, this is the perfectly correct thing to do, this may lead to unwanted consequences in the future. Let’s look at the Rectangle class.

If you inherit your Square from Rectangle, what would your SetWidth and SetHeight functions do? Most probably both will affect width and height at the same time, because in the context of a Square height and width are the same. The catch is that you may have code somewhere that will make assumptions based on width and heightfields being independent. And if that’s the case, and you refer to an instance of Square with your reference of type Rectangle, you will end up breaking that code.

To avoid this, be very careful when modeling your abstractions. If you feel that at some point such problems may occur, don’t inherit your abstractions from each other. In this example, just make Shape the base class of your Square.

4. Interface Segregation Principle

*Clients should not have to implement interfaces that they don’t use.*

Let’s look at the geometric shapes hierarchy once again.

There’s too much happening in this interface. It defines your 2D shape model, how it looks on the screen, how is it rendered and how is it animated. What if you want to pass it to a module that saves the state of your geometric canvas on a disk, so that it can be restored after restarting the application? If your data saving module relies on the IShape2D interface, the class realizing your interface will have to define all the unnecessary drawing and animation logic. It makes sense to separate such concerns by introducing new interfaces for them.

Now that you’ve redefined your interfaces, you can model your Ellipse abstraction like this.

To sum up, you end up with a much more robust object-oriented design. You can attach different behaviors to your classes by implementing just the interfaces you want. On the other hand, you’re not forced to implement anything that you don’t need.

5. Dependency Inversion Principle

*High level modules should not depend on lower level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

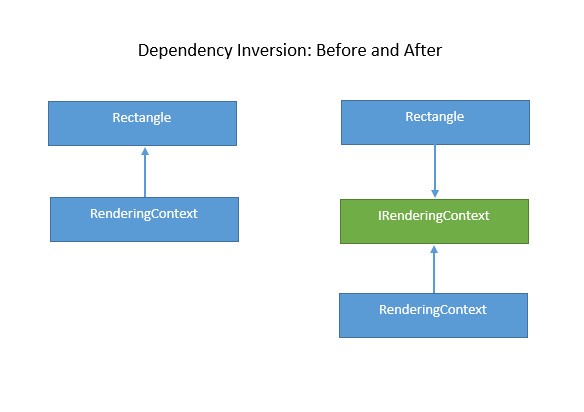
This last principle is a topic for a dozen articles of a similar length in and out of itself. During the last couple of decades, this principle has led people to rethink the way dependencies are introduced and managed in a complex software system and gave rise to concepts of Dependency Injection and IoC Containers. For the scope of this article, we will look at a small example which demonstrates a hard dependency between two entities and how we can redesign the hierarchy to get rid of it.

Rectangle is a class that models the abstraction of the corresponding geometric shape. RenderingContext encapsulates all the logic associated with displaying your shape on the screen (incl. communicating with the graphics device, allocating buffers for the geometry etc). Here Rectangle has a dependency on RenderingContext. That means that each time something is changed in the RenderingContext, there’s a chance that something will break in your Rectangle. It may take significant time and effort to identify that something has gone wrong, localize the issue and fix it. To demonstrate the dependency between the classes, we can draw a class diagram.

To overcome this problem, you need to abstract away the implementation details of your RenderingContext in an IRenderingContext interface. After doing that, you need to modify your Rectangle to work with IRenderingContext instead of RenderingContext.

Before I end this oversimplified explanation of Dependency Inversion, I want to show what happens to the dependencies after introducing the interface.





I hope that after looking at this diagram, it will become clear why the term is called *Dependency Inversion.*