Review: SOLID Principles

SOLID Principles

In this article, we'll have another look at the definitions of each of the five SOLID principles that we covered in the video lectures in order to allow you to review them at your own pace.  If anything is unclear, or if you have any additional questions on something not covered in the lectures, please post a question on the course - I'll be happy to try and help you out.

Before we get back to the principles themselves, let's quickly run through the benefits of SOLID principles again.

* Encourages good design.
* Drives consistency in design.
* Increases cohesion in classes.
* Reduces coupling between classes.
* Reduces likelihood of breaking changes.
* Simplifies debugging.
* Simplifies maintenance.
* Simplifies unit testing.

Remember that well-designed code might take a bit longer to write - but by writing good code, we can derive future benefits.  Code is something that evolves and that we often have to come back to (even if it's just to fix a bug), so by ensuring that we write good code, we can make our lives so much easier down the line.

Single Responsibility Principle (SRP)

* The Single Responsibility Principle (SRP), described by Robert C. Martin in *Agile Software Development, Principles, Patterns, and Practices*, states that *"A class should have only one reason to change."*\*
* I also have my own personal definition for this principle - a class or a method should be responsible for doing only one thing, and it should do that one thing really well.  
    
  \*Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. 1st ed. Upper Saddle River, N.J.: Pearson Education, 2003. Print.

Code that conforms to the Single Responsibility has a clear, well-defined responsibility - and this can be applied at various levels of your code - from packages and namespaces right down to methods.

When you look at a class, ask yourself what it's doing - if it doesn't serve a single, well-defined purpose, you might have to refactor it into a couple of different classes, each with a clear purpose.

The same applies when you're looking at a method - ask yourself if you can divide it into distinct chunks - such as code that sets up a request to invoke a service, validates the response, and persists something to a database.  If a method is doing multiple things, you can break it down into separate methods, each with a name that clearly expresses its intent.  Not only is code like this easier to read and understand, but it also becomes much easier to test - after all, instead of trying to test one method doing a multitude of things, you test specific methods, each intended to do something specific.

A metric that you can use in order to check if your code satisfies the Single Responsibility is the size of your methods.  Many people (myself included) are of the opinion that any method should fit on a single screen - in other words, you shouldn't have to use the scrollbar when you're reading a single method.

Open/Closed Principle (OCP)

* The Open/Closed Principle (OCP) states that *"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."*\*
* In essence, we want to design our systems in such a way that we can easily add features without having to modify, recompile, and redeploy core components.  While this might sound complex, it's quite simple in practice - we can use inheritance to extend classes and add behaviour to them, or we can design our classes in such a way that bits of logic can be plugged straight into them - which we can accomplish by employing certain design patterns or a plugin architecture.  
    
  \*Meyer, Bertrand. Object-Oriented Software Construction. 1st ed. New York [u.a.]: Prentice Hall, 1995. Print.

Code that conforms to the Open/Closed Principle is easy to expand upon, without having to modify the core parts of the code.  If you encounter code that seems to know a great deal about different bits of logic that do similar things, ask yourself if you'll have to modify that code every time that some or other small business rule changes.  If the answer is 'yes', that code might be in violation of this principle.

One approach to get your code to satisfy the Open/Closed Principle might be to write it in such a way that it depends on abstract classes or interfaces, where each implementation can specify its own logic, allowing you to expand on the code by creating different subclasses.  However, this is not the ideal approach - why extend classes to modify simple rules instead of core functionality?

A better way to solve this problem is to take some inspiration from plugin-based architectures.  Some of the applications that we use allow us to download plugins to add additional features, even though it doesn't involve modifying the core parts of the application - we can simply download something that 'plugs in' to our application.  We can write our code in the same way (think of the Strategy pattern) - we can write code that uses interfaces that define methods to execute business rules, and then pass implementations of those interfaces to the classes that need them.  Those classes can then simply delegate requests to the implementations that know how to deal with those requests.  This allows us (and anyone else in our organisation) to expand on the features of the application without having to modify core components.  In other words, this approach leaves our code closed to modification, but open to extension.  As with the other principles, it also makes our code easier to test.

Liskov Substitution Principle (LSP)

* The official description of the Liskov Substitution Principle (LSP) sounds rather technical and hard to understand - "*...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."*\*
* The simplified explanation that I prefer to use to describe the principle is as follows: when code depends on class/interface X, any subtype of X should be entirely valid in the same context (i.e. implementations can be substituted).  
    
  \*Liskov, B.  Data Abstraction and Hierarchy, 1988.

Code that violates the Liskov Substitution Principle can't reliably deal with a situation where we have code that refers to a super class, but substitute that with different sub classes at runtime.

In object-oriented programming, we talk about inheritance as an "is-a" relationship - in other words, a subclass is the same type of thing as its parent.  If we have two classes, Dog and Cat, that extend an Animal class, both a Dog and a Cat "is" an animal.  In code that violates this principle, we will effectively be trying to treat two different thing as the same thing - and we might try to instruct a Cat to bark, in which case our code will probably throw an error.  This points to a flaw in the design of our code - we're trying to treat two things as being the same, but actually, they are quite different and cannot behave in the same fashion.

In order to fix issues like this, we have take a step back and review our designs, in order to extract the things that we need to extract, and refactor our classes to accurately represent the things that they are actually supposed to represent.  A good sign of a violation of this principle, in my experience, tends to be code that has to do explicit "instance of" checks in order to decide how to operate on the objects that it is working with.

Interface Segregation Principle

* According to the Interface Segregation Principle (ISP), *"no client should be forced to depend on methods it does not use."*\*
* In essence, this means that each class should serve a well-defined purpose and only expose behaviour that is aligned with that purpose.  Consumers of that class shouldn't be forced to depend on methods which they do not require.  This ties in nicely with the **Single Responsibility Principle**.  
    
  \*Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. 1st ed. Upper Saddle River, N.J.: Pearson Education, 2003. Print.

In order to conform to the Interface Segregation Principle, our code needs to have well-defined interfaces that serve a specific purpose.  In theory, this principle sounds similar to the Single Responsibility Principle.  It's implication is, however, slightly different.

The reasoning behind the Interface Segregation Principle is that clients should not be forced to depend on methods that they don't use.  In other words, our interfaces (and this includes the public methods exposed on our classes) should provide functionality specific to what users of that interface will need.  If we bundle a bunch of things together in one place, changes to any of those methods will affect all of the users of those interfaces.

In practice, we might have a class that knows how to create both customers and accounts.  While this might seem okay, it introduces a potential issue.  Let's assume that we have two different screens on the front-end for capturing accounts and customers.  Both of these screens will have to depend on one class that knows how to deal with both accounts and customers.  Firstly, this introduces the risk that the customer screen (if a developer is careless) might accidentally call some code that will operate on accounts, simply because it has access to that code.  Secondly, should we decide that we need to change the account creation logic, the customer screen will also have to be recompiled, since it depends on the code being modified, even though it has no reason to do so.

A good sign of a violation of this principle tends to be 'god classes' - massive, complicated objects that can contain business logic for an entire application, all bundled into one place.  If you come across code that looks like this, you might want to consider refactoring that class into multiple smaller classes, each with clearly defined purpose.

Dependency Inversion Principle (DIP)

* The Dependency Inversion Principle (DIP) consists of two distinct statements\*:
  + "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."
* In simple terms, this principle says that we should aim to reduce coupling between components by introducing abstractions (think of interfaces) between them.  In other words, classes should not depend on each other directly.  This is related to both the **Open/Closed Principle** and the **Liskov Substitution Principle**.  
    
  \*Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. 1st ed. Upper Saddle River, N.J.: Pearson Education, 2003. Print.

The Dependency Inversion Principle allows us to reduce the level of coupling between our classes, by forcing code to depend on abstractions rather than concrete implementations.  This resembles the Open/Closed Principle, in that we write code in such a way that we allow it to operate on different concrete implementations without requiring in-depth knowledge about those implementations.  "Inversion of Control" is another term that describes the same concepts as the Dependency Inversion Principle.

There are numerous tools that support dependency inversion and inversion of control that you may have come across already - two of these are quite popular in the Java world - Spring (the IoC container) and the concept of dependency injection in Enterprise Java Beans (EJB).

This also comes back to a phrase that you might have heard elsewhere in the world of object-oriented programming - "coding to an interface".  By writing our code in such a way that it depends on interfaces (or abstractions) rather than concrete implementations, it becomes easier to swop out different implementations.  In effect, the code knows that it depends on an object that can do something, but it doesn't care about how that something is done.  One place where this can prove to be particularly useful is when we want to use mock objects in our test cases; we can easily inject those mocks without affecting the production code being tested - the code doesn't know that its dependency is actually a mock object.