Class on 10/07/15. These notes go with a slide presentation.

Slide presentation: <http://wcccedu.github.io/CPT-163-SOLID/#/>

**Slide 1**

S.O.L.I.D., Not Just a State of Matter by Paul Scarrone

@PaulSCoder [paul.scarrone@gmail.com](mailto:paul.scarrone@gmail.com)

**Slide 2**

This is a big picture view of design

<http://www.kirkk.com/modularity/2009/12/solid-principles-of-class-design/>

**Slide 3**

S is for Single Responsibility Principle

**Slide 4**

Classes should change for only a single reason.

The basis for this principle is cohesion. Cohesion represents the measure to which a class performs a single function. Classes that are highly cohesive are easier to understand. But they are also easier to maintain. This is the motivating force behind SRP. If a class has more than one reason to change, then it stands to reason that the responsibilities of that class that are the cause of change should be separated into multiple classes.

Cohesion is not a concept new to objects. In fact, the concept is taught in most introductory programming courses. Ironically, I’ve found that while most developers can easily define cohesion and explain it’s benefits, few developers actually apply it. Cohesion measures the degree to which an entity does a single thing. Given this definition, it’s no surprise that if some entity is responsible for performing only a single thing, most of our entities should be fairly small. Yet I commonly come across methods that run well over 100 lines of code, and classes that run orders of magnitudes larger than that. I struggle to convince myself that either are very cohesive. When you’re designing a highly cohesive system, you have to think small.

**Slide 5**

O is for Open Closed Principle

**Slide 6**

Classes should be open for extension, but closed for modification.

The Open Closed Principle (OCP) is undoubtedly the most important of all the class category principles. In fact, each of the remaining class principles are derived from OCP. It originated from the work of Bertrand Meyer who is recognized as an authority on the object-oriented paradigm. OCP is states that we should have the ability to add new features to our system without having to modify our set of preexisting classes. As stated previously, one of the benefits of the object-oriented paradigm is to enable us to add new data structures to our system without having to modify the existing system’s code base.

Let’s look at an example to see how this can be done. Consider a financial institution where we have to accommodate different types of accounts to which individuals can make deposits. Figure 4.1 shows a class diagram, with accompanying descriptions of some of the elements and how we might structure a portion of our system. For the purposes of our discussion in this chapter, we will focus on how the OCP can be used to extend the system.

Our Account class has a relationship to our AccountType abstract class. In other words, our Account class is coupled at the abstract level to the AccountType inheritance hierarchy. Because our Savings and Checking classes each inherit from the AccountType class, we know that through dynamic binding, we can substitute instances of either of these classes wherever the AccountType class is referenced. Subsequently, Savings and Checking can be freely substituted for AccountType within the Account class. This is the intent of an abstract class and enables us to effectively adhere to OCP by creating a contract between the Account class and the AccountType descendents. Because our Account is not directly coupled to either of the concrete Savings or Checking classes, we can extend the AccountType class, creating a new class such as MoneyMarket, without having to modify our Account class. We have achieved OCP and can now extend our system without modify its existing code base.

Figure 4.1 Therefore, one of the tenets of OCP is to reduce the coupling between classes to the abstract level. Instead of creating relationships between two concrete classes, we create relationships between a concrete class and an abstract class or, in Java, between a concrete class and an interface. When we create an extension of our base class, assuming we adhere to the public methods and their respective signatures defined on the abstract class, we have essentially achieved OCP. Let’s take a look at a simplified version of the Java code for this example, focusing on how we achieve OCP, instead of on the actual method implementations.

public class Account {

private AccountType \_act;

public Account(String act) {

try {

Class c = Class.forName(act);

this.\_act = (AccountType) c.newInstance();

} catch (Exception e) {

e.printStackTrace();

}

}

public void deposit(int amt) {

this.\_act.deposit(amt);

}

}

Here, our Account class accepts as an argument to its constructor a String representing the class we wish to instantiate. It then uses the Class class to dynamically create an instance of the appropriate AccountType subclass. Note that we don’t explicitly refer to either the Savings or Checking class directly.

public abstract class AccountType {

public abstract void deposit(int amt);

}

This is the abstract AccountType class that serves as the contract between our Account class and AccountType descendents. The deposit method is the contract.

public class CheckingAccount extends AccountType {

public void deposit(int amt) {

System.out.println();

System.out.println();

System.out.println("Amount deposited in checking account: " + amt);

System.out.println();

System.out.println();

}

}

public class SavingsAccount extends AccountType {

public void deposit(int amt) {

System.out.println();

System.out.println();

System.out.println("Amount deposited in savings account: " + amt);

System.out.println();

System.out.println();

}

}

Each of our AccountType descendents satisfies the contract by providing an implementation for the deposit method. In the real world, the behaviors of the individual deposit methods would be more interesting and, given the preceding design, would be algorithmically different.

**Slide 7**

WHA?

* You should design modules that never change
* When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

**Slide 8**

Open

A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.

**Slide 9**

Closed

A module will be said to be slosed if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

**Slide 10**

L is for Liskov’s Substitution Principle

**Slide 11**

Subclasses should be substitutable for their base classes.

We mentioned in our previous discussion that OCP is the most important of the class category principles. We can think of the Liskov Substitution Principle (LSP) as an extension to OCP. In order to take advantage of LSP, we must adhere to OCP because violations of LSP are also a violation of OCP, but not vice versa. LSP is the work of Barbara Liskov and is derived from Bertrand Meyer’s Design by Contract. In its simplest form, it is difficult to differentiate OCP and LSP, but a subtle difference does exist. OCP is centered around abstract coupling. LSP, while also heavily dependent on abstract coupling, is also heavily dependent on preconditions and postconditions, which is LSP’s relation to Design by Contract, where the concept of preconditions and postconditions was formalized.

A precondition is a contract that must be satisfied before a method can be invoked. A postcondition, on the other hand, must be true upon method completion. If the precondition is not met, the method should not be invoked, and if the postcondition is not met, the method should not return. The relation of preconditions and postconditions has meaning embedded within an inheritance relationship that is not supported within Java, outside of some manual assertions or nonexecutable comments. Because of this, violations of LSP can be difficult to find.

To illustrate LSP and the interrelationship of preconditions and postconditions, we need only consider how Java’s exception handling mechanism works. Consider a method on an abstract class that has the following signature:

public abstract deposit(int amt) throws InvalidAmountException

Assume in this situation, that our InvalidAmountException is an exception defined by our application, is inherited from Java’s base Exception class, and can be thrown if the amount we try to deposit is less than zero. By rule, when overriding this method in a subclass, we cannot throw an exception that exists at a higher level of abstraction than InvalidAmountException. Therefore, a method declaration such as the following is not allowed:

public void deposit(int amt) throws Exception

This method declaration is not allowed because the Exception class thrown in this method is the ancestor of the InvalidAmountException thrown previously. Again, we cannot throw an exception in a method on a subclass that exists at a higher level of abstraction than the exception thrown by the base class method we are overriding. On the other hand, reversing these two method signatures would have been perfectly acceptable to the Java compiler. We can throw an exception in an overridden subclass method that is at a lower level of abstraction than the exception thrown in the ancestor. While this does not correspond directly to the concept of preconditions and postconditions, it does capture the essence. Therefore, we can state that any precondition stipulated by a subclass method cannot be stronger than the base class method.Therefore, any postcondition stipulated by a subclass method cannot be weaker than the base class method.

To adhere to LSP in Java, we must make sure that developers define preconditions and postconditions for each of the methods on an abstract class. When defining our subclasses, we must adhere to these preconditions and postconditions. If we do not define preconditions and postconditions for our methods, it becomes virtually impossible to find violations of LSP. Suffice it to say, in the majority of cases, OCP will be our guiding principle.

**Slide 12**

Precondition

* A contract that must be satisfied before a method can be invoked.
* If the precondition is not met, the method should not be invoked.

**Slide 13**

Postcondition

* Must be true upon method completion
* If the postcondition is not met, the method should not return.

**Slide 14**

I is for Interface Segregation Principle

**Slide 15**

Many specific interfaces are better than a single, general interface.

Put simply, any interface we define should be highly cohesive. In Java, we know that an interface is a reference data type that can have method declarations, but no implementation. In essence, an interface is an abstract class with all abstract methods. As we define our interfaces, it becomes important that we clearly understand the role the interface plays within the context of our application. In fact, interfaces provide flexibility: They allow objects to assume the data type of the interface. Subsequently, an interface is simply a role that an object plays at some point throughout its lifetime. It follows, rather logically, that when defining the operation on an interface, we should do so in a manner that does not accommodate multiple roles. Therefore, an interface should be responsible for allowing an object to assume a single role, assuming the class of which that object is an instance implements that interface.

While working on a project recently, an ongoing discussion took place as to how we would implement our data access mechanism. Quite a bit of time was spent designing a flexible framework that would allow uniform access to a variety of different data sources. These back-end data sources might come in the form of a relational database, a flat file, or possibly even another proprietary database. Therefore, our goal was not only to provide a common data access mechanism, but also to present data to any class acting as a data client in a consistent manner. Doing so would clearly decouple our data clients from the back-end data source, making it much easier to port our back-end data sources to different platforms without impacting our data clients. Therefore, we decided that all data clients would depend on a single Java interface, depicted in Figure 4.3, with the associated methods.

Figure 4.3

At first glance, the design depicted in Figure 4.3 seemed plausible. After further investigation, however, questions were raised as to the cohesion of the RowSetManager interface. What if classes implementing this interface were read-only and didn’t need insert and update functionality? Also, what if the data client were not interested in retrieving the data, but only in iterating its already retrieved internal data set? Exploring these questions a bit further, and carefully considering ISP, we found that it was meaningful to have a data structure that wasn’t even dependent on a retrieve action at all. For instance, we may wish to use a data set that was cached in memory and wasn’t dependent on an underlying physical data source. This led us to the design in Figure 4.4.

Figure 4.4

In Figure 4.4, we see that we have segregated the reponsibilities of our RowSetManager into multiple interfaces. Each interface is responsible for allowing a class to adhere to a cohesive set of responsibilities. Now, our application can implement the interfaces necessary to provide the desired set of functionality. We are no longer forced to provide data update behavior if our class is read-only.

**Slide 16**

Specific Interface

An interface should be responsible for allowing an object to assume a single role, assuming the class of which that object is an instance implements that interface.

**Slide 17**

D is for Dependency Inversion Principle

**Slide 18**

Depend upon abstractions. Do not depend upon concretions.

The Dependency Inversion Principle (DIP) formalizes the concept of abstract coupling and clearly states that we should couple at the abstract level, not at the concrete level. Abstract coupling is the notion that a class is not coupled to another concrete class, or class that can be instantiated. Instead, the class is coupled to other base, or abstract, classes. In Java, this abstract class can be either a class with the abstract modifier or a Java interface data type. Regardless, this concept is actually the means through which LSP achieves its flexibility, the mechanism required for DIP, and the heart of OCP.

In our own designs, attempting to couple at the abstract level can at times seem like overkill. Pragmatically, we should apply this principle in any situation where we are unsure whether the implementation of a class may change in the future. We have encountered situations during development where we know exactly what needs to be done. Requirements state this very clearly, and the probability of change or extension is quite low. In these situations, adherence to DIP may be more work than the benefit realized. At this point, there exists a striking similarity between DIP and OCP. In fact, these two principles are closely related. Fundamentally, DIP tells us how we can adhere to OCP. Or, stated differently, if OCP is the desired end, DIP is the means through which we achieve that end. While this statement may seem obvious, we commonly violate DIP in a certain situation and don’t even realize it.

When we create an instance of a class in Java, we typically must explicitly reference that object. Only after the instance has been created can we flexibly reference that object via its ancestors or implemented interfaces. Therefore, the moment we reference a class to create it, we have violated DIP and, subsequently, OCP. Recall that in order to adhere to OCP, we must first take advantage of DIP. There are a couple of different ways to resolve this.

The first way to resolve this impasse is to dynamically load the object using the Class class and its newInstance method. However, this solution can be problematic and somewhat inflexible. Because DIP doesn’t allow us to refer to the concrete class explicitly, we must use a String representation of the concrete class. For instance, consider the following:

Class c = Class.forName("SomeDescendent"); SomeAncestor sa = (SomeAncestor) c.newInstance();

In this example, we wish to create an instance of the class SomeDescendent in the first line, but reference it as type SomeAncestor in the second line. This was also illustrated in the code samples in the section “Open Closed Principle (OCP),” earlier in this chapter. This is perfectly acceptable, as long as the SomeDescendent class is inherited, either directly or indirectly, from the SomeAncestor class. If it is not, our application will throw an exception at runtime. Another more obvious problem occurs when we misspell the class of which we want an instance. Yet another, less apparent, obstacle eventually is encountered when taking this approach. Because we reference the class name as a string, there isn’t any way to pass parameters into the constructor of this class. Java does provides a solution to this problem, but it quickly becomes complex, unwieldy, and error prone.

Another approach to resolving the object creation challenge is to use an object factory. Here, we create a separate class whose only responsibility is to create instances. This way, our original class, where the instance would have previously been created, stays clear of any references to concrete classes, which have been removed and placed in this factory. The only references contained within this class are to abstract, or base, classes. The factory does, however, reference the concrete classes, which is, in fact, a blatant violation of DIP. However, it is an isolated and carefully thought through violation and is therefore acceptable.

Keep in mind that we may not always need to use an object factory. Along with the flexibility of a factory comes the complexity of a more dynamic collaboration of objects. Concrete references are not always a bad thing. If the class to which we are referring is a stable class, not likely to undergo many changes, using a factory adds unwarranted complexity to our system. If a factory is deemed necessary, the design of the factory itself should be given careful consideration.

**Slide 19**

Wha?

Use a “factory” to create instances of your classes so they do not have to be coupled on the concrete level.

**Slide 20**

Resources

* Examples
* SOLID Source