Building SOLID Foundations of OOP, Part 3: LSP

05 December 2012

So far we've looked at Single Responsibility Principle and the Open / Closed Principle, which brings us to the "L" of SOLID, the Liskov Substitution Principle.

The Liskov Substitution Principle (LSP) is named after Barbara Liskov. She first introduced the idea that became the Liskov Substitution Principle in 1988 with a description of a particular scenario:

> What is wanted here is something like the following substitution property: If for each object O_1 of type S there is an object O_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O_1 is substituted for O_2 then S is a subtype of T.

That's cool, and perfectly clear, but how about a little more simple way of thinking about it?

> If my cookie jar takes cookies, and I can fit a Girl Scout thin mint cookie into my cookie jar, and I can also fit an Oreo into my cookie jar, then in the context of my cookie jar, thin mints and Oreos are subtypes of each other.

This example is a little contrived, because there are plenty of ways in which we could also say a thin mint cookie is not a subtype of an Oreo, but if our application is a cookie jar, and our expected behavior for cookies is that they fit into the cookie jar, then thin mints and Oreos are subtypes. This is still an abstract idea though, which is good, we want to embrace abstraction as one of the greatest tools we get to use as a OOP developer. However, to further flush out the idea of LSP, let's talk about a simple, physical example:

VS.

In some ways flour and sugar are subtypes of each other, depending on how we define them and what we expect of them. Both products are white, and have similar textures, are both used in baking, and both are derived from plants. Based on this set of criteria, we might be tempted to think we could easily substitute sugar for flour, or flour for sugar with no negative side effects. That is until we wanted to make cookies, but discovered to our dismay that we had run out of sugar last week, and being one of those items that isn't bought often, was sadly left off the grocery list for yesterday's shopping trip. We might ask ourselves, what could we substitute for sugar? We see flour in the cupboard, would that work? Well, the obvious answer is no, but that's because our brains are smart and understand how to look at similarities and differences at the same time. A program is only as smart as we make it, however, and unless we specifically made our program aware that flour is not a suitable substitute for sugar, it would use flour.

This example serves to highlight a very important fact about LSP - context. In determining if two classes are subtypes of each other, we need to understand the context in which we're making expectations on them. If we expect two classes to act in the same so that we can consider them to be subtypes, we must not only be careful in how we define our classes, but also how we define the contexts in which they behave.

We've said before that abstraction is our friend in the world of OOP, and having properly defined abstractions in our applications allows us to preserve LSP. I figure this is as good a time as any to list the three main benefits of OOP:

1. **Encapsulation** (classes) give our objects purpose and direction, and create contexts in which the various objects of applications carry expectations of the messages they are capable of sending and receiving. Classes are the logical building blocks of our application's design.
2. **Inheritance** serves as the core foundation for creating abstractions. With inheritance, we can define abstract interfaces that concrete classes inherit. This gives

our classes pre-defined behavior via a minimum set of virtual methods (defined, but unimplemented methods). Inheritance defines how related classes should behave without needing to concretely define that behavior for each class (think DRY).

3. **Polymorphism** extends the use of inheritance by allowing us to customize how various pre-defined behaviors are implemented in our concrete classes. Most commonly this is achieved through method over-riding.

With flour and sugar in mind, the importance of context, and our three benefits of OOP, let's take a look at some code. In yesterday's post we worked with our CheckingAccount example to show how we could preserve SRP and OCP by building an interface. This interface allowed us to *extend* CheckingAccount without *modifying* it. We completed our user story requesting a monthly playment feature, and the client was happy with the feature demo. At this week's iteration planning meeting, a new user story emerged:

```
As a bank
I want the ability to calculate interest earned for both checking and savings accounts
```

Hmm, well so far we've only been concerned with checking accounts, but now we are to add savings accounts into the mix, and calculate interest earned. Now feeling more comfortable with abstraction in OOP along with SRP and OCP, we think of a few options:

1. Create a new SavingsAccount class.
2. Create an Account superclass to serve as an abstract interface for both CheckingAccount and SavingsAccount.
3. Create an interface for CheckingAccount and SavingsAccount classes to talk with our CalcMonthlyInterest class.

Let's look at each option. Does simply creating a new SavingsAccount class right now violate any of the SOLID principles? Nope, and it is an excellent example of SRP and if we play our cards right, also LSP. Does providing an abstract interface for CheckingAccount and SavingsAccount violate any of the SOLID principles? Nope, and if we consider that our SavingsAccount class will have very similar behavior to that of our CheckingAccount class, we consider this to be a good way to preserve the DRY principle while reinforcing LSP (both checking and savings accounts are subtypes for each other if we carefully design interfaces for them). It also has the added bonus of taking advantage of one of OOP's benefits - inheritance.

But what about the third option? Does creating an interface for CheckingAccount and SavingsAccount classes to talk with our CalcMonthlyInterest class preserve SOLID principles? Yes, and it is another example of how we can preserve the Open / Closed Principle. In this case, we decide it's a good idea to build a SavingsAccount class, an Account super class, and implement an interface for another new class we are to create, CalcMonthlyInterest. Let's get started.

```
class Account
  attr_accessor :owner, :balance, :account_num, :routing_num
  def initialize(options)
    self.owner       = options[:owner]
    self.balance     = options[:balance]
    self.account_num = options[:account_num]
    self.routing_num = options[:routing_num]
  end

  def debit(amount)
    self.balance -= amount
  end

  def deposit(amount)
    self.balance += amount
  end
end
```

Look familiar? It's identical to our current CheckingAccount class. Here we reason that all accounts will at least utilize the basic behaviors of depositing and debiting, and will need to contain the owner's name, balance, account number and routing number. Other than that, we can allow different account types to add more specific behavior as they need it when we create those classes. Now that we have Account superclass to inherit from, let's see how CheckingAccount and SavingsAccount appear now:

```
class CheckingAccount < Account
end

class SavingsAccount < Account
end
```

With inheritance, we've managed to abstract the behaviors and data of CheckingAccount and SavingsAccount. As they stand at this time both classes are subtypes of each other, meaning they can be substituted for each other given the right context. This might look funny, but the distinction between name spaces is important. We also don't know what types of accounts we may need to account for in the future with our application. It might turn out that other accounts will require more complex behavior than our Account superclass defines, but will still require the fundamental behaviors of debit and deposit.

Now that we have a SavingsAccount class and an Account superclass, let's think about the interface between our accounts and CalcMonthlyInterest. In this case, I'd prefer to use a top-down development approach. Basically I want to think through what my application needs starting at the highest level of abstraction (the interface) and what messages I want to send to the next level of abstraction (the CalcMonthlyInterest class). For a very interesting read, check out the article Up and Down the Ladder of Abstraction written by the equally thought provoking dude, Brett Victor. Since we have so far preserved LSP, our interface can handle both account types like so:

```
class AccountCalcMonthlyInterestInterface
  attr_accessor :account, :calc_monthly_interest
  def initialize(options)
    self.account               = options[:account]
    self.calc_monthly_interest = CalcMonthlyInterest.new
  end

  def apply_interest
    if account.class == "CheckingAccount"
      calc_monthly_interest.apply_checking_interest(account)
    else
      calc_monthly_interest.apply_savings_interest(account)
    end
  end
end
```

We followed the basic ideas of the SOLID principles we have learned so far, but something still doesn't quite seem right about this code. Even without implementing the CalcMonthlyInterest class, we see a big "if" conditional statement in our `apply_interest` method. All of a sudden, we see that our CheckingAccount and SavingsAccount classes really aren't that substitutable because checking accounts have a different interest rate calculation than savings accounts do. Now our AccountCalcMonthlyInterestInterface class requires *knowing* what kind of account it's dealing with. Even though at this time this "if" conditional is fairly simple, it can quickly grow out of control. This is because as we add more account types to our application, we will be required to continue performing conditional checks like this, until we find ourselves in very long chains of conditional checking that are a headache to sort through.

Are CheckingAccount and SavingsAccount really not subtypes of each other afterall? Well, we know that CheckingAccount and SavingsAccount are defined in the same way, and so they should be subtypes. Yet, the context in which we expect them to be subtypes is not so well defined, and suddenly we violate LSP. What can we do in this situation? The solution will appear tomorrow when we dive into the Interface Segregation Principle. Consider this user story half-complete, it's already 5:30 pm and you desperately want to catch the Alan Turing film being screened later in the evening...

Tags

← Previous Archive Next →

SOLID [5]

Liskov Substitution [2]

LSP [1]

Barbara Liskov [1]

Encapsulation [2]

Inheritance [2]

Polymorphism [2]

Brett Victor [1]

Up and Down the Ladder of Abstraction [1]

Top Down Development [1]

Alan Turing [1]