9 LSP: The Liskov Substitution Principle

In 1988, Barbara Liskov wrote the following as a way of defining subtypes.

> *What is wanted here is something like the following substitution property: 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.*[1]

To understand this idea, which is known as the Liskov Substitution Principle (LSP), let's look at some examples.

GUIDING THE USE OF INHERITANCE

Imagine that we have a class named `License`, as shown in Figure 9.1. This class has a method named `calcFee()`, which is called by the `Billing` application. There are two "subtypes" of `License`: `PersonalLicense` and `BusinessLicense`. They use different algorithms to calculate the license fee.

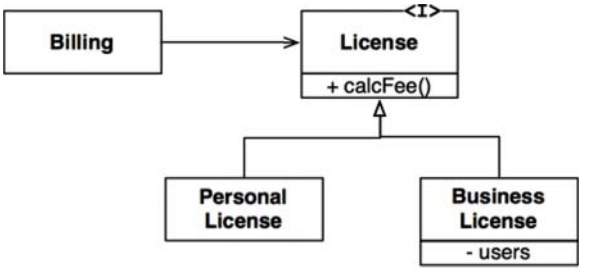

Figure 9.1 `License`, and its derivatives, conform to LSP

This design conforms to the LSP because the behavior of the `Billing` application does not depend, in any way, on which of the two subtypes it uses. Both of the subtypes are substitutable for the `License` type.

1. Barbara Liskov, "Data Abstraction and Hierarchy," *SIGPLAN Notices* 23, 5 (May 1988).

THE SQUARE/RECTANGLE PROBLEM

The canonical example of a violation of the LSP is the famed (or infamous, depending on your perspective) square/rectangle problem (Figure 9.2).

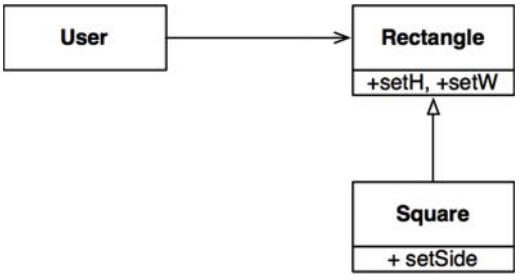

Figure 9.2 The infamous square/rectangle problem

In this example, `Square` is not a proper subtype of `Rectangle` because the height and width of the `Rectangle` are independently mutable; in contrast, the height and width of the `Square` must change together. Since the `User` believes it is communicating with a `Rectangle`, it could easily get confused. The following code shows why:

```
Rectangle r = …
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

If the … code produced a `Square`, then the assertion would fail.

The only way to defend against this kind of LSP violation is to add mechanisms to the `User` (such as an `if` statement) that detects whether the `Rectangle` is, in fact, a `Square`. Since the behavior of the `User` depends on the types it uses, those types are not substitutable.

LSP AND ARCHITECTURE

In the early years of the object-oriented revolution, we thought of the LSP as a way to guide the use of inheritance, as shown in the previous sections. However, over the years the LSP has morphed into a broader principle of software design that pertains to interfaces and implementations.

The interfaces in question can be of many forms. We might have a Java-style interface, implemented by several classes. Or we might have several Ruby classes that share the same method signatures. Or we might have a set of services that all respond to the same REST interface.

In all of these situations, and more, the LSP is applicable because there are users who depend on well-defined interfaces, and on the substitutability of the implementations of those interfaces.

The best way to understand the LSP from an architectural viewpoint is to look at what happens to the architecture of a system when the principle is violated.

EXAMPLE LSP VIOLATION

Assume that we are building an aggregator for many taxi dispatch services. Customers use our website to find the most appropriate taxi to use, regardless of taxi company. Once the customer makes a decision, our system dispatches the chosen taxi by using a restful service.

Now assume that the URI for the restful dispatch service is part of the information contained in the driver database. Once our system has chosen a driver appropriate for the customer, it gets that URI from the driver record and then uses it to dispatch the driver.

Suppose Driver Bob has a dispatch URI that looks like this:

```
purplecab.com/driver/Bob
```

Our system will append the dispatch information onto this URI and send it with a PUT, as follows:

```
purplecab.com/driver/Bob
        /pickupAddress/24 Maple St.
        /pickupTime/153
        /destination/ORD
```

Clearly, this means that all the dispatch services, for all the different companies, must conform to the same REST interface. They must treat the `pickupAddress`, `pickupTime`, and `destination` fields identically.

Now suppose the Acme taxi company hired some programmers who didn't read the spec very carefully. They abbreviated the destination field to just `dest`. Acme is the largest taxi company in our area, and Acme's CEO's ex-wife is our CEO's new wife, and … Well, you get the picture. What would happen to the architecture of our system?

Obviously, we would need to add a special case. The dispatch request for any Acme driver would have to be constructed using a different set of rules from all the other drivers.

The simplest way to accomplish this goal would be to add an `if` statement to the module that constructed the dispatch command:

```
if (driver.getDispatchUri().startsWith("acme.com"))…
```

But, of course, no architect worth his or her salt would allow such a construction to exist in the system. Putting the word "acme" into the code itself creates an opportunity for all kinds of horrible and mysterious errors, not to mention security breaches.

For example, what if Acme became even more successful and bought the Purple Taxi company. What if the merged company maintained the separate

brands and the separate websites, but unified all of the original companies' systems? Would we have to add another if statement for "purple"?

Our architect would have to insulate the system from bugs like this by creating some kind of dispatch command creation module that was driven by a configuration database keyed by the dispatch URI. The configuration data might look something like this:

URI	Dispatch Format
`Acme.com`	`/pickupAddress/%s/pickupTime/%s/dest/%s`
`*.*`	`/pickupAddress/%s/pickupTime/%s/destination/%s`

And so our architect has had to add a significant and complex mechanism to deal with the fact that the interfaces of the restful services are not all substitutable.

CONCLUSION

The LSP can, and should, be extended to the level of architecture. A simple violation of substitutability, can cause a system's architecture to be polluted with a significant amount of extra mechanisms.