Object-Oriented Design Principles
Brahma Dathan and Sarnath Ramnath

Subtyping and the Liskov Substitution Principle

Perhaps the most insidious consequence of freely extending classes is the creation of false subtypes. Requiring every subclass instance to be used as a superclass type is the essence of **substitutability**. One of the rules that is implicit in the use of inheritance is the **Liskov Substitution Principle (LSP)** which can be summarized as follows:

Any superclass object should be substitutable by a subclass object.

The seeds of this originated with Bertrand Meyer, who first proposed the idea in his1988 book. Meyer, and later on, Pierre America, made attempts to formalize the notion of subtype, but the issue was finally settled by Liskov and Wing in 1994.

Polymorphic assignments can result in methods being invoked on a subclass object. Certifying substitutability is complicated, because we cannot think of all possible contexts in which the subclass object may substitute a superclass object. It is therefore useful to formalize the notion. If we view the superclass object's expected behavior as a contract, the subclass object must uphold this contract.

From a practical point of view, what we understand is extending an existing class to accommodate new requirements can be dangerous. It is therefore preferable to define an abstract supertype with all abstract methods and extend this to define any number of concrete subtypes. In all contexts, the code is dependent only on the abstract supertype, and therefore we do not face the problem of a concrete subclass extending another concrete superclass. In situations where an existing concrete implementation has to be reused by another concrete class, we prefer to use composition, i.e, the composite-reuse principle, often summarized by the phrase “Favor composition over inheritance” (Gamm et al., Design Patterns).

This principle advocates that designers should achieve code reuse by their composition (by containing instances of other classes that implement the desired functionality) rather than inheritance from a base or parent class.

Now, let us look at inheritance from concrete classes. Consider the class Counter given below.

```
public class Counter {
  protected int count;

  public void up() {
    count++;
  }

  public int getValue() {
    return count;
  }
```

```
    public void reset() {
        count = 0;
    }
}
```

Here is client code using the above class.

```
public class CounterClient {
    public void doSomething(Counter counter) {
        while (counter.getValue() < 10) {
            System.out.println("doing work cycle " + counter.getValue());
            counter.up();
        }
    }
}
```

Now we extend the Counter class to create Counter2, which counts up by 2, as opposed to 1.

```
public class Counter2 extends Counter {
    public void up() {
        count += 2;
    }
}
```

Suppose we call the `doSomething()` method with an instance of Counter2 as argument to the method.

Obviously, the client code does not produce the same result as it did with an actual type of Counter as its parameter. Therefore, making Counter2 a subclass of Counter is inappropriate.

Next, consider the following code.

```
public class EnhancedCounter extends Counter {
    public void down() {
        if (count > 0) {
            count--;
        }
    }
}
```

We could pass an instance of `EnhancedCounter` as parameter to the parameter to the `doSomething()`method. The behavior is the same as before with an actual type of Counter as parameter. The reason is that the methods of Counter are left unchanged.

Let us look at one more example.

We have a class named Set below.

```
public class Set {
    public Object remove() {
        // code not shown
    }

    public void add(Object object) {
        // code not shown
    }

    public int size() {
       // code not shown
    }
}
```

Let us extend this as below.

```
public class List extends Set {
    public Object remove() {
        // code not shown
    }
    public void add(Object s1) {
        // code not shown
    }
    public int size() {
       // code not shown
    }
}
```

Simply declaring the methods as above cannot help us decide whether List is a proper subtype of Set. Instead, we can get a better handle on the problem by writing the expectations on the methods.

Suppose the postconditions for the three methods of Set are as follows.

add(): Adds the object to the collection if there is room.
remove(): If the collection is not empty, some element of the collection is removed and returned. If the collection is empty, the method returns null.
size(): Returns the number of elements of the collection.

Assume that the postconditions for the three methods of Set are as follows.

add():Adds the object to the end of the collection if there is room.
remove(): If the collection is not empty, the first element of the collection is removed and returned. If the collection is empty, the method returns null.
size(): Returns the number of elements of the collection.

A close look at the postconditions tells us the following for the corresponding methods of Set and List:

The postconditions for the two size() methods are identical.
The postcondition for add() in List is a special case of the postcondition for add() in Set.
The postcondition for remove() in List is a special case of the postcondition for remove() in Set.

Thus in each method, the List postcondition is the same as or a special case of the postcondition of the respective Set method. The methods of List don't do anything that is not expected of a Set instance. Therefore, as far as the postconditions are concerned, List is a proper subtype of Set.

Another condition that we need to check is whether the methods of List would take as parameters any parameter that could be supplied to the corresponding method of Set. Here, it is easy to verify that the List methods are just as generous.

What Does Java Permit?

Suppose C is a subtype of D. If a class A has a method

```
public D m() {
}
```

A subclass B of A may have a method

```
@Override
public C m() {
}
```

If A has a method

```
public void m(D d1) {
}
```

The subclass B cannot have the method

```
@Override
public void m(C c1) {
}
```

If A has a method

```
public void m(C c1) {
}
```

The subclass cannot have the following method either. (This is permitted by LSP, though.)

```
@Override
public void m(D d1) {
}
```

That is, the parameters have to be of the same type for the override to work.

Some Guidelines for Subclassing

Do not rush in too soon. Remember that inheritance is a relationship between well-understood abstractions and the hierarchy usually emerges ``naturally" in our process. This takes time, except in situations where our data has a pre-existing taxonomy. This implies that we have a clear data abstraction in mind before constructing the hierarchy.

Allow for future expansion. Keep in mind that we cannot guess how our system might be used; the best way to plan for that is to be generous when allowing for variations. The rules for this are

- Define methods to be as general as possible at each level of an inheritance hierarchy. When writing methods avoid details that are too specifically tailored for the current set of subclasses; the methods should abstract out common functionality so that subclasses can invoke the superclass method to perform some of the task.
- Be generous in defining data types and storage to avoid difficult changes later on. For example, you might consider using a variable of type double even though your current data may only require a float variable.
- Choose the right access modifiers for your attributes. Applying the optimal access levels to members of a class hierarchy makes the hierarchy easier to maintain by allowing you to control how such members will be used. Declare class members with access modifiers that provide the least amount of access feasible.
- Only expose items that are needed by derived classes. Keeping fields private helps descendants and clients by reducing naming conflicts and protects them from using items that may need to be changed at a later stage. Members that are only needed by descendants should be marked as protected. This ensures that only the derived classes are dependent on these members and makes it easier to update these members during development.

The functionality provided by the methods of the base class should not depend on features that can be overridden. Make sure that base class methods do not depend on features that can be changed by inheriting classes.

The Open-Closed Principle

An important guiding tenet of object-oriented design can be summed up in what is referred to as the Open-Closed Principle (OCP):

A module must be Open for Extension but Closed for Implementation

Extension is the process by which new features are added to existing software, and implementation is the process that converts an abstract design into concrete code. What this statement implies is that our classes and modules must be written in such a manner that they can be extended, i.e., new features can be added, without re-opening the completed implementation, i.e., without need for modification of the existing code. It is obvious that OCP is highly desirable, but a deeper understanding is needed when we apply it. Adding new features to software often requires changes to existing classes. This is a non-trivial task, but the propagation of the effects of change can be contained by proper encapsulation. Sometimes changes can mostly be handled by defining a separate class that incorporates the new features and have only a minimal effect on other classes. This new class, however, needs to be related in some way with the existing classes, without changing their implementation.

The primary mechanism for enabling this is to define the abstraction in a superclass and place the variations in concrete subclasses. Without an inheritance hierarchy, such a feat is impossible to accomplish; inheritance allows us to extend software by adding features through new implementations of the same abstraction, even though the abstract ancestor class and the existing concrete implementations remain closed.

A naive application of inheritance will not satisfy OCP; a thorough understanding of how the implementation will be extended is essential.

The Visitor pattern is a good example of achieving extension of functionality without modification to existing code.

The Stable-Dependency Principle

Changes are almost always inevitable in systems. As user requirements change, we may need to modify existing code or as the open-closed principle suggests, try to accommodate all or most of the changes by extending the system. But this is not always realistic.

If we modify existing code in some classes (hopefully not in interfaces), it is likely that other classes dependent on the modified classes may require changes. In this sense, we would like to think of classes and interfaces in terms of how “stable” they are.

A class that depends on no other classes or interfaces is relatively stable in the sense that it needs to be modified only in some cases, perhaps in situations involving user requirement changes. An interface is even more stable because it does not have any implementation and is just a specification.

We can, in general, say that when we design a class, we should attempt to make it depend less on other classes. We should try to restrict its dependency to interfaces, because interfaces are more stable than classes. For example, when we write code for a collection, instead of writing the following code

```
LinkedList myList1 = new LinkedList();
```

We should write,

```
List myList2 = new LinkedList();
```

This way, myList will depend on the interface List, and we will not use methods that are unique to LinkedList. For example, if someObject is a valid reference, myList1.addFirst(someObject) is a valid statement, whereas myList2.addFirst(someObject) is not. This makes the code using myList2 less dependent on the classes and we can more easily modify the code. For example, we can declare myList2 as below.

```
List myList2 = new ArrayList();
```

Such a change may not be possible for myList1, because the code is more dependent on the actual class.

The idea can be extended to complete systems. Consider the implementation of the Drawing Program. For each package, we determine the number of dependencies on classes in other packages, from classes in this package. For example, in the package controller, we have two members: UndoManager.java and Controller.java. We ignore the interface Controller.java. (Interfaces are any way stable.) UndoManager.java uses the classes controller.commands.Command and model.Model. So this package has two dependencies, one to controller.commands and one to model.

The complete data for all packages in the Drawing Program is shown in Table 1.

Martin has applied the idea of SDP in the context of packages. If organization of the packages and stability among them is important, his ideas are worth considering. According to him, the dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable that it is. His reasoning is that modules that are designed to be unstable (i.e. easy to change) should not be depended upon by modules that are more stable (i.e. harder to change) than they are.

For each package, we compute the following:

1) DEPENDS_ON = the number of dependencies on classes in other packages, from classes in this package.
2) DEPENDENT_ON = the number of dependencies on classes in this package, from classes in other packages

Martin computes an instability index, but we reformulate his idea to compute the stability of a package as DEPENDENT_ON/(DEPENDS_ON+DEPENDENT_ON)

A stability value of 1 is ideal.

The hope is that we can use such information to help us decide

1) if the design is sound
2) on plan to redesign the system, if necessary, and
3) on a way to test the system

	controller	controller.commands	controller.events	controller.states	gui	gui.controller.buttons	gui.controller.handlers	gui.panels	gui.view.renderers	gui.view.views	model	model.shapes	view	# dependencies from this package	Stability
controller	0	1	0	0	0	0	0	0	0	0	1	0	0	2	0.67
controller.commands	0	0	0	0	0	0	0	0	0	0	3	3	0	6	0.40
controller.events	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00
controller.states	4	8	51	0	0	0	0	0	0	2	1	4	1	71	0.05
gui	0	0	0	0	0	2	3	2	0	0	0	0	0	7	0.36
gui.controller.buttons	0	0	9	9	0	0	0	0	0	0	0	0	0	18	0.18
gui.controller.handlers	0	0	7	3	0	0	0	0	0	0	0	0	0	10	0.29
gui.panels	0	0	2	1	0	0	0	0	2	0	1	0	2	8	0.33
gui.view.renderers	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00
gui.view.views	0	0	0	0	0	0	0	1	0	0	0	0	2	3	0.57
model	0	0	0	0	0	0	0	0	0	0	0	1	1	2	0.67
model.shapes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00
view	0	0	0	0	0	0	0	0	0	0	1	1	0	2	0.67
#dependencies on this package	4	9	69	13	0	2	3	3	2	2	7	9	6		

Table 1

The numbers do not necessarily paint a correct picture. The stability of controller.states is 0.05, but that is not quite reflective of the stability of the package. The classes in the package are highly dependent on the events and to some extent on the shapes, and both packages are highly stable. The low stability value is due to the relatively few classes that use classes in the package. In reality, we hardly expect to make many changes to classes in this package.

On the other hand, looking at the dependencies yields some useful information: controller.states is dependent on gui.view.views, which is highly undesirable. No matter what the numbers may tell us, the gui package and its sub-packages are prime candidates for change in the future. We know this, not from any numbers in the table, but our knowledge of how things have changed in Java over the years. Therefore, this calls for a redesign. (The FXView and SwingView instances should be constructed directly by the menu items, rather than going through the context and DrawingState.)