JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox.html

Why extends is evil

Improve your code by replacing concrete base classes with interfaces

By Allen Holub, JavaWorld.com, 08/01/03

The `extends` keyword is evil; maybe not at the Charles Manson level, but bad enough that it should be shunned whenever possible. The Gang of Four *Design Patterns* book discusses at length replacing implementation inheritance (`extends`) with interface inheritance (`implements`).

Good designers write most of their code in terms of interfaces, not concrete base classes. This article describes *why* designers have such odd habits, and also introduces a few interface-based programming basics.

Interfaces versus classes

I once attended a Java user group meeting where James Gosling (Java's inventor) was the featured speaker. During the memorable Q&A session, someone asked him: "If you could do Java over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the `extends` relationship). Interface inheritance (the `implements` relationship) is preferable. You should avoid implementation inheritance whenever possible.

Losing flexibility

Why should you avoid implementation inheritance? The first problem is that explicit use of concrete class names locks you into specific implementations, making down-the-line changes unnecessarily difficult.

At the core of the contemporary Agile development methodologies is the concept of parallel design and development. You start programming before you fully specify the

program. This technique flies in the face of traditional wisdom—that a design should be complete before programming starts—but many successful projects have proven that you can develop high-quality code more rapidly (and cost effectively) this way than with the traditional pipelined approach. At the core of parallel development, however, is the notion of flexibility. You have to write your code in such a way that you can incorporate newly discovered requirements into the existing code as painlessly as possible.

Rather than implement features you *might* need, you implement only the features you *definitely* need, but in a way that accommodates change. If you don't have this flexibility, parallel development simply isn't possible.

Programming to interfaces is at the core of flexible structure. To see why, let's look at what happens when you don't use them. Consider the following code:

```
f()
{   LinkedList list = new LinkedList();
    //...
    g( list );
}
g( LinkedList list )
{
    list.add( ... );
    g2( list )
}
```

Now suppose a new requirement for fast lookup has emerged, so the `LinkedList` isn't working out. You need to replace it with a `HashSet`. In the existing code, that change is not localized since you must modify not only `f()` but also `g()` (which takes a `LinkedList` argument), and anything `g()` passes the list to.

Rewriting the code like this:

```
f()
{   Collection list = new LinkedList();
    //...
    g( list );
}
g( Collection list )
{
    list.add( ... );
    g2( list )
}
```

makes it possible to change the linked list to a hash table simply by replacing the `new LinkedList()` with a `new HashSet()`. That's it. No other changes are necessary.

As another example, compare this code:

```
f()
{   Collection c = new HashSet();
    //...
    g( c );
}
g( Collection c )
{
    for( Iterator i = c.iterator(); i.hasNext() ;)
        do_something_with( i.next() );
}
```

to this:

```
f2()
{   Collection c = new HashSet();
    //...
    g2( c.iterator() );
}
g2( Iterator i )
{   while( i.hasNext() ;)
        do_something_with( i.next() );
}
```

The `g2()` method can now traverse `Collection` derivatives as well as the key and value lists you can get from a `Map`. In fact, you can write iterators that generate data instead of traversing a collection. You can write iterators that feed information from a test scaffold or a file to the program. There's enormous flexibility here.

Coupling

A more crucial problem with implementation inheritance is *coupling*—the undesirable reliance of one part of a program on another part. Global variables provide the classic example of why strong coupling causes trouble. If you change the type of the global variable, for example, all functions that use the variable (i.e., are *coupled* to the variable) might be affected, so all this code must be examined, modified, and retested. Moreover, all functions that use the variable are coupled to each other through the variable. That is, one function might incorrectly affect another function's behavior if a variable's value is changed at an awkward time. This problem is particularly hideous in multithreaded programs.

As a designer, you should strive to minimize coupling relationships. You can't eliminate coupling altogether because a method call from an object of one class to an object of another is a form of loose coupling. You can't have a program without some coupling.

Nonetheless, you can minimize coupling considerably by slavishly following OO (object-oriented) precepts (the most important is that the implementation of an object should be completely hidden from the objects that use it). For example, an object's instance variables (member fields that aren't constants), should always be `private`. Period. No exceptions. Ever. I mean it. (You can occasionally use `protected` methods effectively, but `protected` instance variables are an abomination.) You should never use get/set functions for the same reason—they're just overly complicated ways to make a field public (though access functions that return full-blown objects rather than a basic-type value are reasonable in situations where the returned object's class is a key abstraction in the design).

I'm not being pedantic here. I've found a direct correlation in my own work between the strictness of my OO approach, quick code development, and easy code maintenance. Whenever I violate a central OO principle like implementation hiding, I end up rewriting that code (usually because the code is impossible to debug). I don't have time to rewrite programs, so I follow the rules. My concern is entirely practical—I have no interest in purity for the sake of purity.

The fragile base-class problem

Now, let's apply the concept of coupling to inheritance. In an implementation-inheritance system that uses `extends`, the derived classes are very tightly coupled to the base classes, and this close connection is undesirable. Designers have applied the moniker "the fragile base-class problem" to describe this behavior. Base classes are considered fragile because you can modify a base class in a seemingly safe way, but this new behavior, when inherited by the derived classes, might cause the derived classes to malfunction. You can't tell whether a base-class change is safe simply by examining the base class's methods in isolation; you must look at (and test) all derived classes as well. Moreover, you must check all code that *uses* both base-class *and* derived-class objects too, since this code might also be broken by the new behavior. A simple change to a key base class can render an entire program inoperable.

Let's examine the fragile base-class and base-class coupling problems together. The following class extends Java's `ArrayList` class to make it behave like a stack:

```
class Stack extends ArrayList
{   private int stack_pointer = 0;
    public void push( Object article )
    {   add( stack_pointer++, article );
    }
    public Object pop()
    {   return remove( --stack_pointer );
    }
    public void push_many( Object[] articles )
    {   for( int i = 0; i < articles.length; ++i )
            push( articles[i] );
    }
}
```

Even a class as simple as this one has problems. Consider what happens when a user leverages inheritance and uses the `ArrayList`'s `clear()` method to pop everything off the stack:

```
Stack a_stack = new Stack();
a_stack.push("1");
a_stack.push("2");
a_stack.clear();
```

The code successfully compiles, but since the base class doesn't know anything about the stack pointer, the `Stack` object is now in an undefined state. The next call to `push()` puts the new item at index 2 (the `stack_pointer`'s current value), so the stack effectively has three elements on it—the bottom two are garbage. (Java's `Stack` class has exactly this problem; don't use it.)

One solution to the undesirable method-inheritance problem is for `Stack` to override all `ArrayList` methods that can modify the array's state, so the overrides either manipulate the stack pointer correctly or throw an exception. (The `removeRange()` method is a good candidate for throwing an exception.)

This approach has two disadvantages. First, if you override everything, the base class should really be an interface, not a class. There's no point in implementation inheritance if you don't use any of the inherited methods. Second, and more importantly, you don't want a stack to support all `ArrayList` methods. That pesky `removeRange()` method isn't useful, for example. The only reasonable way to implement a useless method is to have it throw an exception, since it should never be called. This approach effectively moves what would be a compile-time error into runtime. Not good. If the method simply isn't declared, the compiler kicks out a method-not-found error. If the method's there but throws an exception, you won't find out about the call until the program actually runs.

A better solution to the base-class issue is encapsulating the data structure instead of using inheritance. Here's a new-and-improved version of `Stack`:

```
class Stack
{   private int stack_pointer = 0;
    private ArrayList the_data = new ArrayList();
    public void push( Object article )
    {   the_data.add( stack_pointer++, article );
    }
    public Object pop()
    {   return the_data.remove( --stack_pointer );
    }
    public void push_many( Object[] articles )
    {   for( int i = 0; i < o.length; ++i )
            push( articles[i] );
    }
```

```
}
```

So far so good, but consider the fragile base-class issue. Let's say you want to create a variant on `Stack` that tracks the maximum stack size over a certain time period. One possible implementation might look like this:

```
class Monitorable_stack extends Stack
{
    private int high_water_mark = 0;
    private int current_size;
    public void push( Object article )
    {   if( ++current_size > high_water_mark )
            high_water_mark = current_size;
        super.push(article);
    }

    public Object pop()
    {   --current_size;
        return super.pop();
    }
    public int maximum_size_so_far()
    {   return high_water_mark;
    }
}
```

This new class works well, at least for a while. Unfortunately, the code exploits the fact that `push_many()` does its work by calling `push()`. At first, this detail doesn't seem like a bad choice. It simplifies the code, and you get the derived class version of `push()`, even when the `Monitorable_stack` is accessed through a `Stack` reference, so the `high_water_mark` updates correctly.

One fine day, someone might run a profiler and notice the `Stack` isn't as fast as it could be and is heavily used. You can rewrite the `Stack` so it doesn't use an `ArrayList` and consequently improve the `Stack`'s performance. Here's the new lean-and-mean version:

```
class Stack
{   private int stack_pointer = -1;
    private Object[] stack = new Object[1000];
    public void push( Object article )
    {   assert stack_pointer < stack.length;
        stack[ ++stack_pointer ] = article;
    }
    public Object pop()
    {   assert stack_pointer >= 0;
        return stack[ stack_pointer-- ];
    }
    public void push_many( Object[] articles )
```

```
    {   assert (stack_pointer + articles.length) < stack.length;
        System.arraycopy(articles, 0, stack, stack_pointer+1,
                                                articles.length);
        stack_pointer += articles.length;
    }
}
```

Notice that `push_many()` no longer calls `push()` multiple times—it does a block transfer. The new version of `Stack` works fine; in fact, it's *better* than the previous version. Unfortunately, the `Monitorable_stack` derived class *doesn't* work any more, since it won't correctly track stack usage if `push_many()` is called (the derived-class version of `push()` is no longer called by the inherited `push_many()` method, so `push_many()` no longer updates the `high_water_mark`). `Stack` is a fragile base class. As it turns out, it's virtually impossible to eliminate these types of problems simply by being careful.

Note you don't have this problem if you use interface inheritance, since there's no inherited functionality to go bad on you. If `Stack` is an interface, implemented by both a `Simple_stack` and a `Monitorable_stack`, then the code is much more robust.

I provide an interface-based solution in Listing 0.1. This solution has the same flexibility as the implementation-inheritance solution: you can write your code in terms of the `Stack` abstraction without worrying about what kind of concrete stack you actually manipulate. Since the two implementations must provide versions of everything in the public interface, it's much more difficult to get things wrong. I still have the benefit of writing the equivalent of base-class code only once, because I use encapsulation rather than derivation. On the down side, I have to access the default implementation through a trivial accessor method in the encapsulating class. (`Monitorable_Stack.push(...)` (on line 41) has to call the equivalent method in `Simple_stack`, for example.) Programmers grumble about writing all these one-liners, but writing an extra line of code is a trivial price to pay for eliminating a significant potential bug.

Listing 0.1. Eliminate fragile base classes using interfaces

```
 1| import java.util.*;
 2|
 3| interface Stack
 4| {
 5|     void push( Object o );
 6|     Object pop();
 7|     void push_many( Object[] source );
 8| }
 9|
10| class Simple_stack implements Stack
11| {   private int stack_pointer = -1;
12|     private Object[] stack = new Object[1000];
13|
14|     public void push( Object o )
```

```
15|     {   assert stack_pointer < stack.length;
16|
17|         stack[ ++stack_pointer ] = o;
18|     }
19|
20|     public Object pop()
21|     {   assert stack_pointer >= 0;
22|
23|         return stack[ stack_pointer-- ];
24|     }
25|
26|     public void push_many( Object[] source )
27|     {   assert (stack_pointer + source.length) < stack.length;
28|
29|         System.arraycopy(source,0,stack,stack_pointer+1,source.length);
30|         stack_pointer += source.length;
31|     }
32| }
33|
34|
35| class Monitorable_Stack implements Stack
36| {
37|     private int high_water_mark = 0;
38|     private int current_size;
39|     Simple_stack stack = new Simple_stack();
40|
41|     public void push( Object o )
42|     {   if( ++current_size > high_water_mark )
43|             high_water_mark = current_size;
44|         stack.push(o);
45|     }
46|
47|     public Object pop()
48|     {   --current_size;
49|         return stack.pop();
50|     }
51|
52|     public void push_many( Object[] source )
53|     {
54|         if( current_size + source.length > high_water_mark )
55|             high_water_mark = current_size + source.length;
56|
57|         stack.push_many( source );
58|     }
59|
60|     public int maximum_size()
61|     {   return high_water_mark;
62|     }
63| }
64|
```

Frameworks

A discussion of fragile base classes would be incomplete without a mention of framework-based programming. Frameworks such as Microsoft Foundation Classes (MFC) have become a popular way of building class libraries. Though MFC itself is blessedly fading away, MFC's structure has been ingrained in countless Microsoft shops where programmers assumed that the Microsoft way was the best way.

A framework-based system typically starts with a library of half-baked classes that don't do everything they need to do, but rather rely on a derived class to provide missing functionality. A good example in Java is the `Component`'s `paint()` method, which is effectively a place holder; a derived class must provide the real version.

You can get away with this sort of thing in moderation, but an entire class framework that depends on derivation-based customization is brittle in the extreme. The base classes are too fragile. When I programmed in MFC, I had to rewrite all my applications every time Microsoft released a new version. The code would often compile, but then not work because some base-class method changed.

All Java packages work quite well out of the box. You don't need to extend anything to make them function. This works-out-of-the-box structure is better than a derivation-based framework. It's easier to maintain and use, and doesn't put your code at risk if a Sun Microsystems-supplied class changes its implementation.

Summing up fragile base classes

In general, it's best to avoid concrete base classes and `extends` relationships in favor of interfaces and `implements` relationships. My rule of thumb is that 80 percent of my code at minimum should be written entirely in terms of interfaces. I never use references to a `HashMap`, for example; I use references to the `Map` interface. (I use the word "interface" loosely here. An `InputStream` is effectively an interface when you look at how it's used, even though it's implemented as an abstract class in Java.)

The more abstraction you add, the greater the flexibility. In today's business environment, where requirements regularly change as the program develops, this flexibility is essential. Moreover, most of the Agile development methodologies (such as Crystal and extreme programming) simply won't work unless the code is written in the abstract.

If you examine the Gang of Four patterns closely, you'll see that many of them provide ways to eliminate implementation inheritance in favor of interface inheritance, and that's a common characteristic of most patterns. The significant fact is the one we started with: patterns are discovered, not invented. Patterns emerge when you look at well-written, easily maintainable working code. It's telling that so much of this well-written, easily maintainable code avoids implementation inheritance at all cost.

This article is adapted from my forthcoming book, tentatively titled Holub on Patterns: Learning Design Patterns by Looking at Code, *to be published by Apress (www.apress.com) this fall.*

About the author

Allen Holub has worked in the computer industry since 1979. He currently works as a consultant, helping companies not squander money on software by providing advice to executives, training, and design-and-programming services. He's authored eight books, including Taming Java Threads (Apress, 2000) and Compiler Design in C (Pearson Higher Education, 1990), and teaches regularly for the University of California Berkeley Extension. Find more information on his Website (http://www.holub.com).