Polymorphic Hierarchy

Subimplementor methods are the key

Bobby Woolf

I've noticed that at least half the methods I write are not very original. Sure, lots of the methods I write are real methods that do real work and have real names that are unique. Yet in the process of that real programming. I also write a ton of methods that don't require much thought. I implement lots of getter and setter methods, and initialization and instance creation methods, and I also write a fair number of methods that subimplement methods already defined in a superclass. I'm finding that it is these subimplementor methods that are the key to polymorphism. Used aggressively and consistently, polymorphic methods lead to polymorphic classes and ultimately polymorphic hierarchies.

I'll discuss what a polymorphic hierarchy is as well as a pattern I call "Template Class". The top class in a polymorphic hierarchy is a Template Class.

REUSING METHOD DESCRIPTIONS

Let's use printOn: as an example. VisualWorks implements printString in Object via printOn:. Since Object>>printOn: is generic, I frequently subimplement it int my classes to tell me not just what class this instance come from, but to also provide some clue as to which instance this is. Being a good corporate citizen, I comment my new implementor of printOn: with a description to give the next programmer some clue as to what this method does.

The description in Object>>printOn: says, "Append to the argument aStream a sequence of characters that describes the receiver." I couldn't have said it better myself. In fact, why should I try? What do you think this method does? Having seen a hundred other implementors of printOn: throughout the system, all of which do the same thing, what do you think this one does? My philosophy is to duplicate as little effort as possible, so why should I comment a method that's already been commented elsewhere? Insisted, I just describe my method by saying, "See superimplementor." It's my way to let the method say, "Look I do the same thing my superimplementor does. What else would I do? If you don't know what that is, see the description in the superimplementor."

Another clue that my implementor of printOn: does the same thing as Object's is that both implementors appear in the same method protocol, printing. The printing protocol in Object says, "These are all of the printing-type things the receiver knows how to do." By putting my implementor in the printing protocol, I'm saying that my method is also used for printing. That's a good thing, because I'm subimplementing a method that is already used for printing. Not only should a subimplementor do the same thing as it's superimplementor, it should also appear in the same protocol.

ONE DEFINING IMPLEMENTOR

When I've written all the implementors of a message, there's only one description, and that's in the method in the superclass that defines the hierarchy. This implementor is usually pretty lame: it just returns self, or the subclassResponsibility error, or some default implementation that won't cause trouble when subclasses inherit it. But even though the implementation is lame, the method still defines what this message does for the whole hierarchy. Thus, any subimplementors had better do the same thing: their implementations are different, but their purposes are the same. Thus, their descriptions are, "See superimplementor."

The same principle applies to the Smalltalk habit of having one method simply call another method, which has also the same name except for an extra parameter. For example, Object>>changed sends changed: then Object>>changed:; sends changed:with:. There is no need to describe the purpose of all three of these methods. Once you know what changed:with: does, you know that changed and changed: work in the same way, except that they use defaults for some of the parameters. So my description for Object>>changed would be "See changed:."; "See changed:with:." describes Object>>changed:. If I subimplemented any of these methods, the description would be "See superimplementor."

ANATOMY OF A METHOD DESCRIPTION

So what kinds of immodest boasts and secret confessions end up in a method description anyway?

First of all, I try to avoid rephrasing the method name. Unfortunately, I've written many methods like productCode, whose description says “Return the code for the receiver's product.” Oh, really? These sorts of methods are usually just returning an instance variable’s value, so when I can't think of anything else to say,

Used aggressively and consistently, polymorphic methods lead to polymorphic classes and ultimately polymorphic hierarchies.

I describe them with, “Getter.” Ditto for setter methods. Given that you should already know what the instance variable does, there's really not much to say about getter and setter methods.

Second, I try to use method description to describe the entire method. I generally don't like to comment individual lines of code. It's tempting to do so when I write a line or two of code that is so bizarre that another programmer has no chance to understand what he's looking at. Unfortunately, I'm not going to be able to cover this blunder by including an equally convoluted comment. Instead, I hide the bizarre code in a new method with a descriptive name. The comment I would have included to explain the code becomes the method's description.

PURPOSE AND IMPLEMENTATION DETAILS

Third, I split my method description into two parts: purpose and implementation details. *Purpose* explains what the method does. I phrase it as, “If you send this message to this object, here's what'll happen. This method will...” Examples might be, “Sort the receiver's elements.” “Read the next item and return it.” and “Answer whether the receiver contains errors.” *Implementation details* is optional. This is my litany of excuses for why the code is so strange. Luckily, I write reasonably good code most of the time so only a few of my methods need these explanations.

A method's purpose is reusable. All of the implementors of a message within a hierarchy had better have the same purpose. This, I only document the message's purpose in the implementor at the top of hierarchy. Other implementors document their purpose with “See subimplementor.”

A method's implementation details are not reusable. Two implementors of a message should not have the same implementation details. If they do, they're duplicating code.

Thus, you can easily split your own method descriptions into purpose and implementation details. To do so, separate the commentary that explains what the method does from that written explains how. The *what* commentary is the method's purpose; it should be the same for every implementor in the hierarchy. The *how* commentary describes the implementation details, so it should be different (or negligible) for every implementor. In every implementor, except the topmost one, change the purpose comment to “See subimplementor” because the purpose is always the same. Add implementation details comment if necessary. As I read other programmers' method descriptions (such as the vendor code), I mentally perform this separation to understand the method better.

DESCRIPTION REUSE FOR POLYMORPHISM

One of the things that took me a while to learn about Smalltalk is what subclasses are for. To me, each class was its own packet of implementation detail. Anything two classes had in common was coincidence. The way I chose a superclass was to select whichever class would let my class inherit the most stuff “for free.” I didn't create hierarchies, just groups of classes.

Now, I can't seem to think about a class without also needing to understand its superclasses, I don't use class browsers anymore: I use hierarchy browsers. Super classes are like a linear set of mixins, and I want to see what's being mixed into this class. I find that trying to understand a class without knowing its superclasses is like hearing a private joke without knowing its context.

When I create a subclass, I don't just think about how the class should work. I also think about how it should differ from its superclass. The superclass should already do pretty much everything the class needs to know how to do (although the subclass may add additional behavior). The problem is that the superclass

knows what to do, but not how to do it. The subclass implements the how.

Consider the Collection hierarchy. A Collection can accept requests to add and remove elements, iterate over them, and answer how many elements it has, but it does not know how to fulfill many requests. The how depends on the implementation; is the collection a list, a tree, a hash table, or what? Subclasses of Collection implement the details of how. Set (a hash table) implements add:, remove:, do:, and size one way. OrderedCollection (a list) implements them another way, Collection defines what a collection can do while the subclasses define how that gets done.

How do I make sure that the subclass does the same thing the superclass does? Each method that the subclass subimplements (extends or overrides from the superclass) has to do the same thing that the method in the superclass does. The subimplementor doesn't do it the same way, of course, but it should produce the same result. In other words, the subimplementor should have the same purpose as the superimplementor.

PURPOSE IS POLYMORPHIC

When all implementors in a hierarchy have the same purpose they're polymorphic. When all of the methods that subclasses subimplement are polymorphic with their inherited versions, the hierarchy is polymorphic. Thus means that a collaborating object can use one instance of the hierarchy just as easily as another instance. Because all of the instances behave the same, one works just as well as another.

For example, consider an object (Employee) that should maintain a list of things to do (toDoList). How should the object sort the list, by priority or first-come-first-serve? Who knows? But you know it will be some sort of Collection, probably either an OrderedCollection or a SortedCollection. So, however you eventually decide to implement the toDoList, you already know what its behavior is. You can add a task to be done (add:), remove a task that's been done (remove:), ask how many tasks there are left to do (size), and ask for the next one to do (first). Later, you can work out the more domain specific details of how the list should be ordered, and use an OrderedCollection or SortedCollection as appropriate.

DEFINING POLYMORPHISM

This definition of polymorphism is more extensive than the ones you usually hear. The sound bite I learned was that two methods are polymorphic if their names are the same. This is clearly not always true. Think about the messages value and value:. They have lots of implementors, but are those implementors polymorphic? If they are polymorphic, what do those messages do? That depends on the receiver. In a ValueModel, value and value: are accessor (getter and setter) methods. In a block (a BlockClosure), value and value: are evaluation methods, clearly not getters and setters. These implementors of value and value: have the same name, but they are not polymorphic.

For two methods to be polymorphic, they not only need to have the same name, they must behave the same way as well. This means that they not only accept the same number of parameters, but each parameter is of the same type in both methods. Both methods must produce the same side effects, such as changing the state of the receiver in the same way. And, both methods must return the same type of result. Only then are the two methods truly polymorphic.

As described earlier, I contend that two classes can be polymorphic. Two polymorphic classes understand the same messages, and their implementors of those messages are polymorphic. Because the two classes share the same interface and behave the same way, they are polymorphic. In practice, two classes often do not share the same complete interface, but they do share the same core interface and it is polymorphic. A *core interface* is an interface that several classes share, so that they can be used interchangeably. As long as a collaborator only uses that core interface, it can use an instance of one class as easily as any other class that has the core interface.

MAKING A HIERARCHY POLYMORPHIC

This brings me back to my “See subimplementor” method comments. As I got used to commenting my methods this way, I got used to thinking of my methods polymorphically, which made me more careful to implement them polymorphic. As the methods in my hierarchies because more polymorphic, the hierarchies became more polymorphic. As my hierarchies became more polymorphic, they became more

flexible, reusable, extensible, easier to learn, and all of that other good OO stuff.

The problem comes when there is no superimplementor to see. Sometimes I'm implementing a method and I remember that I've already implemented another method with the same name. They're doing the same thing, so they have the same purpose. If one of them is a subimplementor of the other, I describe the subimplementor with "See subimplementor." But I can't do this when they are in peer classes and there is no superimplementor. When this happens, the code is telling me, "You're missing a superimplementor." I don't want to duplicate any effort I can avoid, which includes defining what this message is supposed to do in this hierarchy. So I introduce a superimplementor, document the purpose there, and give it a default implementation. Now I can document the subimplementors with "See superimplementor."

Of course, another problem I sometimes run into is that there's no superclass to put the superimplementor in. Two classes with two supposedly polymorphic implementations of the same message are peer classes that have no relationship to each other in the hierarchy. Their first common superclass is something generic like ApplicationModel or Object. I'm definitely not going to add

Most of the hierarchies we know and love are polymorphic hierarchies. This is no coincidence.

this domain-specific message to such a general class, as though all subclasses should support this message. So what can I do? If these two implementors are really polymorphic, then their classes are probably going to need to be polymorphic as well. If so, the most maintainable way to make two classes polymorphic is to put them in the same hierarchy and make the whole hierarchy polymorphic. Since this hierarchy does not exist, I need to create a new abstract class that describes the polymorphic behavior of this hierarchy, then subclass my two concrete classes off of it. The hierarchy now exist, so now I can add the superimplementor to the superclass. The purpose of the message goes in the superimplementor, and the superimplementors just say "See superimplementor."

THE TEMPLATE CLASS PATTERN

The abstract class I introduced to make the hierarchy polymorphic is what I call a "Template Class." Template Class is a pattern that creates polymorphic hierarchies. It is similar to the Template Method pattern.[1] Whereas a Template Method defines the interface for a method while deferring the details to subclasses, a Template Class defines the interface for a class –a new type– while deferring the implementation details to subclasses. This leads to a whole hierarchy of classes that are polymorphic. A Template Class is typically full of Template Methods.

THE VALUEMODEL HIERARCHY

The ValueModel hierarchy in VisualWorks is a good example of a polymorphic hierarchy. The class ValueModel defines the hierarchy by saving that all instances will understand messages such as value, value:, and onChangeSend:to:. All subclasses implement these messages according to how each class works. A class will inherit some of the implementations if they already work appropriately.

Because all ValueModel subclasses support this core interface, a collaborator can use a ValueModel without having to worry about what subclass the instance came from. Whether the instance is a ValueHolder, an AspectAdaptator, or a TypeConverter, it supports the core value interface. This is the benefit of polymorphism at the class level. Because the hierarchy is polymorphic, you can use any class in the hierarchy as easily as any other and the collaborating ones (such as the value-based widgets) will never know the difference.

Most of the hierarchies we know and love –Collection, Magnitude, ArithmeticValue/Number, Boolean, String, etc– are polymorphic hierarchies. This is no coincidence. The fact that we know them so well and employ them so often is in large part because their polymorphism makes them so easy to use.

CONCLUSION

To summarize:

- Many methods have not only the same name, they do the same thing. This is where polymorphism comes from.
- For two methods to be polymorphic, they need to have not only the same name, but also the same parameter

types, the same side effects, and the same return type.

- Two methods that are polymorphic should appear in the same method protocol.
- A method description documents two things: the purpose and the implementation details.
- For two methods to be polymorphic, they need to have the same purpose. They should not have the same implementation details.
- For two classes to be polymorphic, they need to share the same core interface polymorphic messages.
- A collaborator can use the two classes interchangeably if it only uses messages in their core interface.
- For a hierarchy to be polymorphic, all of its classes must share the same polymorphic core interface.
- The polymorphic hierarchy and its core interface are defined by an abstract class. I call an abstract class that fulfills this role a Template Class.
- A polymorphic hierarchy encapsulates code that is highly reusable, flexible, extensible, and just plain good OO.

To learn more about implementing your own polymorphic hierarchies, I suggest reading the paper "Reusability Through Self-Encapsulation" by Ken Auer.[2] It is a pattern language that describes how to develop a class hierarchy that achieves reuse via inheritance, while maintaining each class's encapsulation. Although polymorphism is not an explicit goal of the pattern language, hierarchies developed this way tend to be polymorphic ones.

REFERENCES

1. Gamma. E., Helm, R., Johnson, R., and Vlissides.) DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1995. http://www.aw.com/cp/Gamma.html
2. Coplien, James, and Schmidt D., (Eds.), PATTERN LANGUAGES OF PROGRAM DESIGN, Addison-Wesley, Reading, MA, 1995. http://hegschool.aw.com/cseng/authors/coplien/patternlang/patternlang.htm

---

Bobby Woolf is a senior member of technical staff at Knowledge Systems Corp. in Cary, North Carolina. He mentors clients in the use of VisualWorks, ENVY, and design patterns. He welcomes your comments at woolf@acm.org or at http://www.ksccary.com