Variations on a Polymorphic Theme: An Etude for Computer Programming

Joseph Bergin
Pace University
Computer Science
New York, NY USA
1 (212) 346 1499
berginf@pace.edu

ABSTRACT
This paper describes a technique by which instructors and programmers can increase their skill with Object-Oriented programming by practicing writing polymorphic programs in an intensive way.

Categories and Subject Descriptors
D.1.5 [**Programming Techniques**]: Object-oriented Programming

D.3.3 [**Programming Languages**]: Language Constructs and Features – *polymorphism, Control structures, Patterns*

General Terms
Algorithms, Design, Languages

Keywords
Polymorphism, Object-Oriented Programming, Learning

1. INTRODUCTION
Suppose you are a programmer. Suppose you learned a language like C or Pascal and are good at it. Suppose that you now want to become a good Java programmer using the language to its full effect. This implies moving your programming style from procedural programming to object-oriented (OO) programming. How can you do it if object thinking is very different from procedural thinking? Suppose you are a teacher. Suppose you want to teach object-oriented programming to your students, but you are a much better procedural programmer than an OO programmer. After all, when faced with a programming problem, it isn't difficult for you to come up with a (procedural) solution. It probably involves *if* statements or the equivalent. Your lectures and answers to student questions probably reflect this understanding. Your facility with selection constructs gets in the way of OO programming since you naturally see a procedural solution first. How can you make the transition to be a more effective programmer or educator in the OO paradigm?

Many people moving to Java think that if you program with classes and inheritance you are an object-oriented programmer. In fact, it takes more. Object-orientation is not about classes or even inheritance. It is about polymorphic run-time message dispatch. All programs need to handle situations in which more than one action is possible at a given program point. Procedural programmers handle this situation with *if* statements and similar selection mechanisms. Object programmers handle it, instead, with polymorphism. This requires thinking differently about programming, but it can lead to better structured more maintainable programs.

My intention here is not to argue the above statements, which are not controversial among skilled OO programmers, but to try to show you a way to develop the skill that will give you a basis for judging them for yourself. An implication of the above, however, is that OO programs have fewer *if* statements than procedural programs. Just as functional programs solve problems without iteration, using recursion, OO programs solve problems without selection, using polymorphism.

Switching dimensions for a moment, in music an *etude* is a piece of music that in itself may have no intrinsic musical value, but is valuable to a musician for the exercise in practicing some particular technical skill. A student flutist, for example will be given an etude with many sixteenth notes and complex combinations, just to give the student valuable finger and lip exercise. Professional musicians use these as well and they are not just for novices. A clarinetist who wants to play the flute will find etudes especially useful, since their hand and lip training on clarinet is all wrong for the flute.

Many of the exercises we set to students, in fact, are etudes. The program has no intrinsic value but will teach the student some important technical skill. Even the student project in a first compiler course usually fits this mold. So I propose an etude for anyone who is a good procedural programmer who wants to become a good OO programmer.

2. THE POLYMORPHISM ETUDE
Find some old program that you have around and that you are proud of. I'll suppose that it is several hundred lines; say about 10 pages. Strictly as an etude, rewrite that program with NO if/switch statements: no selection at all. Solve all of the problems your ifs solve with polymorphism.

Note also that an etude may need to be practiced more than once to get the required skill, so I don't suggest that doing this once is sufficient, but it will be instructive to do it

even once. When you can do it consistently, you have developed a new skill that you can use as appropriate in your regular programming. I'm not suggesting that good OO programs never have if statements, in fact, but until you can write one without if statements you are not sufficiently skilled with an important technique.

Let me put it another way. In Plato's *Allegory of the Cave*, people were constrained throughout their lives to look only straight ahead at a wall, upon which shadows were cast by people walking by behind them carrying various objects. They had no knowledge of the reality behind them, only the shadows before. Being intelligent, they made suppositions about the reality that they experienced, but had no way to posit that there were three dimensional beings casting the shadows, nor the mechanism by which the shadows were cast, or even that their reality was just shadows.

In the same vein, it is hard to judge whether OO thinking is really different from procedural thinking until you are skilled in both. Only then can you really judge. If you think it can't be that different, then you have an exciting experience still ahead of you. The above etude is an attempt to give you a wonderful "aha" experience.

3. IMPLEMENTATION

If you take this challenge it will be helpful to have a bit of guidance as to how to proceed. You might want to try it first before continuing, though. The hints below are not a substitute for practicing this etude. It is practice that will teach you the skill. When you no longer struggle with this you can think of yourself as an expert.

The first step is somewhat simple and it involves the language in which you discuss the problem. We normally think of *if* and *when* as interchangeable in the following: "If the account is a user account, then add interest, else add interest and deduct taxes," versus "When the account is a user account add interest, and when it is a business account add interest and deduct taxes." While the semantics is the same (assuming only user and business accounts exist, of course) the situation is subtly different if this is a discussion leading to a program to model the action.

In the first case you are led immediately and directly to model it with an *if* statement and your language seems to imply that you have already made this *design* decision. The second form is much more interesting, however, as it is less tied to any program syntax, so it leaves you some mental wiggle room in coming up with a solution. You could map the "when" to a Java *if* statement, or you could alternately think that when the object is a user account it should itself add interest and when the object is a business account it should itself add interest and deduct taxes. This is in fact a more OO solution.

So the first, simplest, recommendation is replace *if* in your problem discussions with *when*. Again, I'm not suggesting that you always think this way. It is just for the purposes of the etude. You may, however, find it useful generally, and if you are a teacher, it may help to state problems to your students in this more "solution neutral" way.

The rest of the advice on how to succeed with the etude is a bit more complicated. One of the implications, however, is that you will be writing quite a few classes, though they will be very simple, and you will have lots of objects. I've been in discussion forums in which some of the respondents (who have probably not tried the etude) press others to write few classes, create few objects, and write long (read complex) methods. While it is possible to program this way, it isn't the OO way. There is nothing wrong with creating a lot of objects. In fact, in OO programming we celebrate objects.

The key to programming with polymorphism alone is to capture each possible action when there are several in a different object. Sometimes these different objects are from different classes, but many times we can just parameterize the construction of the objects to capture the difference. If we do need different classes then they can, perhaps, derive from a common superclass that defines the protocol. In Java, however, we prefer interfaces to inheritance for this situation as it leaves us more flexibility. In situations when you can factor common behavior into the superclass the inheritance solution may be preferred, but think of the interface first.

While it is somewhat off the topic here, it is also a good idea to use the following heuristic. When you implement an interface or extend a class (perhaps abstract) the only public methods in the new class should be those defined in the interface or inherited from the superclass. Private helper methods for the re-implementations of the public methods are fine, but don't extend the public protocol of the superclass unless you can think of no other way to proceed. If you are able to do this successfully you will never need to cast a value and that will save you both *if* statements and errors.

And by the way, I'm not suggesting you replace all *if* statements with try blocks where the *else* is just the body of a *catch* clause. None of that. We are after the polymorphism etude here, not the exception handling etude. You are, however, allowed to use library code even though you are pretty sure that it was implemented with selection.

The next thing you need to know is that your program must manipulate (non-final) objects, not primitives. The *int* and *char* types in Java are not objects and therefore have no opportunity to behave polymorphically. The *Integer* and *Character* classes are no better, since, being final, they cannot be extended to provide new behavior. So your program must be built out of objects defined in non-final classes

Now that you have the behaviors factored into different objects, the only thing needed is to bring the right object to bear at the right time. Passing different objects to a method can result in polymorphic action. The trick is to learn how to pass the right object. If you need an *if* statement to choose it, then you are still not being polymorphic enough.

This takes some creativity in general. An extremely simple case is that when you write a Java GUI program write distinct listeners for different buttons. Another simple case is discussed in the next section. It depends on a common situation in which the flow of state is well determined in the program. In general, you may need to map some values into the objects that have the behaviors. Hash maps are good for this. A common situation is that you need to treat negative integer values differently from non-negative values. You can use *max* and *min* library routines to map the negatives all to 0 and the non-negatives all to 1, and then use these values to pick out an object appropriate for negatives (printing error messages, perhaps) and a different object for the other values. An array, Vector, or HashMap can hold the objects, for example. Section 7 shows some details of how to use this idea.

4. A SIMPLE EXAMPLE

Let me next present a simple example that will get you started in thinking appropriately to be successful with this exercise. A while ago, Ward Cunningham, described to me his experience in helping his son, a high-school senior, learn to program. They both wanted something interesting to do. Ward suggested that they might take their own home and program it as if it were a dungeon game. A Person object could move through various Place objects (rooms, hallways, etc.) and find and use various Thing objects. Things can be picked up and carried. One of the more creative Things was a Transporter object. The idea is that the first time you *activate* a Transporter it enters the *charged* state in which it remembers the room in which it was activated (not necessarily the room in which it was found). The next and subsequent times that you activate it, you are transported immediately to the room in which it was charged. Said another way, "*When* you activate it for the first time..." and "*When* you activate it subsequently..."

You can solve this simply with a flag variable and an *if* statement, of course, and this solution probably naturally occurred to you. An OO programmer, however, would think like the following. We have two different actions and either is possible when the transporter is activated. We will therefore choose to put these two actions into different objects, but not into two different transporter objects as we only want one of these. Instead, the *strategy* that a transporter will use when activated will be abstracted into an object, and we will have two of these (or more, if the problem evolves into something more sophisticated.) The transporter will delegate its *activate* action to one of these strategy objects.

Note that these strategy objects follow a well-known Strategy design pattern that you can find in [3]. And note further that we have just implied that delegation is a really big idea in this etude. An object can and should delegate some of its actions to other objects. When you need different behavior, delegate to a different object. The object that does the delegation will seem to change its behavior. In effect, the transporter object is built by composition (holding a strategy object at any given point in time) rather than inheritance.

So a *TransportStrategy* interface can be defined by

```
interface TransportStrategy {
   public void perform(Transporter transporter);
}
```

A Transporter object will hold a reference to one of these in a field as its *currentStrategy*, and, when asked to activate, it will just fire a *perform* message to its *currentStrategy*.

We implement the interface twice to create *InitialStrategy* and *ThereafterStrategy*, where the *perform* of the first has the transporter remember the room it is in and also set the *currentStrategy* of the transporter to be a new *ThereafterStrategy* (replacing itself). After the transporter is first activated (delegating to the initial strategy) it automatically holds a *ThereafterStrategy* in place of the original, so when asked to activate again it will now delegate to one that will transport the owner (a Person object) back to the remembered room.

The key to this is that we know the point in the program at which to replace one strategy with another and therefore don't need to execute any test to see which state we are in or which should be next. The strategy object itself represents the state: it is a "flag with behavior". So we don't *test* the "flag," we just ask it to execute the behavior it knows. The key is that each value of the flag is an object that has distinct behavior.

To summarize, first map the different behaviors into different "strategy" objects and then find a mechanism for associating the object that needs the behavior with the appropriate strategy. Many programs can be built so that this is easy to do, since there may be well-defined points, though more complex than the above, where such transitions occur. Any program for which you can draw a state transition diagram fits this description, in fact. When something happens that causes you to change your state, program that "something" to also replace one strategy object with another that represents the next state.

When you think about why you normally put *if* statements in your programs, you will discover that many of them are simply to regain information that you had earlier in your program that you lost. An *if* statement captures knowledge about state, of course. Think of executing an *if* statement as "climbing information hill" as you have a higher level of information within the *if* statement than without. However, as soon as you leave the *if* statement you normally fall down the other side of the hill and lose the information again. The key to good OO practice is that when you do have information, you capture it in the form of objects that know how to properly manipulate it. These are then passed to the places where the information is needed.

Somewhat harder cases to handle are those in which an action depends on several conditions. In procedural programming this leads to complex conditions. In object programming you may need to delegate more than once to reach a state in which you find the appropriate action. You might also be able to hold the strategy objects in a hash map with keys representing the possible sets of conditions for which that strategy is appropriate. You can use this trick to build a Turing Machine with no if statements. Well, maybe you will need a test for null, which cannot be polymorphic. The details are left as an exercise. And of course, if you can build a Turing Machine this way, you can build anything. You can also build a lisp like (car, cdr, cons) linked list entirely without *if* statements. You will want to investigate the Null Object pattern to do so, however. [4]

Chapter 4 of *Karel J Robot* [2] also gives hints about what is possible to do with simple polymorphism, especially strategies, as well as more on some elementary design patterns that help.

5. CONCLUSIONS

Prefer *when* to *if*. Prefer objects to primitives. Prefer interfaces to inheritance. Prefer delegation and composition to complex hierarchies. Prefer many simple classes to a few complex ones. Prefer many simple methods to a few complex ones. Make the parts simple. Put complexity into the interactions. Have fun.

Once you can program polymorphically you will be a better judge of when selection and polymorphism are best applied. You will be a better programmer for having better skills.

6. ACKNOWLEDGMENTS

I'd like to thank Kent Beck, the creator of Extreme Programming (XP), for nudging my gray matter in a direction that made this possible. In a talk on XP, Kent described the twelve practices in their pure form as just etudes that need to be practiced by any developer and that will improve your skill when you do so. He sometimes describes XP as turning all the knobs up to 10. It is in this sense that XP is extreme. The etude becomes the actual practice.

7. APPENDIX – TEXTBOOK EXAMPLE

Here is the crux of the solution of a simple example. It is taken from Barnes and Kölling's book [1]. The problem is to build a simple ticket machine that takes money and prints tickets. One wrinkle is that *when* negative amounts of money are entered we need to print error messages. I won't show a complete solution, but will show the key steps that transform this into a program using only polymorphism as a control structure (beyond sequence and message passing, of course). I will develop it as a programmer might.

One key operation in which the negative/non-negative problem arises is in the insertMoney method. For non-negative amounts we increment a balance. For negative amounts we print an error message. As mentioned above there are two steps: (1) divide the behaviors into different classes, and (2) map the problem values into the objects so defined. We handle these in order.

Let us define an interface that specifies the method and that will be implemented twice for the different behaviors. The interface and its implementations will be inner classes within the TicketMachine class.

```
private interface Amount {
   public void incrementBalance();
}
```

We next implement this twice in separate classes

```
private class NonNegative implements Amount
{
        public NonNegative(int value)
        {
             rememberedValue = value;
        }

        public void incrementBalance()
        {
             balance += rememberedValue;
        }
        int rememberedValue = 0;
   }
```

Here balance is a field of the containing class. The Negative class just prints an error message.

```
public class Negative implements Amount
   {
        public void incrementBalance()
        {
             System.out.println("Negative insert not
                  allowed");
        }
   }
```

So now we have our two different behaviors. To bring the right object to bear at the right time we need to map negative integers into a Negative object and non-negatives similarly. We will do this with a *factory*. Assume we have an AmountFactory class that knows how to do this mapping and return an appropriate Amount object. Also assume that a TicketMachine holds a reference, *factory*, to an object of this class. Then we can implement insertMoney quite easily in TicketMachine. We delegate the operation to an object provided by the factory

```
public void insertMoney( int amount)
{
     Amount discriminator =
               factory.getAmount(amount);
     discriminator.incrementBalance();
}
```

Finally we need to see how to build the factory that implements step 2 of our plan. This is where the actually mapping is done. Note that the max function will map a portion of the integers into a single value, as will min. We use both here. We ultimately map the negatives all to 0 and the non-negatives all to 1.

```
private  class AmountFactory
{
     public Amount getAmount(int amount)
     {
          int check = Math.max(-1, amount);
          check = Math.min(check, 0);
          check += 1; // 1 or 0;

          Amount negative = new Negative();
          Amount nonNegative =
               new NonNegative(amount);
          amounts.put(one, nonNegative);
          amounts.put(zero, negative);
          return amounts.get(new Integer(check));
     }

     private  HashMap amounts = new HashMap();
     private static final Integer zero = new Integer(0);
     private static final Integer one = new Integer(1);
}
```

First we use max and min to map the original amount into either 1 or 0. We then create new objects of the two above classes and put them into a hash map with appropriate keys. We then extract the one we want, based on the *check* value. The

inefficiency of this can be easily overcome. There is no need to create new Amount objects if we are willing to make them mutable. Then the hash map can be loaded at startup with two (mutable) values.

While the above seems like a lot of work to avoid an *if* statement, remember that this is just an etude. In fact, your etude may even result in overly complicated and ugly code, though it is worth the effort to think about how to improve it. The author does not advocate trying to use this with novice students unless the results are expected to be elegant code and a good design.

However, this does have benefits in practice, as what we have seen here can remove many *if* statements, not just the one. Suppose we now tackle another problem in which the distinction between negative and non-negative values is critical: printing tickets. The machine should only print a ticket *when* there is sufficient money. Continuing our development of the above, we find that we have all of the infrastructure in place to do this easily. We add a new method to the Amount interface:

```
public void printTicket();
```

Now the above factory can be used to do the discrimination in the printTicket method of the TicketMachine class and it will again delegate to the appropriate Amount object. In TicketMachine, all we need is:

```
public void printTicket()
{
    Amount discriminator =
        factory.getAmount(balance - costOfTicket);
    discriminator.printTicket();
}
```

We use the difference between the current balance and the cost of a ticket to get an Amount object from the factory. When this is non-negative the machine can issue a ticket. Otherwise it must complain.

We then implement the new interface method twice. Within NonNegative it looks like this:

```
public void printTicket()
{
    System.out.println("*************");
    System.out.println("* BlueJ Line");
    System.out.println("Admit one");
    System.out.println("*************");
    balance = balance - costOfTicket;
}
```

And within Negative it just prints an error message (omitted here).

In fact, we can handle all negative/non-negative situations in the program the same way. So we amortize the effort to build the factory/mapper over a set of usages. Moreover we bring all of the negative handling/error processing together in one class and the valid actions together in another.

Note that while a procedural programmer programming Java would probably write one class, we have an interface and four classes here. Adding a GUI adds six more (five listeners and a Frame extension). But all methods are short and straightforward. We create quite a number of objects, though that can be greatly reduced with a little thought. And code locality is very good, though very different from procedural code locality.

8. REFERENCES

[1] Barnes, D., Kölling, M., *Objects First with Java: A Practical Introduction Using BlueJ*, Pearson Education, 2002

[2] Bergin, J., Stehlik, M., Roberts, J., Pattis, R., *Karel J Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java*, Unpublished manuscript. Available on the web at: http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html

[3] Gamma, Helm, Johnson, and Vlissides, *Design Patterns*. Addison-Wesley, 1995.

[4] Bobby Woolf, Null Object, *Pattern Languages of Program Design 3*, Robert Martin, Dirk Riehle, Frank Buschmann (eds),