Teaching Polymorphism with Elementary Design Patterns

Joseph Bergin
Pace University
http://csis.pace.edu/~bergin
berginf@pace.edu

Abstract

Polymorphism is often treated as an advanced topic by educators. Many feel that if statements are in some sense more "fundamental" to computing. On the contrary, polymorphism is both fundamental to object programming and is an elementary topic that can be easily understood by students. Previous papers [1] have shown how role-play exercises can remind students that they already have a deep understanding of dynamic polymorphism. The question then becomes how do we find effective teaching techniques to present this topic when we move from the level of metaphor to that of programming. A few elementary patterns [2] can be used to teach this topic even before the student is introduced to ad-hoc selection with if statements. Teaching these patterns early has the added benefit that they are pervasive in the Java libraries, so understanding them eases the student's later work.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object Oriented Programming *–polymorphism, control structures.*

General Terms: Design, Languages.

Keywords: Teaching, Design Patterns.

1. Think polymorphically using elementary design patterns.

Consider the following simple polymorphism pattern:

Context: You are at a point in a program at which one of several things needs to be processed. These things are objects (rather than ints...). You may have different kinds of things and in any case different things need to be processed in different ways.

Problem/Forces: You want the program to be simple and extendable. You want the object autonomy to be maintained as usual. *You want a single point of change for each logical element as the program evolves.*

Therefore, organize your code so that all of the different object types share an interface, perhaps because they derive from a common ancestor. At the point of commonality define a method to perform the processing you need done, and implement this method as appropriate in each type. If you need different kinds of information (parameters) for each type, then you can delegate the processing to an auxiliary object called a Strategy. In many cases this additional object can be created when the original object is and can be held as part of its state. The strategy object can encapsulate all of the additional information.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

For example, imagine a library processing books. There are CheckableBooks and PermanentReserveBooks. When books are brought back to the desk after use, they need to be returned. In this case we probably already have these two classes extending a common class, Book. If the same processing is required for all books, we can put a doReturn method into the Book class. You can now say simply

```
    book. doReturn();
```

Even in the case in which one or more of the subclasses must process returned books differently, we can simply override this method as appropriate, provided the parameter structure (here no parameters) of the method can properly be the same. Even with different implementations of doReturn in the different classes, the invocation is identical.

We would like to maintain this simplicity in harder cases as well.

2. Harder Cases: Strategy

In some cases different things need to be done on return of a book, however. For example, a reserve book needs to be returned immediately to a certain shelf, and a checked book needs to be processed through the patron's library records for late fees and outstanding fines. In this case we can delegate the process to a strategy. If we write methods with different parameters in the two classes it will be hard to use them polymorphically unless we provide an intermediary, a strategy.

For flexibility, strategies are best defined in interfaces. In this case, we can say

```
interface ReturnStrategy
{   public void returnBook();
}
```

Then individual strategies can be defined in classes that implement this, such as:

```
class CheckableReturn implements ReturnStrategy
{   public CheckableReturn(Patron p, CheckableBook b)
    {   this.patron = p;
        this.book = b;
    }
    public void returnBook()
    {     book.returnCheckable(this.patron);
    }

    Patron patron;
    CheckableBook book
}

class PermanentReserveReturn implements
ReturnStrategy
{     public PermanentReserveReturn(Shelf s,
PermanentReserveBook b)
    {     this.shelf = s;
          this.book = b;
    }
    ...
}
```

Then, when the book is given to a patron, one of these strategy objects is created and saved within the book object itself. In fact, the strategy could be created by an appropriate method of the individual class itself. So the CheckableBook class can have a method

```
    void createReturn(Patron p),
    {
          this.returnStrategy
                    = new CheckableReturn(p, this);
    }
```

Then later, when the book is returned, the book can be asked to

```
    book.doReturn();
```

This method is defined in the Book class as

```
    public void doReturn()
    {     this.returnStrategy.returnBook(); //Delegation
          this.returnStrategy = null;
    }
```

What do we gain? The original problem was that we were at a point in a program at which different things could occur. Now, the code for this is just a simple command like

```
    book.doReturn();
```

We do not here need to ask what kind of book it is. To see why this is important, read on.

The above is a bit different from the way many programmers would solve the problem. When the program is originally written an if statement distinguishing the cases would be simple enough, with different methods called depending on the test. However, this means that when the problem changes or is extended (common occurrences) this point in the program would need to be visited again for update. If a new type of book is introduced, we would need to replace the if with a switch or more complex if structure. If the processing of any book changes we might need to change the bodies of the if/else clauses if they were more than just simple messages. This sort of programming, with frequent changes to many places in a program is very error prone as has been shown in practice over many years.

With the strategy solution, however, we don't need to modify this point in the program for future changes. If we create a new kind of book we create a new kind of strategy as well if necessary. If the processing of some kind of book changes we update or extend the appropriate strategy class that handles returns for that kind of book. In a large program that will change, this is hugely beneficial as we have localized the points of change to the classes in which the change occurs and not those places where objects of the classes are used.

The key to thinking above was that in a situation in which many things are possible, one object delegates an action to another kind of object (the strategy). Delegation is the key. You arrange to create the new object at the early point at which you have the necessary knowledge to do so. One consequence is that you have more objects, but the individual objects are simpler. Simple objects are easy to understand and maintain. They may also be easy to extend through inheritance or other mechanisms.

We note that there are other ways to use the strategies than have them held within the book objects. For example, we could put the return strategies into a hashmap using the book as the value and the strategy as the key. Since the strategy contains all the information necessary to return a book, we could, when asked to return a book, find the corresponding strategy in the hashmap and then send it a returnBook() message. There are many other possibilities, of course.

3. New Strategies From Old: Decorators

As your program develops you will find yourself with several strategy classes and the need to define new ones. Sometimes this can be done with inheritance, but there is another way that often works. Suppose you have a strategy that does one thing and you need a strategy that does that thing, but also some independent thing as well. For example, suppose that we discover that some (but perhaps not all) of our book return actions also need to record how much time the book was out. It might even be the case that not all CheckableBooks need this extra action, but only some. One way to proceed is to write a Decorator for strategies that holds another strategy and performs its action in addition to another.

For example, let's build a timer Decorator for return strategies. For simplicity we will assume that the time the decorator is created is the desired check out time.

```
class TimerStrategy implements ReturnStrategy
{     public TimerStrategy(ReturnStrategy d)
      {     this.decorated = d;
      }

      public void returnBook()
      {     decorated.returnBook();
```

```
            timeRecorder.record(this.timeOut, new Date());
        }

        private ReturnStrategy decorated;
        private Date timeOut = new Date();
        // Time the strategy is created.
}
```

Then when we need such a thing, we can create it, wrapping any other kind of return strategy, for example

```
        returnStrategy = new TimerStrategy( new
CheckedStrategy (aPatron));
```

We can now return the book exactly as before with

```
        this.returnStrategy.returnBook();
```

and both actions will be done.

4. Cleaner Code: Null Object

Note that we have left the returnStrategy object of the Book class null after a book is returned. The value null is difficult to work with. If you dereference it by accident or poor design your program will crash (actually, throw an exception). We assumed that an invariant of the field was that it was null when the book was in, and not null otherwise. We can do a bit better, actually, and lessen the chance of using null incorrectly, by not using it at all. Here we will apply another design pattern called Null Object [3]. We let a special object take the place of null. Here we will develop an NullStrategy.

```
class NullStrategy implements ReturnStrategy
{       public void returnBook()
        {       // do nothing
        }
}
```

Now the returnStrategy field of Book could be initialized with one of these

```
ReturnStrategy returnStrategy = new NullStrategy();
```

And the doReturn method could be modified to:

```
        public void doReturn()
        {       this.returnStrategy.returnBook(); //Delagation
                this.returnStrategy = new NullStrategy();
        }
```

5. Keep it Simple: Singleton

Well, we can do even better than this. Note that the NullStrategy has no state and so always behaves in exactly the same way. If we have two or more of these they all behave exactly alike. Therefore it is really unnecessary to have more than one of these, so we can and should turn the NullStrategy (as is true of most null objects) into a Singleton. We do this by giving the class a private constructor but a public static instance.

```
class NullStrategy implements ReturnStrategy
{       public void returnBook()
        {       // do nothing
        }

        public static  NullStrategy value = new NullStrategy();

private NullStrategy()
        {
        }
}
```

It is now impossible to create more of these since the constructor is private. But we have exactly one of them available defined by the class itself, hence it is a Singleton.

Now wherever else we previously said new NullStrategy() we now say NullStrategy.value.

Note that Singleton objects are useful in other places. If there is no need to create more than one object of a certain class, or the logic of the program suggests it is incorrect to do so, then the class should be a singleton.

6. Keep it Safe: Immutable Object

Finally we note that all of the strategy classes we have shown, including the decorator, were Immutable. An object is immutable provided that its state cannot be changed after it is created. The constructor must supply all data necessary to create it (which it must do anyway) and there are no "mutator" methods that change the state later. Immutable objects are very useful in a program since they make the program easier to reason about. If objects don't change state, we know that the state we find them in at any point, including after a crash, is the state they were created with. Don't provide mutator methods for your classes unless they are really necessary and try to design your code overall so that they are mostly not necessary. Programs with few mutatorsare easier to maintain and debug.

It is hard to do much of this if you are processing things that are not objects or even for final objects. This is because you can't attach additional methods to them, though if you have access to the source code, you can modify the definition of a final class. There may be other reasons for not doing so, however.

7. References

[1] Andrianoff, Levine, Bergin, Role Playing: Easing the Paradigm Shift, OOPSLA 2002 Educator's Symposium, Seattle, WA

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

[3] Woolf; "Null Object"; Pattern Languages of Program Design 3, edited by Martin, Riehle, and Buschmann, Addison-Wesley, 1998.