Workbooking Your Way Through Design Patterns

James W. Cooper

When I picked up a copy of the *Design Patterns Java Workbook* by Steven John Metsker, I admit I was trepidated. You sort of think of tear-out sheets for the teacher, blanks to fill in and graphs to color. But this isn't what Metsker is up to at all, and he has written an excellent, lucid book on Design Patterns and some of the important OO culture that surround them.

So what makes this book as good as it is? It explains every little pattern idea clearly, and then asks you review questions in the form of "challenges." All of the code for solving these challenges is presented in an Appendix, but taking the time to think through these rather thought-provoking questions is where you will gain the most from this book. Some of the early challenges seem at first elementary, but in no time, he is posing questions that stimulate some really significant thought, even for experienced OO programmers.

Here's an early one:

Write down 3 differences between abstract classes and interfaces in Java.

That's pretty good. This is the sort of idea that every new OO student stumbles through. All these things start to sound suspiciously similar and you get mired in worry about what is really going on here. Let's take the KoolAid. Here are my answers, without peeking:

1. Abstract classes may have some executable methods and others left unimplemented. Interfaces contain no implementation code.
2. A new class can inherit from an abstract class, but not also from another class, while a new class can inherit from any class and still implement one or more interfaces.
3. Interfaces do not specify constructors.

Metsker's answers are shown in Table 1. Maybe you should try this before you peek. His list is pretty complete.

Table 1 – Differences Between Abstract Classes and Interfaces in Java

1. A class can implement any number of interfaces but subclass at most one abstract class.
2. An abstract class can have nonabstract methods. All methods of an interface are abstract.
3. An abstract class can have instance variables. An interface cannot.
4. An abstract class can define constructors. An interface cannot.
5. An abstract class can have any visibility: public protected, private or non (package). An interface's visibility must be public or none (package).
6. An abstract class inherits from Object and includes methods such as clone(0 and equals().

Even before we get to patterns this is pretty thorough and thought provoking.

Flyweights

Some of the thought questions take you on some pretty interesting side trips and cause you to think more about what Java is really about. The Flyweight pattern is an example of one that doesn't really occur that much in Java code, and for which it is hard to think of good pedagogical examples. Essentially, you create only a few instances of a Flyweight object and use some simple code to set the details of a particular appearance. This is important when you have hundreds or thousands of appearances of essentially the same object, such as a large set of folders or a set of characters on the screen. You really only need on instance of each kind of folder or character, and the various positions can be passed in as each is drawn.

Metsker clearly points out that the Flyweight is about "sharing." "The intent of the Flyweight pattern is to use sharing to support large numbers of fine-grained objects efficiently."

Then, he asks:

Name a Swing class that provides an example of a Flyweight.

I didn't know this one, but he has done his homework here. The API description of the javax.swing.BorderFactory class contains the following note:

Factory class for vending standard Border objects. Wherever possible, this factory will hand out references to shared Border instances.

Shared instances! Aha! A Flyweight!

Factories to the Fore

Once we touch on that idea, it is interesting that he uses the BorderFactory again a few pages later in discussing the Factory Method pattern. Remember that the Factory Method pattern is one where you have two parallel hierarchies of classes, and each subclass in one hierarchy chooses a specific instance of a class in the other hierarchy. Thus, there is no specific place where code decides which class to generate. Instead, the decision is made at coding time for each class. So then, he asks

Explain how the intent of the Factory Method is different from the intent of the BorderFactory class.

The reason I like this question is it pulls the reader up short and helps them realize that what I call a Simple Factory is quite different than a Factory Method pattern.

Not all of Metsker's answers are clear cut. He is perfectly capable of planting his feet firmly on both sides of the fence, and letting you decide for yourself. For example, he poses the question as to whether credit checking computations should be in separate packages for each country or whether they should all be in the same package. He argues both sides pretty well.

More Fireworks

A lot of the examples in this book are about a hypothetical fireworks company that has software to do the mixing and preparation of its products. He calls the company Oozinoz, which continually reminded me of Li'l Abner's "The Lizard of Ooze," even after he explained that the name came from the sounds the crowds make. The penny finally dropped about an hour later in my fevered brain: he means "oohs and aahs."

In this company context, he discusses the Template pattern and then discusses a press that discharges its chemical paste and flushes itself with water. He notes that in order to save the paste for other products, you need to move it away before the area flushes with water. The question is posed as

Write a note to the developers, asking for a change that will let you safely collect discarded paste before the machine flushes its processing area.

This is a pretty ingenious question when you see the answer:

What you want is a hook. You might phrase your request something like: "I wonder if you could add a call in your shutDown() method… If you call it something like collectPaste(), I can use it to save the paste we reuse."

The whole idea here is that a hook method is one that is supposed to be overridden, and may not have any real code in the base class. However, if you derive a new class, you can implement the *collectPaste()* method to do what you want without in any way needing to modify the original vendor's code. This re-emphasizes the real utility of the Template method pattern with a real example. I think this is a great example.

The Liskov Substitution Principle

Metsker spends a number of pages making sure you understand the basic premise of the Liskov Substitution Principle. In essence, this principle says that

An instance of a class should function as an instance of its superclass.

In other words, you should be able to use any derived class as if it had the exact same methods as its parent class. It should behave the same way, although some of the overridden methods may act somewhat differently, they should at least be analogous. An example that may violate the LSP in this book is the idea of deriving a class from a parent class and giving the child class different behaviors. For example, he suggests that the Firework class has a setExplode() method that determines whether a particular kind of shell actually explodes. However, if you derive the Sparkler class from Firework, it cannot utilize the setExplode method, because Sparklers by definition can never explode. This example may be a little strained but the implication is clear: your subclasses should usually not go off on a behavioral tangent. If they do, more powder to them!

Threads and Iterators

I've never spent much time in my own private coding worrying about thread safety: it just hasn't come up that much. But it is very important in production-level code, and the idea of a thread-safe iterator comes up in Metsker's text. In fact, thread safe iterators are a part of Java since version 1.2. You use the synchronizedList() method to get a thread safe

list that you can synchronize on. This is all described in the Javadoc API, but you probably never read it. I know I never did.

For example, suppose that you want to iterate through a list that another thread could conceivably change. You do it as follows:

```
List list = Collections.synchronizedList(new ArrayList());
          ...
      synchronized(list) {
         Iterator i = list.iterator(); //must be in synchronized block
          while (i.hasNext()) {
               foo(i.next());
          }
       }
```

Conclusions

My only quibble is the amount of white space on each page and the wide line spacing. I suppose the designers thought this gave the book a more "workbooky" look, with lots of room for marginal notes. I am reminded of the marginal notes found in Mark Twain's copy of Melville's *Moby Dick*. "Get on with it, man!"

Even though I've written a lot of words about Design Patterns in the last few years, I learned a lot of useful stuff from this book, and you will too. I recommend it.

Reference

1. Steven John Metsker, *Design Patterns Java Workbook*, Addison-Wesley, 2002.