Pattern Hatching

COMPOSITE à la Java, Part I

John Vlissides
Java Report, June 2001

Some of the most common patterns[1] are also the simplest, at least on their face. PROTOTYPE amounts to one method, **copy**, though it's easier said than done. Never heard of TEMPLATE METHOD? Don't worry. You're sure to have used it unwittingly if you know anything about object-oriented programming. And if STRATEGY wasn't taught in your Objects 101 class, it must have been the remedial section.

Conversely, the more complicated a pattern is, the less common it tends to be. It's no coincidence that FLYWEIGHT, INTERPRETER, and VISITOR are among the most complex, least understood, and least used patterns in our repertoire. When you need them, you need them; that's when most people bother to learn them. Until such time, they're easy to ignore.

But a few patterns are common *and* complex. BRIDGE, COMPOSITE, and OBSERVER come to mind here. Patterns like these tend to offer the biggest payoffs, and the biggest challenges.

I'll spend the next several columns looking at those last three patterns in detail. My aim is to rehabilitate them, as it were—to make them more relevant to Java, and just better all around. I'll be doing a lot of questioning and complaining and thinking out loud. Let me know if it gets annoying.

Rethinking COMPOSITE

Of all the structural patterns, COMPOSITE strikes me as most interesting and, strangely enough, the most controversial. It's interesting for a couple of reasons. First off, it describes a fundamental object-oriented data structure—the tree. But it's more than just another data structure because of how central polymorphism is to the pattern. Without polymorphism, there would be no uniformity. Clients would have to treat different elements in the tree differently. Which means you couldn't introduce new kinds of elements without changing clients. Which pretty much demolishes the pattern.

I can't think of a better argument for polymorphism than its role in COMPOSITE. As for the controversy, it's died down in recent years, but it flares up now and then. I'll defer discussing it until much later—till my next column, actually. Suspense is rare in computer discourse, no?

If we had to do COMPOSITE over again (and we do), the first thing I'd change is the Intent section. Right now it goes like this:

> *Compose objects into tree structures to represent part-whole hierarchies. COMPOSITE lets clients treat individual objects and compositions of objects uniformly.*

That's almost okay. It's got the "one or two memorable words" characteristic of a good Intent. In this case the words to remember are "tree" and "uniformly."

The words I don't like are "part-whole"—a Smalltalk-ism from way back—and the repetition of "compose"/"compositions"—too obscure and too complicated. What the heck is a "part-whole" anyway? And what does it mean to "represent … hierarchies" of them? That clause doesn't help enough to offset its vagueness, so let's flush it.

On the other hand, when people read "compose," I bet a lot of them visualize Beethoven scrawling a symphony onto parchment. I claim "assemble" evokes a better picture. Let's see…

> *Assemble objects into tree structures. COMPOSITE lets clients treat individual objects and assemblies of objects uniformly.*

Getting better, but there are still problems here. The first is that there isn't a "problem"—it's not clear *why* you'd want to do this. The clause we dropped contained the sole hint of a problem to be solved, namely that you need to represent something. Unfortunately, that something ("part-whole hierarchies") is meaningful to a minority of readers (hoary Smalltalkers), and even for them, the problem isn't all that explicit. Part-whole hierarchies of what, pray tell? It screams for an example. Too bad there isn't room for one—this is the Intent, you'll recall.

Seeing as we don't have space to be excruciatingly clear about the problem, we can do the next best thing: appeal to motherhood.

What does uniformity really buy us? At the end of the day, it *simplifies the client*. If clients can deal with objects uniformly, they don't have to test for differences and react accordingly. The differences are encoded in the concrete types of elements. Clients don't see concrete types; from their perspective everything's the same (abstract) type.

So why don't we say as much?

> *Assemble objects into tree structures. COMPOSITE simplifies clients by letting them treat individual objects and assemblies of objects uniformly.*

There, isn't that better? Um, well, like I said long ago, the Intent section may be the shortest, but it's the darnedest to get right. Drop me a line if you can do better.

Graphics, assets, or files?

Next up is the Motivation section. Currently it illustrates the pattern with a drawing editor, in which a user draws simple shapes such as lines and circles and assembles them into complete drawings. Classes that implement shapes play the Leaf role in the pattern. The **Picture** class plays the Composite role. The **Graphic** base class serves as the Component, defining the interface that all graphical elements implement to one degree or another.

For a while now I've suspected that this isn't the most accessible example of the pattern. I mean really, how many people use drawing editors, let alone implement them? They rank alongside tax packages and diet managers in the roster of niche applications. We can do better.

Could a more enlightening example be right under my nose? Of the examples mentioned in the Known Uses section, one in particular seems mainstream enough for the Motivation:

> *Another example of this pattern occurs in the financial domain, where a portfolio aggregates individual assets. You can support complex aggregations of assets by implementing a portfolio as a Composite that conforms to the interface of an individual asset.*[2]

Most people have assets they manage, right? (Most people reading this column, anyway.) Thinking of a portfolio as a composite of assets should therefore be easy to grasp. Alas, the whole discussion currently amounts to those two sentences.

"Let's rectify that," I say to myself. So off I go to rewrite the Motivation with an eye toward asset portfolios. Several false starts later, I realize I need to take a closer look at that cited known use.

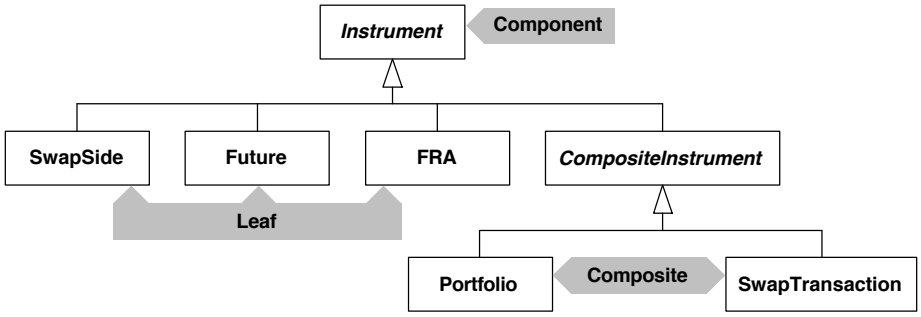

Figure 1: Financial instrument hierarchy

To make a long story short, Figure 1 depicts the COMPOSITE class hierarchy from the paper that describes the original example. I've added the gray boxes to indicate the roles each class plays in the pattern. Without explanation, you probably have more than a few questions about those classes, even if you know the pattern well.

Clearly there's some non-obvious domain knowledge here. Equally clearly, these classes do nothing to make the example more accessible. For a while I toy with simplified versions of the same basic example, but nothing really clicks. The drawing editor example is looking better all the time.

Then I recall the example in *Pattern Hatching* of an object-oriented file system API.[3] That example has proven accessible all right—too accessible. From the beginning, Erich complained that it's contrived, that no one would really design a file system API that way. I've countered that, while his complaint may have been valid at one time, several folks have since told me they've modeled their file system interface after precisely that example.

Funny, though, how nobody's ever sent me a follow-up saying how well it turned out. Makes you wonder how successful they've been. "Maybe a file system API isn't the best-documented example," I lament.

Then it hits me: The Motivation example doesn't have to be corroborated. It just has to be, well, motivating. That means it's *understandable* (so you can recognize when the pattern's needed) and *plausible* (to make you believe it's ever needed). I also realize that I don't have to talk about a file system API per se. I can discuss the design of something similar but more familiar to mere mortals: A file browser.

The Motivation won't be understandable if the example isn't simple, and a file browser is surely that, at least from the user's perspective. It's plausible too, since (a) everybody who's used a PC has used a file browser called "Windows Explorer," and (b) an object-oriented design for one shouldn't surprise anybody who programs in Java. The Motivation doesn't have to implement or even substantiate the example—it can leave those duties to other sections.

I think a file browser example can succeed on these counts. See if you agree.

Motivation

A file browser (Figure 2) manages information on disk. It lets you group files of different types into folders. These in turn may be grouped with files and folders into other folders, and so on. A simple implementation would define classes for files of different types, such as Document and Executable, plus classes that act as containers, like Folder and Trash.

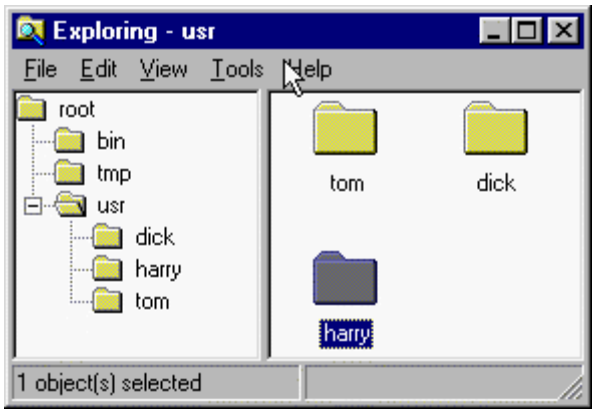

Figure 2: A file browser

But there's a problem with that approach. Browser code that works with files and folders must handle them differently, because they have different types. Yet most of the time, users of the browser treat files and folders identically. For example, users can "open" any file system object just by double-clicking, and they can view its name, creation time, and other common attributes using the same menu operations.

The code in the browser that responds to a double-click shouldn't have to check whether it's a document or a folder before opening it. That would complicate the code unnecessarily, just as it would complicate the user interface to have to left-double-click to open a file but right-double-click to open a folder. A common programming interface for telling file objects to "open" themselves has similar benefits. Browser code could ignore the details of opening different types of files, relying on the objects themselves to respond appropriately. The COMPOSITE pattern makes that possible. It describes how to assemble file objects so that browser code doesn't have to distinguish their types.

The key to the COMPOSITE pattern is an interface that represents individual objects *and* containers. For the file browser, this interface is **Node**. **Node** declares operations like **open** and **getSize** that make sense for all file objects but may nevertheless be implemented differently for different types. It also declares operations that all container objects share, such as operations for accessing and managing the objects they contain.

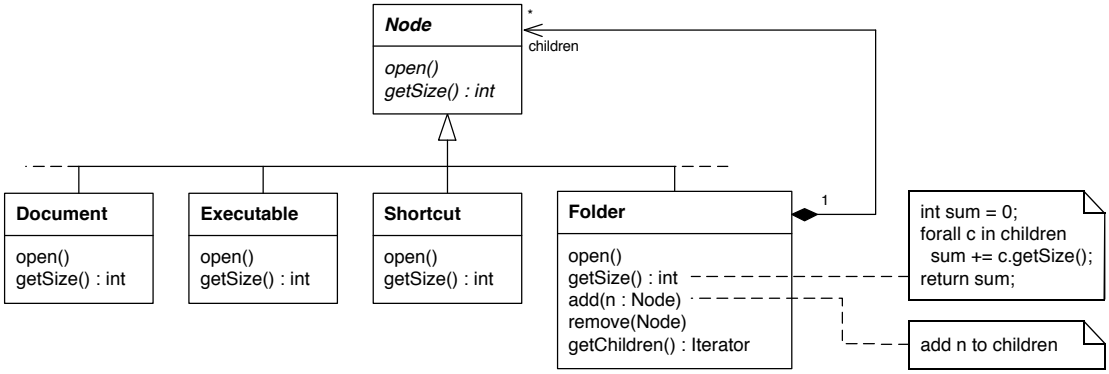

*Figure 3: **Node** hierarchy for file browser objects*

The subclasses **Document**, **Executable**, and **Shortcut** (see Figure 3) define simple file system objects. These classes implement **open** and **getSize** to open each kind of file object and to calculate its size, respectively. Since these objects aren't containers, none of these subclasses implements container-related operations. The **Folder** class defines a container of **Node** objects. **Folder** implements **getSize** by calling **getSize** on the objects it contains and summing the results. **Folder** also implements container-related

operations appropriately. Because the **Folder** interface conforms to **Node**'s, **Folder** objects can contain other **Folders** recursively. You can also add new subclasses of **Node** without changing existing ones.

Figure 4 shows a sample hierarchy of **Node** objects.

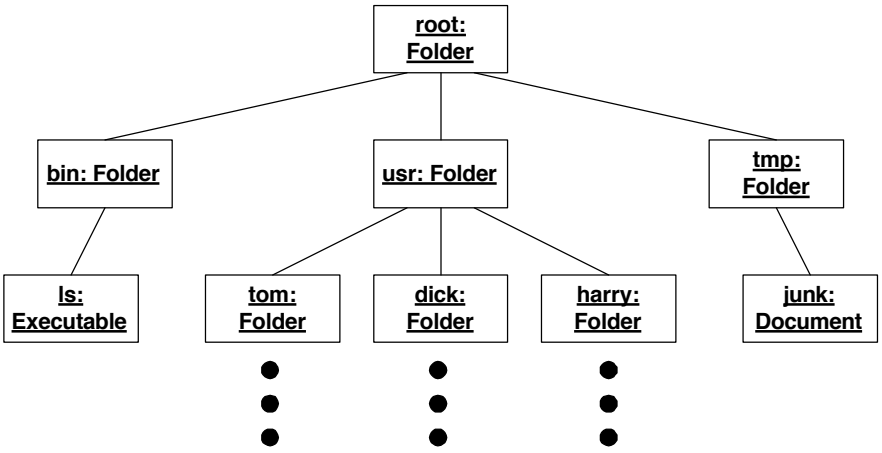

Figure 4: Sample object structure

Applicability

(The original version is pretty good, except for another irksome "part-whole" and a few other details. It's easily fixed.)

Use COMPOSITE when

- you want to work with hierarchies of objects.
- you want clients to be able to ignore the difference between groups of objects and individual objects. Clients can treat all objects in the hierarchy uniformly.

Auf wiedersehen

I'll tackle the rest of the pattern next time, including the all-important Consequences, Implementation, and Sample Code sections. Nothing will be sacred, I assure you.

Acknowledgments

My thanks to Dirk Riehle for his timely feedback.

References

[1] Gamma, E., et al. *Design Patterns*, Addison–Wesley, Reading, MA, 1995.
[2] Birrer, A. and T. Eggenschwiler. Frameworks in the financial engineering domain: An experience report. In *European Conference on Object-Oriented Programming*, pages 21–35, Kaiserslautern, Germany, July 1993. Springer-Verlag.
[3] Vlissides, J. *Pattern Hatching*, Addison–Wesley, Reading, MA, 1998, pp. 13–59.