Pattern Hatching

COMPOSITE à la Java, Part II

John Vlissides

Java Report, September 2001

In case you're tuning in late, we are in the midst of revamping COMPOSITE, one of the most useful and complex and controversial patterns out there. That it's useful and complex should be clear from last time.[1] So what's the controversy?

In a word, it's *fat*. A fat Component interface, to be precise.

Recall the Structure diagram for COMPOSITE, shown in Figure 1. Note the operations in the Component interface. After a nondescript `operation` come three others: `add`, `remove`, and `children`. They're related in that they deal with children—adding them, removing them, and enumerating them.

The question most people ask at this point is, Why are such methods declared in Component? After all, they don't make sense for Leaf classes, by definition. This appears to be a classic "fat" interface, one that declares any and all operations that subclasses might implement without regard for coherence or type safety. Academics will add gravitas to the criticism by branding this a violation of Liskov's Substitution Principle (LSP).[2]

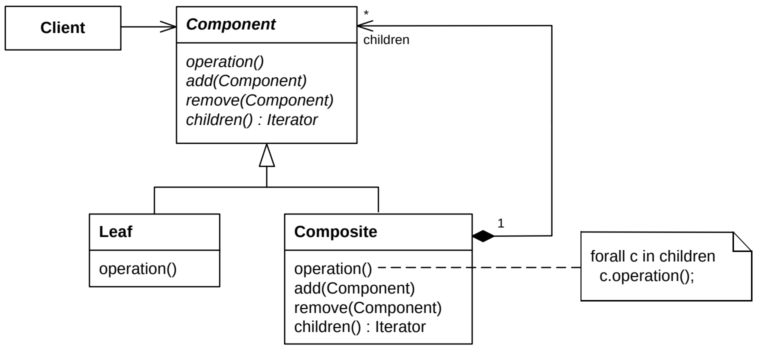

Figure 1: Classic COMPOSITE structure

How bad is it?

Not bad at all, necessarily. We're not talking about a random collection of operations here; all in question are child-related. Whether they "make sense" or not for Leaf classes depends on the application. I've argued that the semantics of calling `add` on a Leaf object could well be "do nothing," or "throw an exception," or "print an error message."[3] As for LSP, all that formidable-sounding dictum says is that a supertype's behavior must be supported by its subtypes, and that you should be able to substitute subtype objects for supertype objects without affecting client code. If those properties hold, you avoid the wrath of Liskov.

Good thing, too, because there may be splendid reasons to declare child-oriented operations in the Component interface. In the context of COMPOSITE, the primary reason is to achieve a key aspect of the pattern's intent—uniformity.

Here again is the full Intent statement as doctored up last time:

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

The trouble with barring child-related operations from the Component interface is that it defeats the very uniformity we seek. So why would one want to do it? Apart from blind obedience to LSP, the most common pretext is static type safety, or at least the hope of it. To wit, if `add` appears in Composite interfaces exclusively, then the compiler will disallow any call to `add` on a Leaf. Trouble is, it'll do the same for any call to `add` on a Component.

Suppose you're traversing a composite structure. The first thing you need is a way to get at a component's children. This alone argues for a `children` operation in the Component interface:

```
public void traverse (Component c) {
    Iterator i = c.children();

    while (i.hasNext()) {
        Component child = (Component) i.next();

        // do something with child, and/or...

        traverse(child);
     }
}
```

Note that we're already flouting static type safety here thanks to the downcast of `next`'s return value. Chalk it up to Java's lack of parameterized types, or its creators' fascination with casting, or whatever—but it's not statically type-safe. And yet you could argue, rightly I think, that if a component contains something other than components as children, something is very wrong indeed. The `add` operation guarantees as much by taking only `Component` objects as parameters.

Anyway, let's assume this level of type safety is acceptable, since traversal is definitely something we want to do in Java. In other words, we're okay with declaring `children` in our Component interface.

Now suppose we want to add something to a child during the traversal. Here's the rub: Even if we know statically that said child is a composite, we'll nonetheless have to cast it to `Composite` to get at its `add` operation. I'll illustrate by replacing the comment in the code above with the cast I'm talking about:

```
        ((Composite) child).add(new Leaf());
```

If we're not sure the child is a composite, we might assuage our fears with a test.

```
        if (child instanceof Composite) {
            ((Composite) child).add(new Leaf());
        } else {
            // so much for static type safety
        }
```

My point is that static type safety can be an illusion, LSP or no LSP. Rather than kidding ourselves as to its preeminence, maybe it's better to admit we can't guarantee it and err on the side of uniformity.

Therein lies the beauty of a pattern: it doesn't have to come down on one side of a controversy. It can dispassionately relate the available trade-offs and allow you to make an informed choice. You can trade off uniformity for static type safety and vice versa, depending on the circumstances. One size needn't fit all.

An alternate Structure

That said, there may well be times and programming languages in which erring on the side of static type safety (and hence omitting child-related operations from the Component interface) makes perfect sense. A reader who reaches that conclusion may find Figure 1 misleading. That's especially likely for those who labor under the misconception that the Structure diagram is a *specification* of the pattern's implementation rather than just an *example* thereof.

We made it a point to include just one class diagram per Structure section. Now I'm thinking it's justifiable to include two here, one for the "uniform" case and another for the "type-safe" case (see Figure 2). The main difference is that type safety requires a separate interface for Composites.

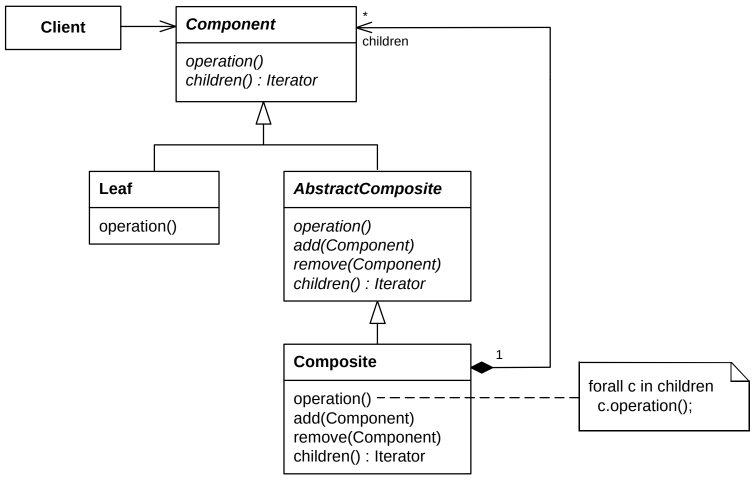

Figure 2: Alternate Structure diagram for the "type-safe" case

Participants, Collaborations, and Consequences

While the Collaborations haven't changed, the Participants section needs updating to reflect the changes to the Structure diagram. In particular, there's a new bullet to describe AbstractComposite, and Composite is tweaked to allow for AbstractComposite's possible presence.

The current Consequences section pretty much covers the good and the bad of the pattern, so I'll leave well enough alone there as well. I do have a cosmetic change, however.

Nowadays I like to segregate a pattern's benefits and liabilities. I particularly like how the Siemens folks use separate itemized lists set off by an introductory sentence containing a form of the word "benefit" or "liability" in bold.[4,5] It's easy to distinguish the good consequences from the not-so-good ones. We already use bold to identify glossary words, though. So I'll use italics, even though they don't stand out as well as bold. The result looks like this:

COMPOSITE offers these *benefits:*

- It defines class hierarchies consisting of primitive objects and composite objects….
- It simplifies the client….
- It makes adding new kinds of components easier….

The pattern has the following *liability:*

- It can make your design overly general….

COMPOSITE's dearth of liabilities introduces a small inelegance, namely a bulleted list with only one bullet. The standardized benefits/liabilities format lets us get away with it, but ideally there'd be at least one more bullet and a liability to go with it. One possibility is the threat of inefficiency if the pattern is applied at too fine a granularity. That's awfully close to the existing liability, I fear.

In any case, if a lone bullet really gets your goat, we could merge its contents with the introductory sentence ("The pattern's chief *liability* is that it can make your design overly general…."). I can certainly live with that.

Implementation

"There are many issues to consider when implementing the Composite pattern." No joke. Truth be told, I've never been happy with the current hodgepodge of implementation items. Whenever you have more than four or five items, you're almost certain to get that grab-baggy feel.

One way to reduce the number of items is to group them into related sets, which are then substructured. Here are the nine items we have currently:

1. *Explicit parent references.*
2. *Sharing components.*
3. *Maximizing the Component interface.*
4. *Declaring the child management operations.*
5. *Should Component implement a list of Components?*
6. *Child ordering.*
7. *Caching to improve performance.*
8. *Who should delete components?*
9. *What's the best data structure for storing components?*

These items are interdependent, naturally, as they concern the same pattern. But there are subtle commonalities that we can exploit.

The most obvious one lies in items 3 and 4, both of which address the fat interface issue. In fact, the distinction between these two seems downright arbitrary. They could easily be consolidated into an item titled "Type safety versus uniformity," which cuts to the crux of the issue. While we're at it, we should mention the role of Java interfaces and abstract classes, two features that weren't available in the original languages. It's the usual admonition, but it bears repeating: use interfaces for base classes, putting any default functionality in abstract classes that partially implement those interfaces.

Other commonalities are subtler. Items 2 and 8 have to do with child ownership. Shared components cannot be deleted indiscriminately, and (lack of) garbage collection plays a big role in both items. Accordingly, we can group them under the rubric of "Child ownership." Meanwhile, items 1, 5, 6, and 9 are all about associations between parent(s) and child(ren)—whether they're explicit, ordered, how they're implemented and where. We'll unite these guys under "Associations between Composite and Leaf components." The original items show up as subitems of these groups (suitably tweaked and updated, of course).

That leaves item 7, the only item that focuses on performance per se. Seeing as there's no law that says every item must have subitems, it can certainly stand by itself. Nevertheless, I can think of a few other performance issues worth addressing—applying the pattern at too fine a granularity, for example, and the attendant bad juju. Together, such issues can form a comprehensive discussion of performance, which the pattern underemphasizes right now, IMHO.

So now we have just four main implementation items:

1. *Type safety versus uniformity.*
2. *Child ownership.*

3. *Associations between Composite and Leaf components.*
4. *Performance.*

Much better.

Sample Code

Currently, this section of the pattern breaks an unwritten but cardinal rule: The Sample Code should illustrate the example in the Motivation section. Why? Three reasons:

1. It takes verbiage to give an example and make it understandable. The Motivation section already does it, so why do it again for a different example in the Sample Code?
2. Repetition is the mother of understanding, or something like that.
3. I'm lazy. Coming up with another good example is work I'd rather not do.

Clearly then, Sample Code should illustrate the file browser example of the new Motivation given in Part I.[1] Oddly enough, that example suggests a bias toward type safety. Figure 3 in Part I shows `add`, `remove`, and `children` operations declared in the `Folder` Composite class but not in the `Node` interface it implements. The Sample Code should illustrate that implementation along with one that showcases the uniform approach.

Alas, you'll have to use your imagination for the rewrite, because I'm nearly out of space. One thing you won't have to guess about, though, is the predominant implementation language. Only stuff that can't be expressed in Java—a variation using parameterized types, for example—would call for a different language, C++ in that particular case. If you're one of those hoary Smalltalkers, let me point you to an excellent resource: *The Design Patterns Smalltalk Companion*.[6] It does a better job of rendering our patterns in Smalltalk than I can ever do.

Known Uses

COMPOSITE's original known uses are as valid as ever, but they're a bit long in the tooth. Happily, there's no shortage of examples in the JDK. Note which side of the fat interface controversy each use falls on.

- java.awt.Component/Container errs on the side of type safety.
- javax.swing.text.View/CompositeView errs on the side of uniformity.
- javax.sound.sampled.Control/CompoundControl is a variation in which the set of children is immutable—they're supplied in the CompoundControl constructor. There's no interface for changing their number. Otherwise this use is strongly biased toward static type safety, as only CompoundControl includes so much as a `getMemberControls` method.
- javax.swing.Border/CompoundBorder is similar to Control/CompoundControl but more degenerate—there are just two constructor-specified children.

Related Patterns

Material for this section was easy to come up with as we wrote *Design Patterns*—the only patterns we had to relate to were our own! Things are much different now. One thing we purpose to do in all our patterns is link them to the now-vast pattern literature. Because the literature gets added to every day, Related Patterns has become the most open-ended section. Still, we think readers will appreciate pointers to patterns from whatever source, as long as they're good and relevant. And not just pointers to patterns but also to known uses, case studies, and anything else that sheds light on the pattern in question. The entry for COMPOSITE in *The Pattern Almanac*[7] is a foreshadowing of the new version.

Acknowledgments

Hats off to Erich Gamma, Richard Helm, Heinz Kabutz, and Dirk Riehle for many helpful suggestions.

References

[1] Vlissides, J. "COMPOSITE à la Java, Part I," *Java Report*, June 2001, pp. 72, 69–70.
[2] Liskov, B., J. Guttag. *Program Development in Java*, Addison–Wesley, 2001.
[3] Vlissides, J. *Pattern Hatching*, Addison–Wesley, 1998, pp. 18–24.
[4] Buschmann, F., et al. *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.
[5] Schmidt, D., et al. *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.
[6] Alpert, S., et al. *The Design Patterns Smalltalk Companion*, Addison–Wesley, 1998.
[7] Rising, L. *The Pattern Almanac 2000*, Addison–Wesley, 2000.