Pattern Hatching

Notation, Notation, Notation

John Vlissides
C++ Report, April 1998

I've been itching to write a column about design notations for ages, but I couldn't bring myself to do it. I'm not sure why. Maybe the notation wars that the methodologists fought a few years back turned me off to the whole issue. Or maybe it didn't seem worthy of an entire column, which would betray a certain chauvinism on my part—that real men *build systems*; they don't agonize over how to draw a class, or the semantic subtleties of bubble-headed lines. Maybe I was afraid of looking less than macho.

Still, I think the true reason is that I just didn't appreciate the importance of the topic. But that's changed. Writing columns and answering people's questions have forced me to go back and examine tons old of e-mail. In the process I've noticed that a small but significant percentage of questions concern the notations we use in *Design Patterns*—mostly misunderstandings about them. A similar percentage ask how to make patterns explicit in design documentation. Yet another, larger percentage involve confusion about how, or even if, certain patterns really differ from one another.

These percentages add up to a slim majority of the e-mail, so there's no doubt the issues here are noteworthy. What's particularly interesting, and something I realized only recently, is that the answers to all these questions involve notation. Meanwhile, Robert Martin has inaugurated a series of columns on UML, which is destined to become the gold standard for object-oriented design notations. And Jim Coplien explored the geometric nature of patterns in his last column[1] and will delve even deeper in his next. Harmonic convergence? Beats me, but if I'm ever going to write about this stuff, now's the time.

The whys and wherefores

What good are graphical notations anyway? What are their benefits, and what do they cost?

I haven't heard of an empirical study that proves anything about OO notations in particular. But it is well known that diagrammatic presentation can help people grasp information more quickly than straight text. Edward Tufte drives this point home in his landmark book, *The Visual Display of Quantitative Information*:[2]

> *Modern data graphics can do much more than simply substitute for small statistical tables. At their best, graphics are instruments for reasoning about quantitative information. Often the most effective way to describe, explore, and summarize a set of numbers—even a very large set—is to look at pictures of those numbers. Furthermore, of all methods for analyzing and communicating statistical information, well-designed data graphics are usually the simplest and at the same time the most powerful.*

Tufte is talking about presenting statistical data, but his observations hold true for almost any information. In fact, graphics are arguably more important for conveying object-oriented design. The only alternatives are code or pseudocode. Both are fine for specifying an implementation, but they are pretty poor at communicating design. While dedicated design languages exist, they have not proven popular; the ones I've seen offer little expressiveness over code or prose. They just don't carry their weight.

It's hard to argue against the general goodness of graphical notations. What's easier to refute is their suitability for a particular purpose. Code may be inferior to graphical notations for expressing design

information, but the flip side is that graphical notations have proven inferior to code for general-purpose programming.

Research into what became known as “graphical programming” came into vogue about twenty years ago. This coincided with the advent of relatively cheap semiconductor memory, which in turn made bitmapped displays and their connection to minicomputers possible at places like SRI, Bell Labs, and Xerox PARC. Once researchers had a viable graphical medium at their disposal, they began to dream in earnest of programming by drawing pictures instead of typing. There was widespread belief that a drawing metaphor would bring programming to the masses by making it more natural. Expert programmers too would benefit from a greater expressiveness.

Alas, most efforts at graphical programming amounted to naïve transliterations of conventional programming constructs. Would-be graphical programmers metamorphosed into electricians of another color, laboriously wiring up boxes labeled “function”, “if”, and “+”. Had they the tenacity to create a complete program that way, the resulting mass of lines and boxes was difficult to read, let alone debug and maintain. The expressiveness of text proved hard to beat. I guess that’s one reason this article is written in Roman characters and not hieroglyphics.

Not much has come of graphical programming research per se, but there have been successful offshoots. What’s made them successful is catering to a more restricted domain than general-purpose programming. Graphical user interface (GUI) builders are a good example; they let you assemble an application’s GUI by manipulating its elements interactively. This is a form of graphical programming, but it’s limited to expressing the user interface. Developers revert to conventional programming to implement other parts of the application.

The moral here is simple. If you want to get nontrivial information across, you’ll want to use any and every means to convey it accurately and in the smallest space. To succeed is to achieve a delicate balance of graphical depictions and descriptive text. The two should act synergistically, neither overpowering the other. Readers should come away feeling informed without having worked at it. They will be persuaded, refreshed, even energized—and probably impressed too.

Notation in patterns

We purposed not to make a big deal about notation in *Design Patterns*.[3] We would stick to standard notations and would change them only if we had to. Our motives were purely pragmatic. We didn’t want to get into the methodology business, with which notations seemed inextricably linked. We also wanted to avoid making notation an end in itself, for we had bigger fish to fry.

We ended up using three formal notations:

1. **Class diagrams** for specifying classes and their static relationships.
2. **Object diagrams** for depicting snapshots of objects and their interconnections.
3. **Interaction diagrams** for showing object interactions.

We went with OMT[4] (with small modifications) for class and object diagrams, and OOSE’s notation[5] for interaction diagrams. Had the UML standard[6] been around in 1993, I’m sure we would have gone with that.

As indispensable as formal notations are, though, we found informal diagrams equally important, especially in a pattern’s Motivation section. By “informal” I mean pictures, sketches, screen shots, and anything else that’s graphical but doesn’t conform to rigid conventions. For example, DECORATOR’s Motivation talks about adding a border and scrollbars to a text-editing component of a user interface. The discussion includes a screen shot of the final product—a bordered and scrollable text component—alongside an exploded view of its constituent components, including a border decorator, a scrollbar decorator, and the original text component. Elsewhere, MEDIATOR’s Motivation section shows a dialog box with interdependent elements to make the argument for a mediator concrete.

Informal diagrams are good in these contexts because they are non-threatening. You don't have to be comfortable with a formal notation to understand and appreciate them. Informal diagrams ease the reader into the pattern, while formal notations are best applied where precision is more important than accessibility.

And therein lies a rub, because formal notations can suggest a level of precision that isn't there. Compounding the problem is their succinctness—they can say a lot in a small space. Put the two together and you've set the stage for "pattern legalism." The symptoms are obvious. Someone asks whether a minor deviation from a pattern's Structure diagram means he's "not following the pattern." Or an implementation is contorted to reproduce a particular interaction diagram. In the extreme, a design includes a class "ConcreteClass" that defines `PrimitiveMethod1` and `PrimitiveMethod2` operations, in almost comical adherence to the TEMPLATE METHOD pattern.

I guess such misunderstandings are our fault. It seems you can't overemphasize that a pattern's Structure diagram is just an example, not a specification. It portrays the implementation we see most often. As such the Structure diagram will probably have a lot in common with your own implementation, but differences are inevitable and actually desirable. At very least you will rename the participants as appropriate for your domain. Vary the implementation trade-offs, and your implementation might start looking a *lot* different from the Structure diagram.

As for whether one is following the pattern or not, who cares? The pattern is a means to an end, not an end itself. Following it in any strict sense is immaterial. If the pattern solves your problem directly, that's great; if you have to bend it a bit, that's great too. Even if the pattern merely inspires you toward an altogether different solution, it has still proven useful. The only potential problem here lies in the documentation phase, when you're describing your solution in terms of patterns. You don't want to mislead someone with irrelevant patterns. If you identify a set of classes as adhering to a pattern, make sure they fulfill the pattern's intent. If the connection is tenuous, don't mention the pattern; otherwise you're sure to confuse more than clarify.

There's a related source of confusion that's at least as prevalent. It is the desire to deduce pattern differences solely by comparing Structure diagrams. Compare STATE's diagram to STRATEGY's, for example, and you'll note few differences. Indeed, both are of the form shown in Figure 1: a Context class delegating functionality to one of several subclasses of an AbstractClass. The structures are virtually identical, right down to the aggregation relationship between the two base classes.

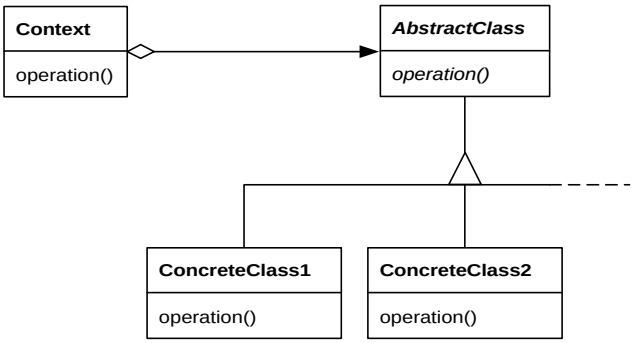

Figure 1: Common structure of STATE and STRATEGY

Does that mean there is no real difference between STATE and STRATEGY? Of course not. All it means is that delegation is commonly used to implement both patterns. Remember that the structure diagram defines only static implementation relationships, and these are expressed in terms of a small set of OO language mechanisms: classes, inheritance, references, and polymorphic operations. Thus the diagram tells only part of the story. It says little or nothing about dynamics—how and when objects get created, how they communicate, and when they go away. Implementations of similar Structure diagrams may produce widely varying object structures at run-time. Even object structures that look similar can exhibit radically different behaviors. If you're looking to distinguish a pattern from others, look first to its intent, the most concise differentiator.

STATE's intent is to let an object alter its behavior when its internal state changes, making the object act as if it can change its class. The pattern accomplishes this by encapsulating state-specific behavior in ConcreteState classes and replacing their instances at run-time. Clients are oblivious to state transitions and the rules by which they occur.

In contrast, STRATEGY's intent is to encapsulate algorithms and let them vary independently of clients. The pattern focuses not on modeling state and transitions but on issues of encapsulation. The treatment of transitions that STATE spends so much time on is not relevant to STRATEGY. Moreover, STRATEGY explicitly charges clients with providing the strategy to use up-front. If a different strategy were needed, the client would have to supply it. Thus the replacement frequency for strategies is usually much lower than that of states.

The Structure diagrams, by the way, reflect none of this. They are each just a part of a broader exposition. Every pattern is peppered with insights that distinguish it from others. You have to study the whole pattern to master it.

Notation for patterns

The last topic I'll touch on is also the one I find most interesting. It's about making patterns explicit in design documentation. Specifically, how do you identify patterns in a design graphically?

We didn't talk about this topic in *Design Patterns*, and now I'm wondering why. We appreciated the need for graphical notation at the class and object level; why didn't we use it at the pattern level? Maybe we just hadn't gotten that far. OMT et al. were still rather new at the time, and we found it necessary to augment them to express basic design relationships, never mind patterns. We had also recognized the need for modest cues to a pattern's presence in code, such as prefixing primitive operations of template methods with "Do-".[3] But we stopped short of recommending naming conventions that identify patterns in code, even though I for one had used such conventions on occasion. For example, I've written quite a few Strategy classes with "Strategy" in their name to identify them as such. The problem with such conventions is that they don't scale well. If a class happens to participate in more than one or two patterns, the name that results gets unwieldy fast—"AbstractFlyweightFactorySingleton" being not the worst example I've seen.

While the jury is still out on a good way to identify patterns in code, there are several viable notations for identifying patterns in design diagrams. The one I adopted early on had Venn diagrams as its inspiration. I used it in one of my early columns[7] to show how the PROXY and COMPOSITE patterns manifested themselves in the design of a simple OO file system. Figure 2 is the diagram from that column. It shows that the Node, File, and Directory classes participate in the COMPOSITE pattern, while Link and Node are participants in the PROXY pattern.

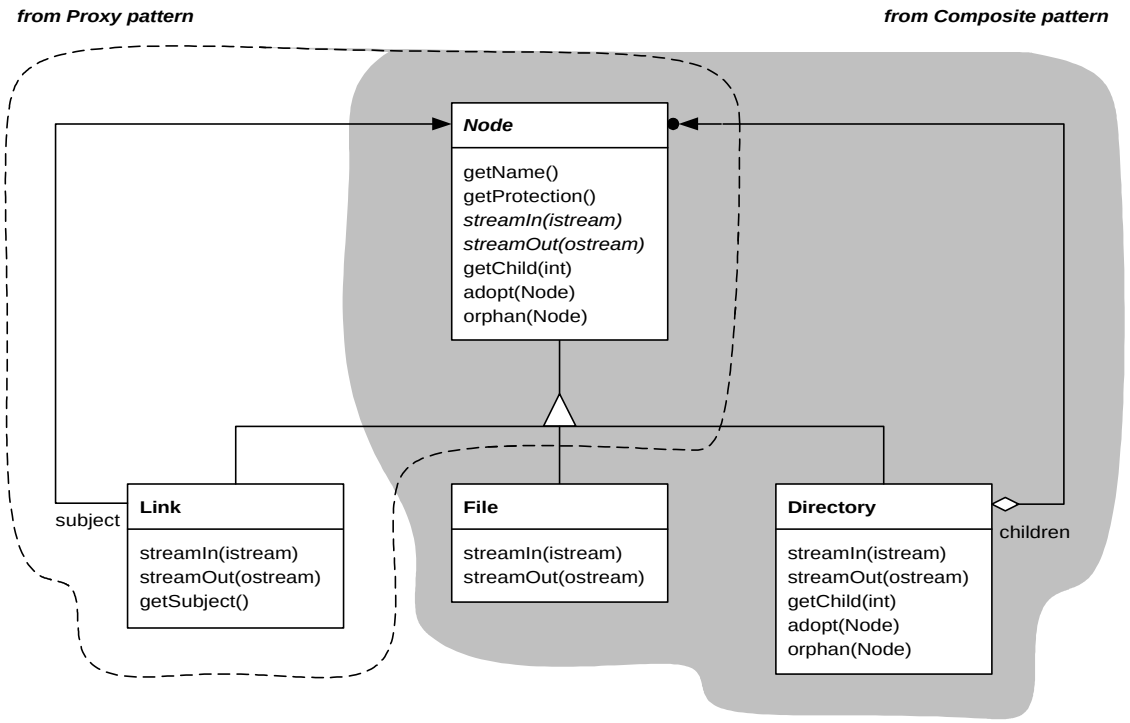

Figure 2: Venn diagram-style pattern annotations

While this notation works okay when there are few patterns per class, it too suffers from scalability problems. The more patterns a class participates in, the more overlapping regions you get, and the more gerrymandering you have to do to cover them. It's also hard to identify precisely the participant roles a class plays. Did you realize that the Link class in Figure 2 plays the role of a Proxy? You're pretty astute if you did, because it's not explicit. You have to know the pattern well to deduce which classes play which roles from the class structure alone.

UML offers an alternative notation that addresses these problems. UML's **collaboration diagrams** can depict design pattern structure by identifying patterns and their participants in a class diagram. Figure 3 shows how. Pattern names appear in dashed ellipses, and dashed lines labeled with participant names associate the patterns with the appropriate classes. This is a marked improvement over the Venn diagram approach, since you no longer have to guess about the role(s) of each class.

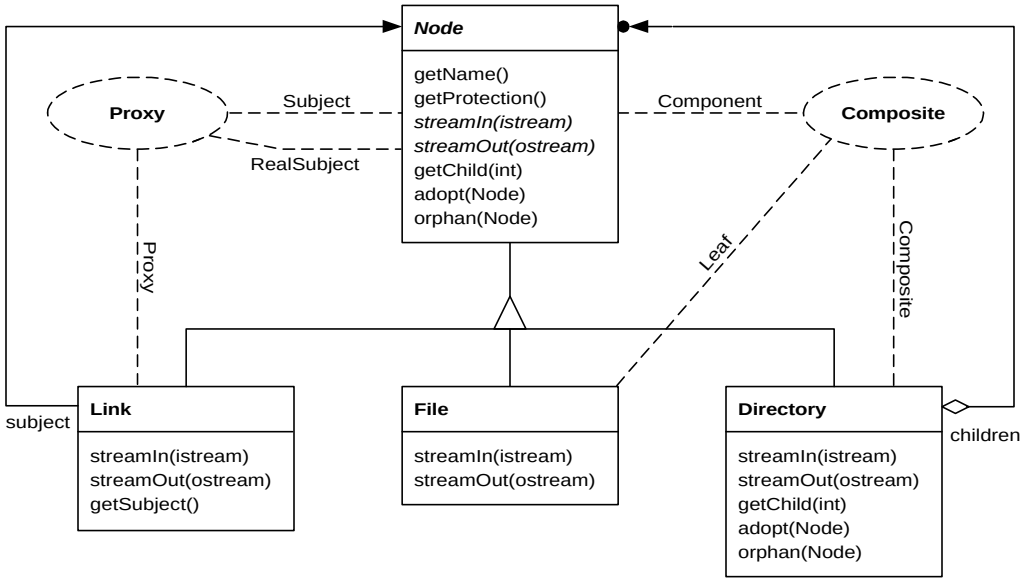

Figure 3: UML collaboration notation

There's still room for improvement, however, because the dashed lines clutter the presentation significantly. The pattern information competes with the class structure, making both harder to see. Figure 4 shows another approach that Erich came up with a couple of years ago. He calls it "pattern:role annotations." It tags classes with shaded boxes containing the pattern and/or participant name(s) associated with a given class. For brevity, only the participant name is shown if there's no ambiguity. By avoiding added lines and by using boxes with a contrasting background, clutter and interference are minimized—pattern-related annotations appear to occupy a different plane from the class structure. I've found this approach highly readable, informative, and scalable. The only drawback I've noticed is that the gray backgrounds don't fax too well. *Caveat scannor!*

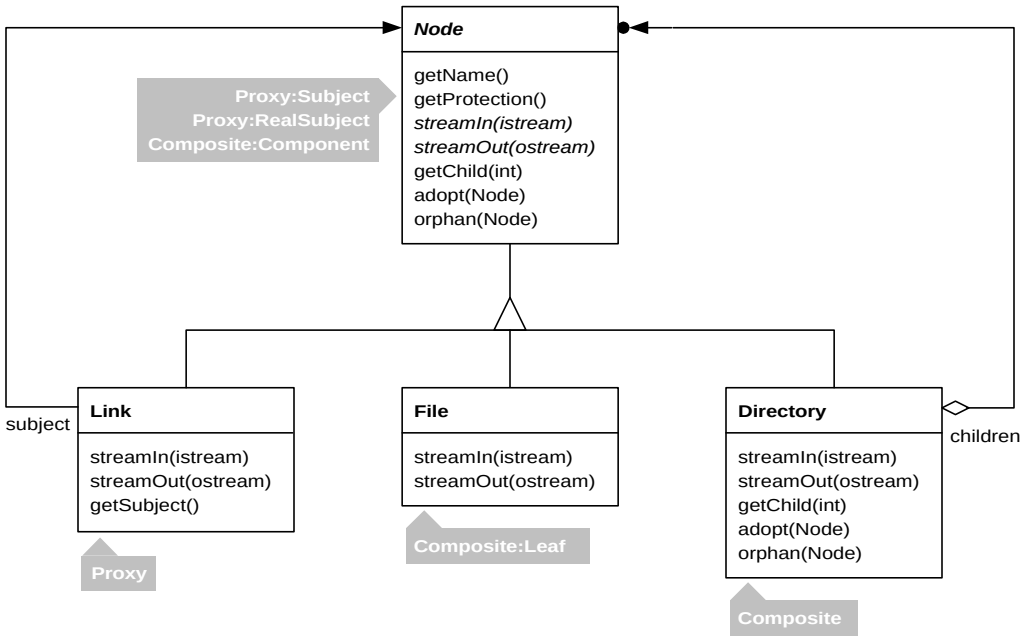

Figure 4: Pattern:Role annotations

Notation beyond patterns

Having come this far in applying notations to patterns, it's natural to wonder what's next. There's a clear trend toward packing more and more meaning into design notations. At present, UML defines seven major notations for just about every aspect of object-oriented design you can think of. The notations we settled on back in 1993 are but a fraction of what can be expressed diagrammatically today. Additional notation could make many of our expositions clearer or more precise, or both. But it's easy to get carried away. We mustn't forget that text is a powerful medium, too.

In fact, I find prose far more difficult to perfect than any diagram. Perhaps it's just me, but I doubt it. There's something very deep about human communication. Sometimes I view it as the ultimate programming language. That analogy probably does violence to natural languages, but it's fun to think about anyway. Consider English. It's rich, it's expressive, and it's widely understood. Development tools are ubiquitous and cheap, but debugging support remains primitive, limited almost entirely to syntax checking. Moreover, the idiosyncrasies of English make C++ look minimalist. And it's really, really hard to predict what you'll get from a "compiler" (or "interpreter"?), because no two (people) are alike. Heck, anybody can do diagrams; writing is the *real* challenge!

Could I have written the wrong column?

[1] J. Coplien. "Space: The Final Frontier," *C++ Report*, March 1998.
[2] E. Tufte. *The Visual Display of Quantitative Information*. Graphic Press, Cheshire, CT, 1983.
[3] E. Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
[4] J. Rumbaugh, et al. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
[5] I. Jacobson, et al. *Object-Oriented Software Engineering—A Use Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.
[6] Rational Software Corp., et al. "UML Notation Guide, Version 1.1," 1997.
[7] J.Vlissides. "Visiting Rights," *C++ Report*, September 1995.