Pattern Hatching

PLUGGABLE FACTORY, Part I

John Vlissides
C++ Report, November–December 1998

Last time I introduced the concept of composite design patterns—patterns that document synergies between other patterns.[1] I also promised to offer up examples of such patterns over several columns. I'll start making good on that promise beginning with PLUGGABLE FACTORY.

Before doing that, though, I'd very much like to propose an alternative name for these beasties. Recall that Dirk Riehle coined "composite design pattern" to refer to assemblages of patterns as distinct from the indivisible patterns we're familiar with. He chose "composite" partly because he felt it was the OOP term of choice for aggregations of things, be they objects, domain models, components—whatever. An informal straw poll confirmed his intuition. But several of those polled also expressed concern over a potential ambiguity.

What's in a name?

One thing I've learned over the years is the value of pattern names as an aid to communication, especially *verbal* communication. Alas, when you say "composite pattern" out loud, a listener can't tell whether you're referring to a particular pattern, namely COMPOSITE, or to a composition of patterns. The distinction is both crucial and context-dependent, making it an impediment to clear expression. And an unnecessary impediment at that; choose something other than "composite" to denote pattern compositions, and the ambiguity evaporates.

Many alternatives have been proposed, including "aggregate patterns," "combination patterns," "compound patterns," "pattern teams," and "pattern ensembles." I like "compound patterns" best, for three reasons. Most importantly, it's a natural usage of the term "compound." You don't have to be clairvoyant to infer what's intended. Second, "compound" avoids ambiguity, since there are no widely cited patterns by that name.

The third reason isn't a particularly good one, but I have to be honest: "compound" has nostalgic value for me. Table 1 shows an early attempt to categorize the design patterns that appeared in our original paper on the topic.[2] The attempt is so early, in fact, that you probably won't recognize some of the names: "Wrapper" became DECORATOR, "Glue" became FACADE, "Solitaire" became SINGLETON, and "Walker" became VISITOR. You may notice a few more unfamiliar names, like "characterization" and "jurisdiction." These were provisional terms for what we ended up calling "purpose" and "scope" in *Design Patterns*.[3]

Purpose classifies what a pattern does. For as long as I can remember, there have been three kinds of purpose: *creational*, *structural*, and *behavioral*. Creational patterns abstract the instantiation process, whereas structural patterns focus on how classes or objects are *put* together, and behavioral patterns tell you how they *work* together. Meanwhile, *scope* suggests the domain over which a design pattern applies. Patterns with *class* scope focus on static relationships, which mostly means inheritance. Patterns involving more dynamic relationships are classified under *object* scope.

That, in a nutshell, is all there is to this classification scheme today. But once upon a time there was a third kind of scope—you guessed it, *compound*. It was meant to distinguish recursive relationships from static or dynamic ones. Most patterns so branded had something to do with traversal, which in C++ is commonly implemented using recursion. Only Wrapper (DECORATOR) didn't involve traversal; it was recursive in the sense that decorators can decorate other decorators, ad infinitum. Under this definition, classifying Walker (VISITOR) as a compound pattern was presumably a gaffe, or at least a stretch, because a pure visitor doesn't

		Characterization		
		Creational	**Structural**	**Behavioral**
Jurisdiction	**Class**	Factory Method	Adapter (class) Bridge (class)	Template Method
	Object	Abstract Factory Prototype Solitaire	Adapter (object) Bridge (object) Flyweight Glue Proxy	Chain of Responsibility Command Iterator (object) Mediator Memento Observer State Strategy
	Compound	Builder	Composite Wrapper	Interpreter Iterator (compound) Walker

Table 1

recurse apart from an iterator or some other traversal mechanism. It seems the notion of visitor as distinct from traversal hadn't yet gelled.

But this is all moot anyhow, because few people understood compound scope. Even fewer found it useful. Turns out the *purpose* classification offers a lot more intellectual traction than scope does. It's common to refer to "the creationals" as a group, for example, because their commonality—instantiation—is obvious and highly relevant to design. Purpose can also channel you to the right pattern, as I demonstrated way back when we were adding symbolic links to an object-oriented file system API.[4] There, the fact that we were monkeying with the file system's structure suggested a pattern of structural purpose. A subsequent quick-scan of the variabilities imparted by structural patterns led us to PROXY, which did the trick nicely. Seldom is scope so useful.

For reasons that aren't clear to me just now, we never saw fit to abandon scope outright. We agreed however that it didn't warrant a three-way classification. So we ended up abandoning compound scope, the most esoteric of the three. While we were at it, we renamed "jurisdiction" and "characterization"—ten-dollar words if there ever were any—to "scope" and "purpose." Everyone felt much better afterward.

Now then, given that "compound" and "composite" are synonymous for our purposes, and the former avoids the ambiguities of the latter, and there's little danger of confusing any new use of "compound" with the nostalgic old nomenclature—I move we adopt "compound patterns" as a replacement for "composite patterns." All in favor say "Aye!"

Can we look at one now?

Okay, with the administrivia out of the way, let's contemplate PLUGGABLE FACTORY, which I introduced last time as PROTOTYPE–ABSTRACT FACTORY. My aim is to document this compound pattern, of course, and also to reveal issues that are unique to reading and writing such patterns.

If there's any leverage to be had in a compound,* it must build on constituent patterns. There's no point recapitulating those patterns in the compound, because the reader can always consult them directly. A compound would also get very big, very fast if it subsumed its constituents. So it's not just permissible but *essential* that a compound defer to other patterns. It must add value, too. The cardinal rule for compound pattern readers and writers alike is to focus on the synergies between patterns.

Synergy starts with the compound pattern's name. You can always form it by stringing together the names of the constituents, but there's not much value-add in that—where's the synergy? Such a name will be unwieldy to boot when there are more than a couple constituents. PROTOTYPE–ABSTRACT FACTORY is a glowing example.

A good name is as critical to compounds as it is to any other patterns: it should be short, evocative, memorable. It'll be something you can use in conversation without hesitation and without getting tongue-tied.

"Pluggable" is fitting here because it connotes dynamic reconfigurability, which as we'll see is highly appropriate. And we're still talking about a factory in the ABSTRACT FACTORY sense. What is no longer relevant to this compound is the factory's "abstractness" or how pluggability is achieved (using PROTOTYPE in this case). The ABSTRACT and PROTOTYPE monikers are thus safely dropped, leaving "PLUGGABLE FACTORY."

Fine. But having said all that, I'll admit it's still helpful to see the names of a compound's constituents upfront. They can tell you a lot about the pattern before getting to the nitty-gritty. The question is, where should the constituent names appear, assuming we're sticking to the traditional design pattern template? It seems we need a new heading, maybe "Constituents" or something.

Call me a curmudgeon, but I'm loath to change the template just to handle compounds. Besides, a heading for alternative names already exists: "Also Known As." PLUGGABLE FACTORY can be Also Known As PROTOTYPE–ABSTRACT FACTORY. No sweat.

Classification and Intent

Given all the hay I've made about our classification scheme, it's fair to ask where compound patterns fall into it. Is a compound's classification a function of its constituents', or can it be altogether different? Does the current scheme even *apply* to compound patterns, or will it need extension to handle them?

Here again, the guiding light should lead us away from reiterating the obvious and toward capturing synergies. PLUGGABLE FACTORY's constituents are both object-creational patterns, so you might conclude their composition is object-creational as well. I would agree completely. But what if one of them were class-structural instead? Is the result a class-object-creational-structural compound pattern, or some permutation thereof? I don't think so. Simply enumerating the constituent classifications is of precious little worth.

No, a compound's classification should reflect its own unique intent and synergy. And I assume the existing scheme is adequate for compounds, at least until proven otherwise—and proof will require a more dissonant combination than ABSTRACT FACTORY and PROTOTYPE.

So PLUGGABLE FACTORY is an object-creational pattern, just like its constituents. Now for its intent:

> *Specify and change product types dynamically without replacing the factory instance.*

Note how this presumes you know something about ABSTRACT FACTORY. It mentions "product types" and "the factory instance" without explanation, under the assumption that you wouldn't be interested in PLUGGABLE FACTORY if you didn't know what they meant. That, in microcosm, is what building on existing patterns is all about.

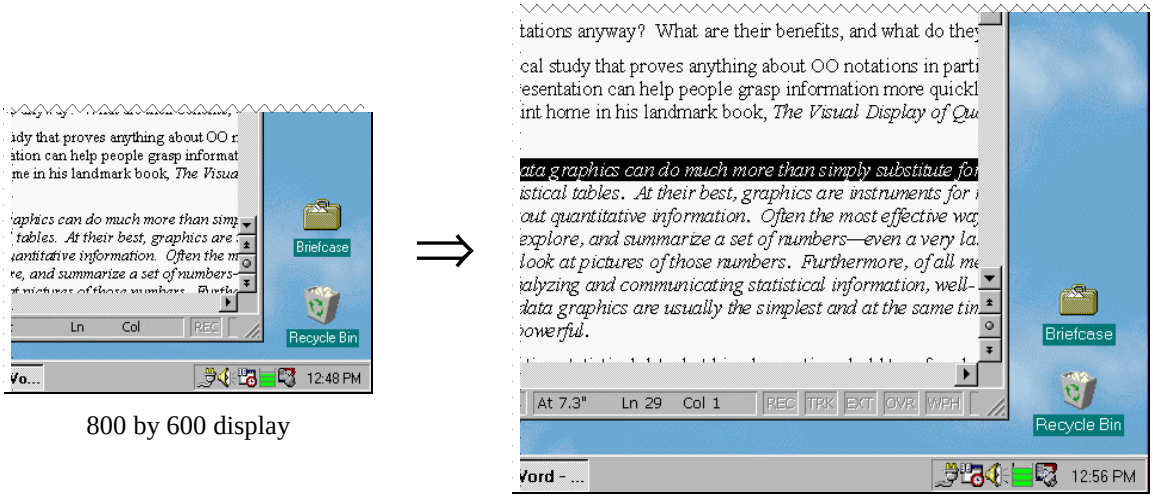
This Intent oozes synergy, too. "Specifying and changing product types dynamically" cuts to the heart of what ABSTRACT FACTORY can accomplish in cooperation with PROTOTYPE and not by itself—not without wholesale replacement of the factory instance. Anyone who's tried that alternative can attest to its draw-

* Please allow me this shorthand. My fingers thank you.

backs, especially the potential for dangling references. PLUGGABLE FACTORY's intent implies a solution to those drawbacks, and its Motivation section reveals the solution in a concrete way.

Motivation

Suppose you're using ABSTRACT FACTORY to provide multiple look-and-feel standards as described in that pattern's Motivation section. Also suppose that applications in your environment should be able to handle changes in screen resolution on-the-fly, as might be required when the user docks or undocks a notebook computer. Undocked, the notebook's display resolution is, say, 800 by 600 pixels, while the monitor attached to the docking station is set for 1280 by 1024 pixels (see Figure 1).



800 by 600 display

1280 by 1024 display

Figure 1

To remain legible, text must be proportionately larger on the monitor than on the notebook. User interface "widgets" that incorporate text, such as buttons and menus, should adopt a larger font, but their look and feel is otherwise unchanged. Similarly, widgets that incorporate a bitmap should use a larger bitmap at higher resolutions. (Compare the icons in the lower right of each display in Figure 1.)

Since widget instances are created using a factory, you could effect these changes in appearance in two steps: Replace the factory instance with one that produces widgets with bigger fonts and bitmaps, and then reconstruct the user interface from scratch. But that's unsatisfactory for two reasons. First, objects that maintain references to the old factory must be updated to refer to the new one. That may be difficult to do, especially if the system was not designed with dynamic updating in mind.

The second reason poses a more fundamental problem. In ABSTRACT FACTORY, you vary the types of products by defining ConcreteFactory subclasses. The pattern's motivating example defines MotifWidgetFactory and PMWidgetFactory in support of two look-and-feel standards. That's fine if you're supporting a small set of standards. Here, however, we are effectively expanding the set of standards when we target multiple display sizes. Clearly we don't want to define a different class of ConcreteFactory for every possible display resolution, any more than we would subclass a button to give it a different text label. Instead, buttons are *parameterized* with the label to display. That approach supports an infinite variety of labels with just one subclass. Parameterization can also let us change the label at run-time without replacing the whole button.

You can use parameterization to lend similar flexibility to ABSTRACT FACTORY. Instead of varying product types by subclassing an AbstractFactory, you define a single ConcreteFactory class—WidgetFactory in this case—and parameterize it with product prototypes that follow the PROTOTYPE pattern. Whenever a client requests a product from the factory, the factory returns a copy of a prototypical instance (Figure 2).

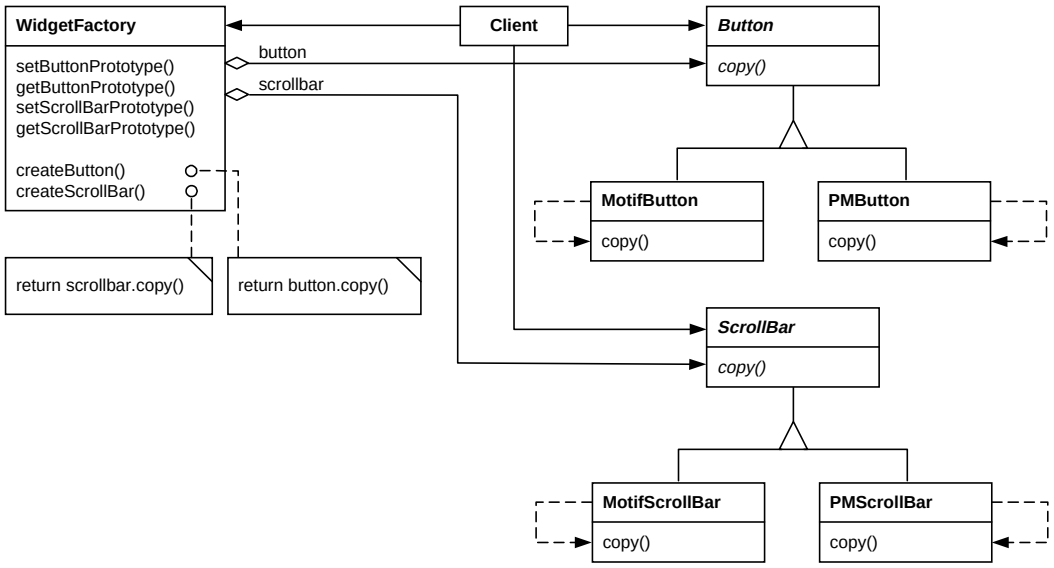


Figure 2

Clients can affect the products that the factory creates in two ways. A client can use a `get` operation to obtain the prototype, which can then be modified directly. For example, an application can change the font of the prototypical button by calling `getButtonPrototype` to obtain the prototype and then calling `setFont` on that prototype. Alternatively, a client can replace a prototype with are more suitable one. To replace the button prototype, just call `setButtonPrototype` with the new button as a parameter. Either way, a pluggable factory gives you finer-grained control over products than a conventional abstract factory.

Applicability

Use PLUGGABLE FACTORY when ABSTRACT FACTORY is applicable *and* any of the following are true:

- Products may vary independently during the factory's lifetime.
- Ad hoc parameterization techniques, such as supplying a class name to a factory operation,[†] are not flexible, expressive, or extensible enough.
- You want to avoid a proliferation of ConcreteFactory subclasses.

More to Come

I'm afraid I'll have to leave this pattern in limbo until next time, space considerations being what they are. I'll pick up with the structure, participants, and collaborations as a review before getting into the real meat of the pattern: the many and varied consequences and implementation issues peculiar to pluggable factories.

Feedback

Referring to my "Notation, Notation, Notation" article,[5] Mark Rodgers writes,[6]

> *One of the key benefits I saw in the UML collaboration notation was that it ties the participant classes together, which had two advantages:*

[†] See item 3 under ABSTRACT FACTORY's Implementation section.

1. *If the same pattern occurs more than once on the same diagram, it is clear in which occurrence a class is participating. "So A is the Subject for Proxy B, and C is the Subject for Proxy D."*
2. *It helps you locate the other participants—"So this is the Proxy. Now where is the Subject?"*

Have you found the lack of these features in Erich's notation (which by the way I do find quite appealing) to be a handicap? Is there a compromise that would get the best of both worlds (e.g., using the UML notation but with a gray oval and thick gray lines)?

Astute and valid points. I can't say as I've ever stumbled across such shortcomings of Erich's notation (shown once again in Figure 3), but then again maybe I haven't used it extensively enough. I can certainly imagine it being difficult to find all the players in a given instance of a pattern.

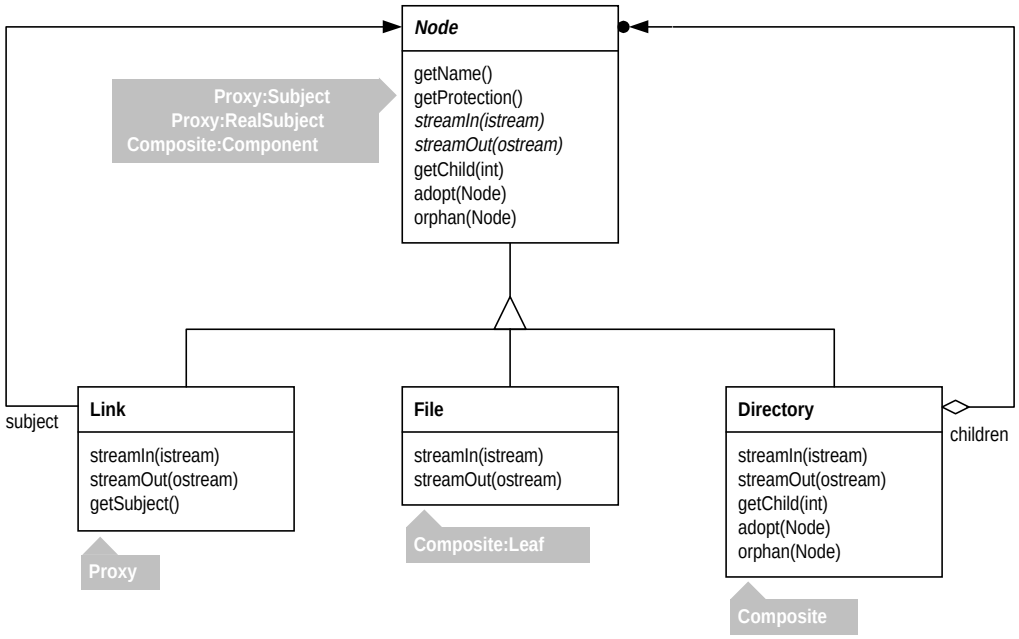


Figure 3

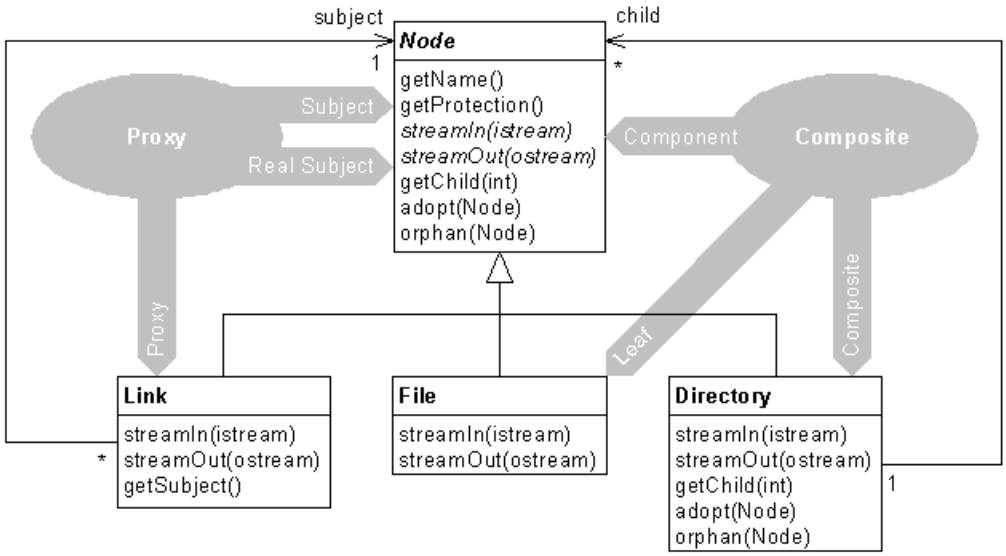


Figure 4

In a later message,[7] Mark offered up the pattern notation in Figure 4 as an alternative. It strikes me as an ingenious hybrid of the UML and Gamma notations. My only reservation is the challenge people will undoubtedly find in drawing such alien-looking gray blobs. What do you think? Is Mark's notation the best of both worlds, or is it too much like something from another world—something best consigned to the X-Files? You decide.

Acknowledgments

Richard Helm offered helpful comments on this article.

[1] Vlissides, J. Composite design patterns (they aren't what you think). *C++ Report*, June 1998, pp. 45–47, 53.
[2] Gamma, E., et al. Design patterns: abstraction and reuse of object-oriented design. *Proceedings of the 7th European Conference on Object-Oriented Programming*, Kaiserslautern, Germany, July 1993, pp. 406–431.
[3] Gamma, E., et al. *Design Patterns*. Addison–Wesley, Reading, MA, 1995.
[4] Vlissides, J. *Pattern Hatching: Design Patterns Applied*. Addison–Wesley, Reading, MA, 1998, pp. 24–27.
[5] Vlissides, J. Notation, notation, notation. *C++ Report*, April 1998, pp. 48–51.
[6] Rodgers, M. E-mail communication, April 14, 1998.
[7] Rodgers, M. E-mail communication, July 22, 1998.