Pattern Hatching

Composite Design Patterns (They Aren't What You Think)

John Vlissides
C++ Report, June 1998

Dirk Riehle presented an interesting paper titled "Composite Design Patterns" at last October's OOPSLA conference in Atlanta.[1] The paper wasn't exactly new to me, as I had reviewed it well before the conference. (I was on the program committee.) But even before that, Dirk and I had discussed its topic on a couple of occasions. Anyhow, I am delighted he wrote that paper, because I think it maps out valuable and largely unexplored territory in the pattern frontier. Dirk has initiated a dialogue, and I can't resist chiming in.

You're probably wondering what the topic is, and the paper's title isn't helping. No, it's not about the COMPOSITE pattern in *Design Patterns*,[2] at least not directly. It has to do with documenting how patterns work together.

It's not uncommon to see the same small set of patterns cooperating again and again in different designs. A Visitor visiting a Composite through an Iterator is one example; a Singleton Mediator is another; a Prototype-based Abstract Factory is yet another. When patterns cooperate, the cooperation itself can give rise to problems, contexts, trade-offs, and consequences. For instance, should a Visitor be a part of an Iterator, or vice versa? Or is neither the case? (They could well be one and the same class.) It all depends on the trade-offs you're willing to make.

Sadly, these issues aren't discussed much in *Design Patterns*. Just where *should* they be discussed? In the VISITOR pattern, in ITERATOR, or both maybe? What about other VISITOR combinations worth documenting—where do they go? Individual patterns are hard-pressed to cover such issues. Indeed, they're hard-pressed to cover their *own* issues. VISITOR is fourteen pages long as it is, and Jim Coplien, for one, thinks that's way too long already.

What Dirk has dubbed "composite design patterns" may be of considerable help here. Composite design patterns (or simply "composite patterns") are themselves patterns in that they name and document a recurring solution to a common problem. The twist is that composite patterns express themselves in terms of other patterns, even other composite patterns. The goal here is to capture synergy between patterns and make it explicit—synergy you'd otherwise have to discern for yourself.

I'll be offering up several composite patterns over the next few columns. I'll describe them informally at first, gradually evolving toward a more structured format, one akin to the pattern template we use in *Design Patterns*.

Before diving in, I should point out one other thing composite design patterns *aren't*: they aren't full-blown pattern languages. It seems the distinction between patterns and pattern languages still confuses people after all these years, despite (or perhaps because of) all that has been written about them. If you're okay with the distinction, feel free to skip the next section.

Patterns versus Pattern Languages

Simply put, a pattern captures a recurring problem-solution pair. It can delve arbitrarily deeply into the problem, the solution, the forces on that solution, its context and consequences, etc.; but at the end of the day, a pattern addresses just one problem. What's more, a pattern cannot be subdivided into finer-grained patterns—not profitably, anyway. Contrast that with a pattern language, which is a collection of patterns

that cooperate to solve a *family* of problems. "Family" implies close relationship. The solutions to these related problems recur and can therefore be expressed as patterns within the language.

If you think individual patterns are helpful, the right pattern language can do far more for you, at least in theory. Unfortunately, many pattern languages have a property that renders them less than helpful. It's analogous to the trouble with many frameworks, should that strike a chord. In a word, it's *ambitiousness*.

The concept of "family" is as important to frameworks as it is to pattern languages, but in a different sense. A framework defines an architecture for a family of *applications*. An accounting framework, for example, defines a common architecture for general ledger, inventory, and other accounting apps. A compiler framework defines an architecture for compilers across languages, machines, and optimization techniques. A graphical editing framework defines an architecture for drawing editors, electronic design aids, project management systems, and other tools for creating drawings and diagrammatic specifications on-screen. A framework expresses such architectures in a very concrete way: as interfaces and/or abstract classes in a programming language, usually with accompanying default implementations.

Why do these or any other applications need an architecture? It's primarily due to size and its attendant cost. Large, complex systems must be scoped out and partitioned in advance. They need a design apart from code, a design within which to understand and reason about the code. A system that's small and cheap to create doesn't justify spending oodles of time architecting it. Any competent programmer can simply hack it out. An architecture is just an expensive diversion unless the system is big or intricate—and if it's either, it's probably both.

Designing a framework is still more expensive, because a framework isn't just any architecture; it's a *reusable* architecture, meaning it has to work across applications. The framework will not succeed unless its architecture draws on experience with those very applications. As Ralph likes to say, you can't *re*use what you haven't used in the first place.

Pattern languages have much the same flavor, and perhaps a stronger one at that. The family of problem-solution pairs that make up a pattern language collectively solve a grander problem. If it's not a sufficiently grand problem, you probably don't need a pattern language to solve it; it would be like using a sledgehammer to kill an ant. But the problem ought to be more than just grand—it must present itself time and again. It must recur. A pattern language should be flexible enough, therefore, to apply in varied settings (as is true of a framework), and that's not bloody likely unless the language draws on real experience. Sheer ambitiousness has made rare commodities of good pattern languages and frameworks alike.

I'm not saying we should give up on pattern languages or frameworks—far from it. But few will argue that real pattern languages aren't a big step up from the patterns in *Design Patterns*. If most pattern writers aren't ready to take a big step, then a baby step should be an attainable and still respectable goal. It would give us a way out of the single-pattern rut, and we'd be that much further toward useful pattern languages. Composite design patterns are one such baby step.

TEMPLATE METHOD-FACTORY METHOD

Here's the simplest composite pattern I know: the combination of TEMPLATE METHOD and FACTORY METHOD. What are the synergies between these patterns? There should be some; otherwise there isn't much point in writing them up in a composite pattern. And there are indeed a few synergies—elementary ones, to be sure, but synergies nonetheless.

Recall that TEMPLATE METHOD lets you separate variant and invariant parts of an operation. The invariant parts are confined to the template method. The template method defers the variant parts to so-called "primitive" operations, which are defined in subclasses. FACTORY METHOD is similar in the sense that it too defers behavior to subclasses. But whereas TEMPLATE METHOD isn't specific about behavior—it's only specific about *responsibility* for the behavior—FACTORY METHOD is highly specific about it: the behavior must culminate in the creation of an object. A factory method always abstracts the instantiation process.

Wherever you find a factory method, a template method can't be far. The reverse is also true. That's hardly surprising, given the patterns' similarities in intent and implementation. Factory methods often serve as

primitive operations to template methods. Less frequently, you'll see a factory method implemented as a template method.

Notice that both patterns have class as opposed to object scope. That means they're generally less flexible than object patterns that do more or less the same thing—like PROTOTYPE in FACTORY METHOD's case, or STRATEGY in TEMPLATE METHOD's. On the other hand, class patterns tend to be simpler and more efficient. The trade-off often boils down to flexibility versus lighter weight. When flexibility isn't an issue, a subsidiary trade-off emerges: lighter weight at compile-time versus run-time.

Both template and factory methods incur relatively little overhead at run-time, since they rely on inheritance rather than delegation for flexibility. The flip side is that you might end up having to subclass solely to override a factory method or the primitive operations. However, one of the synergies of these two patterns is that together, they strengthen the case for subclassing. Neither may justify it alone, but as a team they just might. You might be more willing to abstract the creation process with a factory method when you're already overriding primitive operations. You might also be ready to break up an operation into its primitive constituents when you have to vary what it creates.

Another simple but significant synergy concerns naming. Oddly, *Design Patterns* suggests prefixing factory methods and primitive operations alike with "`do-`" to identify them as such. When using the patterns together, you probably want to distinguish the two. If so, consider prefixing the factory methods with "`make-`" or "`create-`" instead of "`do-`".

PROTOTYPE-ABSTRACT FACTORY

ABSTRACT FACTORY talks briefly about using PROTOTYPE to create products. PROTOTYPE mentions ABSTRACT FACTORY, but only as a competitor. There's a lot more to their confluence than that.

The vanilla implementation of ABSTRACT FACTORY populates the AbstractFactory interface with what amounts to a bunch of factory methods. As a rule, PROTOTYPE will work wherever FACTORY METHOD will, and with more flexibility, but also with higher run-time cost. If you're willing to absorb that cost, you can use PROTOTYPE to reduce the number of classes that ABSTRACT FACTORY introduces.

Figure 1 shows how PROTOTYPE impacts ABSTRACT FACTORY's structure. Note that there's but one ConcreteFactory class, and there's no AbstractFactory at all. Before applying PROTOTYPE, we used AbstractFactory solely to define the "manufacturing interface" for ConcreteFactories, which implemented that interface. The types of products were parameterized by subclassing AbstractFactory. In PROTOTYPE-ABSTRACT FACTORY we use prototypes to parameterize the product types. All we need is a single ConcreteFactory class that we configure with the appropriate prototype objects.

In this case the prototypes are objects of type AbstractProductA and AbstractProductB. These correspond, presumably, to instances of ProductA1 and ProductB1, or ProductA2 and ProductB2. I say "presumably" because, unlike a conventional implementation of ABSTRACT FACTORY, the PROTOTYPE-based implementation allows mixing and matching products from different families—that is, unless deliberate steps are taken to preclude it.

That raises a slew of interesting questions: Who configures ConcreteFactory, and with what prototypes, and when? Is configuration carried out when the ConcreteFactory is instantiated? If so, does its constructor allow the client to supply prototypes of the client's choosing? Is there an interface for setting the prototypes *after* ConcreteFactory instantiation? If so, does that interface include operations for *getting* the prototypes? And are there safeguards against mixing prototypes from different families? How do they work? (Alas, these questions will have to wait till next time.)

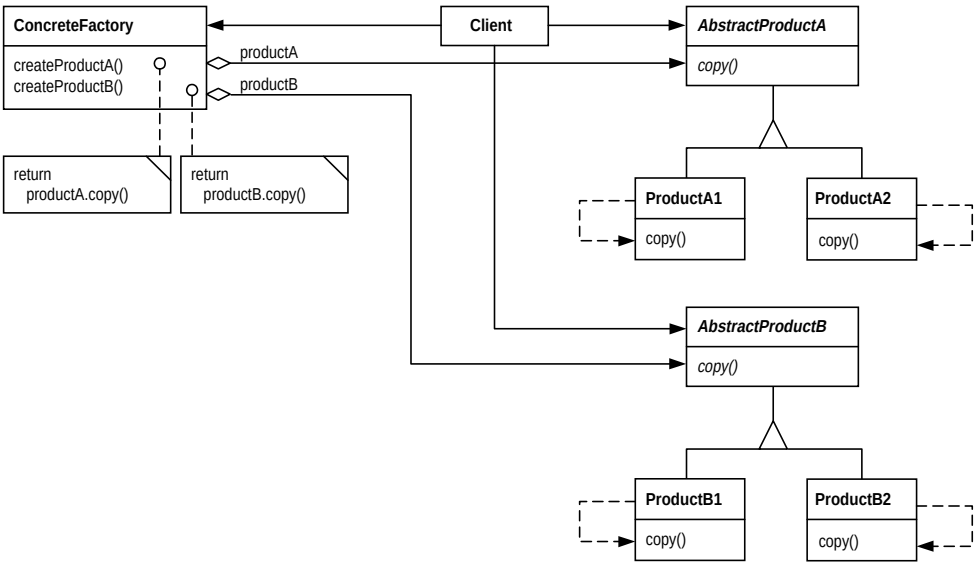

Figure 1:PROTOTYPE-ABSTRACT FACTORY structure

COMPOSITE Composites

The COMPOSITE pattern has a particularly strong affinity for other patterns, and the resulting synergies are invariably rich. Here are sketches of just a few of them:

- COMPOSITE-DECORATOR: These two patterns complement each other as no others. Why, they even share the name of their base class participant—"Component." The same mechanisms for recursive composition that let Composite work its magic do likewise for Decorator. You'll often find the two working together.

  As anyone who has used COMPOSITE can testify, the linchpin of its application is the Component interface. It's key because it defines what you can treat uniformly, which is the very heart of the pattern's intent. The most common quandary in designing the Component interface is whether it includes child manipulation operations like `insert`, `remove`, and their associates. Your decision inevitably rests on a trade-off between uniformity and static type safety. If you opt for uniformity, then you'll declare these (and probably all other) operations in the Component class; otherwise you won't, and you'll suffer the resulting downcasts whenever they're needed.

  The COMPOSITE-DECORATOR composition adds weight to the argument for a uniform interface. DECORATOR works poorly or not at all unless there is a common interface. You can't decorate a component transparently unless both component and decorator share an interface. Yes, it's possible to have several type-specific ConcreteDecorators that cater to different interfaces. But if the ConcreteComponents they decorate have a common base class, as DECORATOR prescribes, then you'll probably have to downcast before you decorate.

- COMPOSITE-FLYWEIGHT: The COMPOSITE pattern can produce a lot of overhead if you apply it at too fine a granularity. The Lexi document processor that's described in Chapter 2 of *Design Patterns* defines Components at the level of individual characters and graphical primitives such as lines and polygons. If a page of a typical book contains 5000 characters, then a 200 page book will allocate one million leaf objects. That doesn't count the internal nodes of the composite, or graphics, or any other objects in the system. Clearly this is not the most efficient way to go.

  Fortunately for us there's FLYWEIGHT. It's applicable here because there's a lot of redundancy in our document structure. There may be a lot of characters, but there aren't many *different* characters. You might see thousands of lowercase "a's", but usually they are exactly the same—same size, same font, same color. That means they can be shared. And you don't have to do a heroic

amount of sharing to get a big reduction in objects. As FLYWEIGHT points out, sharing basic text attributes can let you represent a 180,000-character document with just under 500 objects.

But sharing may introduce a big problem. Many applications of COMPOSITE store a parent link in every component. How do you implement parent links when components may be shared? Typically, you don't; you redesign your protocols to avoid upward traversals. This usually requires rethinking your application of COMPOSITE. It may even lead you away from applying it at so fine a granularity.

Alternatively, components may keep track of multiple parents. When a client asks a component for its parent, the client must supply contextual information so that the component can determine the proper parent for this context. The contextual information can take the form of a location in a virtual tree structure. The component uses this location as a key to look up the corresponding parent. Regrettably, this is seldom a straightforward computation.

- COMPOSITE-ITERATOR-VISITOR: If there's one thing you do with composites, it's traverse them. ITERATOR lets you traverse these structures without regard for how they are linked together. It also lets you reuse common traversals. Meanwhile, VISITOR lets you perform (or not perform) type-specific work at each point in the traversal. The implementation of that work goes into a ConcreteVisitor class, not the Component classes. The separation of concerns thus established promotes extensibility and keeps the Component interface sweet and simple.

  Still, it's entirely possible that your application doesn't need the full generality of separate Iterator and Visitor classes. Clients don't normally use visitors on isolated objects; they use them during traversal, for that's commonly when the concrete types vary in unpredictable ways. Consequently, Visitor and Iterator classes are often lumped into one intrepid workhorse.

  But even if these classes are separate, they are likely to be closely related. What exactly is their relationship? Does the Visitor aggregate an iterator, or vice versa? Certain Visitors might expect certain traversals. In that case, someone must be responsible for coupling visitors with the iterators they require.

More to Come

This is just the start of a start, as you've probably gathered. There are at least as many composite patterns as there are other kinds of patterns. Here are few more that I've noticed and that I'll be talking about:

- COMPOSITE-STRATEGY-OBSERVER, better known as Model-View-Controller, from the Smalltalk world. Buschmann & Co. wrote this up as an architectural pattern[3], but casting it as a composite pattern yields a more compact discussion with little or no disadvantage.
- MEDIATOR-OBSERVER, which is already discussed a bit in OBSERVER, but rather superficially—probably because of that pattern's already considerable girth.
- COMMAND-MEMENTO, another hand-in-glove combination given short shrift in its constituent patterns.

It bears emphasizing that composite design patterns are first and foremost *patterns*. As such they should reflect real usage, not synthetic mind games (although those can be fun, too). As we contemplate these patterns, I'm counting on you to keep me honest. Feel free to send in your known uses of these and any other combinations you've encountered. We've got a lot of new country to explore, and Dirk and I can't do it alone!

References

[1] D. Riehle. “Composite Design Patterns.” In *OOPSLA ’97 Conference Proceedings*, published as *ACM SIGPLAN Notices*, 32(10):218–228, October 1997. ACM Press.

[2] E. Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[3] F. Buschmann, et al. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, Chichester, England, 1996.