IEEE Software

www.computer.org/software

Refreshing Patterns

Rebecca J. Wirfs-Brock

Vol. 23, No. 3
May/June 2006

Editor: Rebecca J. Wirfs-Brock ■ Wirfs-Brock Associates ■ rebecca@wirfs-brock.com

Refreshing Patterns

Rebecca J. Wirfs-Brock

Language is as changeable an entity as cloud formations even in its mundanest, most "vanilla" aspects such as the words dog *or* since.
—*John McWhorter,* The Power of Babel *(W.H. Freeman, 2002)*

When the published form of a pattern seems dated due to new language features or new experiences, it needs refreshing. Unfortunately, just as in spoken languages, once standard definitions are widely circulated in print, it becomes much harder to update them.

When I wrote a prepublication review of *Design Patterns* (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1994), I recommended publishing the book in a loose-leaf notebook to allow for frequent updates and additions. Now, as I write this column in April 2006, *Design Patterns,* despite its age, is still a best-seller, ranking 616 on Amazon.com's book list. However, some find it difficult to read owing to its dated graphical examples and scholarly tone. This is likely why it was recently edged out of its spot as Amazon's top pattern book by *Head First Design Patterns* (Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra, O'Reilly, 2004), whose popularity is based on its goofy examples, humor, and annotated doodles on code. This book makes learning design patterns fun, yet it skimps on presenting all the original patterns in depth. *Design Patterns* is still the most comprehensive source for the original 23 design patterns.

Instead of updating a single source (*Design Patterns*), as I had naively hoped for, pattern authors expanded and grew the literature. Today, over 70 pattern books are in print, covering the gamut from messaging and architecture to domain modeling and code refactorings. Furthermore, if you're looking for new patterns, want to explore patterns in the making, or want to discuss pattern nuances, print publications aren't your best source. You need to look online. The Portland Pattern Repository (http://c2.com/ppr) is the largest pattern Web site with over a thousand patterns—most of which haven't been published elsewhere, because they're not fully cooked. But they aren't half-baked either. This repository is a lively place to explore patterns as they are stewed, skewered, debated, and nuanced. Although some prefer sources of tried-and-true patterns, ready for immediate consumption, if you only look in published pattern books, you miss out on a lot of design ideas and good discussions.

Pattern adaptations

If you're looking for the definitive source for a specific pattern, you're probably not going to find it. One rarely exists. Different authors make different design choices, and their descriptions of the same pattern will vary. John Vlissides, one of the *Design Patterns* authors, advises:

> *You can't overemphasize that a pattern's structure diagram [class 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 im-*

plementation, but differences are inevitable and actually desirable. At the 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.

Although I assign my students the original versions of patterns as presented in *Design Patterns*, I also expose them to other published variations and examples that illustrate adjustments I've made to patterns I've incorporated into my own designs. Reading and studying multiple pattern definitions and examples helps convey the important point that it's quite natural for patterns to wiggle around a bit as they're applied.

Consider the Composite pattern. It supports composing objects into tree structures. In such a structure, an object can either be a container for other objects (playing the role of composite) or not (playing the role of leaf). Figure 1 illustrates the original class diagram for the Composite pattern.

The abstract class named Component provides default implementations for adding, removing, and retrieving children. Leaf and Composite classes are two different kinds of components. If, when implementing the pattern, you strictly follow the composite form outlined in *Design Patterns*, it's tricky to decide what attributes and operations the Component class should define. Default methods are expected to be inherited by different classes of leaf objects. Any Composite class is expected to override the defaults to implement composite-specific behavior.

When *Design Patterns* was published, interfaces weren't a part of any popular programming language. So, the authors relied on abstract class definitions—which opened the door for declaring default method implementations and common attributes. Today, I spend most of my time with C# and Java, both of which support the declaration of interfaces that only define method signatures. In these languages, I prefer to cast many patterns using interfaces in lieu of abstract classes. This is especially true when there isn't reasonable default behavior to shove into an abstract class. *Design Patterns* devotes a couple of pages to discussing meaningful defaults for the Component class, concluding that, "usually it is better to make `add` and `remove` fail by default (perhaps by raising an exception)… or if the argument of `remove` isn't a child of the component."

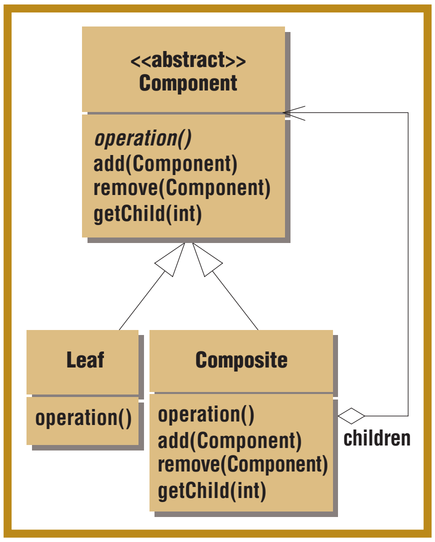

Figure 1. The original Design Patterns Composite pattern form.

Well, maybe. Wouldn't it be simpler if you didn't have to spend time concocting meaningful default behavior? Clearly, a composite has two separate responsibilities—perform specific operations on the basis of its type and manage children. Bob Martin, in *Agile Software Development Principles, Patterns and Practices* (Prentice Hall, 2002), illustrates a variation of the Composite pattern that only defines common operations in an interface (see figure 2).

Unwilling to force-fit child management behavior into a common abstraction, Martin implemented composite-specific behavior only in the CompositeShape class. After struggling to define meaningful operations common to both leaf and composite objects, some of my design students throw up their hands and grumble that the *Design Patterns* authors made the wrong trade-offs.

Recently, I asked Ralph Johnson, a *Design Patterns* coauthor, about this dilemma. He replied, "The book just got that wrong, as I have been saying for 10 years. I argued as much while we were writing the book and lost to the other three. They agree with me now. At the time, most of the C++ frameworks did it like the book said, but Smalltalkers never did it that way, which is why we disagreed. Later, C++ frameworks shifted to putting `add()` in the Composite interface, and of course Java does it that way, too. I'm not sure why people made the change, but I think it has to do with runtime type checking. As long as C++ didn't have a safe way to downcast, it seemed better to make all Components support `add()`."

Clearly, the *Design Patterns* authors had healthy debates while writing their book. For a look into how patterns are conceived, polished, perfected, or perhaps put aside, see John Vlissides' *Pattern Hatching* (Addison-Wesley, 1998).

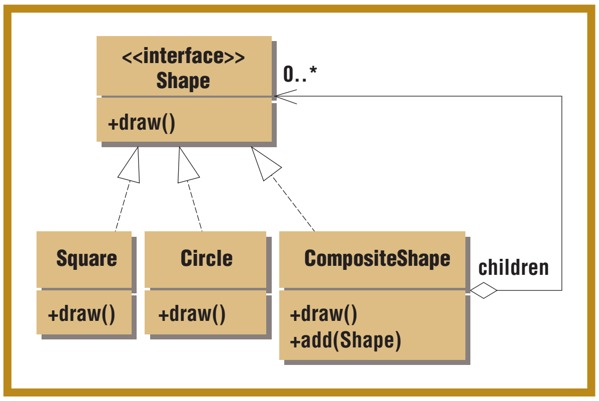

Figure 2. Using an interface to define only type-specific operations.

Pattern morphing

But what makes a form of a particular pattern a "standard" or "preferred" form, and when does an adaptation change a pattern into something else? Patterns aren't just a single structural form. Ideally, a pattern describes a set of roles and responsibilities assigned to one or more classes and offers advice on common adaptations and trade-offs. A pattern's structural form is less important than its intended design purpose. Any time you apply a pattern, you sort through implementation trade-offs and are likely to end up with a solution that differs from any canonical example. In *Refactoring to Patterns* (Addison-Wesley, 2004), Joshua Kerievsky implements the Composite pattern using a single multipurpose class (see figure 3).

This bears little resemblance to the composite structure defined in *Design Patterns*. Kerievsky coded just what was necessary for his application and nothing more. Evolutionary design, which Kerievsky advocates, pushes minimalism to the limits. An evolutionary designer makes successive refinements to working code until it's good enough. Successive refactorings might move code closer to or away from a particular pattern, depending on the current design goal.

In *The Power of Babel*, McWhorter identifies processes that cause spoken languages to change. One process, called *sound change*, occurs when, over time, sounds tend to drop off of words. This is especially common when the accent doesn't fall on a syllable. That's one reason why the Latin word for woman, *femina*, has morphed into the single syllable *femme* in modern French. Kerievsky's implementation of single-class-as-composite-or-leaf has eroded any distinction between the implementation of a leaf and composite role.

But is a single class that supports both leaf and composite behavior still a Composite or something else? I'm guessing that a TagNode object can be transformed into an interior node by simply having a child added, or can change back into a leaf node by having children removed. This isn't the case for Martin's shape example. A circle or square has behavior distinct from a composite shape. There are both structural and behavioral reasons to define different shape classes. If there aren't behavioral differences between terminal and nonterminal nodes, it makes sense to combine these concepts into a single class. But perhaps this implementation is far enough away from the intention of the Composite pattern to be called something else.

Another type of language change is word evolution. Over time, words' meaning can narrow or broaden. In English, *hund* originally referred to any old dog, while *dog* meant a particular breed of large canine. Over time, hund narrowed to mean hound (and its spelling changed), while dog gradually broadened its meaning to refer to all dogs. The name Composite pattern has always slightly confused my students. A composite is an entity made up of distinct parts. Any object playing a composite role will be made up of distinct parts, but leaf objects aren't composites. So naming the pattern for that one role seems a misnomer, especially when today's preferred form for the Composite pattern doesn't define any composite behavior in a shared abstraction. Perhaps it's time to retire the somewhat dated meaning of Composite pattern and rename it Tree instead.

There's always a gap between the spoken word and official standard definitions. Dictionaries, despite being constantly updated, are always out of date. What's disconcerting to an unwary pattern consumer is that shifts in pattern meaning and interpretation aren't collected in one place—we don't yet have an online pattern dictionary. Until there is one, designers exploring a pattern should gather information from various sources and expect "preferred forms" to change and evolve. That's why pattern authors and online pattern communities will continue to grow, adapt, and transform patterns. It's as predictable as a change in the weather.

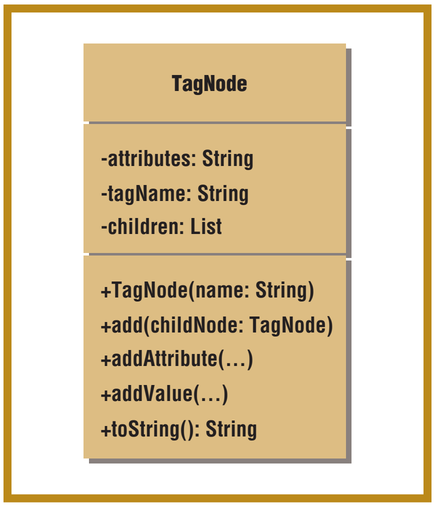

Figure 3. A single class that implements both a leaf and a composite.

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates and an adjunct professor at Oregon Health & Science University. She's also a board member of the Agile Alliance. Contact her at rebecca@wirfs-brock.com.