Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Patterns

Martin Fowler

It should come as no surprise that I'm a big fan of software patterns. After all, I just finished my second book on the subject, and you don't put forth that kind of effort without believing in what you're doing. Working on the book, however, reminded me of many things about software patterns that are not fully understood, so this seemed like a good time to talk about them.

The biggest software patterns community is rooted in the object-oriented world. This community includes the people who wrote the classic Gang of Four book (E. Gamma et al., *Design Patterns*, Addison-Wesley, 1995) and the Hillside group that organized the worldwide PloP (Pattern Languages of Programs) conferences (see http://hillside.net). Most patterns books have come out of this community—but not all. There are good books, such as David Hay's *Data Model Patterns* (Dorset House, 1996), written by people who've had little or no contact with this group.

Even with the Hillside group, there's a lot of disagreement about what's important about patterns—and that mix of views grows even larger with those who don't get involved in the Hillside group's efforts. So there are many opinions on what makes a pattern important. Following are my views.

Why patterns interest me

Patterns provide a mechanism for rendering design advice in a reference format. Software design is a massive topic, and when faced with a design problem, you must be able to focus on something as close to the problem as you can get. It's frustrating to find an explanation of what I need to do buried in a big example that contains 20 things that I don't care about but must understand to complete the thing I do care about.

So for me an important problem is how to talk about design issues in a relatively encapsulated way. Of course it's impossible to ignore all the inter-relationships, but I prefer to minimize them. Patterns can help by trying to identify common solutions to recurring problems. The solution is really what the pattern is, yet the problem is a vital part of the context. You can't really understand the pattern without understanding the problem, and the problem is essential to helping people find a pattern when they need it.

Similarly, abstract discussions of principles—which I often write about in this column—are important, but people need help applying these principles to more concrete problems.

When people write patterns, they typically write them in some standardized format—as befits a reference. However, there's no agreement as to what sections are needed because every author has his or her own ideas. For me, there are two vital components: the how and the when. The how part is obvious—how do you implement the pattern? The when part often gets lost. One of the most useful things I do when understanding a pattern, one I'm either writing or reading, is ask, "When would I *not* use this pattern?" Design is all about choices and trade-offs; consequently, there usually isn't one design ap-

 0740-7459/03/$17.00 © 2003 IEEE

proach that's always the right one to use. Any pattern should discuss alternatives and when to use them rather than the pattern you're considering.

One of the hardest things to explain is the relationship between patterns and examples. Most pattern descriptions include sample code examples, but it's important to realize that the code examples aren't the pattern. People considering building patterns tools often miss this. Patterns are more than macro expansions—every time you see a pattern used, it looks a little different. I often say that patterns are half-baked—meaning you always have to finish them yourself and adapt them to your own environment. Indeed, implementing a pattern yourself for a particular purpose is one of the best ways to learn about it.

Patterns and libraries

This raises an interesting point—if a pattern is a common solution, why not just embed it into libraries or a language? Then people wouldn't need to know the pattern. Certainly, embedding patterns into libraries is common—indeed, usually, it's the other way around. Pattern authors see many libraries doing a similar thing and consequently identify the pattern. This provides value in several ways.

First of all, it can help people understand how the library feature works to extract the library's specific context. A library typically combines many things at once, so again you run into the problem of having to understand a dozen things. A well-written set of patterns can help explain these concepts.

Second, people often move between programming environments, so they might be familiar with a particular solution but not how to implement it in a new environment. Understanding the underlying pattern behind library features helps a great deal in making this connection.

Finally, even if a library implements a pattern, you must still decide how to use it. You might also need to know more about what implementation strategies the library uses and whether they are appropriate for a particular problem. A library can only implement the "how" part of a pattern—you still have to answer the "when." In this case, the presence of a library implementation alters the trade-off. If a library implements a solution that isn't ideal, you can still choose to use it rather than implement the ideal one yourself.

Patterns and the expert

One of the hardest things about patterns is that they are, by definition, nothing new. If you've been working in a field for a while and have become very skilled in it, then a patterns book in that field shouldn't teach you anything new. Are patterns at all valuable to experts? They obviously offer less value to experts than to someone coming to grips with the field, but there is still something to be gained by looking at patterns that merely capture what you already know.

The primary value is that they can help you teach those around you. For me, the driving force behind writing comes from seeing that there's knowledge to be shared. An expert in a team can use written patterns to help educate other team members. The expert can help the team review the general case, which will come with simplified and encapsulated examples, but more importantly, he or she can then show the team how the patterns should or shouldn't be used in the project at hand. So if you're an expert in your field, you might rate the quality of a patterns book based on how it helps you teach your colleagues rather than on what you learned from it.

The other value of patterns to experts is as a standard vocabulary. In the OO world, we can talk about singletons, strategies, decorators, and proxies, confident that a moderately experienced designer will have a good chance of understanding what we mean without a lot of extra explanation. This vocabulary makes it easier to discuss our designs.

Overuse

One of the things that can be a problem is that people can think that patterns are unreservedly good and that a test of a program's quality is how many patterns are in it. This leads to the apocryphal story of a hello world application with three decorators, two strategies, and a singleton in a pear tree. Patterns are not good or bad—rather, they're either appropriate or not for some situations. I don't think it's wrong to experiment with using a pattern when you're unsure, but you should be prepared to rip it out if it doesn't contribute enough.

I often say that patterns are half-baked—meaning you always have to finish them yourself and adapt them to your own environment.

I don't subscribe to the opinion that there are few remaining patterns to gather. In both of my patterns books, I felt I did little more than scratch the surface. There is more room for people to look at the systems that were built and try to identify and describe the patterns involved. This strikes me as an ideal task for academia, although, sadly, the fact that patterns are by definition nothing new seems to make that impossible.

As a field we have much to learn, which is why software development is so much fun. But I think it's frustrating when we don't take the time to learn from our own efforts

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org.