Proceedings of Object Expo Europe

London, England, September 26-30, 1994

AT&T Bell Laboratories

Progress on Patterns: Highlights of PLoP/94

James O. Coplien
Software Production Research Dept., AT&T Bell Laboratories
1000 East Warrenville Rd., Naperville, IL 60566 USA
cope@research.att.com

Abstract

Software patterns support an emerging family of software design techniques. At the first annual conference on Pattern Languages of Programming, the pattern community refined its understanding of what makes a good software design pattern. Attendees explored and refined patterns in writers' workshops, using techniques borrowed from literature and poetry. Many different pattern forms were found to be work well, though most exhibited the basics of the Alexandrian form. There was a disproportionate representation of patterns for distributed processing, client/server architectures, and finite state machines. These are all maturing domains, for which patterns capture established good practice. Good patterns aren't necessarily inventive, but capture hard-won experience in software domains of broad interest.

1. Introduction

Patterns have long been viewed as an important part of how humans understand and shape the world around them. Leading software designers throughout the industry have recently recognized the value of patterns in software design as well. Mature designers are able to see the broad patterns in the software architectures they work with. Patterns are broader and more abstract than paradigms because they look beyond the pieces of a system, to how the pieces fit together. Procedures, objects, and the abstractions of other paradigms provide the building blocks: patterns provide the overall architecture. A generative software design pattern can be used in the same way we use a dress pattern or a boat pattern: to guide a designer to build a new dress or boat. We can weave patterns together into generative pattern languages, families of related patterns which together provide a solution set for system-building problems in a single domain.

Designers have discovered strong parallels between software architecture patterns, and the building architectural patterns of Christopher Alexander.[1] Alexander's works have been a rallying point for the software patterns movement. The movement had its start in grass-roots activities worldwide[2], which reached critical mass at OOPSLA workshops convened by

1. Alexander, Christopher. *The Timeless Way of Building*. New York: Oxford University Press, ©1979.
2. Coad, Peter. "Object-oriented patterns". *Communications of the ACM 35*, 9 (Sept. 1992), pp. 152-159.

AT&T Bell Laboratories

Bruce Anderson, Peter Coad et al. An extended subset of the workshop attendees came together to form the Hillside Generative Patterns Group, a consortium of software designers from across the industry who are evaluating patterns as an answer to the question, “What comes after objects?”

One goal of the Hillside group is to contribute to a better understanding of how patterns can and should be used to support software development. To engage the industry in this endeavor, the Hillside Group organized a conference called Pattern Languages of Programming, or PLoP. The conference was held in Monticello, Illinois, in the United States, in early August 1994. PLoP contributed heavily to the body of knowledge about patterns and their practice in software development. This paper summarizes highlights of what we learned at PLoP, and where patterns might unfold in the future.

This is the first conference I’ve been to in recent memory where there were no overhead slides, no microphones, and no boring keynote address. The food was good and the surroundings superb; even the weather cooperated. The focus wasn’t on sales, marketing, and posturing, but on sharing, learning, and firming up the foundations for a fledgling technique. As Paul Chisholm noted, PLoP was a conference where the “Smalltalk lions lay down with the C++ lambs.”[3] The conference even had an ethics session, run by Norm Kerth. Much of the session focused on how we could guard against inflated industry expectations for patterns. There was a predominant view that it was either too late or futile to contain pattern hype in the industry. The consciousness of how patterns might fail, or how they might succeed in the worst possible way, was on the minds of the attendees. We exhorted each other to lead by example, by developing good patterns and publishing them when opportunities arise.

3. Or was it the other way around?

The conference format was unusual, more similar in structure to a workshop than a conference. Most of the work happened in writers’ workshops. The conference also featured “home groups” which met every day to reflect on what we had learned. There was little time for leisure: this was an intellectually intense, hard-working conference. We probably didn’t take enough time to introspect, but the home groups were a source of powerful insights.

2. Workers’ Write Shops

Review meetings are an important activity of most software design methods. Most reviews focus on a structured document—not as a piece of literature, but as an exposition of the technical details of the document’s subject. At the Hillside Group workshop in early April, we experimented with a review format called writers’ workshops. Richard Gabriel introduced this activity to the pattern community, drawing on his experience from conferences that review English literature such as poetry. Software should be viewed as literature, too, as should the documentation that guides practitioners in its creation and use. The term “pattern” in software refers both to the thing that lives in the code of a well-written program, and the instructions that help us understand or build that thing. Gabriel felt we should review software literature with care and compassion comparable to that used for English literature, such as poetry, that is built with closely guarded construction. It worked well at the Hillside workshop, and we found that it worked well at PLoP.

Writers’ workshops follow a format designed to help authors improve their works. First, the author reads a selection from his or her work. Next, a workshop participant is asked to briefly summarize the entire pattern or pattern language. The group then focuses on what works in the pattern: What should the author retain on the next iteration? The group may then offer suggestions to the author on how to improve the pattern, its form, or the design itself. Last, the author may ask questions of the gallery to clarify remarks made during the review. We allotted about an hour for each work reviewed.

These writer's workshops (wryly coined "workers' write shops" by Panu Viljamaa of Telecom Finland) formed the bulk of the PLoP agenda. Pattern writers critiqued each others' works, in front of an audience comprising non-authors. Non-authors learned about patterns and the pattern form even as passive observers. The review process helped us hone our understanding of patterns and of the pattern form.

3. Celebrating the Pattern Form

What are the criteria for a good pattern? A good pattern helps, guides, or instructs an inexpert practitioner through a problem-solving process. Alexander invented an effective pattern form, which provides the problem solver with a problem definition, a context, a description of the forces at work in the problem, the solution itself, and a rationale explaining the solution and how it resolves the forces. Anyone who has tried can attest that it takes work to shape a problem-solution pair into Alexandrian form. The mental gymnastics of the form help writers refine their understanding and expression of the problem, as the forms of poetry bring focus and conciseness to the poet's expression.

The focus of PLoP/94 was "to celebrate the pattern form." What is magic about the Alexandrian form? Our experience at PLoP suggests that many forms work. While any single pattern language should adhere to a consistent form, different forms may work for different application domains. Source code is more important to some patterns than others. Ward Cunningham started with one pattern (*Half Object Protocol*) and re-cast it into several different forms used by pattern writers at the conference. Not only did he find it easy to translate between forms, but he found that the alternative forms worked equally well.

It was a mild surprise to me that we would "debunk" the sacred Alexandrian form. What, then, is the essence of a pattern that transcends these forms? Our home group explored what had worked well in the patterns we reviewed in the workers' write shops, and came up with the following list:

- The essential Alexandrian elements must still be there, in spite of the form used. We liked patterns that clearly defined the problem and the forces at work.
- A good pattern should start with an abstract, the written equivalent of the oral summary of the workers' write shops. Many patterns go on for four or five pages, too big for readers to wrap their mind around. I found that I could understand most of what was important in a pattern by listening to a 30-second summary. A full 5-minute reading conveys details important to application of the pattern. But the overview, captured as an abstract, is usually enough to tell me whether a pattern is relevant to a problem I seek to solve.
- The solution part of a good pattern has a stratified structure. The solution should start with a general problem description, fortify the presentation with diagrams, and optionally provide code examples. This is the newspaper article style: the further you read, the more detail unfolds. Dennis DeBruler noted that a good pattern with three different examples (e.g., one for each of three popular programming languages) is an ideal teaching tool.

  Patterns start *general*, not *abstract*. We noted that patterns should *have* an abstract up front, not *be* abstract up front. For example, consider the pattern Three Level FSM, presented at PLoP by Robert C. Martin as part of the pattern language, "Discovering Patterns in Existing Applications." It starts with a two-line statement of the pattern's intent. It deepens the intent in a "Motivation" section, that helps readers know whether they should read further. This is followed by a full description of the solution. The solution is supported with a Booch diagram, and followed by several pages of annotated C++ code. If there are multiple examples (e.g., in other programming languages), the generality of the pattern can be preserved without making the pattern uselessly abstract.

- When a pattern involves a lot of collaboration between objects, it is good to have an event diagram. Some of the most subtle structures in a pattern—often the most important ones—capture dynamic object relationships. We noticed that most of the patterns at PLoP failed to capture the essence of pattern dynamics. Most patterns include OMT diagrams or other notations that convey static structure. Dynamic behavior is difficult to capture, but patterns that attempt to do so (using prose or timing diagrams) seem to work better than those that do not.
- Pattern languages work better if they have an index up front. We were particularly attracted to diagrams that laid out pattern dependencies as a graph.
- A pattern should state what its writer assumes about the reader. Patterns must be written with a customer in mind. Given the wide range of expertise and experience in software, it is unlikely that a single pattern can be equally accessible to all audiences. Characterizing the target audience for a pattern improves the chances that it will hit its mark. This will become more important if and when patterns see broader applicability outside the object community.
- The pattern form should be consistent within a single pattern language; this is far more important than following any particular form.
- It is useful to call a given pattern by several different names. Synonyms can be used to highlight aspects of a pattern that are relevant to different domains. For example, Doug Schmidt called his pattern *Reactor*, with *Dispatcher* as a permissible alternative. Nicknames (short or cute names) are also useful, to avoid invoking long descriptive names during informal conversation.

We also agreed that it is important to choose pattern names, synonyms, and nicknames carefully. The nature of pattern names may vary even within a single pattern language by a single author. In Martin's pattern language, some pattern names hint at the problem they solve (e.g., *Rungs of a Dual Hierarchy*, which are problematic in C++), the nature of the solution or result they generate (e.g., *Write a Loop Once*), or the dominant construct in the solution (e.g., *Iterable Container*). There was no consensus on a single best form. The choice of forms should nonetheless be carefully considered—particularly with regard to prescriptive or descriptive emphasis—depending on whether a pattern will be used in a solution catalog or in tutorial material.

4. Brokers, Brokers, Everywhere

I was amazed how many patterns solved related problems. Most patterns broke little new ground, but codified successful practice. While some patterns were technically innovative in their own right, many just did a good job of capturing and expressing successful design techniques—albeit in creative and useful ways. It was no surprise that most patterns were rooted in objects. What was a surprise was that at least five submissions addressed problems in multi-processing or distributed processing systems. At least three patterns or pattern languages contained or featured variations of broker architectures. Multiple patterns appeared for state machines as well. Most PLoP patterns cast their problems and solutions in object-oriented terminology.

Heinz Zuellighoven of the University of Hamburg noted that patterns are different than most advances in computer science of the past couple of decades. Most techniques struggle with first principles to establish new ways of doing things, leaving old ways behind. The patterns movement comes at a time when computer science and software engineering are perched on a mound of carcasses of techniques that no longer are in vogue: process-and-data, finite state machines, and top-down design. One can even find a few people today—including some in the trade press—who believe that the object paradigm is or should be on the heap.

As Dennis DeBruler noted at PLoP, those of us who have been around a while know that new technologies come, grow, embarrass themselves, and give way to their successors. Successor paradigms gain acceptance by differentiating themselves from their predecessors. This differentiation usually falls along old-is-bad versus new-is-good lines. At the advent of a new paradigm, its shortcomings aren't yet known, and the strengths of the old paradigms are forgotten. In short, our industry value system places a higher value on originality than on learning. As Hegel noted, history teaches us that man learns nothing from history.

Patterns break this "pattern" of behavior in emerging software development techniques by adopting an "aggressive disregard for originality," to use Brian Foote's words. We should take advantage of the historical perspective available to us and capture elements of past paradigms that work well for us. In a nutshell, that's what patterns are about.

5. Half Object + Protocol

One of my favorite patterns at the conference was *Half Object + Protocol*, submitted by Gerard Meszaros of Bell Northern Research. It was one of the many patterns solving distributed processing problems. Here is Gerard's statement of the problem, from the proceedings:[4]

> **Sometimes, Objects must appear in more than one computing context (i.e. address space). How can we make the difference between one and multiple address spaces (e.g. single vs. distributed processing) transparent?**

The pattern deals with many forces, from complexity to cost and performance. Gerard's exposition of the pattern includes pictures and a convincing but tight rationale. The solution part of the pattern is:

> **Divide the object into two interdependent half objects (one in each address space) with a protocol between them. In each address space, implement whatever functionality is required to interact efficiently with the other objects in that address space (this may result in duplicated functionality, i.e. implemented in both address spaces). Define the protocol between the two half-objects [for] . . . the essential information that needs to be passed between the address spaces.**

This is a concise yet powerful pattern. Complete with pictures, a list of examples, and a cross-reference to related patterns, it consumed a little more than three pages of the PLoP proceedings.

6. Do patterns work?

At each workers' write shop, a core of reviewers had reactions like, "Wow! I'd really like to try this" or, more commonly, "This really captures what's worked for me before." Doug Lea noted that Doug Schmidt's *Reactor* pattern rose to levels Schmidt's prior articles had failed to reach. In some sessions, reviewers got carried away "gushing," reinforcing what others had already said—something one is supposed to avoid in these workshops. Gushing aside, patterns seem to work well. They give stature to a level of design that is subdued or lost under most contemporary methods.

---

4. The PLoP "Kinko Proceedings", August, 1994.

An ideal pattern conveys expert knowledge to inexpert developers. Most PLoP attendees shared an object-oriented world-view, though their backgrounds were as diverse as CLOS, Smalltalk, and C++. Is this diversity evidence of the broad applicability of patterns? Or does the common theme of objects suggest that the patterns presented at PLoP wouldn't play well outside the object community? While PLoP was a good experiment to gauge the effectiveness of patterns in the object community—with promising results—it's hard to measure whether PLoP attendees were a typical cross-section of the software development community. Time will tell how broadly patterns will apply outside the object community where they have taken root.

Jim Coplien is a Member of the Software Production Research Department at AT&T Bell Laboratories in Naperville, Illinois, and is a member of the Hillside Generative Patterns group. He can be reached at cope@research.att.com. To subscribe to the patterns electronic mail discussion group, send mail to listserver@cs.uiuc.edu with a message body of "subscribe patterns-discussion John Doe", where you should of course substitute your name for 'John Doe'.