A Report on PLoP'94

The first conference on Pattern Languages of Programs was held August 4-6 at Allerton Park in Monticello, Illinois. The goal of the conference was to expand the literature on patterns of programs and to explore the best form for describing software patterns. It succeeded, producing thirty papers on a wide range of topics and using a wide range of styles. Twelve of the papers described pattern languages, while the others presented a single pattern in detail, described a set of related patterns without trying to structure them into a language, analyzed some existing patterns, or discussed some other aspect of patterns.

One of the papers I found most interesting was by Kirk Wolf and Chamond Liu. It described how to build a framework for workstation clients communicating with legacy database systems. Lots of people are building applications like this, which accounts for a lot of my interest. But I learned something about the pattern form from the paper, too.

The pattern language says that the main jobs of a client are searching for business objects, viewing them, and updating and creating them. It says to implement search with a Search object that represents both a search operation and its status. Search objects use mega-scrolling -- they only return small collections of objects and so must often be repeatedly invoked until the entire result is received. It says to ensure loose coupling between business objects and their views by using the Factory Method pattern and the Observer (also called Dependency pattern. It says to avoid the time and screen real estate expense of creating new windows by using a singleton "window keeper" to keep track of windows that have been already created. After explaining how data is exchanged between the database and the client using Proxy, Requests as Objects, and Objects from Records, the pattern language says to use the Observer pattern again to propagate updates from the client to the server.

I'm always interested in comparing other people's patterns with the ones I have found. Some of the patterns in the paper, such as Proxy, Observer, Singleton, and Factory Method, are in the design pattern catalog. Others are special cases of patterns in the catalog. Others are completely different, or are higher-level patterns than those in the catalog.

The Search object is a special case of the Command pattern. A Command is an object that represents an invocation of an operation. Before a Command is invoked, a program can modify its paramters. It can be passed as an argument to the point where it is finally invoked, and it is often saved after it is invoked in case it needs to be undone. Commands are often used to represent operations in menus or in key maps.

A Search object defines the criteria for a search. Although it makes no sense to undo a search, a Search object is often created in one place and invoked in another. Search objects can be reused on different Files by passing them to those files. Searches can return a large number of objects. To keep from running out of memory, Search objects store the state of a search and let you retrieve the results a little at a time. It is unusual for a Command to be reused with different parameters or to return its results incrementally, so a Search object is not the most typical example of Command, but it seems to me to fit, since it represents the invocation of a search operation on a database.

The Exception pattern from Wolf and Liu's paper is common, and would be a good candidate for the design pattern catalog. The Objects from Records pattern describes how to map data from the client to the host, and so is more specialized than the patterns in the catalog. However, it is about the same level of abstraction as those patterns. They are all

"mechanisms", using Booch's definition. Other patterns, such as the Viewing pattern, are both more specialized and more complex, and are at a different level of abstraction.

I learned several things about patterns from this paper. Almost all the papers presented a pattern language top-down. This has the advantage that early patterns set the context for later ones, but has the disadvantage that there are many forward references. This paper presented the patterns bottom-up, and I decided I liked top-down better. Context is more important than closure. If patterns have good names, I don't mind forward references, but I always want to know why a pattern is relevant, which means knowing its context.

The second thing I learned was the difference between a framework and a pattern language. Patterns and frameworks are obviously related, since they are both ways of conveying design information and are reusable. I usually say that frameworks are bigger, more domain specific, more complex, and more concrete than patterns, but this s not very precise. A better distinction might be that a framework *is* a design, while a pattern language tells you how to make one. A framework is like the standard house plans that many builders use. You can move some rooms around, customize the kitchen and put on a patio, but the basic design is fixed. A pattern language says what an architect will do when you tell him that you have eleven kids and your mother-in-law lives with you.

Given a choice between a framework and a pattern language, you should probably pick a framework. Frameworks provide more reuse and require less work to use. Pattern languages teach you how to do something, frameworks do it for you. On the other hand, pattern languages are more adaptible than frameworks. Frameworks are written in particular languages, and often depend on a particular operating system. Moreover, frameworks don't solve all your problems. Sometimes you have to understand why the framework was designed the way it is, and you have to learn how to use the framework before it does you any good. Patterns can help you learn both how to use a framework and why it was designed the way it is.

Wolf and Liu's paper really describes a framework, not a pattern language. It describes the framework in terms of patterns, it describes a bunch of new patterns, and it uses the pattern form, so it looks a lot like a pattern language. But it tells you exactly how many objects to have and how they interact. It doesn't provide choices, but always follows the same path. The paper probably should have followed the form of "Patterns Generate Architectures" from ECOOP'94.

Ward Cunningham wrote a pattern language called "Checks" that is about checking for errors in input data. Checks ensures data integrity while minimizing the number of checks in a program. Putting lots of checks in a program makes it large and inflexible, but not checking input data will corrupt your database. The pattern language resolves these opposing forces.

Checks starts with a fundamental pattern of object-oriented design, Whole Value. This pattern says to represent the meaningful quantities of your domain with special classes. In other words, don't represent money with integers, use a Money class. Don't use a string to represent a telephone number unless you are never going to do anything except display it. Interface widgets should produce and accept these values directly, and not just edit numbers and strings.

Although Whole Value is more fundamental than checking put data, it is crucial to data integrity. All the other patterns in Checks build on it. The next two patterns give more information about these values and how they should behave. Exceptional Value says that each kind of data should allow exceptional values to represent errors. Interface widgets

proudce exceptional values on blank input and should produce blank output when given an exceptional value. Meaningless Behavior says that you shouldn't try to write methods that work under all conditions. Instead, meaningless behavior should result in exceptional values or should raise an exception, and the interface widgets should be expected to recover.

About half the patterns in Checks are about user interface design. Only one (Deferred Validation) is entirely about checking data. The strategy of the pattern language is to design a user interface in which most errors are detected automatically by the user, reducing the need for a lot of error checking. Deferred Validation says that detailed validation of a domain model should be deferrred until an action is requested. This might be when a transaction is posted, or it might be when the user asks "is this OK?", but it will always be at a time when e user will expect it. The pattern language sets the context for Deferred Validation, making it the only part of the program with explicit checks.

The patterns in Checks are very different from the patterns in the design pattern catalog, most of which are what Booch calls "mechanisms". They are patterns of relationships between objects. But Checks is more about the design of a user interface than about the design of objects that implement it. I like Checks partly because it is so different (and thus shows the power of the pattern form) and partly because it is so useful. Error-checking is an important part of most designs, but one that is usually ignored.

There were several more pattern languages that were not about design patterns, including one by Bruce Whitenack on requirements analysis and one by Norm Kerth on transforming analysis to design. Perhaps the most far-out (and the most intriguing!) was written by Jim Coplien. To most people, Jim is known as the author of *Advanced C++ Programming*, but to the patterns community he is known as the "process guy". He has been using object-oriented analysis techniques to study the roles people play in software development teams. He has found some patterns that typify productive teams. He combined those patterns (as well as a lot of well-known patterns) to form a pattern language for a development process. Some of these patterns are a little like object-oriented design patterns. For example, Gatekeeper says that one of the project members should disseminate leading-edge information from outside the project to project members. Firewall says that the job of the manager is to shield other development personnel from being bothered by funders, customers, and other outsiders. These patterns are a lot like the Facade pattern, in which an object acts as the interface to a subsystem or framework. But most of the patterns in his language are not like any object-oriented design patterns. There are patterns about who writes documentation, who can be an architect, and how information flows from the customer to the developer. In contrast to most of what is written about the software development process, I found that Jim's paper told me what to do, and it made sense. That doesn't mean that it is right, of course. But it is at least testable, and as we use it we can find out what works and what doesn't.