

# Design Patterns: Elements of Reusable Architectures

by  
Linda Rising

## Abstract

The notion of a pattern is based on the work of Christopher Alexander, a building architect, and his attempt to capture solutions to recurring problems. The extension of this idea to software provides a new look at reuse and the power of improved communication through a well-defined vocabulary employing successful solutions to recurring design problems.

## Introduction

In the late 1970's, two books appeared, written by Christopher Alexander and his colleagues, who are building architects. These books; "The Timeless Way of Building" [1] and "A Pattern Language" [2], presented the author's view of the recurring problems he saw in building cities and towns, neighborhoods and buildings. Alexander describes these problems and their solutions using an expression he called a pattern[2]:

"Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice."

It's a sad commentary on the field of software engineering that papers written over twenty years ago and papers written yesterday typically both begin by lamenting the fact that software products are over budget, delivered late or not at all, with serious concerns about their quality and reliability. Unfortunately, software engineering gets little help from its underlying discipline, computer science. One common response to this dilemma is to look carefully at other, more mature engineering fields for guidance. We look at hardware engineering, manufacturing technologies, the list goes on.

After examining the field of building architecture and the work of Christopher Alexander, investigators have been observing similar recurring problems and solutions in software. There is clear evidence of patterns in all levels of software design, from high-level architecture down to detailed design.

## Pattern Definition

A pattern is, on the surface, simply another form of documentation. Authors Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides -- the Gang of Four or GOF -- have created a catalog [3] by documenting patterns they have observed

across many successful software products. Experienced designers read these patterns and usually remark, "Sure, I've done that - many times!" The power of this kind of documentation is that knowledge, previously found only in the heads of experienced developers, is captured in an accepted form that can be shared.

Patterns are not theoretical constructs, created in an ivory tower, they are artifacts that have been discovered in more than one existing system. Patterns are successful solutions to recurring problems. The approach calls to mind the notions of cohesion and coupling, developed by Yourdon and Constantine [4] during many late afternoon post mortem sessions, examining lessons learned from past projects. The cohesion and coupling ideas were attempts to capture observed qualities of real systems.

There are several pattern forms that share some common elements:

**Name.** A word or short, meaningful phrase to describe the pattern.

Finding a good name for a pattern is not easy. The name is extremely important, since it is used to reduce communication overhead. The pattern name, like the name of an algorithm or data structure, carries a lot of information. For example, Binary Search, carries information about the appropriateness of its use (only for sorted arrays) and performance (faster than a linear search). This information is assumed by all developers. The implementation of a Binary Search might not be readily recalled by everyone but everyone should know where to find it.

So it is with a pattern. In a design discussion, someone might suggest the use of a Visitor. Everyone would have to recognize whether or not the choice was appropriate, given the inheritance hierarchies involved. The actual implementation might not be trivial and some help from a reference might be required, just as for Binary Search. The use of the name, in both cases, enables discussion at a higher-level of abstraction and a better consideration of trade-offs to produce the best design.

**Problem.** The specific problem to be solved.

**Context.** When to apply the pattern.

A pattern describes a solution to a problem in a context [5], so there must be a description of the problem. The context determines the impact of the forces for a particular setting of the problem. I've been a serious bicyclist for over a decade, so I've developed several patterns of behavior to help me solve problems I encounter on the road. One pattern might be called Tuck and Blast and is used when sighting a large animal (usually a dog) running out of a side yard, barking and showing his teeth. The essence of the solution is: Tuck your body in to be as streamlined as possible and pedal as fast as you can!

The pattern will only solve this problem if the context is appropriate. If I'm in a long paceline then perhaps trying to find shelter behind some larger, more muscular guys who can yell back at the dog might be a better way to solve the problem. If I'm completely out of energy, or if the traffic is heavy, stopping and getting off my bike could be easier than trying to outrun the dog. If the owner is around, simply yelling for assistance might be the best solution.

Some of the forces are:

1. 1. Presence and size of riding companions
2. Amount of traffic
3. Energy reserve of rider
4. Size and aggressiveness of animal
5. Owner of animal in view

These forces must be carefully considered instead of blindly applying the pattern. The importance of the forces varies depending on the context.

**Solution.** The components, their relationships and responsibilities - but no implementation.

The wonderful thing about the use of a pattern is that the implementation is not specified. Just as software design is separate from details such as programming language, operating system, or hardware considerations, so are patterns removed from particular settings, and can be used over and over.

## **Patterns and Object Technology**

Since most of the activity in the patterns community has been tied to object-oriented development, most of the publications and pattern discoveries have been directed toward solving problems in that paradigm. However, the notion of patterns is not tied to any methodology or language. Jim Coplien and Bob Hanmer [6] have been patterns mining the 4ESS, one of AT&T's large telecommunications switches. These patterns are certainly not object-oriented. The same can be said for the patterns we are discovering on the GTD-5®, the large switch we develop for GTE and independent telephone companies.

There is a body of patterns literature [5] concerned with organizational and process patterns also being crafted at AT&T. We are attempting to apply this knowledge in our move toward self-directed teams.

## **Patterns and Reuse**

Mature engineering disciplines have handbooks of solutions to recurring problems. For more on the use of a handbook, please read an interesting article on reuse comparing software engineering to chemical engineering [7].

There has been a sense in the software engineering community that we are continually reinventing the wheel. This phenomenon can be clearly seen in any large company, where projects go their own way, solving problems that are similar or identical to problems being solved by other projects, sometimes down adjacent corridors in the same building. It's easy to point at a lack of communication and the classic not invented here syndrome but the fact is, there hasn't been an appropriate communication medium for transferring this knowledge. The widely applied code libraries did not even begin to tackle this problem.

Patterns form a more flexible foundation for reuse. A handbook of Best Practices Patterns would provide workable solutions for a given domain. A handbook like this would be particularly valuable for capturing domain expertise. What the patterns community is finding, however, is that the same patterns appear across projects and across domains. A group of developers at Siemens AG is writing a book of architectural patterns[8]. Those of us who have been involved in reviewing these patterns can see that these are not specific to any particular domain but would have wide application. Having a vocabulary of architectural patterns would be an excellent way to improve architectural design across the industry.

## **Conclusions**

Among all the benefits of discovering and using patterns, the improvement in communication is the most powerful. The resulting improvement in productivity and quality should be immediate when teams can have design discussions at a higher-level, since the time spent in these discussions will be reduced and the product improved as a result of incorporating solutions with proven records of success.

This communication improvement takes place not only within teams but across teams as well. Moving from one team to another should not involve re-learning design vocabulary. Everyone knows Binary Search. Everyone knows Stacks and Queues. Everyone should also know design patterns. The patterns enable implementors as well as designers. The ideas of the best designers can be more easily communicated to implementors. Instead of having continual consultation with the architectural gurus, a well-defined vocabulary will enable novices to better understand the intent of the design.

Finally, the ultimate benefit of using patterns is that we all become better designers. As Weinberg has wisely noted [9], None of us is as good as all of us! He was talking about the review process but his observation is certainly appropriate here. When we all share our design knowledge, we can all build on that knowledge and thereby

improve instead of re-inventing solutions over and over again. What a wonderful prospect!

## **Addendum**

The following web sites have more patterns information:

<http://st-www.cs.uiuc.edu/users/patterns/patterns.html> - WWW Patterns Home Page

<http://c2.com/ppr> - Portland Pattern Repository

<http://www.cs.wustl.edu/~schmidt/> - Doug Schmidt's Home Page

<http://c2.com/cgi-bin/wiki> - Wiki Wiki Web

<http://www.research.att.com/orgs/ssr/people/cope> - Jim Coplien's Home Page

## **References**

1. Alexander, Christopher. *The Timeless Way of Building*, Oxford University Press, New York, 1979.
2. Alexander, Christopher, et al. *A Pattern Language*, Oxford University Press, New York, 1977.
3. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
4. Yourdon, Edward and Larry L. Constantine. *Structured Design*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
5. Coplien, James. *Generative Pattern Languages: An Emerging Direction of Software Design*, Proceedings of the 5th Annual Borland International Conference, Orlando, Florida, June, 1994.
6. Adams, Michael, James Coplien, Robert Gamoke, Fred Kieve, Robert Hanmer, Keith Nicodemus. *Fault-Tolerant Telecommunication Systems Patterns*, *Patterns Languages of Program Design 2*, Addison Wesley, Reading, MA, 1996, John Vlissides, Norm Kerth, and Jim Coplien, eds, pp. 549-562., 1995.

7. Kogut, Paul. " Design Reuse: Chemical Engineering vs. Software Engineering, " Software Engineering Notes, Vol. 20, No. 5, December 1995, pp. 73-77.

8. Buschmann, Frank, Regine Meunier, hans Rohnert, Peter Sommerland, and Michael Stal, Pattern - Oriented Software Architecture, A System of Patterns, John Wiley & Sons, 1996

9. Weinberg, Gerald, Electronic mail message.

4ESS is a trademark of AT&T. GTD-5 is a registered trademark used under license from GTE Corporation.

**This article appeared in Annual Review of Communications, Vol. 49, 1996, pp. 907-909. Reprinted with permission of the International Engineering Consortium, Chicago, Illinois.**