Patterns and Software: Essential Concepts and Terminology

by **Brad Appleton** <*brad@bradapp.net*>
http://www.bradapp.net/
last modified 02/14/2000

Table of Contents

Introducing Patterns!

Patterns for software development are one of the latest "hot topics" to emerge from the object-oriented community. They are a literary form of software engineering problem-solving discipline that has its roots in a design movement of the same name in contemporary architecture, literate programming, and the documentation of best practices and lessons learned in all vocations.

Fundamental to any science or engineering discipline is a common vocabulary for expressing its concepts, and a language for relating them together. The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge which defines our understanding of good architectures that meet the needs of their users. Forming a common pattern language for conveying the structures and mechanisms of our architectures allows us to intelligibly reason about them. The primary focus is not so much on technology as it is on creating a culture to document and support sound engineering architecture and design.

Pattern Origins

Software patterns first became popular with the wide acceptance of the book **Design Patterns: Elements of Reusable Object-Oriented Software** by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (frequently referred to as the *Gang of Four* or just *GoF*). Patterns have been used for many different domains ranging from organizations and processes to teaching and architecture. At present, the software community is using patterns largely for software architecture and design, and (more recently) software development processes and organizations. Other recent books that have helped popularize patterns are: **Pattern-Oriented Software Architecture: A System of Patterns** (also called the *POSA* book) by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (sometimes called the Siemens *Gang of Five* or just *GoV*); and the books **Pattern Languages of Program Design** and **Pattern Languages of Program Design 2**, which are selected papers from the first and second conferences on Patterns Languages of Program Design (*PLoP* or *PLoPD*). Many of these books are part of the *Software Patterns Series* from Addison-Wesley.

The current use of the term "pattern" is derived from the writings of the architect Christopher Alexander who has written several books on the topic as it relates to urban planning and building architecture:

- **Notes on the Synthesis of Form**, Harvard University Press, 1964
  (hereafter referred to as "*[Notes]*")
- **The Oregon Experiment**, Oxford University Press, 1975
  (hereafter referred to as "*[Oregon]*")
- **A Pattern Language: Towns, Buildings, Construction**, Oxford University Press, 1977
  (hereafter referred to as "*[APL]*")
- **The Timeless Way of Building**, Oxford University Press, 1979
  (hereafter referred to as "*[TTWoB]*")

Although these books are ostensibly about architecture and urban planning, they are applicable to many other disciplines, including software development. In *[Notes]*, Alexander argues that current architectural methods result in products that fail to meet the real demands and requirements of its users, society and its individuals, and are unsuccessful in fulfilling the quintessential purpose of all design and engineering endeavors: to improve the human condition. Alexander wanted to create structures that are good for people and have a positive influence on them by improving their comfort and their quality of life. He concluded that architects must constantly strive to produce work products that better fit and adapt to the needs of all their inhabitants and users and their respective communities. In *[APL]* Alexander describes some "timeless" design ideas to try and realize these goals. In *[TTWoB]* Alexander proposes a paradigm for architecture based on three concepts: *the quality*, *the gate*, and *the way*:

The Quality (a.k.a. "*the Quality Without a Name*")
: This is the essence of all things living and useful that imparts unto them qualities such as: freedom, wholeness, completeness, comfort, harmony, habitability, durability, openness, resilience, variability, and adaptability. It is what makes us feel "alive" and "sated", gives us satisfaction, and ultimately improves the human condition.

The Gate
: This is the mechanism that allows us to reach *the quality*. It is manifested as a living *common* *pattern language* that permits us to create multiform designs which fulfill multifaceted needs. It is the universal "ether" of patterns and their relationships that permeate a given domain. *The gate* is the conduit to *the quality*.

The Way (a.k.a. "*the Timeless Way*")
: Using *the way*, patterns from *the gate* are applied using a technique of *differentiating space* in an ordered sequence of *piecemeal growth*: progressively evolving an initial architecture, which then flourishes into a "live" design possessing *the quality*. Alexander likens it to "a process of unfolding, like the evolution of an embryo, in which the whole precedes its parts, and actually gives birth to them, by splitting." (*[TTWoB]* p. 365). By following *the way*, one may pass through *the gate* to reach *the quality*.

A Bit of Patterns History

The events described here are related in more detail in the HistoryOfPatterns pages at Ward Cunningham's WikiWiki Web.

In 1987, Ward Cunningham and Kent Beck were working with Smalltalk and designing user interfaces. They decided to use some of Alexander's ideas to develop a small five pattern language for guiding novice Smalltalk programmers. They wrote up the results and presented them at OOPSLA'87 in Orlando in the paper *"Using Pattern Languages for Object-Oriented Programs"*.

Soon afterward, Jim Coplien (more affectionately referred to as "Cope") began compiling a catalog of C++ *idioms* (which are one kind of pattern) and later published them as a book in 1991, **Advanced C++ Programming Styles and Idioms**.

From 1990 to 1992, various members of the Gang of Four had met one another and had done some work compiling a catalog of patterns. Discussions of patterns abounded at OOPSLA'91 at a workshop given by Bruce Andersen (which was repeated in 1992). Many pattern notables participated in these workshops, including Jim Coplien, Doug Lea, Desmond D'Souza, Norm Kerth, Wolfgang Pree, and others.

In August 1993, Kent Beck and Grady Booch sponsored a mountain retreat in Colorado, the first meeting of what is now known as the Hillside Group. Another patterns workshop was held at OOPSLA'93 and then in April of 1994, the Hillside Group met again (this time with Richard Gabriel added to the fold) to plan the first PLoP conference.

Shortly thereafter, the *[GoF]* **Design Patterns** book was published, and the rest, is history.

What is a pattern anyway?

In "*Understanding and Using Patterns in Software Development*", Dirk Riehle and Heinz Zullighoven give a nice definition of the term "pattern" which is very broadly applicable:

> A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.

The above authors point out that, within the software patterns community, the notion of a pattern is "geared toward solving problems in design." More specifically, the concrete form which recurs is that of a solution to a recurring problem. But a pattern is more than just a battle-proven solution to a recurring problem. The problem occurs within a certain context, and in the presence of numerous competing concerns. The proposed solution involves some kind of structure which balances these concerns, or "forces", in the manner most appropriate for the given context. Using the pattern form, the description of the solution tries to capture the essential insight which it embodies, so that others may learn from it, and make use of it in similar situations. The pattern is also given a name, which serves as a conceptual handle, to facilitate discussing the pattern and the jewel of information it represents. So a definition which more closely reflects its use within the patterns community is:

> A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.

A slightly more compact definition which might be easier to remember is:

> A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns.

Patterns are usually concerned with some kind of architecture or organization of constituent parts to produce a greater whole. Richard Gabriel, author of **Patterns of Software: Tales From the Software Community**, provides a clear and concise definition of the term **pattern** in the Patterns Definitions section of the Patterns Home Page:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

As Gabriel explains, Alexander describes it a bit more colorfully in *[TTWoB]*:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
>
> As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
>
> As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.
>
> The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing (p. 247).

In **Software Patterns**, esteemed "patternite" Jim Coplien writes:

> I like to relate this definition to dress patterns. I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself.

So we see that a pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints. But a pattern does more than just identify a solution, it also explains *why the solution is needed*!

Generative Patterns

In *[TTWoB]*, Alexander explains that the most useful patterns are **generative**:

> These patterns in our minds are, more or less, mental images of the patterns in the world: they are abstract representations of the very morphological rules which define the patterns in the world. However, in one respect they are very different. The patterns in the world merely exist. But the same patterns in our minds are dynamic. They have force. They are generative. They tell us what to do; they tell us how we shall, or may, generate them; and they tell us too, that under certain circumstances, we *must* create them. Each pattern is a rule which describes what you have to do to generate the entity which it defines. (pp. 181-182)

Generative patterns tell us how to create something and can be observed in the resulting system architectures they helped shape. Non-generative patterns describe recurring phenomena without necessarily saying how to reproduce them. We should strive to document *generative* patterns because they not only show us the characteristics of good systems, they teach us *how to build them*!

For Alexander, however, this "instructive" element is only one facet of what he calls **generativity**. He wants patterns, and especially pattern languages, to be capable of generating whole, living structures. Part of the desire to create architectures that emulate life lies in the remarkably unique ability of living things to evolve and adapt to their ever-changing environments (not only for the sake of individual survival, but also for survival of the species). Alexander wants to impart these same qualities into his architectures. Similarly, in software, good software architectures are all about being adaptable and resilient to change. So another aspect of generativity is about striving to create "living" architectures which are capable of dynamically adapting to fulfill changing needs and demands.

But there is still more to "generativity" than teaching and adaptability. The successive application of several patterns, each encapsulating its own problem and forces, unfolds a larger solution which emerges indirectly as a result of the smaller solutions. It is the generation of such emergent behavior that appears to be what is meant by *generativity*. In this fashion, patterns and pattern languages should guide their users to generate whole architectures that have *the quality*. This particular aspect of Alexander's paradigm seems a bit too mystical for some people's tastes.

Pattern Envy and Pattern Ethics

It is unfortunate (but unavoidable) that the meteoric rise in the popularity of software patterns has led to massive hype. Many use the word "pattern" primarily for its appeal as a hot new buzzword.

Such "patterns-hype" ultimately causes disappointment, resentment, and even disdain when the hype proves different than the reality. This harms the credibility and legitimacy of those in the patterns community making genuine efforts to document "true" patterns. This greatly upsets many "patternites" and, as a result, there is a strong ethic within this community to avoid and dispel hype about patterns and patterns-related work. One might call this the **hype-no-cratic oath**: *First, do no hype!*.

Proto-Patterns and Patternity Tests

So it is important to remember that not every solution, algorithm, best practice, maxim, or heuristic constitutes a pattern (one or more key pattern ingredients may be absent). Even if something appears to have all the requisite pattern elements, it should *not* be considered a pattern until it has been verified to be a *recurring phenomenon* (preferably in at least three existing systems -- this is often called the **rule of three**). Some feel it is inappropriate to decisively call something a pattern until it has undergone some degree of scrutiny or review by others.

If, as Alexander writes, a pattern must be both a process *and* a thing, then a pattern must describe not only the process that creates that thing, but also the thing created by that process. For this reason, many contend that patterns ultimately deal with some kind of visually discernible structure: you should be able to draw a picture of the kind of structure that results from using the pattern in practice.

A "pattern in waiting" that is not yet known to pass the *patternity tests* mentioned above, or some of the ones mentioned below, is often called a **proto-pattern**. Brief descriptions of such proto-patterns are sometimes called **patlets** (pronounced "pat-lets").

Documenting good patterns can be an extremely difficult task. To quote Cope once again (this time from the Patterns Definitions page), *good* patterns do the following:

- **It solves a problem:** Patterns capture solutions, not just abstract principles or strategies.
- **It is a proven concept:** Patterns capture solutions with a track record, not theories or speculation.
- **The solution isn't obvious:** Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns generate a solution to a problem indirectly -- a necessary approach for the most difficult problems of design.
- **It describes a relationship:** Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- **The pattern has a significant human component** .... All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Bear in mind that just because something isn't a pattern doesn't mean it's not any good. If something is a pattern, then (hopefully) it is good. This should by no means imply that everything good should therefore be a pattern. There are many useful things in the world besides patterns; We must not let the surrounding patterns hype lull us into equating patterns with "sugar and spice and everything nice." This simply is not true.

Clearly, the very word "pattern" suggests recurrence; if something doesn't recur, it can't possibly be a pattern. But recurrence is not the sole characteristic of importance. We also need to show that a pattern is fit for use, and that it is a useful fit. Recurrence is purely quantitative characteristic, fitness and usefulness are qualitative characteristics. We can show recurrence simply by using the rule of three; We show fitness by explaining *how* the pattern is successful; and we show usefulness by explaining *why* it is successful and beneficial.

Patterns are Useful, Useable, and Used

So aside from recurrence, a pattern must describe *how* the solution balances or resolves its forces, and why this is a "good" resolution of forces. We need both of these to convince us that the pattern is neither sheer speculation (pure theory) nor is the pattern blindly following others (rote practice). We want to show that the practice is more than just "theory", and that the theory really has been practiced. We might even say that:

> A pattern is where theory and practice meet to reinforce and complement one another, by showing that the structure it describes is useful, useable, and used!

A pattern must be *useful* because this shows how having the pattern in our minds may be transformed into an instance of the pattern in the real world, as something thing that adds value to our lives as developers and practitioners. A pattern must also be *useable* because this shows how a pattern described in literary form may be transformed into a pattern that we have in our minds. And a pattern must be *used* because this is how patterns that exist in the real world first became documented as patterns in literary form.

This yields a continuously repeating cycle from pattern writers, to pattern readers, to pattern users: writers documenting patterns in literary form make them usable to pattern readers, who can then remember them in their minds, which makes them useful to practitioners and developers, who can use them in the real world, and enhance the user's quality of life.

Anti-Patterns

If a pattern represents a "best practice", then an **anti-pattern** represents a "lesson learned." Initially proposed by Andrew Koenig in the November 1995 C++ Report, there are two notions of "anti-patterns."

1. Those that describe a bad solution to a problem which resulted in a bad situation.
2. Those that describe how to get out of a bad situation and how to proceed from there to a good solution.

In *[Notes]*, Alexander writes at length about "forces" and "misfits" (or "misfit variables"), and about the struggle to achieve good fit between a good design structure and its context:

Every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. ... The context and the form are complementary. ... Once we have the diagram of the forces in the literal sense ... this will in essence also describe the form as a complementary diagram of forces.

These *misfits* are the forces which must shape it [the design], and there is no mistaking them. ... We may summarize the state of each potential misfit by a [boolean] variable. Each [misfit] variable stands for one possible kind of misfit between the form and context. These [misfit variables] describe a state of affairs that is neither in the form alone nor in the context alone, but a relation between the two. The state of this relation, fit or misfit, describes one aspect of the whole ensemble. It is a condition of harmony and good fit in the ensemble that none of the possible misfits should actually occur.

These "conditions of harmony" of forces between a form and its context are what Alexander later came to call "patterns." The misfits, and the situations in which misfits dominate the "form" correspond to what many now call anti-patterns. Anti-patterns can be valuable because it is often just as important to see and understand bad solutions as it is to see and understand good ones. This makes for some potentially useful signposts to help us recognize why a particular alternative might seem sensible at first, but turns out not to be the best route to follow. It helps guide us away from the misfits and toward the appropriate patterns that might better solve our problem.

More recently on the anti-patterns front, Brown, Malveau, McCormick and Mowbray have written a book entitled **Anti-Patterns**, which is about "refactoring software, architectures, and projects in crisis." The anti-patterns and refactored solutions described in the book are very similar in nature to many of the "pitfalls" described by Bruce Webster in **Pitfalls of Object-Oriented Development** and the "lessons learned" described by Tom Love in **Object Lessons**.

These books attempt to do more than just identify and diagnose various "anti-solutions"; they try to go a step further by recommending a course of action for correction, recovery, and prevention. The most useful anti-patterns are those that do more than just point out Dilbert-like examples of bad practice (or of good practice gone bad). Anti-patterns that stop short at this point are about the death and destruction of good design and good process. Recognizing when this occurs is certainly valuable, but the most useful anti-patterns are about recovery, refactoring, realignment, and reconstitution of the good from the bad; they show us how to turn lemons into lemonade and how to avoid subsequent lemons, leading us away from the misfits of anti-patterns, and back towards the patterns that resolve them.

Kinds of Patterns

Due to the overwhelming acceptance of the *[GoF]* book, much of the initial patterns focus in the software community has been on **design patterns**. The patterns in the *[GoF]* book are object-oriented design patterns. There are many other kinds of software patterns besides design patterns. Martin Fowler has written a book of Analysis Patterns. There is also a website and mailing list for organizational patterns. Patterns submitted to previous PLoP conferences have encompassed all aspects of software engineering including: development organization, software process, project planning, requirements engineering, and software configuration management (just to name a few). Presently however, design patterns still seem to be the most popular (though organization patterns seem to be gaining momentum).

Kinds of Design Patterns

The *[GoF]* book defines design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." It then goes on to say that:

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not in can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample ... code to illustrate an implementation. Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages ....

The above description is slanted toward *object-oriented design*, but with only minor changes, it could be readily adjusted to describe software design patterns in general (simply remove the words "object-oriented" and replace "class" with "module" or "subsystem"). Since the *[GoF]* book was the first (and currently the most popular) of the software patterns books, the term "*design pattern*" is often used to refer to any pattern which directly addresses issues of software architecture, design, or programming implementation. Many choose to make an important distinction between these three conceptual levels by categorizing them into **architectural patterns**, **design patterns**, and **idioms** (idioms are sometimes called **coding patterns**). The authors of **Patterns of Software Architecture** *[POSA]* define these three types of patterns as follows:

Architectural Patterns
An *architectural pattern* expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Design Patterns

A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.

Idioms

An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

The difference between these three kinds of patterns are in their corresponding levels of abstraction and detail. Architectural patterns are high-level strategies that concern large-scale components and the global properties and mechanisms of a system. They have wide-sweeping implications which affect the overall skeletal structure and organization of a software system. Design patterns are medium-scale tactics that flesh out some of the structure and behavior of entities and their relationships. They do *not* influence overall system structure, but instead define *micro-architectures* of subsystems and components. Idioms are paradigm-specific and language-specific programming techniques that fill in low-level internal or external details of a component's structure or behavior.

In "*Understanding and Using Patterns in Software Development*", Riehle and Zullighoven make similar distinctions, but seem to partition the different kinds of patterns among analysis, design, and implementation. They define the terms **conceptual patterns**, **design patterns**, and **programming patterns** as follows:

Conceptual Patterns

A *conceptual pattern* is a pattern whose form is described by means of terms and concepts from an application domain.

Design Patterns

A *design pattern* is a pattern whose form is described by means of software design constructs, for example objects, classes, inheritance, aggregation and use-relationship.

Programming Patterns

A *programming pattern* is a pattern whose form is described by means of programming language constructs.

Using these definitions, conceptual patterns are based upon metaphors in a restricted application domain. Design patterns complement, or elaborate upon conceptual patterns by delving into the implementation of elements from the conceptual space. And programming patterns descend further into implementation details using a specific implementation language.

When comparing and contrasting these two sets of definitions, it appears that programming patterns are equivalent to idioms. For the other types of patterns described above, the first set of authors choose to delineate them by their architectural scope whereas the latter set of authors choose to delineate them by whether they employ language from the problem space or the solution space.

Elements of a Pattern

Alexander says that "every pattern we define must be formulated in the form of a rule which establishes a relationship between a context, a system of forces which arises in that context, and a configuration, which allows these forces to resolve themselves in that context." (*[TTWoB]* p. 253). Alexander also recommends using pictorial examples: "First there is a picture which shows an archetypal example of the pattern." (*[APL]* p. *x*). Several different formats have been used for describing patterns (see the answer to question 10 of *Doug Lea's Patterns-Discussion FAQ* for one such description). The pattern description format used in Alexander's work is called the "**Alexandrian form**". The format used in *[GoF]* is referred to as "**GoF format**". The section headings of the paragraphs which immediately follow, make up what is called "**canonical form**" (sometimes this too is called "Alexandrian form") and is the format used by *[POSA]*, AGCS, and many others (often with slight adaptations). Despite the use of these differing pattern formats, it is generally agreed that a pattern should contain certain essential elements. Regardless of the particular format/headings used (or lack thereof), *the following essential elements should be clearly recognizable upon reading a pattern*:

Name

It must have a meaningful name. This allows us to use a single word or short phrase to refer to the pattern, and the knowledge and structure it describes. It would be very unwieldy to have to describe or even summarize the pattern every time we used it in a discussion. Good pattern names form a vocabulary for discussing conceptual abstractions. Sometimes a pattern may have more than one commonly used or recognizable name in the literature. In this case it is common practice to document these nicknames or synonyms under the heading of **Aliases** or **Also Known As**. Some pattern forms also provide a **classification** of the pattern in addition to its name.

Problem

A statement of the problem which describes its **intent**: the goals and objectives it wants to reach within the given context and forces. Often the forces oppose these objectives as well as each other (one might think of this as a "wicked problem" reminiscent of DeGrace and Stahl, in their book **Wicked Problems, Righteous Solutions**).

Context

The *preconditions* under which the problem and its solution seem to recur, and for which the solution is desirable. This tells us the pattern's **applicability**. It can be thought of as the initial configuration of the system before the pattern is applied to it.

Forces

A description of the relevant *forces* and constraints and how they interact/conflict with one another and with goals we wish to achieve (perhaps with some indication of their priorities). A concrete scenario which serves as the **motivation** for the pattern is frequently employed (see also **Examples**). Forces reveal the intricacies of a problem and define the kinds of *trade-offs* that must be considered in the presence of the tension or dissonance they create. A good pattern description should fully encapsulate all the forces which have an impact upon it. A list of prospective pattern forces for software may be found in the answer to question 11 of *Doug Lea's Patterns-Discussion FAQ*.

Solution

Static relationships and dynamic rules describing how to realize the desired outcome. This is often equivalent to giving instructions which describe how to construct the necessary work products. The description may encompass pictures, diagrams and prose which identify the pattern's **structure**, its **participants**, and their **collaborations**, to show how the problem is solved. The solution should describe not only *static structure* but also *dynamic behavior*. The static structure tells us the form and organization of the pattern, but often it is the behavioral **dynamics** that make the pattern "come alive". The description of the pattern's solution may indicate guidelines to keep in mind (as well as pitfalls to avoid) when attempting a concrete **implementation** of the solution. Sometimes possible **variants** or specializations of the solution are also described.

Examples

One or more sample applications of the pattern which illustrate: a specific initial context; how the pattern is applied to, and transforms, that context; and the resulting context left in its wake. Examples help the reader understand the pattern's use and applicability. Visual examples and analogies can often be especially illuminating. An example may be supplemented by a *sample implementation* to show one way the solution might be realized. Easy-to-comprehend examples from known systems are usually preferred (see also **Known Uses**).

Resulting Context

The state or configuration of the system after the pattern has been applied, including the **consequences** (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context. It describes the *postconditions* and *side-effects* of the pattern. This is sometimes called **resolution of forces** because it describes which forces have been resolved, which ones remain unresolved, and which patterns may now be applicable (see the answer to question 12 of *Doug Lea's Patterns-Discussion FAQ* for an excellent discussion of *resolution of forces*). Documenting the resulting context produced by one pattern helps you correlate it with the initial context of other patterns (a single pattern is often just one step towards accomplishing some larger task or project).

Rationale

A justifying explanation of steps or rules in the pattern, and also of the pattern as a whole in terms of how and why it resolves its forces in a particular way to be in alignment with desired goals, principles, and philosophies. It explains how the forces and constraints are orchestrated in concert to achieve a resonant harmony. This tells us how the pattern actually works, why it works, and why it is "good". The solution component of a pattern may describe the outwardly visible structure and behavior of the pattern, but the rationale is what provides insight into the *deep structures* and *key mechanisms* that are going on beneath the surface of the system.

Related Patterns

The static and dynamic relationships between this pattern and others within the same pattern language or system. Related patterns often share common forces. They also frequently have an initial or resulting context that is compatible with the resulting or initial context of another pattern. Such patterns might be predecessor patterns whose application leads to this pattern; successor patterns whose application follows from this pattern; alternative patterns that describe a different solution to the same problem but under different forces and constraints; and codependent patterns that may (or must) be applied simultaneously with this pattern.

Known Uses

Describes known occurrences of the pattern and its application within existing systems. This helps validate a pattern by verifying that it is indeed a *proven solution* to a *recurring problem*. Known uses of the pattern can often serve as instructional examples (see also **Examples**).

Although it is not strictly required, good patterns often begin with an **Abstract** that provides a short summary or overview. This gives readers a clear picture of the pattern and quickly informs them of its relevance to any problems they may wish to solve (sometimes such a description is called a *thumbnail sketch* of the pattern, or a **pattern thumbnail**). A pattern should identify its target audience and make clear what it assumes of the reader.

Qualities of a Pattern

In addition to containing the aforementioned elements, a well written pattern should exhibit several desirable qualities. Doug Lea, in his paper *"Christopher Alexander: an Introduction for Object-Oriented Designers"* provides a detailed description of these qualities (which are summarized below):

Encapsulation and Abstraction

Each pattern encapsulates a well-defined problem and its solution in a particular domain. Patterns should provide crisp, clear boundaries that help crystallize the problem space and the solution space by parceling them into a lattice of distinct, interconnected fragments. They also serve as abstractions which embody domain knowledge and experience, and may occur at varying hierarchical levels of conceptual granularity within the domain.

Openness and Variability

Each pattern should be open for extension or parametrization by other patterns so that they may work together to solve a larger problem. A pattern solution should be also capable of being realized by an infinite variety of implementations (in isolation, as well as in conjunction with other patterns).

Generativity and Composability

Each pattern, once applied, generates a resulting context which matches the initial context of one or more other patterns in a pattern language. These subsequent patterns may then be applied to progress further toward the final goal of generating a "whole" or complete overall solution. "Patterns are applied by the means of *piecemeal growth*. Applying one pattern provides a context for the application of the next pattern." (from the PLoP'97 Call for Papers) But patterns are not simply linear in nature, more like fractals in that patterns at a particular level of abstraction and granularity may each lead to or be composed with other patterns at varying levels of scale.

Equilibrium

Each pattern must realize some kind of balance among its forces and constraints. This may be due to one or more invariants or heuristics that are used to minimize conflict within the solution space. The invariants often typify an underlying problem solving principle or philosophy for the particular domain, and provide a rationale for each step/rule in the pattern.

The aim is that, if well written, each pattern describes a whole that is greater than the sum of its parts, due to skillful choreography of its elements working together to satisfy all its varying demands.

Patterns, Rules, and Creativity

It is the combined presence of *all* these pattern elements and qualities that make patterns *more* than just heuristics, rules, or algorithms. Heuristics and principles frequently participate in the forces and/or rationale of a pattern, but they are only one element of the pattern. Furthermore, as Cope writes in "*Software Design Patterns: Common Questions and Answers*":

> Rules aren't commonly supported by a rationale, nor put in context. A rule may be part of the solution in a pattern description, but a rule solution is neither sufficient nor necessary. Patterns aren't designed to be executed or analyzed by computers, as one might imagine to be true for rules: patterns are to be executed by architects with insight, taste, experience, and a sense of aesthetics.

A pattern is the process that generates a solution, but it may generate any one of a vast number of variant solutions (conceivably without repeating the same solution twice). The human element of patterns is what chiefly contributes to their variability and adaptability, and usually requires a greater degree of creativity in their application and combination. So, just as the processes of architecture and design are creative endeavors, so too is the application of patterns. In the same paper quoted above, Cope goes on to say:

> If design is codified in patterns, does the need for creativity go away? Can we replace high-priced expensive designers with less sophisticated programmers who are guided by patterns? The answer is that creativity is still needed to shape the patterns to a given context. Just as a dressmaker tailors a pattern to an individual customer, and perhaps to a specific event where the dress is to be worn, so designers must be creative when using patterns. Patterns channel creativity; they neither replace nor constrain it.

Patterns and Algorithms

The previous section about patterns versus rules also applies in large part to algorithms and their data structures. Certainly, algorithms and data structures may be employed in the implementation of one or more patterns, but algorithms and data structures generally solve more fine-grained computational problems like sorting and searching. Patterns are typically concerned with broader architectural issues that have larger-scale effects. The design patterns in *[GoF]* address people and development issues like maintainability, reusability, communicating commonality and encapsulation variation. These are issues that matter to the people who need to create *and* evolve/grow these software systems over time.

Algorithms and data structures are usually concerned almost exclusively with optimizing space or time or some other aspect of computational complexity and resource consumption. They are about finding the most compact and efficient means to perform some important computation or store and recall its results. Such algorithms and data structures are rarely concerned with compromises and tradeoffs regarding other concerns that other concerns that have little to do with things like performance or memory, and more to do with things like maintainability and adaptability and (re)usability of the architecture.

There are a great many books of algorithms and data-structures that provide source code and quantitative analysis for structures like AVL trees or splay-trees. But in many of these same textbooks there is little mention of how to implement these structures in ways that emphasize maintainability and adaptability and reuse; and when they do there is a whole new set of issues to worry about (like maintainability and reusability and encapsulation). Looking at the computational time and space aspects alone simply doesn't address the fuller set of forces that affect the architects and implementors as well as the users.

This is why algorithms and data structures tend to be more fine-grained than patterns: Because they mostly address issues of computational complexity, and not so much the underlying issues of the people who are both using *and* building the software. Patterns fundamentally address people issues (like maintainability) more so than simple hardware and software efficiency/memory issues.

Of course software developers need to be concerned both with finding appropriate architectures and with finding appropriate solutions to computational problems. So there will always be a need for patterns as well as for algorithms and data structures (and their use together).

Patterns and Frameworks

One thing closely related to design patterns and object-orientation is a **software framework** :

> A software framework is a *reusable mini-architecture* that provides the generic structure and behavior for a family of software abstractions, along with a context of memes/metaphors which specifies their collaboration and use within a given domain.

The framework accomplishes this by hardcoding the context into a kind of "virtual machine" (or "virtual engine"), while making the abstractions open-ended by designing them with specific *plug-points* (also called *hot spots*). These plug-points (typically implemented using callbacks, polymorphism, or delegation) enable the framework to be adapted and extended to fit varying needs, and to be successfully composed with other frameworks. A framework is usually *not* a complete application: it often lacks the necessary application-specific functionality. Instead, an application may be constructed from one or more frameworks by inserting this missing functionality into the plug-and-play "outlets" provided by the frameworks. Thus, *a framework supplies the infrastructure and mechanisms that execute a policy for interaction between abstract components with open implementations*.

A definition of an *object-oriented* software framework is given in *[GoF]*:

> A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes a framework to a particular application by subclassing and composing instances of framework classes.
>
> ... [a framework] dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. A framework predefines these design parameters so that you, the application designer/implementer, can concentrate on the specifics of your application. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize *design reuse* over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.

The difference between a framework and an ordinary programming library is that a framework employs an *inverted flow of control* between itself and its clients. When using a framework, one usually just implements a few callback functions or specializes a few classes, and then invokes a single method or procedure. At this point, the framework does the rest of the work for you, invoking any necessary client callbacks or methods at the appropriate time and place. For this reason, frameworks are often said to abide by **the Hollywood Principle** ("Don't call us, we'll call you.") or **the Greyhound Principle** ("Leave the driving to us.").

Design patterns may be employed both in the design and the documentation of a framework. A single framework typically encompasses several design patterns. In fact, a framework can be viewed as the implementation of a system of design patterns. Despite the fact that they are related in this manner, it is important to recognize that frameworks and design patterns are two distinctly separate beasts: a framework *is executable software*, whereas design patterns represent knowledge and experience *about software*. In this respect, frameworks are of a physical nature, while patterns are of a logical nature: frameworks are the *physical realization* of one or more software pattern solutions; patterns are the instructions for *how to implement* those solutions.

The *[GoF]* book describes the major differences between design patterns and frameworks as follows:

1. *Design patterns are more abstract than frameworks*. Frameworks can be embodied in code, but only *examples* of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. In contrast, design patterns have to be implemented each time they are used. Design patterns also explain the intent, trade-offs, and consequences of a design.

2. *Design patterns are smaller architectural elements than frameworks*. A typical framework contains several design patterns but the reverse is never true.

3. *Design patterns are less specialized than frameworks*. Frameworks always have a particular application domain. In constrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these wouldn't dictate an application architecture.

The Quality Without a Name

The **"Quality Without A Name"** (abbreviated as the acronym QWAN) is *"the quality"* that imparts incommunicable beauty and immeasurable value to a structure. It encompasses all of the following:

- universally recognizable aesthetic beauty and order
- recursively nested centers of symmetry and balance

- life and wholeness
- resilience, adaptability, and durability
- human comfort and satisfaction
- emotional and cognitive resonance

Alexander proposes the existence of an objective quality of aesthetic beauty that is universally recognizable. He claims there are certain timeless attributes and properties which are considered beautiful and aesthetically pleasing to all people in all cultures (not just "in the eye of the beholder"). It is these fundamental properties which combine to generate the QWAN, and which make a structure feel "whole" and "alive".

Alexander conducted some experiments using configurations of colored beads, and also using the design of carpets and tapestries. Those results, combined with his own architectural experience with the users of his buildings, suggested the presence of this "objective beauty" was closely tied to the presence of symmetries and subsymmetries that balance the use of contrasting space, light, and color to form fields of visual centers. Feelings of beauty and order would increase when these visual centers unfolded recursively at multiple hierarchical levels of granular scale throughout a design (much like fractals). However, if the symmetries are too pure (too perfect or exact), it seems to be less desirable than if slight irregularities and imperfections exist. Apparently some imperfections are not only palatable, but can also be utilitarian (just as doped imperfections in precious stones can actually enhance their crystalline structure).

In *The Laws of Architecture from a Physicist's Perspective*, artist and mathematician Nikos A. Salingaros (who has worked closely with Alexander over the past ten years) discusses the rules of beauty and order in past times and, as a result, proposes the following three laws of architecture:

1. Order on the smallest scale is established by paired contrasting elements, existing in a balanced visual tension.
2. Large-scale order occurs when every element relates to every other element at a distance in a way that reduces entropy.
3. The small scale is connected to the large scale through a linked hierarchy of intermediate scales with scaling factor approximately equal to $e = 2.718$.

Salingaros goes on to say that these laws don't govern *beauty* in architecture, they govern *life* in architecture: "They are extracted from physics and mathematics by looking at how nature is put together. I discovered these laws by observing how fundamental particles come together to form a structure." He says that when structures conform to these architectural laws, those who enter them feel a kind of resonant harmony, an almost emotional connection to the structure which feels peaceful and nourishing. This connection between life and architecture is due to the thermodynamics of living forms, but the idea of building structures which possess these intrinsic qualities of living biological organisms comes from Alexander.

In **Reengineering the Application Development Process**, Michael Beedle tries to succinctly describe the QWAN as something that:

> is created when the attributes in the design makes that design "live". That is, designs that are flexible, extensible, adaptable, reusable and have other qualities of living things; except of course self-reproduction and metabolism.

According to this definition, the QWAN is attained when the pattern or pattern language generates a "live" architectural solution; one which emulates the capability of living things to dynamically adapt to fulfill changing needs and demands. This is closely related to Alexander's idea of generativity. In *"Christopher Alexander: an Introduction for Object-Oriented Designers"*, Doug Lea attempts to describe the QWAN as follows:

> ... "the quality without a name", the possession of which is the ultimate goal of any design product. It is impossible to briefly summarize this. Alexander presents a number of partial synonyms: *freedom*, *life*, *wholeness*, *comfortability*, *harmony*, But no single term or example fully conveys meaning or captures the force of Alexander's writings on the reader, especially surrounding the human impact of design, the feelings and aesthetics of designers and users, the need for commitment by developers to obtain and preserve wholeness, and its basis in the objective equilibrium of form.

So another crucial aspect of the QWAN is the effect it has upon the architecture's inhabitants and designers that makes them feel alive, whole, and comfortable. It is this kind of "habitability" that improves user comfort and quality of life (what TQM circles might refer to as "total customer satisfaction", and what Tom Peters means by "to thrill and delight" the customer in the pursuit of "Wow!").

Of course this "ultimate goal" of achieving the QWAN is very elusive. This particular aspect of patterns seems to borrow concepts from Zen Buddhism, Taoism, and Platonic ideals. There are those who feel the whole idea of the QWAN is just a bit too whimsical and metaphysical; that it is not scientific or tangible enough to have a place in true engineering disciplines (especially if it is something that cannot be measured or quantified). The notion of beauty as something that is universally objective can be a trifle hard for some to swallow.

But in many respects, an individual's sense of the QWAN is also about cognitive judgement. Every master designer develops their own highly honed intuition which is borne from extensive experience. Although this "intuition" may be subjective, it can be uncannily accurate and give the designer an almost instinctive sense of what will work and what wont (even before the measures are brought to bear to try and verify it). This stems from the designer being intimate with the design and internalizing it at a visceral level, almost to the point of becoming an inhabitant whose sensory network is "plugged-in" to the system. If a pattern can impart to its readers and users, this same "plugged-in" feeling of being connected to the design and deeply comprehending it, then in theory it will impart to the reader the same cognitive feeling of its aptness that the designer experienced. If a pattern succeeds in this attempt, then all who see and use it will supposedly experience the resonant feeling of beauty and harmony that the QWAN is supposed to evoke.

Pattern Languages

A collection of patterns forms a vocabulary for understanding and communicating ideas. Such a collection may be skillfully woven together into a cohesive "whole" that reveals the inherent structures and relationships of its constituent parts toward fulfilling a shared objective. This is what Alexander calls a **pattern language**. If a pattern is a recurring solution to a problem in a context given by some forces, then a pattern language is a collective of such solutions which, at every level of scale, work together to resolve a complex problem into an orderly solution according to a pre-defined goal. In the Patterns Definitions section of the Patterns Home Page, Cope defines a pattern language as follows:

> A pattern language defines a collection of patterns and the rules to combine them into an architectural style. Pattern languages describe software frameworks or families of related systems.

Cope gives a slightly different definition in *"Software Design Patterns: Common Questions and Answers"*:

> A pattern language is a structured collection of patterns that build on each other to transform needs and constraints into an architecture.

Cope then goes on to say that:

> Good pattern languages guide the designer toward useful architectures and away from architectures whose literary analogies are gibberish or unartful writing. Good architectures are durable, functional, and aesthetically pleasing, and a good combination of patterns can balance the forces on a system to strive towards these three goals. A good pattern language gives designers freedom to express themselves and to tailor the solution to the particular needs of the context where the patterns are applied.

Unlike a mere pattern compilation or catalog, a pattern language includes rules and guidelines which explain how and when to apply its patterns to solve a problem which is larger than any individual pattern can solve. These rules and guidelines suggest the order and granularity for applying each pattern in the language. A pattern language may be regarded as a lexicon of patterns plus a grammar that defines how to weave them together into valid sentences (or artful tapestries if you will). Ideally, good pattern languages are **generative**, capable of generating all the possible sentences from a rich and expressive pattern vocabulary.

A pattern language forms a gestalt in which each of its patterns collaborate to solve a more fundamental problem that is not explicitly addressed by any individual pattern. This helps a pattern language to achieve an **organic order**, which Alexander describes in *[Oregon]* as "the kind of order that is achieved when there is a perfect balance between the needs of the parts and the needs of the whole." So in a sense, a pattern language is like an ecosystem of patterns, all of which are inherently related at some level. This "ecological quality" of pattern languages contributes to their "wholeness" and their ability to help us generate "live" architectures" possessing the QWAN.

Throughout **The Timeless Way of Building**, Alexander remarks on what a pattern language truly embodies:

- "Thus, as in the case of natural languages, the pattern language is generative. It not only tells us the rules of arrangement, but shows us how to construct arrangements -- as many as we want -- which satisfy the rules." (p. 186)
- "A pattern language gives each person who uses it, the power to create an infinite variety of new and unique buildings, just as his ordinary language gives him the power to create an infinite variety of sentences." (p. 167)
- "The structure of the language is created by the network of connections among individual patterns: and the language lives, or not, as a totality, to the degree these patterns form a whole." (p. 305)
- "Each pattern then, depends both on the smaller patterns it contains and on the larger patterns within which it is contained." (p. 312)
- "The language is a good one, capable of making something whole, when it is morphologically and functionally complete." (p. 316)

Michael Beedle, author of **Reengineering the Application Development Process**, likens the effects of using pattern languages to the generation of *emergent behaviors*: spontaneously recurring patterns of dense local interaction between entities, resulting in dynamic, self-organizing systems that are adaptive, open, and capable of multi-scale effects. In other words, pattern languages provide a dynamic process for the orderly resolution of problems within their domain which indirectly leads to the resolution of a much broader problem. The patterns and rules in a pattern language combine to form an architectural style. In this manner, pattern languages guide system analysts, architects, designers, and implementors to produce workable systems that solve common organizational and development problems at all levels of scale and diversity.

Piecemeal Growth

In *[Oregon]*, Alexander describes how his pattern language and corresponding "timeless way" method was applied to a planning project for the University of Oregon. He explains the principle of organic order in which "Planning and construction will be guided by a process which allows the whole to emerge gradually from local acts." He then explains the process of piecemeal growth as one which is based on the idea of repair as opposed to replacement: rather than designing an architecture to replace an existing one, and which will eventually be replaced itself, Alexander prescribes a more evolutionary approach which gradually unfolds a complete structure from an initial foundation by continual embellishment, modification, improvement, and repair. He refers to the "design for replacement" approach as *large lump development* and compares it with piecemeal growth as follows:

> Large lump development hinges on a view of the environment which is static and discontinuous; piecemeal growth hinges on a view of the environment which is dynamic and continuous.... According to the large lump point of view, each act of design or construction is an isolated event which creates an isolated building -- "perfect" at the time of construction, and then abandoned by its builders and designers forever. According to the piecemeal point of view, every environment is changing and growing all the time, in order to keep its use in balance; and the quality of the environment is a kind of semi-stable equilibrium in the flux of time.... Large lump development is based on the idea of replacement. Piecemeal growth is based on the idea of repair.

Those familiar with software development lifecycles might see some similarities between large lump development and the waterfall model, and between piecemeal growth, and the spiral models which involve prototyping and incremental/evolutionary development (or what Booch calls "round-trip gestalt design").

The relationship between patterns and piecemeal growth is that pattern languages are intended to grow and evolve whole architectures through this process of piecemeal growth. The various sequences in which a pattern language instructs the user in the application of the patterns therein should unfold a complete architecture which gradually emerges from the successive application of individual patterns in the appropriate order.

Pattern Catalogs and Systems

The authors of *[POSA]* have classified different kinds of pattern collections that possess varying degrees of structure and interaction into pattern catalogs, systems, and languages:

Pattern Catalogs
: A *pattern catalog* is a collection of related patterns (perhaps only loosely or informally related). It typically subdivides the patterns into at least a small number of broad categories and may include some amount of cross referencing between patterns.

Pattern Systems
: A *pattern system* is a cohesive set of related patterns which work together to support the construction and evolution of whole architectures. Not only is it organized into related groups and subgroups at multiple levels of granularity, it describes the many interrelationships between the patterns and their groupings and how they may be combined and composed to solve more complex problems. The patterns in a pattern system should all be described in a consistent and uniform style and need to cover a sufficiently broad base of problems and solutions to enable significant portions of complete architectures to be built.

A pattern catalog adds a modicum of structure and organization to a pattern collection, but doesn't usually go very far beyond showing only the most outwardly visible structure and relationships (if in fact it shows any of them). A pattern system adds deep structure, rich pattern interaction, and uniformity to a pattern catalog. Both pattern systems and pattern languages form coherent sets of tightly interwoven patterns for describing and solving problems in a particular domain. But a pattern language adds robustness, comprehensiveness, and wholeness to a pattern system. The primary difference is that, ideally, pattern languages are computationally complete, showing all possible combinations of patterns and their variations to produce complete architectures. In practice however, the difference between pattern systems and pattern languages can be extremely difficult to ascertain.

While a pattern system may be a cohesive collection of patterns about a very broad topic, a pattern language has to be about more than just a "broad topic". A pattern language ultimately corresponds to a single-minded collective that forms a kind of "mega pattern" or "super pattern" in that the entire language possesses an underlying, shared problem and its context, forces, solution, resulting context, and rationale (which each pattern in the language addresses at some level or another). This coherence of purpose is what gives the pattern language a sense of closure. A pattern system does not necessarily have *all* of these pattern elements. It may focus on an equally broad or narrow topic, but it may not necessarily have a clear mission or agenda (and may result in many relationships between patterns being harder to find), or it leaves several important gaps unfilled in the problem space (and hence may not attain overall resolution or closure).

But perhaps the most important difference between pattern languages and pattern systems is that pattern languages are not created all at once. They evolve from pattern systems through the process of piecemeal growth (and a pattern system may, in turn, evolve from a pattern catalog in a similar manner). So just as pattern languages help to incrementally grow whole architectures, pattern systems may serve to incrementally grow into whole pattern languages.

Writing Patterns

Writing good patterns is *very* difficult. Patterns should not only provide facts (like a reference manual or user's guide), but should also tell a story which captures the experience they are trying to convey. A pattern should help its users to: comprehend existing systems; customize systems to fit user needs; and construct new systems. The process of looking for patterns to document is called **pattern mining** (or sometimes **reverse-architecting**).

How do you know a pattern when you come across one? The answer is you don't always know. You may jot down the beginnings of some things you think are patterns, but it may turn out that they aren't patterns at all, or they are only pieces of patterns, or simply good principles or rules of thumb that may form part of the rationale of a particular pattern. It is important to remember that a solution in which no forces are present is *not* a pattern.

The best way to learn how to recognize and document useful patterns is by learning from others who have done it well! Pick up several books and articles which describe patterns (don't choose just one) and try and see if you can recognize all the necessary pattern elements and desirable qualities that were mentioned earlier in this paper. When you see one that appeals to you, ask yourself why it is good. If you see one you don't like, try and figure out exactly what it is about the pattern that leaves you unsatisfied. Read as much as you can, and try to learn from the masters. Numerous resources for learning more about patterns are given near the end of this paper. Most importantly, *be introspective about everything you read*! Examine how it is meaningful to you and how it will help you accomplish future goals.

After you have been exposed to a wealth of pattern literature, choose one of the various pattern formats and see if you can flesh out some of your earlier ideas for things you thought might be patterns. If you are trying to compose a language of patterns, start by examining the forces and context of each pattern and try to identify any simple underlying principles that seem to help organize the patterns together into useful configurations.

Further assistance for those who are courageous enough to undertake writing a pattern, or a pattern language, may be found in Ward Cunningham's "*Tips for Writing Pattern Languages*", and in Gerard Meszaros' and Jim Doble's "*A Pattern Language for Pattern Writing*". Both of these papers are indispensable resources for writing patterns and pattern languages.

How do the experts decide what makes a good pattern? The PLoP conferences have several criteria which they feel submitted pattern papers should meet. These are as follows (summarized from Buschmann et. al. in **Pattern-Oriented Software Architecture**):

- **Focus on practicability:** Patterns should describe *proven solutions* to recurring problems rather than the latest scientific results.
- **Aggressive disregard of originality:** Pattern writers do *not* need to be the original inventor or discoverer of the solutions that they document.
- **Non-anonymous review:** Pattern submissions are *shepherded* rather than reviewed. The shepherd contacts the pattern author(s) and discusses with them how the patterns might be clarified or improved upon.
- **Writer's workshops instead of presentations:** Rather than being presented by the individual authors, the patterns are discussed in **writer's workshops**: an open forum where all attending seek to improve the patterns presented by discussing what they like about the patterns as well as other areas in which they are lacking.
- **Careful editing:** The pattern authors have the opportunity to incorporate all the comments and insights during the shepherding and writer's workshops before presenting the patterns in their finished form.

Pattern Futures

The current popularity of software patterns has spurred numerous activities to broaden their use and support within the software development community. There are groups of people using patterns to document their software development processes and engineering handbooks in the form of a pattern language. Several of the leading object-oriented software design notations/methods have added support for the modeling and representation of design patterns. Many software development tools and environments have added similar support. Some research projects are attempting to formally codify design patterns for the purposes of generating source code. Commercial software libraries are being developed which provide reusable implementations of several well known design patterns (Java provides a few of these in its standard library). It may not be long before some programming languages introduce special syntax for representing design patterns as explicit programming constructs.

There is speculation that patterns will one day replace computer programmers. Such speculation has occurred many times in the past regarding various other "new" technologies. The languages and tools that will be used to solve software problems in the future may advance far beyond what is presently available (much like programming languages and development environments have evolved since the days of programming machine-code on punch cards), but they will still require developers with similarly evolved skills and wisdom to use them effectively. As long as software developers keep abreast of emerging software concepts and technologies, there will always be a need for them to skillfully and creatively construct useful software solutions using these new tools and languages.

So while the ability to codify patterns as generic software components may be important, even more important is the *knowledge of how/when to apply and combine patterns*, in conjunction with the *ability to use a shared vocabulary of pattern names* to communicate the nuggets of insight they represent. Because patterns capture knowledge that is primarily intended for *humans*, it is the *social* impact of patterns which largely shapes their technological impact.

What about Christopher Alexander? His writings on patterns and pattern languages were composed more than two decades before they became popular within the software community. During these past few decades, Alexander has been laboring on a new book entitled **The Nature of Order** which significantly advances his earlier ideas about patterns, architecture, and beauty. This new work will span four volumes, some of which are tentatively planned for publication by the end of 1997. In describing the book, one of its editors, Nikos A. Salingaros writes:

> Alexander develops a comprehensive theory of how matter comes together to form coherent structures. Paralleling, but not copying, recent results from complexity theory, he argues that the same laws apply to all structures in the universe; from atoms, to crystals, to living forms, to galaxies.

This book encompasses all the concepts and theories discussed in Alexander's earlier works and extends them to develop a much broader paradigm for creating architectures. It discusses a new concept of wholeness that emerges from fields of *centers* (which are constructed from small pattern languages) with *universal recursive properties*, and from dynamic processes called *sequences* (which relate the order in which to visit centers and apply patterns) that employ *structure-preserving transformations*. Alexander presents his new architectural paradigm with concepts like:

- the degrees of life and how life comes from wholeness
- the fifteen fundamental properties of wholeness and life
- the principle of unfolding wholeness
- structure-preserving transformations
- the fundamental process for creating life
- achieving comfortable, habitable living spaces
- defining and designing essential centers for working out structure
- the emergence of living order and the face of god

Many in the patterns community are awaiting publication of this book with enormous anticipation as it promises to be a work of monumental significance for all walks of architecture.

Concluding Remarks

All mature engineering disciplines draw from a collective compendium of time-honored, battle-tested "best practices" and "lessons learned" for solving known engineering problems. Great engineers don't just design their products strictly according to the principles of math and science. They must adapt their solutions to make optimal trade-offs and compromises between known solutions, principles, and constraints to meet the ever-increasing and ever-changing demands of cost, schedule, quality, and customer needs. Patterns help bring order out of chaos by identifying what is constant and recognizable in the midst of such incessant change. In this sense, patterns appear to resemble *strange attractors*, the convergence of dynamically interacting components into stable configurations, that recur all throughout successful systems.

Patterns represent distilled experience which, through their assimilation, convey expert insight and knowledge to inexpert developers. They help forge the foundation of a shared architectural vision, and collective of styles. If we want software development to evolve into a mature engineering discipline, then these proven "best practices" and "lessons learned" must be aggressively and formally documented, compiled, scrutinized, and widely disseminated as patterns (and anti-patterns). Once a solution has been expressed in pattern form, it may then be applied and reapplied to other contexts, and facilitate widespread reuse across the entire spectrum of software engineering artifacts such as: analyses, architectures, designs, implementations, algorithms and data structures, tests, plans, and organization structures.

Patterns are *not* a "silver bullet"! They *are* extremely valuable tools for capturing and communicating acquired knowledge and experience to improve software quality and productivity by addressing fundamental issues in the development of software. By employing these tools we are better suited to meet challenges like "communication of architectural knowledge among developers; accommodating a new design paradigm or architectural style; resolving nonfunctional forces such as reusability, portability, and extensibility; and avoiding development traps and pitfalls that have traditionally been learned only by experience." (quoted from the PLoP'96 Call for Papers.)

Perhaps the final remarks from **Pattern-Oriented Software Architecture** best describe the significance of patterns for software:

> Patterns expose knowledge about software construction that has been gained by many experts over many years. All work on patterns should therefore focus on making this precious resource widely available. Every software developer should be able to use patterns effectively when building software systems. When this is achieved, we will be able to celebrate the human intelligence that patterns reflect, both in each individual pattern and in all patterns in their entirety.

Further Information

Some good URLs for learning more about patterns are:

- Doug Lea's "Patterns-Discussion FAQ"
  http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html
- Doug Lea's paper, *"Christopher Alexander: An Introduction for Object-Oriented Designers"*
  http://gee.cs.oswego.edu/dl/ca/ca/ca.html
- Dirk Riehle and Heinz Zullighoven's "*Understanding and Using Patterns in Software Development*"
  http://www.riehle.org/1996/TAPOS-96-Survey.html
- Doug Schmidt and Ralph Johnson's introduction to the October 1996, CACM Special Issue on patterns
  http://www.cs.wustl.edu/~schmidt/CACM-editorial.html
- Excerpts from Jim Coplien's SIGS management briefing **Software Patterns**
  http://www.sigs.com/books/wp_patterns_5pp.html

- Jim Coplien's paper "*Software Design Patterns: Common Questions and Answers*"
  ftp://st.cs.uiuc.edu/pub/patterns/papers/PatQandA.ps
- John Vlissides' article "*Patterns: The Top 10 Misconceptions*" in the March 1997 Object Magazine Online
  http://www.sigs.com/publications/docs/objm/9703/9703.vlissides.html
- The "History of Patterns" on Ward Cunningham's WikiWiki Web
  http://c2.com/cgi-bin/wiki?HistoryOfPatterns
- "Pattern Definitions" from the Patterns Home page
  http://hillside.net/patterns/definition.htm
- Steve Berczuk's *"Finding solutions through pattern languages"*
  http://world.std.com/~berczuk/pubs/Dec94ieee.html
- *"Some Notes on Christopher Alexander"*, by Nikos A. Salingaros
  http://www.math.utsa.edu/sphere/salingar/Chris.text.html
- *"Design Patterns: Elements of Reusable Architectures"*, by Linda Rising
  http://www.agcs.com/techpapers/patterns.htm
- Brian Kurotsuchi's Design Patterns Tutorial
  http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/
- Ravi Palepu's *"Modelling the Real World: Application of Patterns to Reduce Complexity in the Software Development Process"*
  http://www.scs.carleton.ca/~palepu/pat.html
- Doug Schmidt's "Pattern Writer's Workshop Tutorial"
  http://www.cs.wustl.edu/~schmidt/writersworkshop.html
- Ward Cunningham's "Tips for Writing Pattern Languages" on the WikiWiki Web
  http://c2.com/cgi/wiki?TipsForWritingPatternLanguages
- "A Pattern Language for Pattern Writing" by Gerard Meszaros and Jim Doble
  http://hillside.net/patterns/Writing/pattern_index.html
- Richard Gabriel's article *"Developing Patterns Studies"* from InfoWorld on-line
  http://www.infoworld.com/cgi-bin/displayArchives.pl?dt_iwe05-97_72.htm
- A "Patterns BookList" on the WikiWiki Web
  http://c2.com/cgi/wiki?BookList

Other more general patterns resources on the web are:

- The Patterns Home Page
  http://hillside.net/patterns/patterns.html
- The Portland Pattern Repository
  http://www.c2.com/ppr
- Ward Cunningham's wonderful WikiWiki Web
  http://c2.com/cgi/wiki?WelcomeVisitors
- Patlets FrontPage - a Patterns Database
  http://hillside.net/patterns/patlet/?FrontPage
- Cetus Links: Patterns, hundreds of links to pattern-related pages
  http://www.objenv.com/cetus/oo_patterns.html
- Brad Appleton's "Software Patterns Links"
  http://www.bradapp.net/links/sw-pats.html
- AG Communications Systems Patterns Pages
  http://www.agcs.com/patterns/
- The OrganizationPatterns FrontPage
  http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns
- The "Design Patterns Mailing Lists by thread"
  http://iamwww.unibe.ch:80/~fcglib/WWW/OnlineDoku/archive/DesignPatterns/
- The "Organization-patterns Mail Archive by thread"
  http://www.bell-labs.com/~cope/Patterns/organization-patterns-archive/

Acknowledgements

This paper is *not* an original work, but rather a synthesis of material from many knowledgeable and well respected members of the patterns community. Much of what was written here is either adapted or quoted directly from the writings of Jim Coplien, Doug Lea, the Gang of Four and the Siemens Gang of Five, and of course Christopher Alexander. Michael Beedle provided me a wealth of material both from the writings of Alexander and from his soon to be published book about using patterns for re-engineering development processes. His numerous comments and those of Linda Rising were immensely helpful. Thanks also to John Vlissides for his helpful comments, and to Desmond D'Souza and Tim Ottinger for sharing their thoughts about frameworks.

About the Author

Brad Appleton lives in the Chicago area and is a senior software engineer with Motorola Cellular Infrastructure Group in Arlington Heights, IL. His first exposure to software patterns was Peter Coad's article in the September 1992 issue of *Communications of the ACM*. He has been trying to "keep up" with the patterns movement ever since. Brad received his BS in Mathematics and Computer Science from the University of Michigan in 1988, and is currently working towards an MS in Software Engineering. His other professional interests include object-oriented design, software configuration management, and software process improvement. He may be reached via e-mail at <*brad@bradapp.net*>.

Index of Terms and Concepts

The following terms and concepts were defined and used in this paper:

Brad Appleton <*brad@bradapp.net*>
http://www.bradapp.net/