

# A Unified Theory of Software Patterns

Michael A. Beedle, e-Architects Inc.  
beedlem@e-architects.com

**Abstract.** A unified theory of software patterns is presented through the unification of other existing pattern theories. The resulting new theory resolves two issues that practitioners often find with software patterns and pattern languages: 1) Lack of specificity of software patterns, 2) Lack of specificity in the joint-points in pattern languages. The theories unified in this paper are: 1) the General Pattern Theory from Grenander, 2) the Lepus software patterns theory by Eden and co-workers, and 3) the Alexanderian pattern theory. This unified theory defines commonalities for 1) pattern structure, 2) invariance of relationships on compositions of pattern instances, 3) comparative measurements on candidate patterns, and 4) pattern composition to form functionally and morphologically complete wholes.

## 1. Introduction

The programming community has seen the rise of patterns as a new systems architecting paradigm that has expanded the capabilities of practitioners around the world. This is especially true for object-oriented practitioners, since it was this community who first advocated patterns [2], and organized directed organizations and conferences to expand the usage of patterns worldwide i.e. the Hillside Group [1].

However, while it is true that this group strongly advocated pattern usage, also, from its very early beginnings, this organization chose purposely to de-emphasize the development and acceptance of a rigid academic papers and structures, including any or all formal methods. This measure was taken to bring patterns to the masses and to avoid the elitism that sometimes is often seen in academic circles. This has been a good approach since patterns are now a practical technique that has benefited many.

However, the end result has been that while these anti-theory, anti-formality and anti-structure policies have worked well for practitioners, because the pragmatic expansion in usage is been explosive and revolutionary, they haven't allowed the patterns community to develop a common unified understanding of what patterns are. The answer of the patterns community to this problem, are rules-of-thumb and guidelines like: Does the proposed pattern have structure and behavior defined? Does it connect to other patterns from the context and resulting context? Does it have good forces defined? Etc. These are good guidelines that do have value for the practitioners but that do not allow concise and objective documentation of patterns.

Clearly, without a deeper understanding of these concepts, the patterns' movement, even though very successful when measured in practitioners numbers, is at risk of falling into the predatory game of subjective arguments among practitioners and academics.

In summary, the problems confronted now by the software patterns community are:

- 1) There isn't a formal theory available that explain how patterns work from first principles in the space of software, and this creates several problem:
- 2) There are anomalies in the current de facto theory, the Alexanderian theory of patterns
  - a) Lack of specificity of software patterns,
  - b) Lack of specificity in the joint-points in pattern languages.
- 3) Patterns discussions, evaluations and specifications are subjective.

From the Kuhnian perspective, the Alexanderian theory has become "normal science" for software patterns because practitioners and pattern writers accept it for guidance and inspiration to write software patterns, patterns are expected to be grouped in pattern languages, and a certain unspecified quality is expected when applying such patterns.

In other words, this "normal science" brings its own "paradigms" and its own "anomalies", things that this paradigm is unable to explain.

However, if the paradigm repeatedly fails to explain above anomalies, a "crisis" ensues and alternative theories must be developed. This paper presents one such alternate theory, but encourages others to propose other competing alternate theories than can prove to be successful in explaining these or other anomalies. In time, these alternate theories can be replace the old paradigm by completing a "scientific revolution" in the field of software patterns.

In this paper we present a *Unified Theory of Software Patterns* that is the combination of three existing pattern theories

- 1) General Pattern Theory from Grenander [3-4],
- 2) The Alexanderian pattern theory [6-10]
- 3) Lepus software patterns theory by Eden and co-workers [5].

We unify these theories by merging many of the concepts that each one provides.

## 2. Brief Introduction to Other Theories

In the last section we advertised a new theory that unify these other theories:

- 1) GPT, General Pattern Theory from Grenander,
- 2) APT, Alexanderian pattern theory, and the
- 3) LSPT, Lepus software patterns theory by Eden and co-workers.

But what are these theories and what challenges present to a unified theory?

The General Pattern Theory from Grenander is a formal abstract mathematical theory that relies heavily on Group Theory. The challenge is to translate the abstract symbolisms to the software space, to synergize their meanings among the established patterns' vocabulary, and to formally merge its concepts with concepts of the other theories.

The Alexanderian pattern theory, is a theory that has its roots in the architecture of buildings, but that extends nicely to other domains. However, the challenge to provide an accurate translation into the space of software is to interpret this theory correctly and to make the proper arrangements in the space of software to account for its interpretation. Also, this theory tends to be less formal, so the challenge is to formalize its concepts according to the other more formal theories concepts and constructs.

The Lepus software patterns theory by Eden and co-workers is the only formal software patterns theory available today [5]. It allows the specification of patterns through Lepus formulae that serve to constraint specific programming constructs like classes and objects structures and behaviors, to work according to the specified constraints. To unify this theory, its programming abstractions will need translation into the abstractions of the other theories.

## 3. Anomalies

While the Alexanderian theory of patterns provide a general framework for software patterns it does have some anomalies as perceived by practitioners and theorists:

- 1) Lack of specificity of software patterns. Lepus, defines a set of rules that the patterns must conform to, and therefore better defines what cannot change in the structure of a pattern. The reverse is also true. By specifying what cannot change, we get a precise definition of what can change [5].

2) Lack of specificity in the joint-points in pattern languages. However, using Grenander's connectors and generators, it is possible to specify how exactly structure is linked together, because patterns become generators at a higher level that are linked through valid connectors [3,4]

## 4. Structural View of the Unified Theory

From a structural and more technical point of view here are some considerations used by the unified theory.

### 4.1 Defining comparative measurements on candidate patterns

Goodness of fit. Goodness of fit is defined as the solutions inside the region defined by misfit variables that correspond to a solution to a problem in a context. This is what gives "goodness of fit" among competing solutions. [6] (NOTES)

The mechanism used in Alexander's work Notes of the Synthesis of Form [6], is used to evaluate the fitness of a solution to solve a problem.

Therefore:

***APT offers a formalism to select pattern candidates that provide a "solution to a problem in a context", that allows the introduction of a relationship of pattern with a human need.***

### 4.2 Defining Pattern Structure

Pattern Structure. Alexander says that patterns must define a structure and that this structure has parts that can change and invariant parts. Therefore, the formal, but flexible description from LEPUS, to describe the invariants of a pattern, is used. Basically LEPUS defines invariants that pattern instances conform in terms of LEPUS formulas that bind the elements of patterns to conform to structural relationships. This is perfectly compatible with Alexanderian theory because Alexanderian theory does not specify a way to define structure. (LEPUS) [5]

The Lepus formulas already define graphs in terms of programming constructs but GPT is expressed in terms of abstract generators and connectors.

For example, Lepus participants are functions, classes, objects; and relationships are things like inheritance, associations, and ground relations among the participants that capture the structure and behavior in a pattern. Behavior includes behavioral relations like: *c is the first argument of f*", "*f<sub>1</sub> invokes f<sub>2</sub>*", "*f<sub>1</sub> forwards the call to f<sub>2</sub>*", "*f is defined in c*". The set of relations is subject to extensions in LSPT. Nonetheless, every ground relation can be mapped to a canonical, straightforward implementation in GPT through a generator. This satisfies the mapping of a LePUS formula to GPT.

All **correlations** of interest between *functions*, *classes*, and *hierarchies*, and sets of any dimension, extend **systematically** from the *ground relations* by two generalizations: *total* and *regular*, that can be modeled in GPT.

Finally, converging *regular relations* can be traced to isomorphisms that **commute** in both LSPT and GPT through the mapping of connectors and generators.

Therefore:

***The participants in LSPT are used as generators in GPT and  
The relationships in LSPT are used as the connectors in GPT***

#### 4.2.1 Defining invariance of relationships on structural components

1. The invariance in GPT defined in terms of Group theoretical Equivalence relationships and symmetry groups is traced to LSPT Lepus formulas.
2. The Lepus formulas in LSPT are made equivalent to what is invariant for a pattern in APT.

Let a and b are programming changes in terms of the programming language, for example for OO programming, in the set of Lepus abstractions:

Functions  
 Uniform sets  
 Inheritance class hierarchies  
 Clans  
 Ground abstractions  
 Extensions  
 Correlations  
 Regular functions

Where:

- x is the binary operation of the groups defined as the multiplication of transformations
- i is defined as the identity or "do nothing" program transformation
- e == belongs to
- G is the group
- Symm is the symmetry group

Programming changes can be indeed a group because they satisfy the following 4 conditions.

#### 1) Binary Operation

$$a \times b \in G$$

- "any two programming changes still belong to the set of programming changes"
- and a programming changes are bijections because, a programming change always leads to a unique modified program.

#### 2) Associativity

$$a \times (b \times c) = (a \times b) \times c$$

"program transformations are associative"

#### 3) Identity

$$i \times a = a \text{ and } i \in G$$

"there exists a 'do nothing' program transformation and it is unique"

Managers don't like it :-)

#### 4) Inverse

for each  $a \in G$ , there exists  $a^{-1}$  such that

$$a \times a^{-1} = e$$

"for any programming change you can program another programming change that will nullify the effects of the first one"

The resulting groups are not Abelian since  $a \times b \neq b \times a$  for all elements of the Group.

This shows that valid programming changes form a group in the set of valid programs in a programming language.

Now define the Symmetry Group, *Symm*, of a pattern implemented in many programs, as the equivalence relationship  $\sim$  among any two pattern implementations  $p$  and  $q$ , such that:

$p \sim q$ , (p is equivalent to q)

if and only if,  $q$  satisfies a Lepus formulae. This relationship can be proved to be reflexive, symmetric and transitive.

In turn, this Equivalence relationship defines isomorphisms on pattern instances, that is bijective transformations, among pattern instances. And it also defines the set of programming changes that a set of programs can undergo while still maintaining the set of constraints and invariants defined in the pattern's Lepus formula.

Therefore:

***The Lepus formula can be directly traced to the symmetry group and the equivalence relationship, and the different ways to identify all programs that implement a pattern in GPT and LSPT are equivalent. Pattern instances in LSPT are considered to implement the same pattern if they satisfy the constraints of the LSPT Lepus formula. This is equivalent to satisfying the equivalence relationship and symmetry group constraints in GPT.***

For the second statement we give a verbal argument, since Alexanderian theory is defined verbally.

Alexander writes:

"Each one of these patterns is a morphological law, which establishes a set of relationships in space.

This morphological law can always be expressed in the same general form:

$X \rightarrow r(A, B, C, \dots)$ , which means:

Whithin a context of type  $X$ , the parts  $A, B, \dots$  Are related by the relationship  $r$ ."

[11 – pg. 90]

This relationships that APT states is made identical to the Lepus formula in LSPT.

Therefore:

***The Lepus formulas in LSPT are made equivalent to what is invariant for a pattern in APT.***

#### 4.3 Pattern composition in Pattern Languages

Resolution of Layered Forces. Resolution of Layered Forces is the filtering of unresolved forces linkage in a hierarchy of solutions that are resolved in layers. This leads to a proto-structure of linked problem/solutions [6], that do not include problems created by the solution (NOTES). Because of that, Alexander factorizes the "requirements" or problems into a structure of its own that has proven to be

erroneous over time. He in fact corrects this problem by using the results of the BART experiment. However.

Pattern Definition and Pattern languages. The triad Context/Problem/Solution defines a pattern. This was primarily the structure adopted as the results of the BART experiment were understood by Alexander because the concept of a pattern, includes in its solution the problems introduced in the domain space by the application of other patterns. This leads to the self-consistent structure of contexts/problems/solutions in layers that have already gone through the process of resolving mutual problems in a self-consistent way. This structure is known formally as a pattern language (BART, TTWOB [7]).

Morphologically and Functionally complete. In addition pattern languages must ensure the generation of Morphologically and Functionally complete wholes (TTWOB) [7]. To accomplish this in a defined way, the formal but flexible description of generators and generator connectors from Grenander [4], is applied to software to define "linked structure" in software layers.

However, three modifications are needed in Grenander's theory to do this: 1) allow software elements to be generators and connectors (as presented in the paper above), 2) make the equivalence that patterns are defined in the context of Lepus formulas, in other words, connectors and generators define pattern instances, with its invariant structure defined by Lepus formulae, and 3) ignore the image layer and use the structure layer in Grenander's theory to directly define pattern structure. (GRENANDER) [3,4]

Solution Overlapping. Solution Overlapping is the allowance of overlapping solutions in a pattern language. This concept was introduced in the paper "A city is not a tree", and allow overlapping and multiple-connectedness of the solutions (CITY) [11].

Differentiating Space. The rules for Differentiating Space, from Alexander (TTWOB)[7], are included. These rules specify that there are rules for choosing sequences of patterns from a pattern language and that only by choosing valid sequences can "valid" structure be generated (TTWOB)[7].

APT argues that patterns can be composed in pattern languages. Because APT does not formally define how these compositions should be made, the GPT formalism is used to allow:

any generator available that satisfies the constraints of a Lepus formula can participate in multiple patterns

GPT argues that once this is done, that patterns themselves are considered generators [5].

Therefore:

*The pattern algebra defined in GPT can be made identical to pattern language concept in APT.*

## 5. A new paradigm of programming: Pattern-Oriented Programming Languages

One of the most interesting aspects of this theory is that once the anomalies are solved it leads to the possibility of creating pattern-oriented programming languages. In our case, we have used LISP to prototype such a language using:

- 1) two new macros defpattern and instantiate-pattern.
- 2) A "behind the scenes" repository of information we call the Meta-Pattern Protocol, paralleling the Meta-Object Protocol in CLOS.

Defpattern takes a name and an LEPUS formula, and generates named operands behind the scenes, so that structures can be linked together at a later time. Behind the scenes, there is a registry of such patterns and structures that keep track of the invariant relationships. This is where the structural pattern theory from Grenander is applied: these pattern definitions are used as generators to create pattern instances, and these generators have the ability to combine to form other patterns. This is a MOP like structure, that instead of keeping classes and their relationships like in a MOP (meta-object protocol), it stores patterns and their relationships. We call this registry of set of structures the meta-pattern protocol, or MPP.

instantiate-pattern, takes a pattern name, the same identifier used to define the macro, and a series of parameters, probably data, function names, class names, and even pattern instance names. instantiate-pattern, creates an instance of the pattern using these parameters, but it also makes sure that the programming elements involved in the pattern are also tracked with some structures, that they adhere to the pattern specifications, and that the pattern instances linkages are kept. This of course, requires some LISP language modifications/extensions, because it happens "behinds the scenes".

The custom code that instantiate-pattern takes as parameters is regular ole-fashioned LISP code: functions, lambdas, classes, methods, other pattern instances, etc.

A program, at a high level is simply a collection of pattern instances working together. The advantage in programming this way is that the overall structure of the program, the application architecture, is driven by pattern instances based on pattern definitions. These pattern instances use regular programmer-supplied code, and language extensions. The end results is that the program structure matches the literary form of a pattern language. This unity of programming has the advantage of matching:

- 1) the knowledge representations of the literary form of a pattern,
- 2) its actual implementation in programs and
- 3) the concept of a pattern in the programmer's brain.

Therefore, it is "literal" pattern programming, in Knuth's terminology. In terms of architecture, pattern-oriented programs are layered like this:

patterns instances
pattern definition
custom supplied code (classes, functions, etc.)
LISP language extensions (all the stuff to keep the MPP)
regular LISP language constructs

In addition, this paradigm of programming has the additional value of storing pattern representations as accumulated knowledge.

Pattern-oriented programming as presented here, should not be confused with code generation or the "automation" or the application of patterns because the programmer still has to 1) think which patterns to use and 2) supply the custom code to make its program work.

In other words, the fact that a LISP programmer would immediately start writing (defun...), (defmacro...), or (defclass...) or (defpattern...), etc.; does not make the literary form of a pattern unnecessary, or does not imply that the goodness or QWAN, as required by the Alexanderian paradigm, of a pattern is gone, does not mean that there isn't any custom programming, or does not mean that one has to "stop thinking".

Quite the opposite, it simply complements the literally form of patterns with the patterns actually used in a program. In other words, it makes it easier to think in terms of patterns and to apply them, because it unifies all forms of patterns: the literally form, the structure(s) found in the program, and the cognitive models in the programmers

## 6. Conclusions

This unified theory provides a minimal set of definitions to formally define 1) pattern structure, 2) comparative measurements on candidate patterns, and 3) pattern composition to form pattern languages. Also, this theory is used to propose pattern-oriented programming languages.

## References

1. Hillside group <http://www.hillside.net>
2. GOF, *Design Patterns*., Addison and Wesley, Boston, 1994.
3. Ulf Grenander, *Elements of Pattern Theory*, John Hopkins University Press, Baltimore, MA, 1996
4. Ulf Grenander, *General Pattern Theory : A Mathematical Study of Regular Structures*, Oxford Mathematical Monographs, Oxford, 1993.
5. Amnon H. Eden Joseph (Yossi) Gil' Yoram Hirshfeld, Amiram Yehudai, *Towards a Mathematical Foundation For Design Patterns*, Technical report 1999-004, Dept. of Information Technology, Uppsala University.
6. NOTES, Christopher Alexander, *Notes on the Synthesis of Form*, Oxford University Press, 1963.
7. TTWOB, Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979
8. APL, Christopher Alexander, *A Pattern Language*, Oxford University Press, 1977
9. TOE, Christopher Alexander, *The Oregon Experiment*, Oxford University Press, 1975
10. NOO, Christopher Alexander, *The Nature of Order*, Oxford University Press, (to be published)
11. CITY, Christopher Alexander, *A City is Not a Tree*, *Architectural Forum* 122 April (1965): No. 1, pages 58-61 and No. 2, pages 58-62. Reprinted in: *Design After Modernism*, Edited by John Thackara, Thames and Hudson, London, 1988; and in: *Human Identity in the Urban Environment*, Edited by G. Bell and J. Tyrwhitt, Penguin, 1992.
12. BART (See bibliography in:) Stephen Grabow: *Christopher Alexander: The Search for a New Paradigm in Architecture*, Oriol Press, Stocksfield, England, 1983.