

# 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 theories considered 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 structural components within a pattern and pattern instances, 3) comparative measurements on candidate patterns, and 4) pattern composition.

## 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.

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.

Simply stated, there is a lack of formal theories that explain how patterns work from first principles in the space of software, and this creates a problem:

*Patterns discussions, evaluations and specifications are now very subjective.*

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.

On the other hand, there are interdisciplinary patterns' theories with incredible insights and concepts that can crossover to software like:

- 1) General Pattern Theory from Grenander [3-4],
- 2) The Alexanderian pattern theory [6-10]

that lack careful formal translations to software space. Also, these theories lack formalized synergies with existing software patterns' theories like the Lepus software patterns theory by Eden and co-workers [5].

This paper presents a *Unified Theory of Software Patterns* built from these theories by merging many of the concepts that the other theories provide.

## 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. A Unified Theory of Software Patterns

The unification of these theories has been summarized in the following 4 points:

### 1. Defining pattern structure

1. The participants in LSPT are used as generators in GPT
2. The relationships in LSPT are used as the connectors in GPT

### 2. Defining invariance of relationships on structural components within a pattern

1. 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.
2. The Lepus formulas in LSPT are made equivalent to what is invariant for a pattern in APT.

### 3. Defining comparative measurements on candidate patterns.

1. 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. Defining pattern composition:

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

### 3.1 Defining Structure

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, straight-forward 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***

### 3.2 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 x b e 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

$$ax(bxc) = (axb)xc$$

"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:

$$p \sim q, \text{ (p is equivalent to q)}$$

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, .... 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.***

### 3.3 Defining comparative measurements on candidate patterns

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.***

### 3.4 Defining pattern composition

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.***

## 3. Conclusions

This unified theory provides a minimal set of definitions to formally define 1) pattern structure, 2) invariance of relationships on structural components within a pattern and pattern instances, 3) comparative measurements on candidate patterns, and 4) pattern composition. Future work of this theory could include more unified or non-unified aspects of each of these theories but this provides a minimal working set.

## 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. Christopher Alexander, *Notes on the Synthesis of Form*, Oxford University Press, 1963.
7. Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979
8. Christopher Alexander, *A Pattern Language*, Oxford University Press, 1977
9. Christopher Alexander, *The Oregon Experiment*, Oxford University Press, 1975
10. Christopher Alexander, *The Nature of Order*, Oxford University Press, (to be published)