A Theoretically-based Process for Organizing Design Patterns*

Sargon Hasso and C. R. Carlson
Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois 60616, USA
{hasssar,carlson}@iit.edu

Abstract

Since their introduction, hundreds of different design patterns have been discovered and documented to address a variety of problems we encounter in software design and construction. However, to use any one of these design patterns effectively and quickly, one must have access to a design pattern definition that unambiguously gives away its intended purpose, and find an easy way to choose one design pattern from among a family of related design patterns. Existing research is either lacking or has not addressed these issues adequately.

We propose a new process by which we can organize design patterns and allow for easy identification and differentiation among them. We rely on and adapt some linguistic theories to analyze design patterns; use some complex algebraic structures to structure hierarchical concepts that describe design patterns; and utilize some general and reliable classification theories that were used successfully in compiling dictionaries and thesauri of natural languages.

1 Introduction

Rising's [21] Almanac adopts a classification scheme that uses categories derived from domains in which the respective design patterns are used, e.g. Accounting, Interactive Systems, GUI Development, Persistence, Security, Time, etc. Furthermore, these categories are intermixed with designation suggested by original authors of these design patterns, e.g. creational, behavioral, architectural, structural, C++ idioms, Java idioms, etc. There are several problems with this approach:

1. This kind of classification scheme is inconsistent. For example, Accounting and Health Care (domain category); Creational, behavioral, and Structural (author's specific category); Design Process and Testing (methodology category); Smalltalk idioms, Java idioms, and C++ idioms (programming language category); Event-Driven Systems and Fault-Tolerant Systems (application category), etc.

2. Some of these categories overlap: Architectural and Client-Server; Air Defense subsumes Event-Driven and Fault-Tolerant Systems; others may be applicable to more than one category, e.g. Creational.

3. For a set of design patterns listed under one category, does that imply that these patterns are not applicable to other categories? It's true that some may be specific to one particular domain, but it would be very hard to imagine that all of these domains have unique problems that don't apply to other domains.

There must be a better way to organize design patterns such that we can eliminate these problems. We will propose a new classification technique and a process to apply it systematically to organize any collection of design patterns. As a corollary of the classification scheme, we will be able to solve this problem: given a set of related design patterns, how can we define them in such a way that they are distinguished from each other precisely and unambiguously? For example, Proxy [8], Remote Proxy, Protection Proxy, Cache Proxy, Synchronization Proxy, Counting Proxy, Virtual Proxy, Firewall Proxy [22], or Visitor [8], Default Visitor, Acyclic Visitor, Selective Visitor, Extrinsic Visitor, Extension Object [17].

In the published literature on design patterns, relationship to other design patterns is often described based on author's knowledge of the existence of other design patterns, even then it is based on author's intuition (which most likely is correct). Our proposed process will explicitly show the relationship between patterns based on a mathematical device that we will define, discuss, and use extensively with the aid of some examples.

This paper is organized as follows: in section 2, we survey some of the most relevant work which addressed, to some degree, design pattern organization problem. In section 3, we will introduce our approach and demonstrate it with some examples. Section 4, will discuss our main contribution. In section 5, we conclude our discussion, and finally, in section 6, we discuss our future work. Additional materials are given in the Appendices that augment the concept lattice presented in section 3, and we present a case study to demonstrate the results of applying our approach to organize a small group of design patterns.

2 Related work on the classification techniques of Design Patterns

The design patterns in Gamma et al. [8] are classified based on two criteria, namely: the *purpose* and the *scope*. However, the granularity of the classification scheme is very coarse-grained; no distinctions are explicitly made in a consistent and unambiguous way between related patterns; and a classification scheme based on relationships between single design patterns is too low-level and too specific to be useful (cf. Figure 1.1 [8, p. 12]).

Coad [5] uses a classification scheme that organizes patterns into four major subclasses: Transaction, Aggregate (structural), Plan (script), and Interaction (communication) patterns, and within each there are a number of patterns. While the approach is novel, it seems rather woefully incomplete, and the patterns within each major category are really instances of the major category. The only distinction is obtained via a role assignment to each participant in the pattern.

Adamczyk [1] classifies design patterns based on the types of problems they solve, such as: limited amount of memory, algorithmic dependency, complex scheduling algorithms, etc. However, the same design pattern may be listed under more than one category.

Zimmer [23] examines the nature of relationships between the design patterns to classify them. The classification is very similar to Alexander's approach [2] where one pattern can serve to pull in one or more related patterns. The approach was tried on the small list of patterns in Gamma [8] and it will take some hard work to try to relate the hundreds of patterns listed, for example, in [21]. Furthermore, it is not clear how two related patterns are similar or different, and in what way?

Jackson's [14, 15] uses a generic classification scheme intended for general software solution problems. His categories are very coarse-grained and general. Each of these problem classes can be designated by a problem frame, e.g. *Required behavior, Commanded behavior, Information display, Simple workpiece, and Transformation.* Most problems' analyses must somehow fall into these general problems. This scheme does not classify design patterns, per se, however, it warrants further research to see its utility to design pattern classification.

3 Linguistics-based Approach to Software Design Patterns Classification

We have evolved our approach to use techniques from linguistics and lattice theory and have come up with an integrative approach that seems to have worked for us. The result is a procedure that allows us to classify an arbitrary number of design patterns. We will describe each step in detail, citing along the way the relevant theory or technique that we used to develop this procedure. We will illustrate each step with examples.

3.1 Process Steps

Step 1: Design Patterns Selection There is a considerable body of literature that contains much of the design patterns we examined. Rising's almanac [21] lists a large number of them although no details are discussed. Therefore, much of the details are consulted by referring the source of the publication for much of these design patterns. Chief amongst them are: Gamma et al. [8], Buschmann et al. [4], the series of pattern languages books, four volumes in all, [6, 22, 17, 11]. Others sources are consulted as well.

Step 2: Attribute Identification Componential Analysis of meaning [18] is a technique that allows two or more words or related meaning to be compared and contrasted through a minimal set of contrasting features. The name of the feature or component is irrelevant as much as the type of values they contain. Generally, it is intuitive to recognize, given a set of related words, what makes them related and what makes them distinct from each other. This analysis method just formalizes this procedure.

Each well-documented design pattern is described with at least one or more general properties that essentially, when juxstaposed, makeup the definition of the design pattern. Generally, these properties are derived from the general behavior of one or more major components of each design pattern. These behaviors are in turn packaged into what we refer to in this proposal as a role. In doing so, these roles are well defined and have formal definitions in terms of operations they perform, messages they react to, and state, represented by attributes, they effect. In other words, these properties are distributed over the participant components of the design pattern. We also refer to these properties as features, cf. Övergaard [19].

In linguistics, the meaning of a lexical term is derived from different domains each of which contributes one or more components. In a similar manner for design patterns, the features constitute the components of each pattern and these may be derived from different domains. Example domains, which we will also refer to as semantic domains, in design patterns are: creation domain, i.e. features or properties responsible for instantiating or creating objects, communication domain, i.e. features responsible message travel, containment domains, i.e. features responsible for storing objects, etc. Of course, our reference to domain is linguistics-based and not software engineering- or business-centric such as telecommunication, accounting, real-time system, data-base, etc. Our linguistics based domains, for the purpose of design pattern semantics derivation, are more general and they cut across different software engineering- or business domains.

As an example, here is a brief description of the properties that we have extracted from Proxy related patterns, i.e. Proxy [8], Remote Proxy, Protection Proxy, Cache Proxy, Synchronization Proxy, Counting Proxy, Virtual Proxy, Firewall Proxy [22]:

Surrogate (s1) A general property of this category and it signifies substitution. More on this when we discuss predicate identification in step 3.1 below.

Access (a1) A general property of this category and it signifies an access to something.

Interface (f1) A property that designates an interface involvement.

Multiple access (s2) A property that describes the scope of access: multiple vs. non-multiple access.

Sharing (s3) This property designates access to a shared resource, e.g. a cache, a server, a variable.

Reference counting (s4) A count can be associated with a shared resource.

Delayed loading (s5) Access to a resource may be fully available or loaded partially on demand.

Relationship between real and surrogate (s6) The type of relationship between the real object and its substitute object can be one-one, one-many, or many-many.

Security (s7) A feature that denotes whether a security aspect is required or not for accessing real objects via their proxies.

It is to be noted that there are no rules on how to extract these properties, nor from what section of the documentation this information is gathered. Generally, key properties are clear from the text and with experience they can be extracted with relative ease.

	s1	a1	f1	s2	s3	s4	s5	s6	s7
PROXY (207)	x		x					one-one (s61)	
REMOTE PROXY	x							one-many (s62)	
PROTECTION PROXY	x	x		no (s20)					x
CACHE PROXY	x				x	no (s40)			
SYNCHRONIZATION PROXY	x	x		yes (s21)					
COUNTING PROXY	x				x	yes (s41)			x
VIRTUAL PROXY	x						x		
FIREWALL PROXY	x	x			x			many-many (s63)	x

Table 1: Properties of Proxy related Design Patterns. Legend: s1 surrogate, a1 access, f1 interface, s2 multiple access, s3 sharing, s4 reference counting, s5 delayed loading, s7 security, s6 relationship between real and surrogate.

Step 3: Relationships Construction We construct a cross-table to represent relationships between objects and their attributes. The intersection of an object and an attribute constitutes a relationship if one exists. Relationships can be as simple as certain objects have a specific property,

in this case we cross-check it; or yes/no to emphasize the presence or absence of the relationship; or a multi-valued relationship. There are several methods by which we can convert a multi-valued relationship into a single-valued relationship, cf. discussion in [9]. The purpose of this object-attribute cross-table will become an input to a software tool that will present the same information in graphical format that is more useful for further analysis as we will demonstrate in step 5 below. As an example, see Table 1 for Proxy related design patterns. Although not very apparent, this cross-table represents concept hierarchies which are critical in the proposed classification scheme. We will define and illustrate these concepts again in step 5 bellow.

Step 4: Predicate Identification Generally, one main concept seems to underlie the meaning of each design pattern. We assign a verbal predicate that best describes this concept. We then add this predicate as another property or attribute to our tables. Adding this concept as an attribute serves two purposes:

1. other design patterns that don't belong in the same category may also share this group with this attribute. This is the basic idea of the semantics of a design pattern is derived from from different domains, and,
2. a concept associated with a verbal predicate is the first level coarse-grained classification scheme.

Step 5: Concept Lattice Generation Looking at separate tables each with many objects and attributes may not be the best way to make conclusions about our analysis. Concept hierarchies are easier to extract from concept diagrams than from their tabular presentation. To formalize the analysis technique and be able to use software tools to automate the process quickly and easily, we use presentation and analysis tools based on the theory of ordered sets and lattice theory, namely Formal Concept Analysis (FCA) used as a formal model for analysis of hierarchies of concepts. Much of theoretical and formal theorems and definitions are treated in details in [9]. We will briefly introduce the main concepts here and use the results of the fundamental theorem of concept lattices directly since the application as a presentation format and analysis tool is much more interesting and software tools exist to construct concept lattices.

The model for conceptual hierarchies formalizes the traditional philosophical definition of a *concept* as a unit of thought determined by its *extent*, the set of all objects belonging to the concept, and its *intent*, the set of all attributes (features) shared by the objects. FCA investigates how objects can be hierarchically grouped together according to common attributes. The following three definitions, taken from [9], are the necessary background to understand the concept lattice, its construction, and how to read it:

Definition 1 *A Formal Context is a triple (G, M, I) where G is a set of objects, M is set of attributes, and a binary relation $I \subseteq G \times M$. If $g \in G$ and $m \in M$ are in relation I, we write $(g, m) \in I$ and say 'the object g has the attribute m'.*

One way of writing a formal context is by means of a cross-table, see Table 1, in which the objects, i.e. design patterns, are listed in row-headings, and the attributes, i.e. design pattern properties, are listed in column-headings. The intersection constitutes the ordered pair (g, m) and is cross-marked iff $(g, m) \in I$, e.g. (Protection Proxy, a1).

Definition 2 *(A, B) is a formal concept of (G, M, I) iff $A \subseteq G$, $B \subseteq M$, $A' = B$, and $B' = A$, where:*
$A' = \{m \in M \mid (\forall g \in A)\ gIm\}$
$B' = \{g \in G \mid (\forall m \in B)\ gIm\}$

A' is the set of attributes common to all objects in A, B' is the set of objects having the attributes in B. The set A is called the extent of the formal concept (A, B), and the set B is called its intent. The set of all formal concepts of (G, M, I) is denoted by $\mathfrak{B}(G, M, I)$

Definition 3 *Let (A_1, B_1) and (A_2, B_2) be formal concepts in $\mathfrak{B}(G, M, I)$. We say that (A_1, B_1) is a subconcept of (A_2, B_2) iff $A_1 \subseteq A_2$. Equivalently, we say that (A_2, B_2) is a superconcept of (A_1, B_1) iff $B_2 \subseteq B_1$. Using $\leq$ to express this relationship, we therefore have:*
$(A_1, B_1) \leq (A_2, B_2) \Longleftrightarrow A_1 \subseteq A_2 \Longleftrightarrow B_2 \subseteq B_1$.
The set of all formal concepts of (G, M, I), partially-ordered by this relation, is denoted $\underline{\mathfrak{B}}(G, M, I)$ and is known as as the concept lattice of the formal context of (G, M, I).

Figure 1 illustrate a Proxy related design patterns using concept lattice. The actual attributes and their meaning are illustrated in Table 1. Each node in the diagram represents a formal concept. Each attribute name (lower-case letters) is attached to exactly one node and drawn above the node; while each object (upper-case letters) is also attached exactly to one node and drawn below the node. Nodes connected by edges express concept order. Here is how to read the extents and intents of each concept from the diagram: Consider the diagram in Figure 1. For any object in the diagram, say CACHE PROXY, we can find out all of its attributes by following the *ascending paths* originating from itself and any time we encounter a node with attribute attached to it, it is part of that object. In this case it is the set {s40, s1, s7, s31}. Similarly, to find out all the objects that have a specific attribute, say s31, we follow all the *descending paths* originating from itself and recording any object we encounter in the path. In this its the set {COUNTING PROXY, CACHE PROXY}.

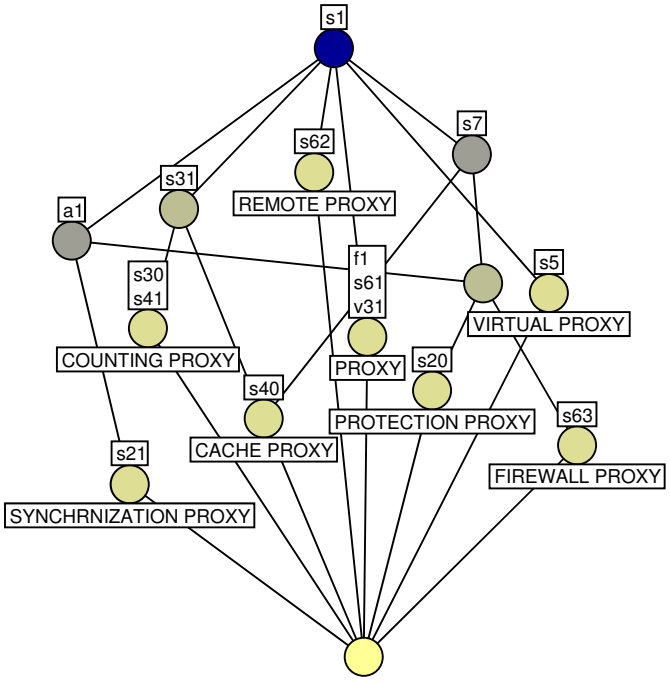

Figure 1: A sub-lattice diagram depicting Proxy related design patterns. Nodes annotated with upper-case letters denote design patterns; nodes annotated with lower-case letters denote attributes. Multi-valued attributes are converted into single-valued attributes, e.g. s30, s31 where s3 is the same attribute from Table 1, and the next digits are the results of multi-value into single-value conversion.

Step 6: Classification We will illustrate how concept lattice diagrams help us in classifying a group of related objects with common attributes. We showed in step 5 above how to read concepts off the concept lattice diagram. For example the lattice diagram in Figure 1 classifies the following concepts in the following format: {{design patterns set}, {attributes set}}

1. {{VIRTUAL PROXY, FIREWALL PROXY, COUNTING PROXY, PROXY, REMOTE PROXY, CACHE PROXY, PROTECTION PROXY, SYNCHRONIZATION PROXY}, {s1}}
2. {{FIREWALL PROXY, PROTECTION PROXY, SYNCHRONIZATION PROXY}, {a1, s1}}
3. {{FIREWALL PROXY, PROTECTION PROXY}, {a1, s1, s7}}
4. {{FIREWALL PROXY}, {s5, s63, a1, s7, s1}}
5. {{PROTECTION PROXY}, {s20, a1, s7, s1}}

It is worth noting that the higher-up in the concept hierarchy (upper nodes in the lattice structure), the more general the concepts are and the less distinctions are made between design patterns. For example, concept (1) has only one attribute that all of the design patterns share, viz. s1. As we descend in the hierarchy, we pick more attributes, become more specific, and drop more objects. Fewer objects have attributes in common, e.g. concepts (2,3). Only if we pick more attributes, therefore becoming more specific, are we able to distinguish between individual patterns, e.g. concepts (4,5).

In 1852 Roget published his *Thesaurus of English Words and Phrases, Classified and Arranged so as to Facilitate the Expression of Ideas.* The Thesaurus presents ideas arranged in certain way to reveal the list of words that can be used to express them [7, p. 266]. In this way, he was able to classify the ideas expressible by language in order to permit quick access to them [3]. Using some of Roget's ideas in classification, Figure 2 is the first level classification diagram for some design patterns based on verbal predicates. We intend to augment this figure, in future work, with additional conceptual hierarchies discussed in this section from formal concept diagrams.

Step 7: Definition Since the features or attributes are components of each design pattern, and these, in linguistics-based definitions, constitute a uniform way of describing each and every design pattern, we derive a definition for the design pattern from its components. The definition is simply the systematic description of the diagnostic features by listing all the values of the *intent* in our concept structure. Since the definition is generated from a sound analysis method, one can, in practice, develop a design pattern dictionary not unlike linguistic dictionaries. A good definition should be concise, precise, atomic, explicit, and unambiguous. A user of a design pattern dictionary does not have to go through pages and

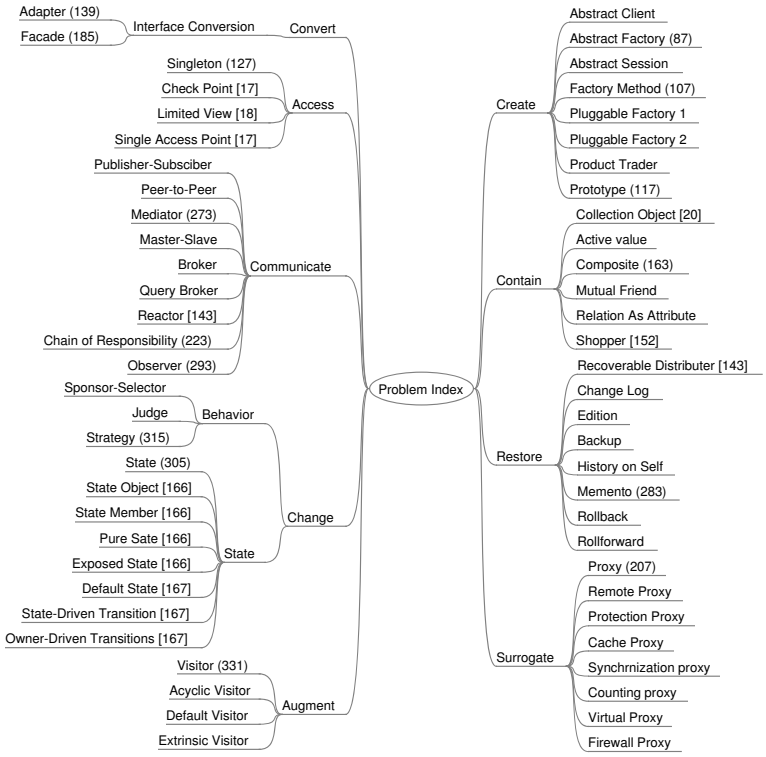

Figure 2: Linguistics-based concepts as the basis for Design Patterns classification.

pages of documentation to figure out the meaning of a design pattern. As an example, consider the definitions of the Connector and Acceptor design patterns [17], and contrast our proposed definition to Rising's [20]:

Proposed definition: CONNECTOR is a design pattern that decouples service initialization from services provided, used for synchronous and asynchronous connection, and it uses active initialization.
Rising's definition [20, p. 40]: "CONNECTOR decouples service initialization from services provided. This is a pattern for active initialization. ACCEPTOR is for passive initialization" [implicit reference that these two are somehow related].
"This pattern is similar to CLIENT-DISPATCHER-SERVER (CDS)." [again an implicit reference to relatedness] "This pattern addresses both synchronous and asynchronous service initialization, while CDS focuses on synchronous connection." [An important property not included in the definition].

Of course, Rising does not define design patterns in the same sense we do, but the *intent* and *related patterns* sections in her documentation is all the user has to go on. Our process will consistently produce a definition in the manner shown above.

4 Contribution

Bushcmann et al. [4, p. 423] states the need for an appropriate method to organize patterns as more design patterns are discovered. Almost all the published literature on design patterns does not address this important area of research. The common practice widely accepted today is to adopt Gamma's [8] classification method (which is quite coarse-grained and inadequate) with additional categories to fit the new patterns. For example, in addition to Creational, Structural, and Behavioral, we find categories like: Concurrency, Partitioning patterns [10], Integration patterns [13], etc. Rising [21] was the first who attempted to collect and catalog most of the publicly available or published design patterns in the literature in one place. However, a careful examination of the catalog reveals several shortcomings: the classification scheme adopted by Rising was not based on any known classification system thus creating the problem for finding the right design pattern amongst several hundreds quickly and methodically; there are several design patterns that are related but their differences were not easy to discern thus creating a problem in selecting one over another similar pattern; the general classification and cataloging framework, or lack thereof, does not allow us to extend the catalog easily.

Our contribution is the creation of an attribute-based, fine-grained classification process that more or less subsumes all other pattern organization schemes. It does this by normalizing all categories, others have used as classifiers, into a

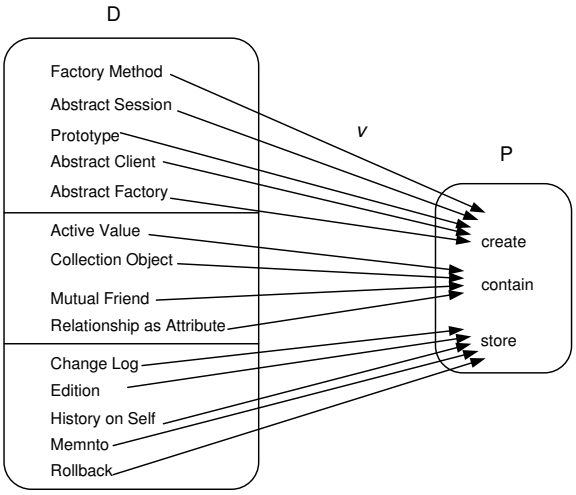

Figure 3: Design Patterns are sorted by a predicate(verb) property.

set of attributes or features. For example, all Concurrency patterns would have 'concurrency' as one attribute. In this way, the category itself becomes part of a self-contained, attribute-based pattern definition. Relationships between patterns are based on feature set which can be determined precisely rather than implicitly. Moreover, we may discover relationships that are not quite obvious if one considers hundreds of design patterns. The process can be applied to any arbitrary set of patterns to produce pattern catalog. It does, however, depend on a consensus of how pattern attributes are determined.

5 Conclusion

Since the first catalog of design patterns published by Gamma et al. [8], hundreds of design patterns were published by the researchers and practitioners. Unfortunately, while most of these patterns are highly useful, designers and other users could not practically nor realistically browse through them and narrow their searches quickly.

We proposed a systematic classification scheme and a process for doing it based on the theory of Componential Analysis of Meaning and Peter Mark Roget's classification scheme adopted in his thesaurus when it was first published in 1852. We enhanced these techniques and integrated them with formal concept analysis to give us a more formal way to do the analysis. In this scheme, we examined several design patterns and identified the one essential concept that seems to underly the problem it intends to address. Then, we associated a verbal predicate to this concept. This step allows us to achieve the first level coarse-grained patterns organizations. This point can be precisely expressed mathematically as follows: Given two sets, say D that contains design patterns,

and a set P that contains concepts represented by predicates (verbs); then the process of attributing (assigning) a design pattern to each concept is the map $D \xrightarrow{v} P$. In this case the *domain* is the set D, and the *co-domain* is the set P. Lawvere [16] refers to this map as *P-valued property* and has the net effect of producing a structure in the *domain* D. In plain language, we say that the map v gives rise to a *sorting* of the set of design patterns, D, into P sorts. Graphically, this mapping is illustrated in Figure 3. It is clear from this figure how the sorting is achieved and hence the desired classification. Additional steps allowed us to identify design patterns' attributes that became diagnostic features to distinguish one pattern from another. We also demonstrated how to define design patterns consistently.

6 Future Work

We have examined about 76 design patterns from Gamma et al. [8], Buschmann et al. [4], Coad et al. [5], the four PLoP books [6, 22, 17, 11], Hay et al. [12], Rising [21], and have applied our proposed technique to classify and organize these design patterns as a proof of concept. We intend to examine several more and produce a new design pattern catalog, or a dictionary, based on these ideas.

Much like what Roget did in his classification scheme for the thesaurus, the verb that we associate with each design pattern becomes an index into a sub-group of related patterns because this property is easier to identify fairly quickly. Continuing our search in the design pattern hierarchy, as we go to the next level, using the lattice as a classifier tool, we are not sure what to look for, but the lattice will help to differentiate the key properties in each pattern. In this way, it will become a forcing function to ask appropriate questions and get answers. We see this process leading us to develop a pattern locator technique. This technique will help us to conceptualize what we are looking for because this is how humans organize ideas.

We realize that many experts on design patterns may disagree on the definitions that we have created for each pattern, but we believe that a standards group within the pattern community could be formed that uses this process and gives us better definitions.

Acknowledgement

I would like to acknowledge the support I received from Dr. Miroslav Kis, of BMO Financial Group, for helping me re-shape and enhance some of the ideas discussed in this paper.

APPENDICES

A Concept Lattice as a classifier tool

We will illustrate the use of concept lattice with a more familiar example that demonstrates how a concept lattice diagram classifies an animal set based on some of their attributes. We will also go through some details that make the concept lattice as a classifier tool more comprehensible.

	mammal	fly	land	sea	lay-eggs	carnivorous	herbivorous
LION	x		x			x	
MONKEY	x		x				
SNAKE			x		x	x	
PIGEON		x	x		x		x
EAGLE		x	x		x	x	
ALLIGATOR			x	x	x	x	
FISH				x	x		
FROG			x	x	x	x	
TURTLE			x	x	x		x
CHICKEN			x		x		x

Table 2: A table representation of some animals and some of their features.

Table 2 is the cross-table representation and Figure 4 is its concept lattice diagram. Both table format and its corresponding concept lattice diagram convey the same information, but, as we will demonstrate with the help of a software tool, the concept lattice diagram is much superior in discovering related concepts and showing precisely how those concepts are related. In section 3 step 5, we have discussed how to interpret and read concepts from the diagram. There are several observations to notice in the diagram and we will go through some of them. The top most and the bottom most nodes are special concepts: the top most node represents the concept that corresponds answering the following question: what attributes do all animals in the set have in common? there are none. Therefore, using the textual concept representation first discussed in section 3 step 6, we have this concept:
{{FISH, , CHICKEN, EAGLE, MONKEY, ALLIGATOR, FROG, TURTLE, SNAKE, LION, PIGEON}, {}}
Similarly, the bottom most node represents the concept that corresponds answering the following question: which animal does have all the attributes in the set? there are none. This concept can be represented textually as:
{{}, {fly, sea, carnivorous, herbivorous, land, mammal, lay-eggs}}
Intuitively, we know that *some* animals share *some* attributes but not *all* attributes. And this is precisely the role of the intermediate nodes between the top most and bottom most nodes. Every node represents a concept. Nodes connected by edges express concept order. Intuition tells us that abstract objects

tend to represent general concepts, e.g. the concept land animals, represented formally as:
{{CHICKEN, EAGLE, MONKEY, ALLIGATOR, FROG, TURTLE, SNAKE, LION, PIGEON}, {land}},
says that all the animals listed can live on land. It does not tell us how the different land animals differ from each other. To do this, we must descend the concept hierarchy by adding more contrastive features. To show this, from the 'land' node, let us descend along the path connecting this node to the node labelled 'carnivorous'. This concept is represented as:
{{ALLIGATOR, FROG, SNAKE, LION, EAGLE}, {carnivorous, land}}
This clearly shows that as we add more attributes, fewer animals will share these (this relationship was formalized by Definition 3 in section 3). At this point, we are still incapable of differentiating between this smaller set of animals. To do this, we continue the previous exercise and follow the edges connecting this node to the next adjacent nodes in the downward direction. Here we have two options: 1) we can descend to the node labelled 'LION', the concept we encounter is this:
{{LION}, {mammal, carnivorous, land}}
This concept added another feature, mammal, that was enough to separate the LION from the other animals in the previous group. Formally, the reason we picked up the 'mammal' feature is because the node 'LION' was connected to another node, in the upward direction, hence it must have that feature (starting with an object node and traversing all the ascending paths from itself to the next node, and if that node is labelled with an attribute, it means this object node has that attribute). 2) we can descend to the node labelled 'SNAKE', and we have this concept:
{{SNAKE, EAGLE, ALLIGATOR, FROG}, {lay-eggs, land, carnivorous}}
By adding a new feature, 'lay-eggs', it reduced the number of animals that have this feature and it also eliminated the 'LION' from this concept. The two edges leaving 'carnivorous' node clearly shows this division. However, at this point, i.e. at 'SNAKE' node, the concept is still more general because we still don't know how to differentiate each individual animal in this new concept. But we can continue the process and come up with these concepts:
{{EAGLE}, {lay-eggs,land,carnivorous,fly}}
{{ALLIGATOR, FROG}, {lay-eggs, sea, carnivorous, land}}
Note that this last concept will not be able to differentiate between an 'ALLIGATOR' and a 'FROG' because we have no more features that set them apart. This was made deliberate just to show that in order in uniquely identify (by features) each object, we better go back and revise our feature list such that each object-labeled node has only one object attached to it. If for example, we add a new feature say, eating insects vs. subsisting or feeding on animal tissues (carnivore), we would have been able to set them apart.

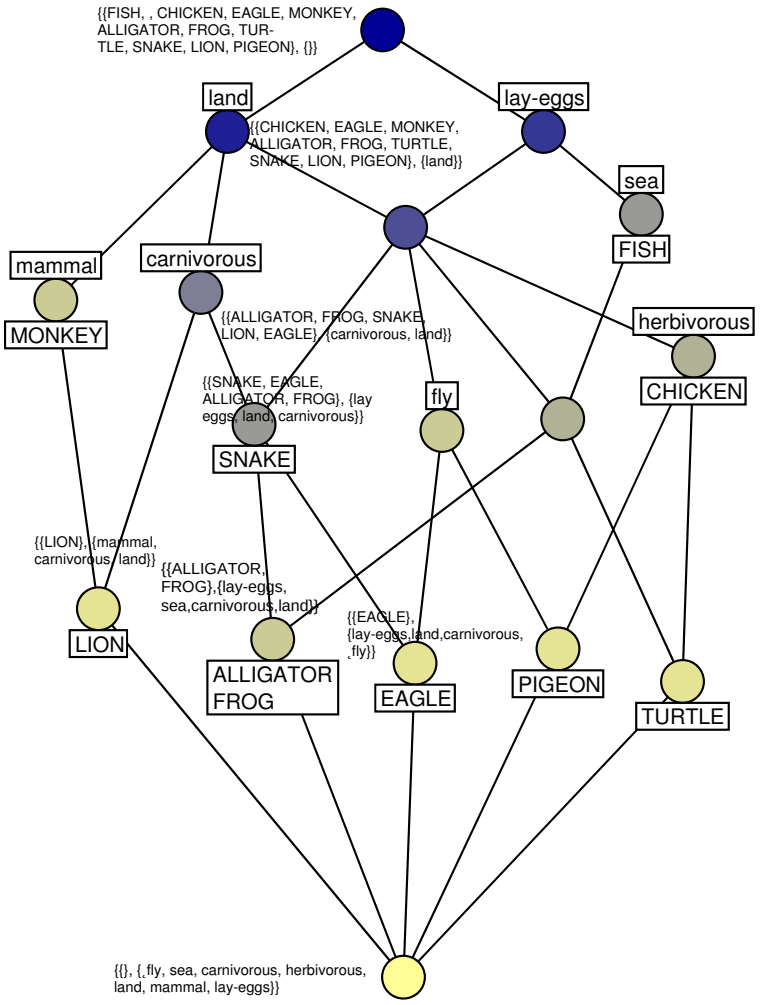

Figure 4: Animal classification using concept lattice diagram

B Case Study: classifying a small set of design patterns

In this appendix, we will demonstrate our process on a small set of design patterns. We will not show all the detailed steps in the process but one has to go through them in order to arrive at the final results. We will examine these 13 design patterns: STATE OBJECT, PROXY, TEMPLATE METHOD, STATE MEMBER, SINGLE ACCESS POINT, REMOTE PROXY, STATE, ADAPTER, FIREWALL PROXY, CHECK POINT, LIMITED VIEW, FACADE, STRATEGY. All of these design patterns are cataloged in Rising's Pattern Almanac [21]. Figure 5 is the concept lattice representation of the cross-table format, which we left out. The following attributes and a brief description for each were identified for each pattern:

Access (a1) A general property of this category and it signifies an access to something.

User Information (a2) Access to user information is encapsulated in an object that handles all user verification and security privileges.

An Object or Application (a3) Control the number of access points to an object or an application for security reasons.

Data (a4) Control the how much of data to expose subject to user privileges.

Change (c1) A general property of this category and it signifies changing some aspect.

Behavior change dependent on state change (c3) The state of the object determines object behavior.

Uses delegation to change behavior (c4) The behavior change of an object is delegated to another object.

Uses inheritance to change behavior (c5) Different subclasses implement different behavior.

State encapsulated in an object (c7) This feature has to do with how the state of an object is implemented: either the object state is part of the object itself, or the state is encapsulated in a separate object (state object) and this object becomes an attribute of the containing state object.

Switch between (c12) This feature determines the types of switched entities. In this case, it switches between states. Other switchers switch between operations.

Convert (v1) A general property of this category and it signifies conversion of something.

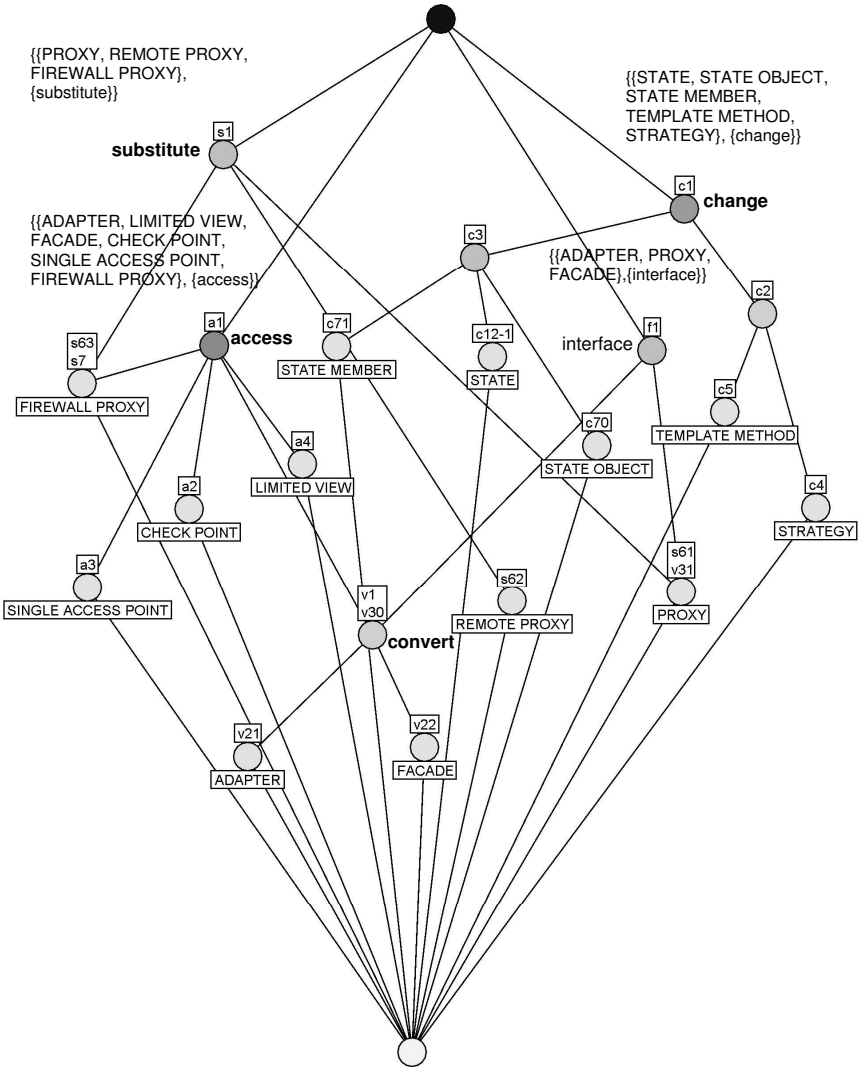

Figure 5: Lattice diagram for access, change, surrogate, conversion, and interface-type design patterns. For attribute names (small letters) and their meaning, please refer to the discussion in Appendix B.

Interface conversion (v2) Interface conversion type can be one-one, or one-many, meaning several interfaces are invoked to match one method or procedure call.

Interface type (v3) This feature denotes whether the interface is adapted is the same or different.

Interface (f1) This a general property to indicate that type of change has to do with interface or something else.

Surrogate (s1) A general property of this category and it signifies substitution.

Relationship between real and surrogate (s6) The type of relationship between the real object and its substitute object can be one-one, one-many, or many-many.

Security (s7) A feature that denotes whether a security aspect is required or not for accessing real objects via their proxies.

A multi-valued attribute are converted into single-valued attribute (this was discussed briefly in section 3 step 3). Nodes labeled like 's63', 'v30', etc. denote attributes 's6' and 'v3', respectively, but with different values. We use abbreviations for attribute names because they are too long and they would clutter the lattice diagram making it incomprehensible. Because of n-dimensional nature of the lattice structure, which a 2-D plane does not do justice, we use a software tool (Toscana is publicly available software tool) to generate and explore the diagram.

The lattice diagram allows us to quickly identify the main properties, in the way we define them, of each design pattern, and, most importantly, it allows us to see its relationship with other patterns. Two design patterns are related if they have at least one or more common features. By the same token, it also allows us to see the differences between a seemingly similar pattern, e.g. the Proxy patterns we discussed earlier in this paper. The attributes attached to higher nodes are more general and hence they apply to more objects attached to lower nodes. It is these higher attribute-nodes that give commonality to design patterns. Higher nodes, therefore, partition the set of design patterns into equivalence classes that, naturally, become the basis for classification as will be discussed shortly. For example, the attribute 's1' is attached to the highest node before the supermum (least upper bound) node as depicted in Figure 5. Commonality and differences between design patterns can be illustrated by the these two concepts:
{{PROXY}, {s61, v31, f1, s1 }}
{{FACADE, ADAPTER}, {v1, v30, f1, a1}}
What this means is this: the PROXY design pattern has common attributes with FACADE and ADAPTER. These two concepts differ by these attributes: 's1', 's6', 'v1', and 'a1'. Their common attributes are: 'v3', and 'f1'.

So what is the implication of these ideas in practice? The common feature tell us that there are choices to pick among many design patterns. For example, FACADE and ADAPTER share some attributes that, say, we are interested in. However, if our problem requirements specify additional features, e.g. conversion is between interfaces in a one to one manner, this forces us to select ADAPTER over FACADE. In general, and in the context of using design patterns as design solutions, given a new requirement in terms of problem statement, one of the first things you need to do is to evaluate a new design based on an existing ones. If a match is found, i.e. it has properties or features similar to it, you use it, or if you recognize that it has essentially some minor variations to the existing one, the framework gives you the ability to traverse the list of attributes (features) you may find desirable in solving your problem.

The complete definition of a design pattern is derived from its attributes. A complete analysis and identification of relevant features may take some time and experience. Generally, the original author of a design patterns is the best authority in this regard. With our process, assigning a verbal predicate to a design pattern which essentially captures its general nature does nothing more than gives it a weighted attribute. This serves as a coarse-grained classifier. Decomposing the pattern into its other features (attribute identification) will permit us to place the pattern into its proper place in the classification hierarchy subject to attribute-names and their values in relation to other patterns. Most widely accepted pattern classification schemes organize patterns based on categories that are too coarse-grained and target-dependent, i.e. technology, platform, programming language, etc. Referring to Figure 5, it would be very hard, if impossible, to see that the {ADAPTER, FACADE} are related to each other and to the set {FIREWALL PROXY, CHECK POINT, SINGLE ACCESS POINT, LIMITED VIEW}. This is because using our lattice diagram, they constitute the same concept represented textually as:
{{FIREWALL PROXY, CHECK POINT, SINGLE ACCESS POINT, LIMITED VIEW, ADAPTER, FACADE}, {access}}.

References

[1] Paul Adamczyk. Quantitative design patterns. Master's thesis, Illinois Institute of Technology, Chicago, IL, May 2002.

[2] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-Kingand, and Shlomo Angel. *A Pattern Language: Towns.Buildings.Construction*, volume 2 of *Center for Environmental Structures*. Oxford University Press, New York, 1977.

[3] Margaret Edna Anderson. *Roget's Thesaurus: An Explanation of Its Purpose and A Study of Some Applications of Its Priciples*. Ph.D. Thesis, Case Western Reserve University, May 1978.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, New York, 1996.

[5] P. Coad, D. North, and M. Mayfield. *Object Models: Strategies, Patterns, and Applications*. Yourdon Press, Upper Saddle River, 1997.

[6] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley Publishing Company, Reading, MA, 1995.

[7] D. L. Emblen. *Peter Mark Roget: The Word and the Man*. Thomas Y. Crowell Company, New York, NY, 1970.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[9] Bernhard Ganter, Rudolf Wille, and C. Franzke. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[10] Mark Grand. *Patterns in Java*, volume 1. John Wiley & Sons, Inc., New York, NY, 1998.

[11] N. Harrison, B. Foote, and H. Rohnert, editors. *Pattern Languages of Program Design 4*. Addison-Wesley Publishing Company, Reading, MA, 2000.

[12] David C. Hay. *Data Model Patterns: Conventions of Thought*. Dorset House, January 1996.

[13] Hohpe, G. and Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. The Addison-Wesley Signature Series. Addison-Wesley, Boston, MA, 2004.

[14] Michael Jackson. Problem Analysis Using Small Problem Frames. In *Proceedings of WOFACS '98*, number 22 in Special Issue of the South African Computer Journal, pages 47–60, March 1999.

[15] Michael Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[16] F. William Lawvere and Stephen J. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, Cambridge, 1997.

[17] R. C. Martin, D. Riehle, and F. Buschmann, editors. *Pattern Languages of Program Design 3*. Addison-Wesley Publishing Company, Reading, MA, 1998.

[18] Eugene A. Nida. *Componential Analysis of Meaning*. Mouton & Co. N. V. Publishers, The Haguge, Netherlands, 1975.

[19] Gunnar Övergaard. A Formal Approach to Collaborations in the Unified Modeling Language. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference*, volume 1723, pages 99–115, Fort Collins, CO, USA, October 28-30 1999.

[20] Linda Rising, editor. *The Patterns Handbook: Techniques, Strategies, and Applications*. Cambridge University Press, New York, 1998.

[21] Linda Rising. *The Pattern Almanac 2000*. Software Pattern Series. Addison-Wesley, Boston, MA, 2000.

[22] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors. *Pattern Languages of Program Design 2*. Addison-Wesley Publishing Company, Reading, MA, 1996.

[23] Walter Zimmer. Relationships Between Design Patterns. In J. Coplien and D. Schmidt, editors, *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1995.