Introducing a General Multi-Purpose Pattern Framework: Towards a Universal Pattern Approach

Alexander G. Mirnig and Manfred Tscheligi
Center for Human-Computer Interaction
Christian Doppler Laboratory for "Contextual Interfaces"
Department of Computer Sciences, University of Salzburg
Salzburg, Austria
Email: {firstname.lastname}@sbg.ac.at

Abstract— **Patterns have been successfully employed for capturing knowledge about proven solutions to reoccurring problems in several domains. Despite that, there is still little literature regarding pattern generation or common pattern quality standards across the various domains available. This paper is an extended version of a short paper presented at PATTERNS 14 [1], in which we introduced an attempt for a universal (i.e., domain independent) pattern framework. Via basic set theory, it is possible to describe pattern sets that are composed of several subsets regarding pattern types, quantities, sequence, and other relevant factors. This further enables us to describe patterns as sets of interrelated elements instead of isolated entities, thus corresponding with the scientific reality of complex problems with multiple relevant factors. The framework can be used to describe existing pattern languages and serve as a basis for new ones, regardless of the domain they are or were created for.**

Keywords-patterns; basics on patterns; pattern framework; set theory; pattern modeling

I. INTRODUCTION

Patterns have been used as a tool for capturing knowledge about proven solutions to reoccurring problems in a multitude of domains and disciplines. Most prominent among these are architecture, design, and software engineering [1][2][3][4][5]. Patterns allow documenting knowledge about methods and practices in a structured and systematic manner and can, therefore, serve as a valuable knowledge transfer tool within or even across disciplines. Another related benefit of patterns is that they can serve to "make implicit knowledge explicit" [6], i.e., they can be used to explicitly capture what is normally only acquired via experience after having worked in a certain field or domain for an extended period of time. They can thus go beyond and supplement the "raw" information contained in guidelines with a more solution- and practice-oriented dimension. The information contained in such patterns can then be provided to others (researchers or other interested parties) in a relatively quick and efficient manner, as it contains information about solutions that are already proven to work.

Having access to a structured collection of information from implicit and explicit knowledge about research practices is useful for any domain in which research is conducted. So it would make sense to extend the pattern approach or even establish patterns as a general field of basic research, with extensions into particular domains and disciplines. This wider potential of patterns has been recognized and has been summarized by Borchers [7] in the following way: "There is no reason why experience, methods or values of any application domain cannot be described in pattern form as long as activity includes some form of design, creative or problem-solving work." Despite this, there is little general (i.e., domain independent) literature available on patterns and pattern finding or creation. This is not a new idea [8] and there have been efforts to go deeper into the commonalities of various pattern approaches and patterns in general by, e.g., the work of Meszaros and Doble [9] and Winn and Calder [10].

Two of the main benefits of patterns are that they facilitate re-application of proven solutions and that they serve to make implicit knowledge explicit. These benefits are of particular importance to those, who do not already have this knowledge themselves, i.e., it is a way to draw from a vast pool of knowledge that would otherwise be gained via experience, over a long period of time. If working with patterns has extensive domain experience as a prerequisite, then those that would need that knowledge the most would arguably benefit the least from it.

We argue for a general strand of research on patterns as a means to capture knowledge about research practices. With such a theoretical basis available, practitioners from any domain could have a pool of knowledge to draw from, which would help them create patterns suitable for their needs. This should not mean that a variety in pattern languages and approaches is not desirable. It makes sense to assume that different domain requirements need different pattern approaches. However, the basics of patterns should ideally be similar for everyone and easily accessible, like, e.g., with general mathematics. A statistician needs and employs different mathematical means than a fruit vendor. But both draw from the same pool of general mathematics as their basis. In our research, we try to look at patterns in a similar way. We want to promote their use as a universal tool to structure knowledge in all kinds of areas and disciplines.

In this paper, we will take a look at pattern approaches in general and the commonalities between them. We will then integrate these into a formal pattern framework, with the aim of providing a formally sound and flexible basis, which allows practitioners and researchers to create their own patterns and pattern collections within their respective domains. To this end, we pursue four main goals in developing our framework:

- the framework should be a suitable basis for and, therefore, be compatible with most (if not all) existing pattern approaches and languages
- it should contain basic functionalities that allow meaningful structuring and referencing of patterns
- the framework must dictate the pattern content only in the most rudimentary way, so that it is not restricted to only one or very few disciplines
- the framework must be formally sound but also easy to work with, so that it can be applied by large number of individuals

The final goal of this research is to arrive at a structured but still easy to understand framework that captures the essence of patterns and makes them understandable as well as usable for practitioners and researchers in any domain. We do this via a basic set theoretic [11] analysis that allows describing patterns and pattern languages in a general manner. Such a general analysis of patterns allows us to treat them as separate phenomena, independent of the domains they are created and used in. Set theory is one of the most basic, but at the same time very powerful, mathematical tools available. By using set theory, we can ensure consistency of our framework, while still keeping things basic and relatively easy to understand. An additional benefit of our approach is that it permits the creation of pattern sets across different pattern languages that address a similar purpose. This can facilitate the consolidation of already existing knowledge within the various domains.

This set theoretic framework serves as a domain independent basis for reflections on how patterns can or should be created and structured. It can be extended to fit the needs of a particular discipline or area, if that would become necessary, but is, at its core, a purely formal tool that is not restricted to any domains or disciplines. In this paper, we begin with an overview of existing general literature on patterns in Section II, followed by some explanations regarding the basics of set theory and why we deem it a suitable tool for the purpose of this paper. In Sections V and VI, we provide an outline of the proposed set theoretic pattern framework. In Section VII, we supplement the framework with general recommendations on how to find patterns for multidisciplinary applications of the framework. In Section VIII, we present an example application of the framework to structure an existing pattern collection. In Section IX, we discuss limitations and future work potentials of the framework, with a brief conclusion at the end in Section X.

II. Related Work

Patterns have been employed in a multitude of application domains [1][4][12] and a good number of extensive pattern collections [7][13][14] have been created in the past. Literature on the pattern generation process itself, sometimes also referred as *pattern mining* [13], is still scarce [15]. Existing literature on pattern generation is mostly focused on specific domains [7][4][8][12]. The work of Gamma et al. [13] can be considered important elementary literature, but it is still centered on software design. Although covering a wide spectrum of software design problems, it is arguably of limited applicability outside of the software engineering domain. The same can be said for other specialized pattern generation guidance [8], which would require adaptation to be employed in other domains (e.g., biology or linguistics). Falkenthal et al. [16] introduced a promising approach for validating solution implementations of patterns in various domains, though provided only one nontraditional use case (Costumes in Films) for their approach.

The advantages of patterns would be both desirable and feasible [7] for these other domains. Vlissides [17] provides a good summary of what patters can and cannot do. The perceived advantages of patterns might be summed up as follows:

- they capture expertise and make it accessible to non-experts
- their names collectively form a vocabulary that helps developers communicate better.
- they help people understand a system more quickly when it is documented with the patterns it uses.
- they facilitate restructuring a system whether or not it was designed with patterns in mind.

Another interesting aspect of patterns is that one single pattern is usually not enough to deal with a certain issue. Alexander [2] himself already expressed this by stating the possibility of making buildings by “stringing together patterns.“ However, the pattern itself does not always include the information of which other pattern might be relevant in a particular case. This information is only available once the pattern is part of an actual pattern language of several related patterns. Borchers [7] introduced the notion of high level patterns, which reference lower level patterns to describe solutions to large scale design issues. This hierarchy is expressed via references in the patterns themselves. Borchers’ view of high and low level patterns is a good way of understanding and describing patterns as interconnected entities. A suitable framework for patterns and pattern languages should ideally be able to capture the – sometimes complex – relations between patterns and allow mapping of individual patterns to higher level or overarching problems or goals.

One concept that is similar to the ideas pursued in this paper is that of ontologies. While term ‘ontology’ itself can have several meanings, the following short definition by Blomqvist and Sandkuhl [18] provides a good summary of how the term is usually understood in ontology engineering: “An ontology is a hierarchically structured set of concepts describing a specific domain of knowledge that can be used to create a knowledge base. An ontology contains concepts, a subsumption hierarchy, arbitrary relations between concepts, and axioms. It may also contain other constraints and functions.” Given this description, one might assume that ontologies would be an ideal tool to capture and transfer domain knowledge universally. However, there are two limitations that make ontology engineering approaches run counter to the goals pursued in this paper. While ontologies can promote the application of good practices [20][21][22], reusability of ontologies is still considered a serious, and as of yet unsolved, challenge in ontology engineering [20][23][24]. Second, actually building an ontology is a very difficult tasks with many potential pitfalls, even for experts [20][21].

Both of these are serious limitations when considering suitability for users from a wide range of disciplines and/or skill levels as well as reusability of existing solutions. Patterns, thanks to their focus on reusability and the problems themselves instead of the abstract structure of a domain or field, seem more well-suited for the problems at hand.

Efforts to provide a general basis for patterns include the work of Meszaros and Doble [9], who developed a pattern language for pattern writing, which serves to capture techniques and approaches that have been observed to be particularly effective at addressing certain reoccurring problems. Their *patterns for patterns* were divided into the following five sections: Context-Setting Patterns, Pattern Structuring Patterns, Pattern Naming and Referencing Patterns, Patterns for making Patterns Understandable, Pattern Language Structuring Patterns. Another interesting approach being quite similar in its aims to the one presented in this paper is the *Pattern Language for Pattern Language Structure* by Winn and Calder [10]. They identified a common trait among pattern languages (i.e., they are symmetry breaking) and built a rough, nonformal general framework for pattern languages in multiple domains. These ideas are similar in concept to what we pursue in our research. The difference is that we want to provide a purely formal framework without or minimal statements regarding its content (such as types or traits). We want to focus on the basics behind patterns and structure these so that they can be applied as widely as possible, although we draw from their work (and that of others) to supplement the framework with general recommendations for pattern finding later in Section VII.

III. SET THEORY – A BRIEF INTORODUCTION

Patterns, despite the term having a rather well established meaning, come in many shapes and forms and can be of varying complexity and verbosity. At the most basic level, they still have one thing in common – They consist of a number of statements, which are divided into several different categories of statements (e.g., pattern name, scenario, problem statement). A pattern is naturally much more than that, but this rather simple and elementary commonality is sufficient to begin building a framework from. A framework is a structure, an empty container that facilitates working with its contents (whatever these might be) in a consistent and organized way. In our case, this container should facilitate organizing and referencing patterns. Set theory is a mathematical method that allows organization of objects or data into so-called sets. Thus, if we understand patterns as collections statements in different categories, the connection to set theory becomes evident when we replace the word 'categories' with 'sets'. In the following paragraphs, we will outline the basics of set theory and highlight some of its advantageous attributes that we will use to build the pattern framework.

A set is an abstract, mathematical entity that contains other entities. These contents can either be sets themselves or singular, irreducible entities – so-called *elements*. Sets that are themselves contained in another set are called *subsets* of the set(s) they are part of (their *supersets*). Let us illustrate these considerations via the following example set S:

$$S = \{a, \{b, c\}\} \tag{1}$$

Sets are commonly denoted via curly brackets ('{' and '}') . In (1), we see two such sets: The set $\{b, c\}$ and the set $\{a, \{b, c\}$. The former is contained in and thus a subset of the latter. Therefore, we can say that $\{b, c\}$ is a subset of S and that S is a superset of $\{b, c\}$. This can more briefly be expressed via the symbols '$\subseteq$' and '$\supseteq$' in the following way: $\{b, c\} \subseteq S$; $S \supseteq \{b, c\}$. A subset is a *proper* subset if it is contained in another set, but there is at least one element that is part of the superset, but not the subset. In our example, a would be such an element, which is why we can furthermore state that $\{b, c\}$ is a proper subset of S. We write this as: $\{b, c\} \subset S$. The fact that a is an element of S can be expressed via the symbol '$\in$' in a similar fashion as: $a \in S$. A set of non-empty subsets is called a *partition*, if each element of the superset lies in exactly one element of the set of subsets. The set $\{\{a\}, \{b, c\}, \{d\}\}$ is a valid partition of the set $\{a, b, c, d\}$.

Sets are defined by their contents and one set is identical to another if every element of the former is also an element of the latter. The order of these elements does not matter. Consider the following examples:

$$\{b, c\} = \{c, b\} \tag{2}$$

$$\{a ,\{b, c\}\} \neq \{b, c\} \tag{3}$$

The sets $\{b, c\}$ and $\{c, b\}$ both contain the same elements, so they are identical. The variable a is contained in $\{a, \{b, c\}\}$ but not in $\{b, c\}$, so these two sets are not identical. Via rather simple operations we can distinguish sets from each other via their contents and make statements regarding these same contents. These operations can be used to distinguish patterns from each other and structure their contents via subsets.

Two other very useful aspects of set theory, which we will employ later on, are ordered sets and the empty set. As mentioned before, the order of the elements in a set is usually irrelevant. Ordered sets can be used to arrange contents in a certain order. To distinguish them from regular sets, they are denoted by angle brackets ('<' and '>'). On a technical level, ordered sets are regular sets, where the order is determined by the number of subsets a certain element is part of. So the ordered set $<a, b, c>$ would really be the regular set $\{\{a\}, \{a, b\}, \{a, b, c\}\}$. a is contained in three subsets, b in two, and c only in one, which results in the order of a, b, c. Let us illustrate this further via the following examples:

$$<a, b> \neq <b, a> \quad (4)$$

$$\{\{a\}, \{a, b\}\} \neq \{\{b\}, \{b, a\}\} \quad (5)$$

$$\{\{c, a\}, \{b, a, c\}, \{c\}\} = <c, a, b> \quad (6)$$

The formula in (4) demonstrates that order is essential in ordered sets and (5) shows why that is the case, since the regular sets in (5) are simply the equivalents to the ordered sets in (4). (6) is meant to emphasize, that the order of the elements in the regular sets does not matter, only their frequency of occurrence does. The utility of being able to put things into sequence is very useful and we can use this to capture problems and the sequence they occur in. It can also be used to map the hierarchy of patterns as sequences from higher to lower level patterns.

A set is an abstract entity. It is defined by its contents but not identical to its contents. It is more than the sum of its elements. Therefore, the following is true:

$$a \neq \{a\} \quad (7)$$

While a is an element of the set that contains only a, it is *not* identical to that set. The set itself is a separate entity. An interesting consequence of this is that we can talk about sets, regardless of whether they actually contain anything. There is exactly one set, which does not contain any elements, and it is called the empty set (usually denoted by '∅' or '{}'). The empty set is a very versatile tool and can be used to, e.g., handle blank fields in a pattern (e.g., an incomplete pattern without keywords)

At this point, we should mention again that the pattern framework is intended for a wide range of individuals from all kinds of backgrounds. Depending on the reader's background, this section might have seemed either a bit complicated, or rather elementary for what can be considered a substantial section of this paper. The most important things to keep in mind for now are:

- regular sets to cluster and organize information
- ordered sets to put information into sequence
- the empty set to handle empty categories

And with that, we have covered elementary set theory to a sufficient degree, so that we can now shift our focus onto the patterns themselves.

IV. Patterns in General

A. Patterns and Pattern Sets

Now that we have provided an overview of elementary set theory and the components that we intend to employ, we want to go more into detail regarding patterns, pattern languages, as well as some of the concepts behind them. Beginning with the basic term "pattern", Alexander [2] characterized patterns in that "each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." In order to fulfill this requirement, patterns are usually held to some minimum standards of what they and their structure should contain. Probably the most common one is the structure suggested by Borchers [7] and van Velie [5], who suggest six main pattern elements: name, forces, problem, context, solution, and examples. Existing pattern structures come in a variety of levels of complexity – from detailed ones with a large number of subcategories [4] to comparably simplistic ones [12] with a small number of subcategories. So on a basic level, patterns can be understood as a structured assortment of statements. The statements contained in each instance of such a structure form a whole that provides a solution to a specific problem, describing both in detail to facilitate reapplication of the solution. It is this structure that we build into a framework via set theory.

A pattern language is a complete hierarchy of patterns, ordered by their scope [12]. Patterns can be divided into high- and low-level patterns [7], depending on the scale of the problem and its solution. High level patterns are more abstract and deal with larger scale problems. Low level patterns are more concrete and focus on smaller problems or parts of problems. In that sense, low level problems can be part of high level problems, which means that low level patterns can address or further specify problems from high level patterns. The distinction between high and low level patterns is not a strict one and depends on the respective individual's or group's perception of the degree of abstraction of a certain problem. In software engineering, the lowest level patterns are also referred to as *idioms*. Idioms contain very concrete solutions – mostly actual code snippets – that address very small scale coding problems.

In addition to "regular" patterns, there are also other types of patterns. The most notable of these are anti-patterns, which differ from regular patterns in that they do not describe a proven working solution to a problem, but rather a solution that is proven *not* to work or not work well. They follow the same structures as regular patterns, but instead of best practices they describe bad practices that should not be replicated. Since they describe solutions to reoccurring problems just as regular patterns do (with the delineating factor that the described solution should be avoided instead of replicated), a framework suitable for regular patterns should also be suitable for anti-patterns. In the framework, it will be possible to represent all these different pattern types.

V. The Basic Framework

We now translate these concepts into a basic set theoretic structure. We do so by employing mostly regular sets, ordered sets, and subsets. Note that the following analysis would work for any number of statements that is or can be structured in a similar way, which includes most, if not all, pattern structure (such as, e.g., the design patterns template laid out by Gamma et al. [4]). We chose to base our analysis on a concrete example, a Contextual User Experience (CUX) pattern structure by Krischkowsky et al. [8] (see Table 1), in order to give a more current and less software engineering centered example. As noted previously, the following set theoretic analysis would apply to any other similarly structured assortment of statements (in this case most – if not all – imaginable pattern structures).

TABLE I. CUX PATTERN STRUCTURE [8]

Instructions on Each Pattern Section		
#	***Section Name***	***Instruction on Each Section***
1	Name	*The name of the pattern should shortly describe the suggestions for design by the pattern (2-3 words would be best).*
2	UX Factor	*List the UX factor(s) addressed within your chosen key finding (potential UX factors listed in this section can be e.g., workload, trust, fun/enjoyment, stress...). Please underpin your chosen UX factor(s) with a definition.*
3	Key Finding	*As short as possible - the best would be to describe your key finding (either from an empirical study or findings that are reported in literature) in one sentence.*
4	Forces	*Should be a detailed description and further explanation of the result.*
5	Context	*Describe the detailed context in which your chosen key finding is extracted/gathered from.*
6	Sugges-tions for Design	*1) Can range from rather general suggestions to very concrete suggestions for a specific application area. 2) The design suggestions should be based on existing knowledge (e.g., state of the art solutions, empirical studies, guidelines, ...). 3) More than one suggestion are no problem but even better than only one. 4) There can also be a very general suggestions and more specific "sub-suggestions".*
7	Example	*Concrete examples underpinned by pictures, standard values etc. Examples should not provide suggestions (this is done in the suggestion part) but rather underpin and visualize the suggestion presented above.*
8	Key-words	*Describe main topics addressed by the pattern in order to enable structured search.*
9	Sources	*Origin of the pattern (e.g., literature, other pattern, studies or results)*

The pattern structure shown in Table I consists of nine categories. This means that each pattern generated in this structure will consist of various statements in each of these nine categories. We now want to generate an actual pattern language set, let us call it CUX Language (and refer to it as *CL* for brevity's sake), based on the structure outlined in Table 1. We can do so by introducing nine subsets (i.e., sets of the set *CL*) CL_1 to CL_9, each subset corresponding to one of the nine categories (from Name to Sources, respectively) described above. This is what the initial structure looks like:

$$CL: \{CL_1, CL_2, \dots, \mathrm{CL}_9\} \tag{8}$$

At this very basic level, a pattern structure is nothing more than a set consisting of a number of (as of yet empty) subsets. To distinguish the *CL*-subsets from each other, they can be filled with a sequence of numbers reflecting the number of categories or actual text strings of the category labels. For the *CL* from our example above, the simplest way to achieve this would be to fill them with the numbers 1-9. For now, the actual contents of the *CL*-subsets only matter inasmuch they are distinct from each other and allow reference later on. The number of subsets depends on the number of distinct categories each individual pattern of the structure is supposed to have. This structure can be adapted for other pattern structures, by adapting the number of subsets accordingly.

After having defined the pattern structure, we now also need a set-theoretical representation of the individual patterns. Since we assumed patterns to be statements arranged in certain categories (sets), we now assign said statements to each of the nine subsets for *CL*. We start out by assuming a set that contains all these statements; we define it as follows:

$$S: \{S_1, S_2, \dots, S_n\} \tag{9}$$

We obtain a pattern by simply assigning certain elements of *S* to each subset of *CL*. For this purpose, we introduce a function *p* from a subset of the set of statements *S* into *CL* into a partition S^p of the sets of statements *S*:

$$S^p = \{Sx_1, \dots, Sx_n\} = S \tag{10}$$

$$p_i: CL \rightarrow S^p \tag{11}$$

$$P_i = \sum_{CL_j \subset CL} p_i(CL_j) \tag{12}$$

What happens here is that each subset of *CL* (CL_1 to CL_9) gets filled with actual content by having a certain number of statements (i.e., part of the partition of *S*) assigned to it. Partitioning *S* ensures that none of the categories remain empty, i.e., that the pattern is complete. Note, that we use the term 'statement' in the loosest sense, so it can refer not only to full sentences, but also to single words or sequences of words, which are not full sentences, or even images. Therefore, partitioning *S* into clusters of sensible information (= subsets) is a necessary step that should be reflected in the formal analysis. The relation $\mathrm{p_i}$ determines the assignment of one subset of S^p to each subset of *CL* (CL_1 to CL_9 in our example). The actual pattern is the sum of all values of $\mathrm{p_i}$ returned for all values $CL_i \subset CL$, viz. all categories of the pattern structure (CL_1 to CL_9 in our example). The results is a set of *i* number ordered pairs, which, in our example, might look like the following:

$$P_1 = \{<CL_1, Sx_1>, \dots, <CL_9, Sx_9>\} \tag{13}$$

Note that the P_1 above is only one possible example out of many. The actual values assigned by p_i are left undefined in this framework, since this depends on the context and proposed content of the individual patterns. The framework should be flexible and widely applicable, so the actual pattern generation must be left to the domain experts. We can use the pattern relation to generate as many relations p_1 to p_n as we need, and thus to generate n number of *CL*-patterns P_n.

We can apply this analysis to any other similarly structured pattern language *PL* and its subsets PL_i, thus granting us a basic structure of patterns as collections of sets of statements. Of course, simply arranging patterns into sets and subsets does not in itself guarantee that any of these patterns are actually useful or reasonable. But that is also not the purpose of the framework at this stage. What this analysis can tell us is (a) the pattern language or structure *PL* (*CL* in our example) the patterns are generated in, (b) how many statement categories (viz., subsets of *PL*) a successful pattern generated in that language must contain, and (c) which statements can be found in which category, i.e., the patterns themselves (P_i). As we can see, this elementary analysis has already yielded a quite flexible starting framework, via which we can express a pattern structure, partition the information to be transformed into a pattern, and relate that information to the pattern structure. Most importantly, building the framework did not require us to reference the actual contents of *S* or its subsets, meaning that it is – at least so far – applicable independently of its contents or the context it will be used in.

VI. Descriptors

Even at this elementary level, we can do more than merely put statements into a certain structure and add them to a collection of similar statements. We can also lay the groundwork for the relations between individual patterns of a certain collection. A pattern collection is more than an unstructured assortment of statements and needs some kind of inner structure. At this point, there are three important aspects that an individual *CL*-pattern does not tell the reader at this stage, but which we can capture even at this basic formal level. These are (a) which *other* patterns might be useful or even necessary for a given purpose, (b) exactly at which point during a given task or activity and (c) in which order will they be needed. A design pattern for, e.g., menu depth might sensibly be followed by a pattern for hierarchical structures, and preceded by a pattern for menu types and their suitability. But with only one pattern at hand, one can only guess what else they might need upon being presented with only a single pattern or depend on prior experience. It would be undesirable and arguably defeat the purpose of patterns, if extensive meta-knowledge were necessary to be able to use them successfully. Patterns usually contain references to other patterns as a separate field, though the reliability of this depends on the respective pattern collection. To capture (a), (b), and (c) on a more general level, why we enrich the basic set theoretic framework with specialized descriptor sets, which serve to understand patterns in context with each other. These will allow us to add an additional layer of expressiveness and flexibility to the language.

At its core, a descriptor is nothing more than an ordered set containing several subsets with patterns. By employing ordered sets, we can distinguish its subsets solely by virtue of which position they have within the ordered set. The general idea is to use this property of ordered sets to implicitly add auxiliary information to any given pattern collection, simply by arranging that collection in a certain order. That way, the general structure of a descriptor set needs to be defined only once and one can add additional information to a pattern collection by arranging them in a certain order according to the descriptor. Let us illustrate the basic idea via an example descriptor set. Remember that angle brackets ('<' and '>') denote an ordered set, as opposed to regular sets, which are denoted by curly brackets ('{' and '}').

$$D^E = <\{P_1, P_2, P_3, P_4\}, \{P_5, P_6, P_7\}> \quad (14)$$

Assume that there is a pattern collection that consists of seven patterns. Of these, four are regular patterns and three of them antipatterns. We can now define a descriptor as an ordered pair consisting of exactly two subsets. The first of these subsets contains only patterns, the second contains only antipatterns. By applying this knowledge to D^E, we learn that patterns P_1 to P_4 are regular patterns, and that P_5 to P_6 are antipatterns. We have thus provided an easy way to categorize patterns as regular patterns and antipatterns, that can be applied independent of context, and which is as simple as arranging the patterns in a certain order. Furthermore, we have added information to the pattern collection without having to edit the patterns themselves. (Please note that whether a pattern is a regular one or an antipattern is usually considered essential information and already part of the pattern itself. This is merely an illustrative example that employs two obviously distinctive pattern attributes). In the framework, we will use this structure to make a more general distinction of mandatory vs. optional patterns (see Section VI.B).

Structuring the pattern collection in this way allows for added efficiency when generating new collections, and also facilitates sharing and consolidation of pattern collections. E.g., if one would program a pattern database in this framework, new pattern collections can be categorized by arranging the pattern labels in a certain order, mandated by the predefined descriptor sequence. Similarly, new patterns can be added and enriched with information by simply assigning them to an appropriate subset of a descriptor. One could even consolidate patterns from different sources and/or authors into one collection and categorize them by simply arranging them in a certain order. There are often many pattern collections dealing with similar topics yet the valuable knowledge in these patterns is often difficult to consolidate, simply because pattern approaches vary so much.

Therefore, we view the added advantages gained by adding descriptors as an important quality of the framework and a necessary step towards a pattern framework that facilitates exchange both within and between disciplines. In the following subsections, we will build a standard descriptor set structure in a step-by-step manner. To begin, we postulate a descriptor as an ordered set, which consists of a number of subsets that contain either individual statements or sets of statements.

A. Targets

One single problem rarely occurs in isolation, but is instead often part of a higher-level problem or occurs while trying to achieve a certain overall goal. These are often nothing more than a single sentence or a few words, but they serve as a good overall indicator about where to find a solution to a particular problem one may have. E.g., a programming pattern might be part of the larger problem of trying to avoid pointer errors in C++. Another example would be Tidwell [12], who structured their design patterns as part of several categories, such as “Organizing the content” or “Showing complex data”. One individual pattern can conceivably be part of several such higher-level problems or be used in similar or different context to achieve different goals. This is different from the problem described in a pattern, since a given high-level pattern could very well reference a lower-level problem that addresses a different problem, while both serve the same general purpose. In the following, we will label these high-level problems or overall goals *Targets* (or *T* for short).

Finding, iterating and validating patterns is a lengthy and multi-stage process. Whereas finding a new context a pattern can be used in might be as simple as trying to apply it and succeeding. This is were Targets as separate and standardized entities come in very handy. The Target(s) of a pattern should not be part of the pattern itself, since that would entail having to change and subsequently revalidate a pattern each time a new application possibility for it is discovered. Instead, Targets are separate from the actual patterns, which can be assigned or mapped to them. This allows reusing and reapplying patterns (one of their key aspects) in different contexts without having to modify the patterns themselves each time. Whenever a new application for a certain problem strategy is found, a new Target expressing that application area can be created and the pattern (or several) assigned to it. Due to their general nature and labeling function, Targets are the first entities that will be part of our standardized descriptor structure. This is also one of the reasons why we postulated descriptors as containing either statements or sets of statements. Each pattern in this framework is, per definition, a set of statements. But not everything, which might be a sensible addition to the descriptor, is necessarily a pattern (such as Targets). The first set of statements in a standardized descriptor set is thus always an expression of the Target of a pattern collection. At this point, the descriptor structure looks like:

$$D = <T> \qquad (15)$$

‘T’ is a placeholder for a set containing one or several statements, so a descriptor at this stage could read, e.g., $D^E = <\{S_{37}\}>$ or $D^E = <\{S_{28}, S_{29}\}>$. Of course, a Target without any patterns is rather useless, so we need to add these to the descriptor as well.

B. Mandatory and Optional Patterns

The second subset in the descriptor will contain the actual patterns that are supposed to contribute to the Target expressed in the first set. However, there is one important distinction that we can make at this stage, which consists in separating the patterns into mandatory and optional patterns. Patterns can be part of solutions to higher-level problems and are ideally applicable in similar contexts. It is reasonable to assume that a high-level Target might cover a high-level context and thus a range of several low-level contexts. But not all of these low-level contexts might be similar enough to be interchangeable, thus excluding some patterns depending on which subcontext it is applied to. In addition, solving one problem via a pattern, might (and often will) pose a new problem, for which there are several possible solutions (and thus, several possible patterns).

To illustrate the concept of mandatory and optional patterns via a brief example, assume that we design an interface and want to display items and their contents on screen at once. We decide to take a look at Tidwell’s pattern [12] collection and find a pattern titled “Two-Panel Selector”, which suits our needs. Following the pattern solution, we divide our interface into two parts; one showing the items, and the other their contents. We then find that our item structure is multi-layered and rather complicated, and that two panels are not sufficient to display it in an adequate fashion. Conveniently, we find a pattern section that contains solutions for displaying complex data. Among these, we find a pattern showing the use of cascading lists and another showing the use of tree-tables. Depending on other considerations (e.g., horizontal space, consistency with the rest of the interface), we will then decide for one of the two solutions, but very likely not both. They are two solutions for a similar (and in our case the same) problem and we are free to choose the one we deem more appropriate for our purpose.

If, however, it is – for whatever reason – impossible or undesirable to separate the patterns into mandatory and optional ones, the set of optional patterns can also simply be left empty. Since the empty set is an abstract entity, these standardized descriptors will always remain compatible with each other. Even when one ore several of their subsets are empty, the *number* of these subsets never changes. Thus, the standard sequence and meaning of each subset is preserved, regardless of whether any of its preceding subsets is empty or not. Again, we further illustrate this via example descriptors:

$$D^E_1 = <\{S_{37}\}, \{P_1, P_2\}, \{P_3\}> \qquad (16)$$
$$D^E_2 = <\{S_{38}\}, \{P_1, P_2, P_3\}, \{\}> \qquad (17)$$

Both of these example descriptors are of the same structure. They differ in that they have two different targets, and that P_3 is an optional pattern in D^E_1, and a mandatory pattern in D^E_2. They both have the same number of subsets, so if we were to add another set to the descriptor set, we could so without worrying about potentially empty set, since the sequence is preserved.

C. References to other pattern collections or other sources

Patterns are not the only things that can be considered optional when tackling a problem. Other sources and references are often needed as well. Patterns usually contain references to sources they draw from, aside from references to related patterns from the same pattern collection. But there is additional benefit when these references are added at the descriptor level. That benefit is flexibility. The descriptors are not part of the patterns themselves and can be generated at any time, once a pattern collection is available. Thus, any information that can be added by modifying the descriptor does not necessitate modifying the patterns contained in the descriptor. Thus, a seemingly outdated pattern collection can be updated *ex post*, by generating descriptors containing references to material that was not available at the time the patterns were originally generated.

The same can be done with patterns from other pattern collections that handle similar issues. Patterns from other pattern collections can be added to the set of optional patterns. Since the descriptor allows inclusion of any set of statements, this could, in semantical terms, be the full pattern or merely a link to its website or bibliographical reference. Even if the *CL*-structure of both pattern collections were the same, the pattern relations p_i would be different. This means that the "foreign" patterns would not be part of the set of patterns P_i and, thus, are merely sets of statements and easily distinguishable from one's own patterns. The descriptor structure allows consolidation of knowledge from different sources and goes beyond the possibilities gained by referencing only within the patterns themselves.

We show how such references might work via another brief example: Assume that we intend to design a car interface, for which we have our own car interface design pattern collection. While designing, we notice that the interface structure has become very deep and rather difficult to navigate. We now intend to solve this problem by either reducing the menu depth or presenting the information in a more effective way, but cannot find an appropriate pattern in our collection. However, we find such solutions in other, more generic interface design pattern collections. One of these turns out to be particularly to our liking and can be easily applied without any modifications. Both pattern collections were printed and published several years ago, so a revision would not be a trivial task and require substantial effort.

Then, we decide to collect our car design patterns in a database and arrange them via descriptors. To keep the required effort at a minimum, we add the patterns simply as uniquely identifying labels for the original patterns. The resulting database allows us to search for design problems via their Target. We create on descriptor, which has "Designing car interfaces with high menu depth" as its Target and reference the foreign pattern we found in this descriptor. Thus, anybody who faces the same problem and uses the database will know that there is a different pattern collection that provides a solution to a certain subproblem. Such information is normally either included when a pattern is generated or not at all. With the descriptor structure, it is as simple as new descriptor. We can include newly created patterns by inputting them directly or referencing them, as well as draw from knowledge from related fields by referencing other patterns in this way.

By adding the two additional subsets for mandatory and optional patterns and references to the initial descriptor structure, the updated descriptor structure is:

$$D = <T, M, O> \qquad (18)$$

D. Pattern Sequence

Finally, problems do not always occur at random, but can appear in a certain sequence. Thus, a solution to one problem might be followed by another solution, dictated by the underlying sequence of problem occurrence. This might be as simple by one problem followed by the other, or it could also reflect a hierarchical structure of high to low-level problem solutions, where depending on how a higher-level problem is solved, different lower-level problems occur. We can find such sequences, e.g., in Breitenbücher et al. [25], who propose a method to organize low-level solutions (so-called *idioms*) in sequence, while taking into consideration the preceding idioms.

To handle sequences at the descriptor-level, we add another subset to the descriptor. The purpose of this set is to put our patterns and other information into a sequence; therefore, we label this additional set *S*, which overlaps with the previously introduced sets *M* and *O*. Since S is supposed to handle only sequences, it would make little sense for something to be part of *S* but not of *M* or *O*. Therefore, we postulate that every element of *S* must also be element of either *M* or *O*. Unlike *T*, *M*, and *O*, *S* is not a regular set but an ordered one. Since the order of their contents does not matter for regular sets, but does matter for ordered sets, arranging the contents as an ordered set is a simple and efficient way to express a sequence.

If we wanted to express that, e.g., P_1 from $D^E{}_1$ above would be needed after the solution described in P_2, we could add the sequence set $<P_2, P_1>$ to it and arrive at the following descriptor $D^E{}_3$:

$$D^E{}_3 = <\{S_{37}\}, \{P_1, P_2\}, \{P_3\}, <P_2, P_1>> \qquad (19)$$

As we can see, the sequence set contains nothing that is not also in *M* or *O*, and merely puts some (but not all) of the parts of the pattern collection into sequence. We can use the sequence set to not only express linear sequences, but also hierarchical structures. Assume, that we have one high-level pattern P_1 and a number of low-level patterns P_2 to P_7. P_1 proposes three possible solutions to the high-level problems. Depending on which solution is chosen, new low-level problems occur that are described in P_2 to P_4. Each of these solutions is then followed by another set of possible problems, described in P_5 to P_7. We can express this hierarchical structure via the following sequence set $S^E{}_1$:

$$S^E{}_1 = <\{P_1\}, \{P_2, P_3, P_4\}, \{P_5, P_6, P_7\}> \qquad (20)$$

The contents of S^E_1 represent a hierarchical structure from high to low. Since the contents of S^E_1 are regular sets, the order within these sets does not matter and they can be considered as being of the same level. We can use the sequence set to specify such a hierarchy even further. If we know which low-level problem leads into which, we could also formulate the alternative descriptor S^E_2:

$$S^E_2 = <\{P_1\}, \{<P_2, P_5>, <P_3, P_7>, <P_4, P_6>\}> \quad (21)$$

In S^E_2, we find P_1 is still the highest-level pattern and can see additionally the sequences between the individual patterns from the lower-level subsets. While it might not be immediately obvious, the three-level structure from S^E_1 is also preserved in S^E_2, since the first element of each ordered pair in S^E_2 is also an element of the middle set in S^E_1.

E. Adding it up

By combining all of the sets *T* to *M*, we arrive at the following, final descriptor structure:

$$D = <T, M, O, S> \quad (22)$$

Target, mandatory patterns, optional patterns and references, as well as the sequence set allow for a good amount of expressive possibilities. We will once again illustrate the potential use of the final descriptor structure via a simple example. Assume that we have four patterns, which would help us in conducting a user study in the car. P_1, P_2, and P_3 are patterns to reduce user distraction and part of our own pattern collection. We have also access to another pattern about processing the data gained from the study. This pattern, we label it P^F_1, was generated in a different pattern structure and is, therefore, not part of our patterns P_i. We can now specify which of these patterns we want or need and in which order by introducing a descriptor. To do that, we need to specify the contents of each of its subsets. We further want to express that we definitely need P_1 and P_2, as well as P^F_1 and that the *DL*-pattern will be needed after the *CL*-pattern. We thus arrive at the following example mandatory pattern set M^E:

$$M^E: \{P_1, P_2\} \quad (23)$$

We also know that P_3 has proven useful in several similar cases in the past, but not in all of them, so we consider it as an optional pattern. Having one optional pattern (P_3) and one foreign pattern (P^F_1), gives us the following example set O^E:

$$O^E: \{P_3, P^F_1\} \quad (24)$$

We further know that P_3, should it be needed, is always needed after P_1. P_2, on the other hand, has no fixed position in the sequence, but occurs after P_3 in a few specific cases. P^F_1 is always needed last. This results in the following two sequence sets:

$$S^E_3: <P_1, P_3, P^F_1> \quad (25)$$
$$S^E_4: <P_1, P_3, P_2, P^F_1> \quad (26)$$

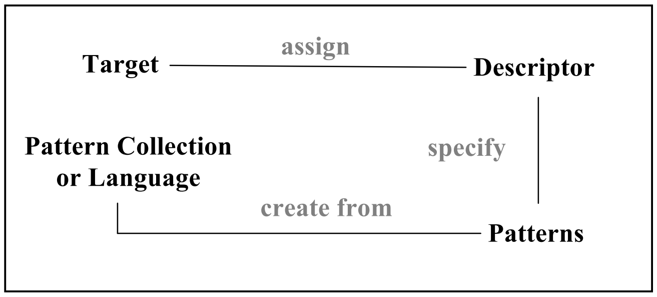


Figure 1. The Pattern Framework – a high-level overview

But how do we now specify which of these sequences is the appropriate one for a given scenario? Since patterns are created for a certain purpose, we need to map each sequence to its most appropriate purpose We can specify this via the Target, which contains the general purpose or overall problem of a collection of patterns. We can now introduce two targets, T^E_1 and T^E_2, with T^E_1 outlining the general high-level problem and T^E_2 specifying the contexts in which P_3 is followed by P_2. These can be any statements; in our example we specify them as the sets $\{S_1\}$ (for T^E_1) and $\{S_1, S_2\}$ (for T^E_2). As a result, we get the following two example descriptors D^E_3 and D^E_4:

$$D^E_3 = <\{S_1\}, \{P_1, P_2\}, \{P_3, P^F_1\}, <P_1, P_3, P^F_1>> \quad (27)$$
$$D^E_4 = <\{S_1, S_2\}, \{P_1, P_2\}, \{P_3, P^F_1\}, <P_1, P_3, P_2, P^F_1>> \quad (28)$$

In addition to being able to specify the relations between patterns from a single pattern language, we are not confined to that single pattern language. Furthermore, we can describe hierarchical and sequential pattern structures from different domains and pattern languages in the same framework.

D: <T, M, O, S>

D: Descriptor – contains 4 subsets
(contains 4 subsets, ordered set)

T: Target
(contains a finite set of statements, regular subset of D, may not overlap with M or O)

M: Set of mandatory patterns
(contains a finite set of CL-patterns, regular subset of D, may not overlap with T or O)

O: Set of optional patterns and references
(contains a finite number of sets of statements, regular subset of D, may not overlap with T or M)

S: Sequence set
(contains a finite number of sets of statements, ordered subset of D, every element of S must also be element of M or O)

Figure 2. The Descriptor structure

Furthermore, patterns can be clustered and applied for different purposes in an efficient manner by simply altering the structure of these pattern clusters, and without having to change the patterns directly. By adding one additional layer (targets and descriptors) to what was already available before, we have arrived at a highly modular and flexible pattern framework. Figure 1 provides an overview of the interrelation of pattern languages, patterns, descriptors, and targets. Figure 2 contains a summary of the standard Descriptor structure.

Descriptors can be generated on an as-needed basis, which means that they can be used to categorize the initial pattern collection, as well as update it with additional information, to keep a pattern from becoming outdated quickly without being cumbersome to maintain. Most of the information is contained in the patterns themselves, but we tried to elevate information that might change or be in need of frequent updates to the descriptor-level. This makes expanding and updating pattern collections easier, since most of these changes will require either changing or generating descriptors, which are nothing more than strings in a predefined sequence. This approach has the additional advantage that descriptors can be used to consolidate knowledge from different sources, since the descriptor structure is based solely on the set theoretic framework and, therefore, not bound to any particular field or context. We have thus provided a framework that is formally sound, based on only elementary mathematical principles, flexible regarding its content, and with additional means of referencing and consolidation via the descriptor structure. In order to actually use the framework for structuring patterns, however, one still requires a means to collect and structure data regarding working solutions. We provide some general recommendations on how to do that in the following section.

VII. Adding the Content

We mentioned in Section V that the pattern relation p_i is left undefined in the framework and is only specified insofar as it is a relation from CL into S^p. Similarly, the specifics of the partitions in S^p are undefined since these vary depending on pattern structure and content. This ensures flexibility of the framework, but it also means that actually generating patterns cannot be done with only the framework itself. The framework ensures consistency and easier means of referencing and consolidation across domains, disciplines, and pattern languages. The pattern framework is intended for a wide audience, which includes those who are not yet familiar with pattern approaches but would want to apply them in their field. Therefore, we use this section to provide a consolidated overview of some of the literature regarding pattern generation, to supplement the formal framework. This is not meant to be a comprehensive summary, but a general aid to generate meaningful patterns in this particular framework. In the following sections, we mainly draw from Meszaros and Doble's [9] pattern language for pattern writing, Winn and Calder's [10] pattern language for pattern language structure, and some of Borchers' [7] considerations regarding pattern generation.

A. Defining the structure

One of the primary steps when beginning to build a pattern language – and elementary to partitioning the statements that will later constitute the individual patterns – is to define the pattern structure. By that, we do not mean the relations or hierarchy between patterns, but rather the number of sub-categories or fields of each pattern. Pattern structures exist in a wide variety of granularity. Tidwell's [12] pattern structure is minimalistic but effective, with only six subcategories (what, use when, why, how, examples, in other libraries), whereas the structure introduced by Gamma et al. [4] propose 13 subcategories for each of their patterns. The exact number of subcategories should be decided on individual needs, preferences, and also available resources (more subcategories = more complicated and longer pattern mining process). However, there are a few basic subcategories, which each pattern structure should contain. We present these, together with the reasons why we consider them to be essential, in the following.

Name: Patterns should be uniquely identifiable, so that they can be referred to and structured with regard to other patterns. Therefore, each pattern should have a unique name that clearly distinguishes it from other patterns. It is furthermore helpful if that name is not obtuse or even presents an image of the suggested solution in the reader's mind (Meszaros and Doble refer to this as an "evocative pattern name" [9]).

Problem: One of the major distinguishing features of patterns is their problem-centric nature. If the pattern does not present a solution to a (reoccurring) problem, then it only provides general guidance and serves the same purpose as a guideline, but without the comprehensive character a guideline usually provides. Therefore, a separate description of the problem is considered essential for a successful pattern.

Context and/or Forces: Patterns contain proven, working solutions, which means that these solutions solved the problems in particular cases. Therefore, understanding and documenting this context is elementary for being able to decide whether a particular solution is suitable for a different (even when similar) context. Forces are the aspects of the context that the solution is supposed to optimize. They are important, but not always considered as separate pattern subcategories (e.g., [1], [12]). Therefore, we only consider one of them as essential – unlike Borchers [7] and van Velie [5]. The bottom line is that each pattern should, at the very minimum, contain some kind of description of its context as a separate entry – whether it be *context* or *forces* (or both).

Solution: A seemingly obvious point that is never the less worth pointing out. Each pattern should contain a description of the actual solution as a separate entry. This is not the same as a simple screenshot of a working example, but rather a detailed textual description of the steps taken to solve the problem in its particular context.

Examples: Since the solution described by the pattern is supposed to be a proven one, concrete examples (preferably more than one) should be provided to show the end result of the implemented solution. These examples are closely related to, but not the same as, the solution. They help to put the general solution into more practical terms and link the solution to its context. In the case of several implementation examples being available, they can also aid the designer in identifying essential commonalities between application contexts. This can be an additional aid, when a designer is not sure whether a pattern would be suitable for their particular context.

A successful pattern structure can have as many pattern subcategories as needed, though the ones listed above should be considered a reasonable minimum for any pattern structure. The minimum requirements we presented here are very similar to those given by Gamma et al. [4] (pattern name, problem, solution, consequences), but with slight extensions and modifications for wider applicability. We also decided to not include an implementation's consequences as a necessary component, since that might be a confusing concept for patterns outside of areas in which consequences can be traced more easily (such as in software code, where changes and the parts they affect can be more or less fully described).

B. *Mining and Iteration*

In order to generate meaningful patterns, the solutions contained therein need to be discovered first. Pattern generation is a difficult and lengthy process, which usually occurs in several phases. Köhne [25] describes the pattern generation process as consisting of the following 8 stages: pattern mining – pattern writing – shepherding – writer's workshop – review by pattern author – collection of pattern in repository – peer review – pattern book publication. While this is a good overall summary of how pattern generation or finding occurs, Pattern creation does not always follow these exact steps in reality. There is no single accepted method or process for pattern generation, but there are several useful recommendations for generating successful patterns by Borchers [7], Martin [15], Vlissides [17], and others. In the following, we present what we consider the bare minimum of what a pattern generation process should entail.

The first step in generating a pattern is recognizing the problem and its reoccurring nature. There is no standard procedure for this and Appleton [18] even notes that the best way to learn how to recognize patterns is to learn from others who were able to do so successfully. This is why pattern generation should happen in several stages. Anyone, who has worked in a certain field for some time, should be able to eventually spot problems that have manifested themselves over and over in the past. They might also be able to recognize certain regularities in the solutions that were employed to solve the problem in all its past occurrences. To go from this initial pattern assessment to a complete pattern, examination and iteration should happen in several steps and by several people, so that the essence of the solution can be extracted and adequately described. Furthermore, reexamination and iteration should be done by several individuals. These pattern iterators will then rework the patterns to suit their readability requirements, i.e., the resulting pattern will automatically be written and formatted for easier readability for a wider audience. Even if the pattern started out as a simple assumption about a potential solution, at the end, the pattern contains the know-how of all its iterators and a quantitative component that complements the pattern content. After all, if multiple experts came to similar conclusions about a problem and its solution, then this lends support to the assumption that the solution is indeed a working one and the problem a reoccurring one. Thus, it can be possible even for people who are inexperienced in pattern generation to come up with successful patterns.

Therefore, the most important steps any successful pattern generation process should contain are (a) *problem identification* to define the elementary parts, context, and eventually the solution; (b) *structuration* to guarantee a uniform format, good readability, and completeness of patterns with the same structure; and (c) *reflection and feedback* to examine whether the solution is a working one and ensure sufficient detail of its explanation to allow easy application.

C. *Piecemeal Growth*

This point is based on Winn and Calder's [10] suggestion by the same name. They suggest, "if new structure needs to be added to the system, then add it gradually, piece by piece, evaluating the effect of the change on the whole." In their work, Winn and Calder have applied this to systems (software, architectural, biological), as well as pattern languages. In this paper, we adapt their ideas only for the generation of pattern languages.

Building a full pattern language is a lengthy process, which begins with a few patterns. As more solutions are discovered, more patterns can be created, which culminates in a full pattern language, once a certain number and level of comprehensiveness of patterns is reached. This means that new solutions and, therefore, new patterns must be considered in light of already existing solutions. It is possible that a new solution is incompatible with an already established solution, where both problems usually occur together. In such a case, parameters must be provided that allow deciding when one or the other solution should be applied. Similarly, a newly introduced solution might be superior to a previous solution, rendering its respective pattern obsolete. This must be reflected in the pattern language, as they would otherwise seem like equally effective solutions to the same problem. Therefore, changes and additions to any existing patterns should occur in small steps, while re-evaluating the existing patterns in light of these new additions.

In terms of the pattern framework, this means that newly generated patterns should ideally entail review and potential modification of descriptors. Since descriptors allow mapping patterns to overall goals, modifications to the existing patterns themselves should seldom be necessary. An initial pattern collection might only have a single descriptor, since the patterns are likely to be generated with one overall goal or problem in mind. However, it is very possible that a new pattern presents a solution that often, but not always, occurs with other problems for which patterns are available. In such cases it is recommended to create to separate descriptors that cover both cases – those, in which both problems occur and those, in which they do not. The same is true for conflicting patterns. These can be put into different descriptors, thus making these conflicts visible without a need for modification of any of the patterns themselves. In the case of outdated patterns, these can simply be left as they are, but not made part of any descriptor. Therefore, they are still available for reference purposes, but not part of any recommended set of solutions.

Cases, such as the ones described above, which necessitate a restructuring of both new and existing patterns, can happen at any stage in the pattern language development process. However, the additional pattern does not necessarily entail a new descriptor. It could simply be added to an existing one or prompt the creation of several new ones, all depending on the individual case. Therefore, the growth of a pattern language's complexity cannot be considered linear in regard to the number of patterns it contains.

The development of a pattern language can be seen as an organic process, where changes and additions can have wide-ranging consequences. Therefore, such changes and additions should happen in small steps, followed by a reexamination of the pattern collection. In the framework, this reexamination should almost always happen at the descriptor-level.

D. *Cross Linkage*

This point ties in with the previous one and is, once again, strongly grounded in Winn and Calder's [10] suggestion by the same name. They state, "if the system structure is complex, then overlap and use cross linkages to capture complexity." The general idea is that linear or linear-hierarchical structures cannot be a catch-all for complex structures. A pattern structure should allow cross-linking and overlaps between its elements, so that it can support complex structures.

In the previous section, we explained that even a single new pattern could potentially entail fundamental changes to the overall pattern collection. In the framework, this can manifest itself as the creation several new descriptors or the vanishing of older, outdated descriptors. In order for this flexibility to be possible, overlaps and links between the descriptors must be possible, which is the case for the framework due to its basis in set theory.

Different descriptors can largely have the same contents, with only minor differences, to satisfy different Targets. For example, two descriptors might differ in one only containing one more pattern than the other, thus dealing with a special case of the other's, more general, Target. They might even be identical regarding their elements, but with different sequence sets. One of these descriptors could then serve as a solution to a hierarchical occurrence of the problem, the other to a differently structured overall problem. Patterns are supposed to be reused in similar contexts; the descriptors, therefore, support that reuse and allow multiple occurrences of the same pattern and overlaps between descriptor contents. To adequately support the nonlinear growth in pattern language complexity when new patterns are added, it is important to generate as many new descriptors as necessary once new links between patterns or Target hierarchies are discovered.

It is not uncommon that a pattern language would start out as a neatly organized string of patterns that all serve one universal goal. As the language's complexity grows, so should its level of detail. A neatly organized descriptor variety helps structuring and reapplication of patterns for different contexts. It is also an invaluable aid for efficient and quick searching and finding of solutions to particular problems. A designer or practitioner will likely not need the whole pattern language for any given task, but also not know which individual patterns they do need, unless they read through the whole pattern catalogue. By employing the proposed method, only the individual descriptors need to be read to identify, whether a patter cluster that provides a solution to a certain goal or not. Once one is found, the reader is led through all relevant patterns, their links, and in the proper sequence via the descriptor's structure.

VIII. The Framework applied – Car User Experience Patterns

An actual pattern collection usually takes either the form of a (often online) pattern database or a printed volume. The framework was constructed mainly with databases in mind, since the added flexibility by using descriptors is easier to realize when existing input can be added to (which is difficult to do with published paper collections). In addition, the sets of statements that make up each pattern category are directly translatable into data fields and the descriptors can then point to these data. While the framework loses some of these advantages when applied to a paper-based pattern collection instead of a database, it is still feasible to use it for that purpose. In this section, we present an example of a paper-based User Experience design pattern collection, which was structured using the universal pattern framework.

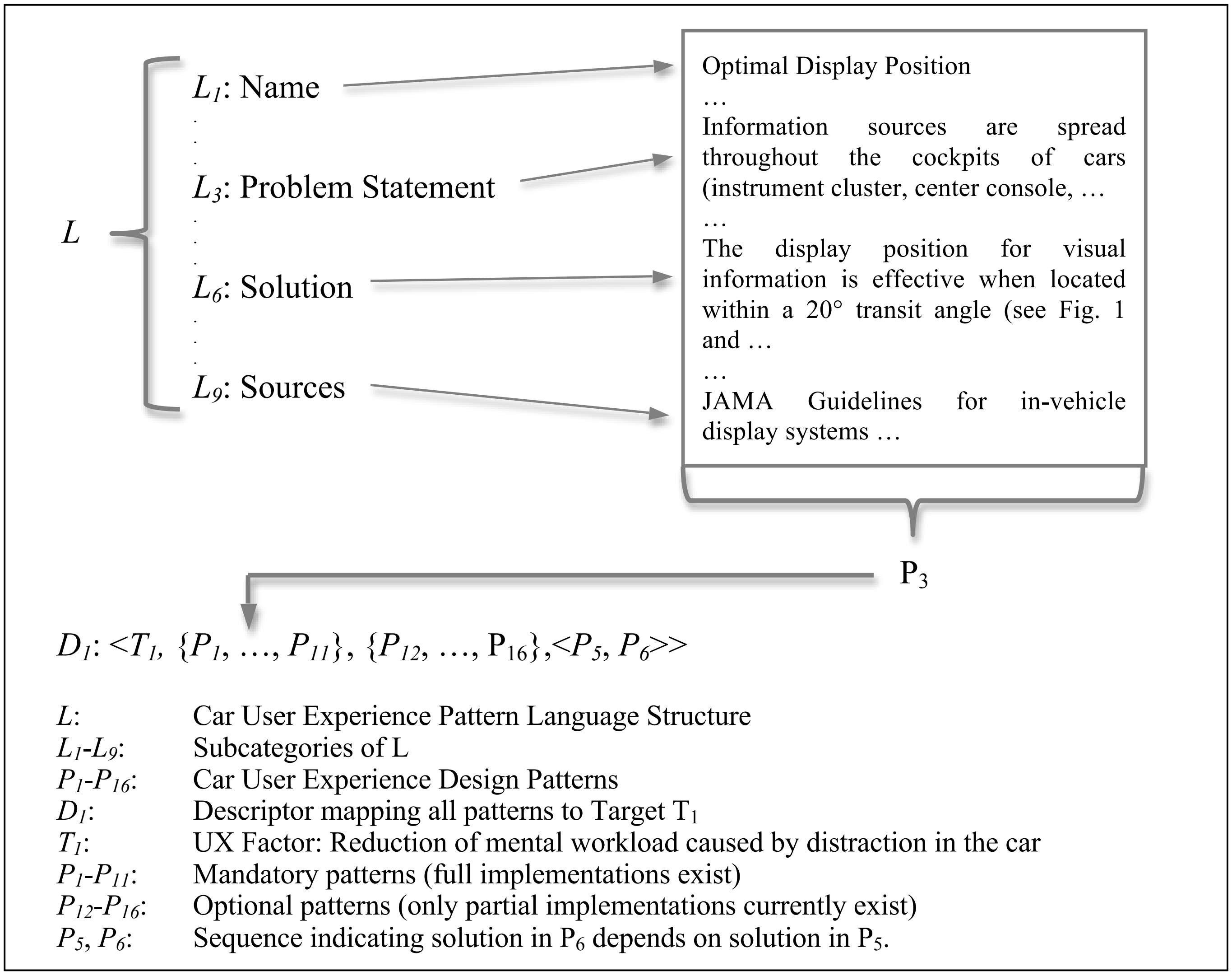
The pattern collection consisted of 16 individual patterns. All of them were about design problems in the car with the aim to reduce mental workload while interacting with the interface. The actual pattern finding process is described in detail in [26]. The resulting patterns all followed the same structure, which consisted of nine categories of statements (Name, Intent, Topics, Problem, Scenario, Solution, Examples, Keywords, Sources). Since the descriptor still enables structuring towards overall goal and regarding pattern sequence and status (mandatory vs. optional), we created one descriptor to serve as an index for the whole pattern collection.

The overall goal of every pattern was to provide design solutions that reduce mental workload, so the appropriate Target became *UX Factor: Reduction of mental workload caused by distraction in the car*. 'UX Factor' was added since this is one of several factors that are postulated to influence UX and to distinguish these from later patterns that address different influences on UX. The patterns were findings from scientific works, supplemented with implementation examples, and iterated in collaboration with industry stakeholders. Due to this somewhat nonstandard approach, there were some patterns that had more implementation examples and more straightforward instructions on how to put their respective solutions into practice. However, others provided less such examples and were perceived to be more suited for more experienced designers. Therefore, this second type of patterns was considered optional and only for those who have the necessary skills to put the proposed solutions into practice.

Thus, we described these two sets of patterns via the descriptor's sets for mandatory and optional patterns and references. Finally, there was one pattern that relied on another pattern from the same collection. Using the solution in the first pattern could sometimes create an additional problem, which the second pattern would help to solve. But it would have been misleading to imply a necessary connection and write one single pattern for both problems, since they occur together only sometimes, but not always. In order to adequately represent this relation, the two patterns were put into the sequence set to indicate that reading the first should always entail reading the second one afterwards. In the text, we indicated this with one sentence between the patterns explaining the possible link. With all this taken into account, the resulting descriptor looked as follows:

$$D_1: <T_1, \{P_1, \ldots, P_{11}\}, \{P_{12}, \ldots, P_{16}\}, <P_5, P_6>> \quad (29)$$

This was then transformed into an index. The Target served as the overall headline, patterns 1 to 11 and 12 to 16 were put into separate subsections, and patterns 5 and 6 were put into sequence and linked explicitly with additional text between the two patterns. By using the framework approach, we were able to easily structure the pattern collection in a meaningful way, even though the framework contains no information that would be specific to the car or UX-domains. Moreover, the thusly-structured pattern collection can still be put into a database, without a need for



L
L_1: Name
L_3: Problem Statement
L_6: Solution
L_9: Sources
Optimal Display Position
…
Information sources are spread throughout the cockpits of cars (instrument cluster, center console, …
…
The display position for visual information is effective when located within a 20° transit angle (see Fig. 1 and …
…
JAMA Guidelines for in-vehicle display systems …
P_3
$D_1: <T_1, \{P_1, \ldots, P_{11}\}, \{P_{12}, \ldots, P_{16}\}, <P_5, P_6>>$

L:	Car User Experience Pattern Language Structure
L_1-L_9:	Subcategories of L
P_1-P_{16}:	Car User Experience Design Patterns
D_1:	Descriptor mapping all patterns to Target T_1
T_1:	UX Factor: Reduction of mental workload caused by distraction in the car
P_1-P_{11}:	Mandatory patterns (full implementations exist)
P_{12}-P_{16}:	Optional patterns (only partial implementations currently exist)
P_5, P_6:	Sequence indicating solution in P_6 depends on solution in P_5.

Figure 3. Car User Experience Pattern Descriptor and Pattern Example

any substantial restructuring work, since the set- and descriptor structures are consistent among all pattern collections that are based on the framework.

This collection of 16 patterns is part of a prospected larger collection spanning two more UX factors in addition to the first one. At the writing of this paper, the pattern finding process for these additional two UX factors had not yet been completed. Nevertheless, the end result will be a collection of several patterns, which are mapped to the three different UX factors and structured internally via the framework's proposed descriptor structure. Figure 3 provides an additional overview of the descriptor structure, along with an example for how an individual pattern relates to the pattern structure in the framework. To reiterate, an individual pattern is a set, which contains several subsets of statements. A pattern collection or language consists of several such sets. Descriptors are separate sets, which are used to map individual patterns to an overall goal, and can thus be used to structure the pattern set as well as map lower- to higher-level patterns. There is no a priori limit to the amount of descriptors that can be created for any given pattern collection. The number of descriptors depends on the amount of goals, which are identified and/or deemed necessary for any given purpose.

IX. Discussion

Next, we discuss the proposed framework, address benefits, possible shortcomings, and future work potentials.

A. Benefits of using the framework

The framework provides a flexible basis purely by virtue of its formal features. The basic and uniform structure enables any adequately structured set of statements to be considered a potential pattern, so as long as an area or discipline can satisfy this minimal requirement, it can use the framework to structure its patterns. This general applicability also means that the framework cannot serve as a suitable means to verify a pattern's (or structure's) validity or soundness on its own. What the framework offers is a consistent basis, which the individual disciplines can build upon. Pattern languages can be described as partitioned sets of statements within the framework. As long as the structure of a certain pattern collection is known, its individual patterns and sub-parts of patterns can be referenced within the framework by referencing the appropriate set or sub-set.

Thanks to the descriptor structure, linkage between patterns and pattern languages within the framework is possible, even between patterns from different disciplines. In such a case, the differences in pattern structure must be known and appropriately modeled in the framework, since it is very likely that their structures do not consist of the same sub-categories. Mapping patterns to overall Targets can reduce redundancy and allows mapping of lower-level patterns to higher-level goals. The standard structure of Descriptors allows structuring patterns with regard to priority (mandatory vs. optional) as well as sequences of problems or their solutions. Finally, all these features are available on the very basic framework-level, and are thus not dependent on any particularities of the actual pattern content or the discipline they belong to. Thus, the initial goal of the framework not being bound to any individual discipline or domain is achieved.

B. The set-theoretic basis and its multi-domain suitability

As initially stated, the framework is intended as a basis for patterns as a general knowledge transfer tool, suitable for a multitude of disciplines and domains. However, employing mathematical methods might seem to limit the framework to only those disciplines already familiar with such methods, which is why we briefly discuss the need for this mathematical basis and its consequences for applications of the framework. The framework was developed with databases, as well as paper-based pattern collections in mind. Therefore, a suitable framework should fulfill the minimum requirements of consistency and division of information into separate categories or data fields. This ensures that any pattern from such a framework can be used as input for a database, by treating the pattern subcategories as datafields in the database. By keeping that structure the same for both database and paper-based pattern collections, compatibility and consistency between the two is ensured. This also permits any paper-based collection built in this framework to be incorporated into a database of the same format.

The formulae in Sections V and VI are accompanied by explanations, so that the purpose of the theoretical basis can be understood without necessarily having to understand the methods themselves. Thus, the framework does not require knowledge of mathematics or formal methods to be applied, as long as the separation of patterns into statement categories and the meaning of the descriptor contents are understood by the reader. Such an application of the framework would likely result in a well-structured paper pattern collection, like the one shown in our example in Section VIII. However, as example also showed, a paper-based collection loses some of the framework's advantages. This issue is inherent to the medium, as it is generally difficult to update or crosslink published volumes (short of releasing updated reprints). We do not think that there is any framework that could solve this fundamental issue, so the minimum requirement of handling databases must be fulfilled by anyone who intends to apply the framework to its full extent.

The set theory employed in this framework is elementary and based on conventional (Boolean) logic. The reason for this is, once again, the desire to keep the framework as easy to understand and handle as possible. But furthermore, we believe that for achieving the goals outlined in Section I, conventional elementary set theory is absolutely sufficient, as we merely arrange statements in sets and a statement is then either present in a given set or it is not. There are no degrees involved here that would warrant employing fuzzy operations or sets. The same goes for other extensions to conventional logic and set theory: unless they are needed, they would only complicate matters without adding any tangible benefit (and since they are often supersets of conventional set theory, the framework could still be extended on an as-needed-basis in special cases). A more complicated underbelly would probably not matter for the average reader with an IT-background and who is already familiar with logic to some degree. But for those with different backgrounds, it might create an additional hurdle that we would rather avoid. The framework is rather simple on a formal level but it achieves what it was meant to do just as well. In this regard, we see the framework to strike the best possible balance between necessary skill level of the user and application possibilities.

C. *Finding patterns and descriptors – it's not that easy*

Putting patterns into a meaningful structure is only one step in any pattern finding process, although a rather important one. The purpose of this paper was to provide a *basis* for patterns as a universal tool, and not a complete guide for discipline-independent pattern finding. Nevertheless, if we want the framework to be successful, then it should ultimately be applied in areas, in which there have been no (or few) pattern approaches in the past. In such areas, simply providing a framework without any guidance on how to actually *find* patterns would arguably be of little use. Therefore, we included a number of recommendations based on existing pattern approaches in Section VI. We consider these recommendations elementary enough to be sensible for any pattern collection and, therefore, a suitable supplement to the framework. On the other hand, the elementary and general nature of these recommendations also means that they are, at best, necessary (but not sufficient) conditions for successful pattern finding. We acknowledge that the recommendations given in Section VI constitute a sensible starting point but not a complete pattern finding guideline, and that more work on pattern finding (both within and across disciplines) is needed.

D. *Tool support*

The framework is, in its current state, not supported by a tool or any other automated means that could aid the user in finding patterns or creating a pattern language. The framework provides a basis that is consistent among disciplines but most of the necessary legwork still has to be done by the individuals themselves. This is not something that cannot be fully eliminated, but a completely unassisted framework is a lot less accessible than it could be, esspecially considering our aim of domain-independent applicability.

There are specialized tools that can aid solution finding in certain contexts; the EXPLAINER tool by Redmiles et al. [27][28] is one such tool. Tools like this one might be reusable in other disciplines as well, but it can be expected that full tool support from pattern finding to arrangement in a language, can probably not be handled on a universal level by one single tool. However, since the basic framework is essentially a means to structure statements and set them in relation to each other, there is no reason why it shouldn't be possible to simply provide a database input mask that assists with the most common operations (defining number of category-subsets, labels, adding the statements, defining descriptors with predefined subsets, etc.). This is something that would greatly aid users in applying the framework and we hope to be able to provide such an aid further down the line.

E. *Wider application*

In Section VIII, we provided an example of an actual application of the framework in practice. The example was for a paper-based pattern collection, which illustrated how the descriptor structure can be used for meaningful categorization within a pattern collection with relatively little effort. Overall, the example might seem rather unspectacular, especially since it only resulted in the creation of one single descriptor. What we did not show was an actual pattern database that makes full use of all of the framework's advantages (most importantly, multiple descriptors for overlapping pattern sets and reference to sources or patterns from outside). We intend to use the framework for many more future pattern collections (including databases), so that more application examples will eventually become available. At this point, the framework is still very new and we do not have a complete database that would be suitable for demonstration purposes. However, we do think that the framework is outlined in sufficient detail in this paper to allow successful application at this stage and we encourage the community to use (and criticize) the framework, as only actual use can really show it suitability (or lack thereof).

X. CONCLUSION

In this paper, we have provided a formal framework that supports finding and structuring patterns independent of their domain, field or discipline, supplemented with information on how to generate actual content (i.e., finding patterns) and gave an example of an application of the framework in practice.

In our framework, patterns are separate from descriptors, which are themselves separate from their targets. This means, that patterns can be generated as usual and assigned on an as-needed basis. For the pattern user, this means that they do not have to scour vast databases of patterns for those they might need. All they need is to have a look at the descriptor(s) that is/are assigned to the target they have in mind. For the pattern provider, there is also the added advantage existing pattern databases can be expanded with descriptors, which help make them more usable and reduce the amount of domain experience and previous knowledge required in order to employ patterns successfully. The example we provided in Section VIII is one such case. The paper version can be made into a database using the same structure and format. Additional descriptors and/or patterns can then be added and the collection expanded as needed.

Descriptors can function similarly to references contained in the patterns themselves (as suggested by Borchers [7]), but enable additional or alternative references to other patterns at any time, since they are not actual parts of a pattern. This means that descriptors can be used to describe virtually any pattern set, regardless of which domain(s) its patterns came from or when the pattern was created. Not only it is possible to capture the hierarchical order of existing pattern languages via descriptors, but also reference patterns from other languages that might fit a certain purpose. This means that the framework is not tied to a single pattern language or even a single domain and permits references to patterns from multiple pattern languages. The framework still needs to be adopted and used on a wider scale, in order to prove its suitability in practice. Nevertheless, due to its general basis and viability for both pattern databases and paper-based pattern collections, we consider it an appropriate basis for patterns as a domain independent knowledge transfer tool. We will use the framework as a basis for our future pattern collections (including a pattern database implementation) and further iterate the framework, as new insights from such use cases are gained.

ACKNOWLEDGMENT

We gratefully acknowledge the financial support by the Austrian Federal Ministry of Economy, Family and Youth and the National Foundation for Research, Technology and Development (Christian Doppler Laboratory for „Contextual Interfaces").

REFERENCES

[1] A. Mirnig and M. Tscheligi, “Building a General Pattern Framework via Set Theory: Towards a Universal Pattern Approach,” The Sixth International Conference on Pervasive Patterns and Applications (PATTERNS 2014) IARIA, Venice, Italy, May 2014, pp. 8-11.

[2] C. Alexander, The Timeless Way of Building, New York: Oxford University Press, 1979.

[3] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, A Pattern Language: Towns, Buildings, Construction, Oxford: University Press, 1979.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: elements of reusable object-oriented software, Boston: Addison-Wesley Professional, 1995.

[5] M. van Velie and G. C. van der Veer “Pattern Languages in Interaction Design: Structure and Organisation,” Proc. Ninth Int. Conf. on Human-Computer Interaction, IOS Press, IFIP, Zürich, 2003, pp. 527-534.

[6] D. May and P. Taylor, “Knowledge management with patterns,” Commun. ACM 46, 7, July 2003, pp. 94-99, DOI=10.1145/792704.792705, retrieved: May 2015.

[7] J. Borchers, A pattern approach to interaction design, New York: John Wiley & Sons, 2001.

[8] A. Krischkowsky, D. Wurhofer, N. Perterer, and M. Tscheligi, “Developing Patterns Step-by-Step: A Pattern Generation Guidance for HCI Researchers,” Proc. PATTERNS 2013, The Fifth International Conferences on Pervasive Patterns and Applications, ThinkMind Digital Library, Valencia, Spain, May 2013, pp. 66–72.

[9] G. Meszaros and J. Doble, “A pattern language for pattern writing,” Pattern languages of program design 3, Robert C. Martin, Dirk Riehle, and Frank Buschmann (Eds.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, November 1997, pp. 529-574.

[10] T. Winn and P. Calder, “A pattern language for pattern language structure,” Proc. 2002 Conf. on Pattern Languages of Programs - Volume 13 (CRPIT '02), James Noble (Ed.), Volume 13. Australian Computer Society, Inc., Darlinghurst, Australia, June 2003, pp. 45-58.

[11] K. Devlin, The Joy of Sets: fundamentals of contemporary set theory, 2nd ed., Springer, 1993.

[12] J. Tidwell, “Designing Interfaces : Patterns for Effective Interaction Design,” O’Reilly Media, Inc., 2005.

[13] A. Dearden and J. Finlay, “Pattern Languages in HCI: A Critical Review,” HCI, Volume 21, January 2006, pp. 49-102.

[14] S. Günther and T. Cleenewerck, “Design principles for internal domain-specific languages: a pattern catalog illustrated by Ruby,” Proc. 17th Conf. on Pattern Languages of Programs (PLOP '10). ACM, New York, NY, USA, Article 3, pp. 1-35, DOI=10.1145/2493288.2493291, retrieved: April, 2014.

[15] D. Martin, T. Rodden, M. Rouncefield, I.Sommerville, and S. Viller, “Finding Patterns in the Fieldwork,” Proc. Seventh European Conf. on Computer-Supported Cooperative Work, Bonn, Germany, September 2001, pp. 39-58.

[16] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, and F. Leymann, “Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains”, International Journal on Advances in Software, issn 1942-2628, vol. 7, no. 3 & 4, 2014, pp. 710-726, http://www.iariajournals.org/software/, retrieved: May 2015.

[17] J. Vlissides, Pattern Hatching: Design Patterns Applied (Software Patterns Series), Addisoin-Wesley, 1998.

[18] B. Appleton, Patterns and Software: Essential Concepts and Terminology, February 2000 http://www.bradapp.com/docs/patterns-intro.html, retrieved February 2015.

[19] E. Blomqvist, K. Sandkuhl, “Patterns in Ontology Engineering: Classification of Ontology Patterns,” Proc. Seventh Int. Conf. on Enterprise Information Systems (ICEIS 2005), Miami, USA, May 2005, pp. 413-416, retrieved: May 2015.

[20] R. A. Falbo, G. Guizzardi, A. Gangemi, V. Presutti, “Ontology Patterns: Clarifying Concepts and Terminology,” Proc. 4th Workshop on Ontology and Semantic Web Patterns (WOP 2013), CEUR-WS, Sydney, Australia, 2013, retrieved: May 2015.

[21] M. Poveda-Villalón, M.C. Suárez-Figueroa, A. Gómez-Pérez, “Reusing Ontology Design Patterns in a Context Ontology Network,” Proc. Second Workshop on Ontology Patterns (WOP 2010), CEUR-WS, Shanghai, China, 2010, pp. 35-49, retrieved: May 2015.

[22] O. Noppens and T. Liebig, “Ontology Patterns and Beyond – Towards a Universal Pattern Language, “ Proc. Workshop on Ontology Patterns (WOP 2009), CEUR-WS, Washington D.C., USA, 2009, pp. 179-186, retrieved: May 2015.

[23] A. Gangemi and V. Presutti, “Ontology Design Patterns,” Handbook on Ontologies, Second ed., S. Staab, R. Studer (Eds.), Springer, 2009, pp. 221-243.

[24] E. Blomqvist, A. Gangemi, V. Presutti, “Experiments on Pattern-based Ontology Design,” Proc. 5th Int. Conf. on Knowledge Capture (K-CAP 2009), September 2009, Redondo Beach, California, USA, pp.41-48, retrieved: May 2015.

[25] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, “Automating cloud application management using management idioms,” Proceedings of the Sixth International Conference on Pervasive Patterns and Applications (PATTERNS), pp. 60–69, May 2014.

[26] A. Mirnig, A. Meschtscherjakov, N. Perterer, A. Krischkowsky, D. Wurhofer, E. Beck, A. Laminger, and M. Tscheligi, “Finding User Experience Patterns Combining Scientific and Industry Knowledge: An Inclusive Pattern Approach,” The Seventh International Conference on

Pervasive Patterns and Applications (PATTERNS 2015) IARIA, Nice, France, March 2015.

[27] D. F. Redmiles, “Reducing the Variability of Programmers’ Performance Through Explained Examples,” Proc. INTERCHI ’93 Conf. on Human Factors in Computing Systems, IOS Press Amsterdam,Amsterdam, The Netherlands, 1993, pp. 67-73.

[28] C. Rathke, D. F. Redmiles, “Improving the Explanatory Power of Examples by a Multiple Perspectives Representation,” Proc. 1994 East-West Conf. on Computer Technologies in Education (EW-ED ’94), Crimea, Ukraine, September 1994, pp. 195-200.