Deriving a Pattern Language Syntax for Context-Patterns

Kristian Beckers, paluno – The Ruhr Institute for Software Technology
Stephan Faßbender, paluno – The Ruhr Institute for Software Technology
Maritta Heisel, paluno – The Ruhr Institute for Software Technology

In a previous publication we introduced a catalog of context-patterns. Each context pattern describes common structures and stakeholders for a specific domain. The common elements of the context were obtained from observations about the domain in terms of standards, domain specific-publications, and implementations. Whenever the domain of a system-to-be is already described by a context-pattern, one can use this context-pattern to elicit domain knowledge by instantiating the corresponding context-pattern. Moreover, we analyzed the common concepts in our context-patterns and created a meta-model to describe the relations between these concepts. This meta-model was the initial step towards a pattern language for context-patterns. In this work, we show the consequent next step for the definition of a pattern language syntax for context-patterns. Thus, we describe how to derive the connections between the existing context-pattern in a structured way and present the results.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*System architectures,Systems specification methodology*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Elicitation methods, methodologies*; H.1.1 [**Information Systems**]: Systems and Information Theory—*General systems theory*

General Terms: Documentation

Additional Key Words and Phrases: Domain Knowledge, Requirements Engineering, Context Establishment

ACM Reference Format:

Beckers K., Faßbender S. 2014. Deriving a Pattern Language for Context-Patterns jn 2, 3, Article 1 (May 2010), 25 pages.

1. INTRODUCTION

The long known credo of requirements engineering states that it is challenging to build the right system if you do not know what right is. Requirements engineering methods have to consider domain knowledge, otherwise severe problems can occur during software development, e.g., technical solutions to requirements might be impractical or costly. It is an open research question of *how* to elicit domain knowledge correctly for effective requirements elicitation [Niknafs and Berry 2012]. Fabian et al. [Fabian et al. 2010] concluded in their survey about existing requirements engineering methods that it is not yet state of the art to consider domain knowledge.

We propose to built patterns for a structured domain knowledge elicitation. Depending on the kind of domain knowledge that we have to elicit for a software engineering process, we always have certain elements that require consideration. We base our approach on Jackson's work [Jackson 2001] that considers requirements engineering from the point of view of a machine in its environment. The machine is the software to be built and requirements

Author's address: Kristian Beckers, Oststrasse 99, 47057 Duisburg, Germany; email kristian.beckers@uni-duisburg-essen.de; Stephan Faßbender, Oststrasse 99, 47057 Duisburg, Germany; email: stephan.fassbender@uni-due.de; Maritta Heisel, Oststrasse 99, 47057 Duisburg, Germany; email: stephan.fassbender@uni-due.de

are the effect the machine is supposed to have on the environment. Our patterns do not enforce considering the machine explicitly, but demand a description of the environment.

Our context-patterns support the structured elicitation of domain knowledge and we showed a number of these in the previous works of ours [Beckers et al. 2012; Beckers et al. 2012; Beckers and Faßbender 2012; Beckers et al. 2012]. However, these patterns are isolated from each other and a common language is missing that describes how a problem that requires multiple patterns can be solved by their combination. Our intention is that the pattern catalog should grow, and providing an easy way for engineers to describe further context-patterns is a step towards this aim. Hence, our pattern language has the intention to support engineers in a better understanding of context-patterns, describing further context-patterns, and finding a useful way to combine context-patterns to solve a problem.

In this work, we will focus on the relations between context-patterns. These relations form the syntax of a pattern language. For a definition of the term syntax see Sect. 1. We will show how one can derive these relations using so called pattern relation tables, and which steps one has to take to fill such a table. Additionally, we present the resulting relations between the already existing context-patterns.

Some similar advances exist in the related work. Winn and Calder [Winn and Calder 2003] describe a pattern language for pattern languages in their work. It contains different patterns which can be applied to solve different problems occurring while deriving a pattern language. While some of the patterns emphasise the importance of relations between patterns, no pattern is available for the problem of finding the relations. Pauwels et al. [Pauwels et al. 2010] also stress the importance of relations between patterns. But the method for building a pattern language contains no explicit step for finding relations, nor does any other step embody the relation mining. Zdun [Zdun 2007] proposes, based on an idea of Henney [Henney 2005], to use formal grammars to refine and define pattern relations and subsequently to use the relations for pattern selection. While existing relations are refined when formalising them and additional information for the relations is collected, the initial set of relations has to be known beforehand. Hence, this work might be fruitful for further investigations and refinement of relations between context-patterns.

The remainder of the paper is organised as follows. We provide background on context-patterns in Sect. 2, and contribute a template for describing pattern languages in Sect. 3. The elements of our pattern language include a meta-model (previously introduced in [Beckers et al. 2013]), shown in Sect. 4, and relations between context-patterns and an overview of all possible sequences of context-patterns (see Sect. 5). Section 6 presents lessons learned when applying our pattern language and Sect. 7 concludes.

2. CONTEXT-PATTERNS

We developed a number of patterns for the elicitation of domain knowledge, so-called *Context-Patterns*. The pattern language syntax described in this paper refers to these context-patterns. Note that we view context in the sense of Jackson [Jackson 2001], which is a model-based description of all the domains in the environment of the machine, the thing to be built.

2.1 P2P pattern

Our P2P pattern (see Figure 1 top) is based upon the P2P architecture from Lua et al. [Lua et al. 2005], which is derived from a survey of existing P2P systems. This survey describes P2P systems as layered architectures that contain at least the following layers.

The *Application Layer* concerns applications that are implemented using the underlying P2P overlay, for example, a Voice-over-IP (VoIP) application. The *Service Layer* adds application specific functionality to the P2P infrastructure, for example, for parallel and computing-intensive tasks or for content and file management. Meta-data describe what the service offers, for instance, content storage using P2P technology. Service messaging describes the way services communicate. The *Feature Management Layer* contains elements that deal with security, reliability and fault resiliency, as well as performance and resource management of a P2P system. All

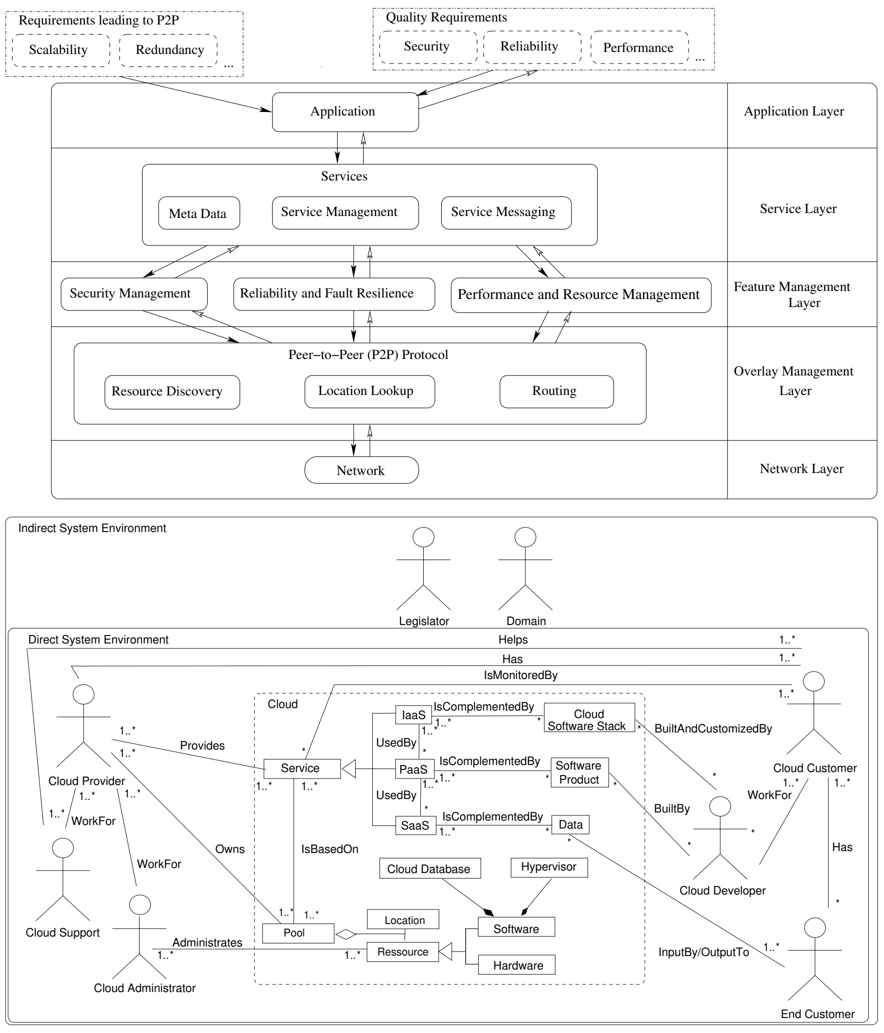

Fig. 1: P2P pattern (top) and cloud analysis pattern (bottom)

these aspects are important for maintaining the robustness of a P2P system. The *Overlay Management Layer* is concerned with peer and resource discovery and routing algorithms. The *Network Layer* describes the ability of the peers to connect in an ad hoc manner over the internet or small wireless or sensor-based networks.

2.2 Cloud Pattern

We also briefly introduce our cloud pattern (see Figure 1 bottom) [Beckers et al. 2011]. A *Cloud* is embedded into an environment consisting of two parts, namely the *Direct System Environment* and the *Indirect System Environment*. The *Direct System Environment* contains stakeholders and other systems that directly interact with the *Cloud*, i.e., they are connected to the cloud by associations. Moreover, associations between stakeholders in the *Direct* and *Indirect System Environment* exist, but not between stakeholders in the *Indirect System Environment* and the *Cloud*. The *Cloud Provider* owns a *Pool* consisting of *Resources*, which are divided into *Hardware* and

Software resources. The provider offers its resources as *Services*, i.e., *IaaS*, *PaaS*, or *SaaS*. The *Pool* and *Service* do not require instantiation. Instead, the specialized cloud services such as *IaaS*, *PaaS*, and *SaaS* and specialized *Resources* are instantiated. The *Cloud Developer* represents a software developer assigned by the *Cloud Customer*. The developer prepares and maintains an *IaaS* or *PaaS* offer. The *IaaS* offer is a virtualized hardware, in some cases it is equipped with a basic operating system. The *Cloud Developer* deploys a set of software named *Cloud Software Stack* (e.g., web servers, applications, databases) into the *IaaS* in order to offer the functionality required to build a *PaaS*. In our pattern *PaaS* consists of an *IaaS*, a *Cloud Software Stack* and a *cloud programming interface (CPI)*, which we subsume as *Software Product*. The *Cloud Customer* hires a *Cloud Developer* to prepare and create *SaaS* offers based on the CPI, finally used by the *End Customers*. *SaaS* processes and stores *Data* input and output from the *End Customers*. The *Cloud Provider*, *Cloud Customer*, *Cloud Developer*, and *End Customer* are part of the *Direct System Environment*. Hence, we categorize them as *direct stakeholders*. The *Legislator* and the *Domain* (and possibly other stakeholders) are part of the *Indirect System Environment*. Therefore, we categorize them as *indirect stakeholders*.

2.3 The SOA Pattern

Our SOA patterns concern domain knowledge for Service-Oriented Architectures (SOA). The following description is taken from [Beckers et al. 2012]. A SOA spans different layers [Beckers et al. 2012], which form a pattern on a SOA with technological focus, as depicted in Figure 2 on the top. The first and top layer is the *Business Domain* layer, which represents the real world. It consists of *Organizations*, their structure and actors, and their *business relations* to each other. The second layer is the *Business Process* layer. To run the business, certain *Processes* are executed. Organizations *participate in* these processes. These processes are supported by *Business Services*, which form the *Business Service* layer. A business service encapsulates a business function, which *performs* a process activity within a business process. All business services rely upon *Infrastructure Services*, which form the fourth layer. The infrastructure services offer the technical functions needed for the business services. These technical functions are either implemented especially for the SOA, or they expose interfaces from the *Operational Systems* used in an organization. These operational systems, like databases or legacy systems, are part of the last SOA layer at the bottom of the SOA stack. These layers form a generic pattern, the SOA layer pattern, to describe the essence of a SOA.

In Figure 2 on the bottom, we adapted problem-based methods, such as problem frames by Jackson [Jackson 2001], to enrich the SOA layer pattern with its environmental context. The white area in Figure 2 (bottom) spans the SOA layers that form the machine. The business processes describe the behaviour of the machine. The business services, infrastructure services, components, and operational systems describe the structure of the machine. Note that the business processes are not part of the machine altogether, as the processes also include actors which are not part of the machine. Thus, the processes are the bridge between the SOA machine and its environment. The environment is depicted by the grey parts of Figure 2 (bottom). The light grey part spans the *Direct Environment* and includes all entities, which participate in the business processes or provide a part, like a component, of the machine. An *entity* is something that exists in the environment independently of the machine or other entities. The dark grey part in Figure 2 (bottom) spans the indirect environment. It comprises all entities not related to the machine but to the direct environment. The Business Domain layer is one bridge between the direct and indirect environment. Some entities of the Direct Environment are part of organisations. Some entities of the Indirect Environment influence one or more organisations. The machine and the Direct Environment form the *inner system*, while the *outer system* also includes the Indirect Environment. The entities we focused on for the stakeholder SOA pattern are stakeholders, because all requirements to be elicited stem from them. There are two general kinds of stakeholders. The *direct stakeholders* are part of the direct environment, while the *indirect stakeholders* are part of the indirect environment. We derived more specific stakeholders from the direct and indirect stakeholders, because these two classes are very generic. Process actors and different kinds of providers are part of the direct environment. Legislators, domains, shareholders and asset providers are part of the indirect

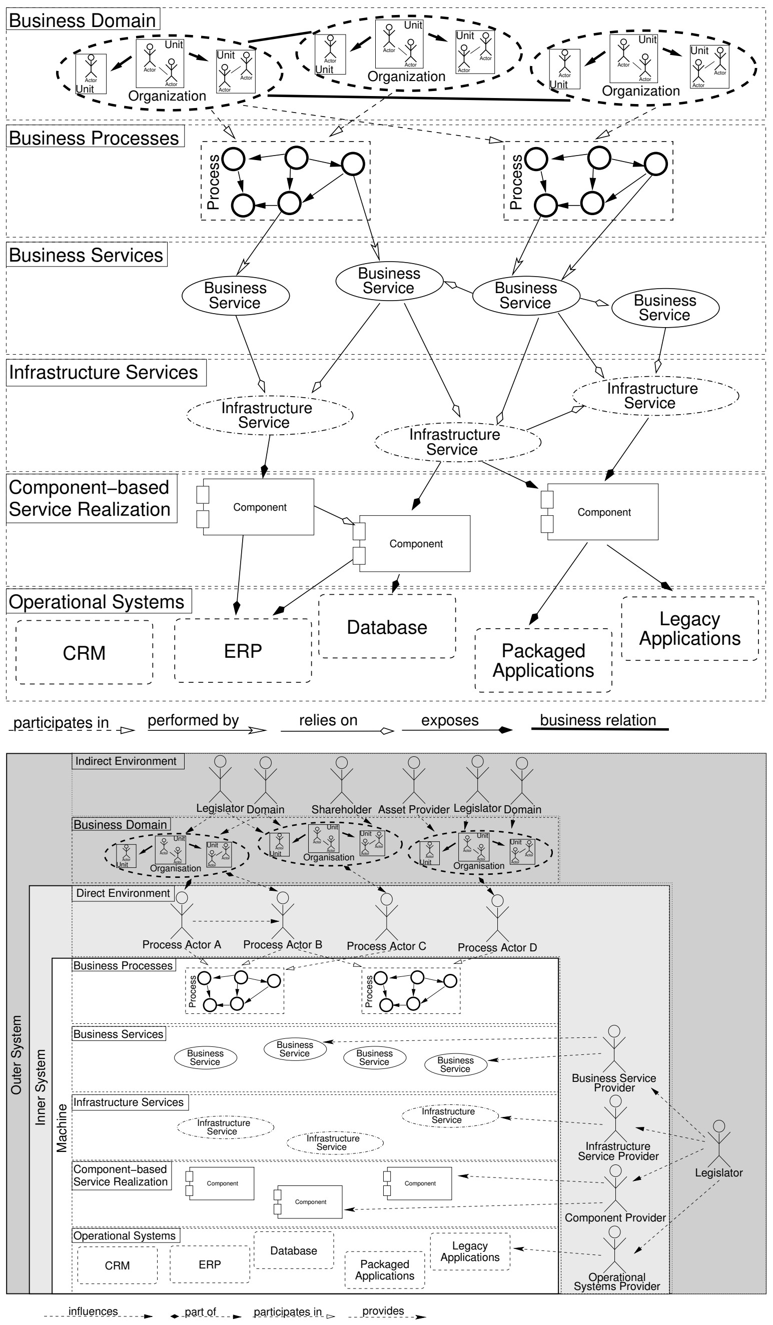

Fig. 2: SOA layer pattern (top) and SOA layer stakeholder pattern (bottom)

environment. In Figure 2 (bottom), the resulting stakeholder classes are depicted as stick figures. For a detailed description of these stakeholders, we refer to our previous work [Beckers et al. 2012].

2.3.1 *Patterns for Requirement-Based Law Identification.* We consider legal domain knowledge using a set of law patterns. Commonly, laws are not adequately considered during requirements engineering [Otto and Antón 2007]. Therefore, they are not covered in the subsequent system development phases. One fundamental reason for this is that involved engineers are typically not cross-disciplinary experts in law and software and systems engineering. To bridge this gap we developed law patterns and a general process for law identification which relies on these patterns [Beckers et al. 2012].

We investigated how judges and lawyers are supposed to analyse a law, based upon legal literature research. These insights lead to a basic structure of laws and the contained sections and how they relate to the context of a system-to-be in terms of requirements. One result of our investigations is a common structure of laws. Based on this structure of laws, we defined a *law pattern* (see Figure 3 left hand). The law pattern itself is discussed in detail in [Beckers et al. 2012]. Every dictate of justice as part of a law states that every *Addressee* who avoids or accomplishes a certain *Activity* which influences a *Target Subject* or a *Target Person*, to which an *Individual* (*Target Person*) is entitled to, has to comply with law. This information forms the *Law Structure*. The artefacts of the *Law Structure* are generalized in the *Classification* part. The addressees of this section are specializations of *Person Classifier*, the activities of *Activity Classifier*, the target subject of *Subject Classifier*, and the target person of *Person Classifier*. A Law itself is enacted by a *legislator* for a *Domain* and related to other laws (*Law Pattern*).

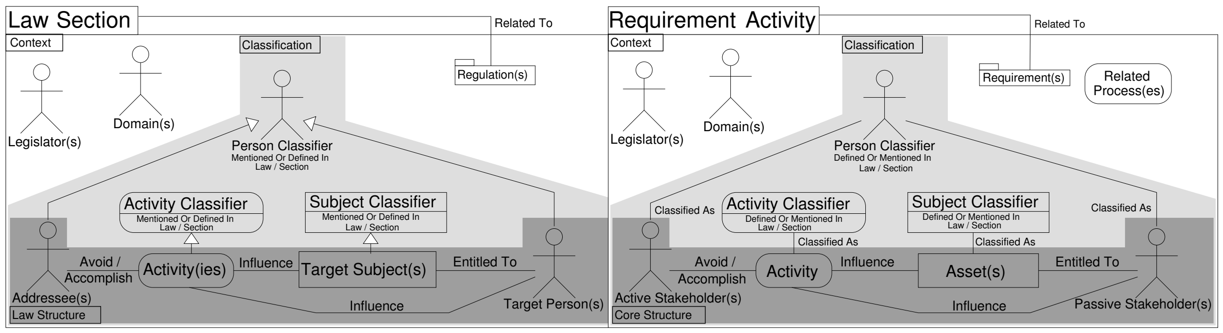

Fig. 3: Law pattern (left) and law identification pattern (right)

To determine the legal context of a system the requirements have to be related to the law pattern. Therefore, we developed the law identification pattern (see Figure 3 right hand). First of all, a *Requirement* can be related to other *Requirements* and dictates a certain behaviour of the machine. A behaviour can be a certain *Activity* or a whole *Process*. A *Process* consists of different *Activities*. An *Activity* involves an *Active Stakeholder* and in some cases an *Asset*. Additionally, an *Activity* influences a *Passive Stakeholder* in a direct way or indirect through an *Asset* which is entitled to the *Passive Stakeholder*. In addition, *Assets* can be related to each other, for example, one *Asset* is part of another *Asset*. All these relations also have to be discovered and documented. They form the *Core Structure* of the law identification pattern. Finally, the gap between the terms and notions of the technical world and the terms and notions of the legal world has to be bridged. Therefore, the parts of the core structure have to be classified using the terms of the legal world. And the context in means of *Countries* and *Domains* the system will operate in has to be set up.

3. A TEMPLATE FOR PATTERN LANGUAGES

We propose a template to describe and compare pattern languages. This template is based on the idea that a pattern language consists of the same elements the natural language does: vocabulary, syntax, and grammar. We looked into the works of Alexander [Alexander 1977] and Buschmann et al. [Buschmann et al. 2007] and their views on pattern languages (see Sect. 3.1). We discuss our template based on inspirations by their fundamental work on the area of pattern languages. In addition, we instantiated our template with several influential existing software engineering pattern languages (see Sect. 3.3) and discuss the results. Finally, we describe in Sect. 3.4 how we show an instantiation of our template for our pattern language for context-patterns.

3.1 Viewpoints of Pattern Languages

[Alexander 1977] described the term *pattern language*, which is a structured method for describing common design practices for a knowledge area. Alexander described a pattern language for creating towns and buildings in [Alexander 1977] and he wanted to empower ordinary people to successfully solve very large, complex design problems. “This language is extremely practical. It is a language that we have distilled from our own building and planning efforts over the last eight years. You can use it to work with your neighbors, to improve your town and neighborhood. You can use it to design a house yourself, with your family; or to work with other people to design an office or a workshop or a public building like a school. And you can use it to guide you in the actual process of construction.” [Alexander 1977, p. x].

Inspired by the work of Alexander we looked into the essential elements of a pattern language and state that these elements are vocabulary, syntax, and grammar. Note that Alexander did not explicitly state in his work that these are elements of a pattern language, but we argue in the following that these elements are referenced in his work. Beforehand, we define these terms for human language based on the Oxford English Dictionary (OED). The term language in the OED[1] is defined as follows: “The system of spoken or written communication used by a particular country, people, community, etc., typically consisting of words used within a regular grammatical and syntactic structure”. In addition, the OED[2] defines the vocabulary of a language as: “the body of words used in a particular language”. Moreover, the OED[3] defines the term semantic as: “relating to meaning in language or logic”. Alexander states in regard to a pattern language that “the elements of this language are entities called patterns.” [Alexander 1977, p. x]. Hence, patterns are the vocabulary of a pattern language. In addition, Alexander states that “A pattern language has the structure of a network. [...] However, when we use the network of a language, we always use it as a sequence, going through the patterns, moving always from the larger patterns to the smaller, always from the ones which create structures, to the ones which then embellish those structures, and then to those which embellish the embellishments. ... Since the language is in truth a network, there is no one sequence which perfectly captures it. But the sequence which follows, captures the broad sweep of the full network; in doing so, it follows a line, dips down, dips up again, and follows an irregular course, a little like a needle following a tapestry.” [Alexander 1977, p. xviii].

Furthermore, Alexander reasons about the use of his language in comparison to the use of the English language. “This language, like English, can be a medium for prose, or a medium for poetry. The difference between prose and poetry is not that different languages are used, but that the same language is used, differently. In an ordinary English sentence, each word has one meaning, and the sentence too, has one simple meaning. In a poem, the meaning is far more dense. Each word carries several meanings; and the sentence as a whole carries an enormous density of interlocking meanings, which together illuminate the whole.” [Alexander 1977, p. xli].

Buschmann et al. [Buschmann et al. 2007] formulate a hypothesis in their work that a pattern language is built up from over several stages. Firstly, pattern stories describe specific examples of the application of patterns in combination. Secondly, the experiences from the stories are abstracted into pattern sequences. Thirdly, numerous sequences of patterns form a pattern language. These show that the patterns can be combined in a way that helps engineers to solve problems with different solutions.

[1]The definition of the term language in the Oxford English Dictionary: `http://www.oed.com/view/Entry/105582?rskey=uuYVrM&result=1&isAdvanced=false#eid`

[2]The term vocabulary defined in the Oxford dictionaries `http://www.oxforddictionaries.com/definition/english/vocabulary?q=vocabulary`

[3]The definition of the term semantic in the Oxford dictionaries `http://www.oxforddictionaries.com/definition/english/semantic`

3.2 A Template for Describing a Pattern Language

Note that the difference between a language and a pattern language is that a language focuses on communication. In addition, a pattern language focuses on complex engineering activities. Complex engineering problems are often split up into sub-problems, which are addressed separately. Different patterns contain problems and solutions for the different granularity levels of a problem (its sub-problems). Hence, the solution to a design problem often requires the combination of different patterns applied in sequence. Moreover, in a pattern language there often exist multiple solutions to a problem, which means multiple pattern sequences. A pattern language contains all these sequences of patterns.

We propose the following template for describing a pattern language:

Patterns (Vocabulary). "The elements of this language are entities called patterns. Each pattern describes a problem which 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." [Alexander 1977, p. x]. Moreover, patterns have to be presented in a consistent format "For convenience and clarity" [Alexander 1977, p. x]. The same format also makes them easier to understand and browse.

Patterns have to describe how a solution solves a problem. This solution has to be described in a way that makes it possible to decide if this solution creates an added value (or a benefit) for the user of the pattern. Hence, the engineer can decide if the solution does create the added value he/she is looking for or if the solution should not be implemented to save time and resources. It is vital "to present the problem and solution of each pattern in such a way that you can judge it for yourself, and modify it, without losing the essence that is central to it." [Alexander 1977, p. x].

Note that Alexander does not state in one precise sentence that problem-solution pairs are an element of a pattern language. However, Alexander states [Alexander 1977, p. x] that a pattern language consists of patterns and in turn that a pattern contains the essence of a problem-solution pair. Moreover, Alexander states that all patterns of a pattern language should be described using the same format. In the following page [Alexander 1977, p. xi], Alexander states that: "There are two essential purposes behind this format. First, to present each pattern connected to other patterns, so that you grasp the collection of all 253 patterns as a whole, as a language, within which you can create an infinite variety of combinations. Second, to present the problem and solution of each pattern in such a way that you can judge it for yourself, and modify it, without losing the essence that is central to it." [Alexander 1977, p. xi]. Hence, it is our understanding that patterns in a pattern language should have the same format and the problem-solution pair is essential to a pattern. We conclude that in turn the problem-solution pair is essential to a pattern form used in a particular pattern language, as well.

Connections between Patterns (Syntax). Each solution includes syntax, a description that shows where the solution fits in a larger, more comprehensive design and which other solutions can refine this design. This relates the solution into a network of other needed solutions. For example, a larger solution might be a house for a place people want to live in. The rooms are part of the house and require ways to get light. One way to get light into a room is an electronic lamp. Another way is a candle. "In short, no pattern is an isolated entity. Each pattern can exist in the world, only to the extent that is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller pattern which are embedded in it." [Alexander 1977, p. xiii].

Pattern Sequences (Grammar). The grammar provides the meaning of sequences of patterns. Meaning with regard to patterns is a solution to a problem, which is derived by applying a sequence of patterns. Note that a pattern language allows that different sequences of patterns exist, which all solve the same problem. This is in line with Buschmann et al. [Buschmann et al. 2007], who state that numerous pattern sequences form a pattern language. This can be compared to a language in which different sentences can have the same meaning, while being syntactically different. For example, the following sentences are syntactically different but have the same meaning; (1) I have not seen the sun in a long time, and (2) It has been ages since I saw the sun. In short, a grammar explains in which places of what sequences a pattern is useful [Eloranta et al. 2014].

In several books regarding pattern languages all possible sequences of patterns are shown in a diagram, such as [Eloranta et al. 2014; Buschmann et al. 1996; Gamma et al. 1994; Schumacher et al. 2006; Schümmer and Lukosch 2007]. In some cases such as [Eloranta et al. 2014] the number of possible sequences of patterns lead to a large diagram. To address this problem, the authors show only a partial view of the diagram in the book and reference the entire diagram on a corresponding homepage. Thus, the scalability issue of the diagram size can be solved.

3.3 Software Engineering Definitions of a Pattern Language

We describe the vocabulary, syntax, and semantics of a pattern language for the related work on patterns for software engineering. We focus in particular on the works of [Fowler 1996], [Gamma et al. 1994], and [Schumacher et al. 2006]. We consider the works of Fowler, because he presented analysis patterns for capturing domain knowledge of enterprise systems. His pattern language for analysis pattern has some impact, to be precise 223 citations are listed in the ACM digital library[4]. Fowler's analysis patterns refer to the analysis phase of software engineering and support in particular the structured re-use of elicited domain knowledge. His work has the closest similarity to our work concerning the re-use of elicited domain knowledge using context-pattern. To the best of our knowledge no work about patterns for re-using elicited domain knowledge for software engineering with more citations exists. This is why we consider Fowler's work. In addition, we decided to consider patterns for the design phase of software engineering with significant impact. The work on design patterns of Gamma et al. [Gamma et al. 1994] has a citation count of 4524 in the ACM digital library[5] and we are not aware of a work regarding patterns in software design with a higher citation count. Similar works, for example, the work of Buschmann et al. [Buschmann et al. 1996] regarding pattern-oriented software architecture, have a lower citation count. Buschmann et al.'s work has 837 citations[6]. We selected the work of Schumacher et al. [Schumacher et al. 2006] as a representative pattern-based work concerning a specific knowledge area in software design, in this case security. In the future, we are planning to include further work regarding pattern languages, e.g., the previously mentioned work of Buschmann et al. in an extended comparison of pattern languages using the structure following parts of a pattern language.

Patterns (Vocabulary). Fowler agrees with Alexander that a pattern language requires to have a common way to describe patterns [Fowler 1996; 2002]. His common way for describing analysis pattern contains a unique name, structural and graphical description, and a textual description of behavior and relations to other patterns. A very similar understanding of how to describe a pattern can be found in [Hafiz et al. 2012] and [Fernandez and Pan 2001].

Even though Fowler does not follow it strictly in his analysis patterns, he identified a meta structure of design patterns: "It is commonly said that a pattern, however it is written, has four essential parts: a statement of the context where the pattern is useful, the problem that the pattern addresses, the forces that play in forming a solution, and the solution that resolves those forces." [Fowler 1996, p. 6].

[Gamma et al. 1994] also state that patterns need a consistent format in agreement with Alexander. The format of the author's design patterns is defined by a template, which is structured into different sections. For example, every pattern has among others a section for its name, intent, motivation, solution, forces, consequences, and known uses.

[4]ACM citation count of Analysis patterns: reusable objects models source: `http://dl.acm.org/citation.cfm?id=265172`

[5]ACM citation count of Design patterns: elements of reusable object-oriented software source: `http://dl.acm.org/citation.cfm?id=186897`

[6]ACM citation count of Pattern-oriented software architecture: a system of patterns source: `http://dl.acm.org/citation.cfm?id=249013`

[Schumacher et al. 2006] use a similar template as [Gamma et al. 1994] for their security design pattern. It is interesting to note that the patterns of the authors have no security specific sections in their template such as security goals, e.g., confidentiality. This allows the assumption that the template could also be used in a more general sense for non security related design patterns.

To sum up, [Gamma et al. 1994] and [Schumacher et al. 2006] follow a well defined set of sections in their template that describes each pattern. In contrast, [Fowler 1996; 2002] defines the structure of his analysis patterns more abstract. His structure just requires a name and some form of structural and behavioural description.

[Fowler 1996] refrains from restricting his *analysis pattern* to a fixed form of a single problem-solution relationship in contrast to design patterns. “A fixed form carries its own disadvantages, however. In this book, for instance, I do not find that a problem-solution pair always makes a good unit for a pattern. Several patterns in this book show how a single problem can be solved in more than one way, depending on various trade-offs. Although this could always be expressed as separate patterns for each solution, the notion of discussing several solutions together strikes me as no less elegant than pattern practice. Of course, the contents of the pattern forms make a lot of sense-any technical writing usually includes context, problem, forces, and solution. Whether this makes every piece of technical writing a pattern is another matter for discussion.” [Fowler 1996, p. 6-7].

However, Fowler states in his meta structure for *design patterns* of other authors (introduced above in the syntax part) that: “This form appears with and without specific headings but underlies many published patterns. It is an important form because it supports the definition of a pattern as ’a solution to a problem in context’, a definition that fixes the bounds of the pattern to a single problem-solution pair.” [Fowler 1996, p. 6].

The pattern template of [Gamma et al. 1994] describes the problem in several separate sections. The section *Intent* describes the general design problem. The section *Motivation* states a scenario in which the design problem occurs and the section *Applicability* refers to specific situations in which the design pattern is useful. The solutions are described in several sections, as well. The section *Structure* describes the graphical representation of the pattern. *Participants* illustrates the elements in the graphical representation. *Collaborations* states the elements’ collaborations and responsibilities. *Implementations* and *Sample Code* illustrate how to represent the pattern in source code.

[Schumacher et al. 2006] consider the sections *Problem* and *Solution* explicitly in their template. The sections are also paired in the sense that the Solution section follows the Problem sections without any section in between. However, several sections refine the solution such as descriptions of *Structure*, *Dynamics*, and *Implementation*.

Overall, design patterns such as the ones from [Gamma et al. 1994] and [Schumacher et al. 2006] seem to follow the guideline of describing one problem and one solution in a pattern. Nevertheless, [Fowler 1996] still embeds a problem solution relationship in his patterns, but not as strict. In some cases he refers to multiple solutions or problems.

Connections between Patterns (Syntax). [Fowler 1996; 2002] states that the relations between patterns are important. Note that Alexander uses the term connection instead of relation. We argue that connection is a synonym for relationship[7] and use the term relationship for the remainder of this chapter. The reason is that software engineering works such as [Fowler 1996; 2002; Schumacher et al. 2006] use the term relationship, as well.

Moreover, according to [Fowler 1996; 2002] a pattern language is indeed about the relations between patterns. Fowler dedicates an entire part of his book [Fowler 1996] to *Support Patterns*, which define the relationships between organizational patterns for, e.g., accounting to software architecture patterns such as the *Layered Architecture* pattern.

[7]Connection and Relationship are synonyms according to dictonary.com: `http://dictionary.reference.com/browse/relationship`

In addition, [Hafiz et al. 2012] agree and elaborate that an enumeration of patterns without defined relations among them is just a pattern catalog. Both, Hafiz et al. and Fowler, basically adopt the view of [Alexander 1977] towards connections between patterns being an essential part of a pattern language.
[Gamma et al. 1994] state also that the relationship between patterns is important. They even create a figure to illustrating the relations between their patterns. Moreover, a section in their template defines the relations to other patterns. The relations between design patterns have many different names such as *single instance*, *adding operations*, or *defining algorithm's steps*.
[Schumacher et al. 2006] dedicate two sections in their template to documenting the relations between patterns. The *Variants* section contains descriptions of variants and specialisations of a pattern. In addition, the *See Also* section in the template references patterns that solve similar problems and patterns that refine the pattern.
In summary, [Fowler 1996], [Gamma et al. 1994], and [Schumacher et al. 2006] agree that relations between patterns have to be defined. However, each pattern language uses different kinds of relations and different ways to document these relations.

Pattern Sequences (Grammar). Fowler uses two different types of patterns: analysis patterns that refer to a particular business domain, and supporting patterns that describe how to apply the analysis patterns. The pattern sequences in Fowler's work can relate different analysis patterns or analysis patterns and supporting patterns. However, Fowler's books do not contain a diagram that shows all pattern sequences, instead the sequences are written in texts of the individual patterns [Fowler 1996; 2002].
Gamma et al. [Gamma et al. 1994] show diagrams in their books, which contain all possible sequences of their patterns. In contrast, Schumacher et al. [Schumacher et al. 2006] use a taxonomy for security and map their patterns to the respective parts of the taxonomy.

3.4 A Pattern Language for Context-Patterns

We describe in the following how our work done so far on context-patterns fits into the required elements of a pattern language and discuss how close we are to having a pattern language for context-patterns.

Patterns (Vocabulary). We analyze in [Beckers et al. 2013] which elements and concepts we used in the context-patterns presented in different works of ours [Beckers et al. 2012; Beckers et al. 2012; Beckers and Faßbender 2012; Beckers et al. 2012]. Further, we described the relations between the identified elements and concepts in a meta-model in [Beckers et al. 2013] (A short summary is given in Sect. 4) and we showed *how* to describe a context-pattern using the meta-model. We claim our meta-model and the pattern catalog contain the vocabulary for our pattern language and published the claim in [Beckers et al. 2013].
Our context-patterns each address a particular problem and have a method that states how to solve this problem. The pattern form which reflects this information is introduced in [Beckers et al. 2014]. We explicitly state the problem and forces for the problem. The grammar is contained in the method description which is part of each of our patterns. Describing the solution in a method provides engineers with descriptions of how to apply the solutions.

Connections between Patterns (Syntax). We analyzed the relations between our context-patterns and present the results in Sect. 5. Context-pattern *can refine* each other. Moreover, the domain knowledge elicited and stored in one context-pattern can be used by another pattern as *input*. This requires that both patterns are combined in a new method and that their elements have to be mapped. We showed how this can work in [Beckers et al. 2012] and combined the cloud system analysis pattern with the law pattern adding a new method. How to derive relations between context-pattern and the relations found for the existing context-pattern are the contribution of this work.

Pattern Sequences (Grammar). We are showing all possible sequences between our context-patterns in Fig. 5, similar to the related works [Eloranta et al. 2014; Buschmann et al. 1996; Gamma et al. 1994; Schumacher et al. 2006; Schümmer and Lukosch 2007]. The figure shows sequences of context-patterns of the types technical, organisational & technical, and organisational. The sequences contain context-patterns that are used jointly, refine or are input for other context-patterns.

As a result, we claim to have defined the vocabulary of a pattern-language via our meta-model and pattern catalog in Sect. 4 and [Beckers et al. 2013], and the syntax via our defined relations between the context-patterns in Sect. 5. We rely on the methods in our context-patterns as grammar for the pattern language [Beckers et al. 2014]. This is quite different from the design patterns by [Gamma et al. 1994] and [Schumacher et al. 2006], which use more explicit sections of their templates to describe problems and solutions. However, the analysis patterns by [Fowler 1996] also contain problems and solutions, but in a less strict format. Our context-patterns also focus on the analysis phase of software engineering and we claim to be in alignment with Fowler. Nevertheless, our view of integrating a method as solution is novel and we have to discuss further with the pattern community if this satisfies as a grammar for our pattern language.

In contrast, not all works follow Alexander's definition of a pattern language. Jackson's work on problem frames [Jackson 2001] provides a different view. He considers his problem frames as a *kind of pattern* and also presents a pattern language. This language for expressing problem frames contains domain-types and interfaces between problem frames. However, Jackson avoids the term *pattern language* in his definition and it is an open debate if his language qualifies as a pattern language, because it differs from [Fowler 1996], [Gamma et al. 1994], and [Schumacher et al. 2006]. This example shows that not all patterns require a pattern language in Alexander's sense. However, we will focus on defining an accepted pattern language for context-patterns via publishing papers in the community and discussing it with the experts of the pattern community.

4. A META-MODEL FOR CONTEXT-PATTERN

In this section, which is a summary of a previous work [Beckers et al. 2013], we present a meta-model for building context-patterns that consider domain knowledge during the analysis phase of software engineering. We consider different kinds of domain knowledge, e.g., technical domain knowledge. Therefore, we used a bottom-up approach, starting with a set of previously and independently developed context-patterns.

We identified the common concepts in our existing context-patterns [Beckers et al. 2012; Beckers et al. 2012; Beckers and Faßbender 2012; Beckers et al. 2012], and aggregate this knowledge into a meta-model of elements one has to talk and think about when describing a new context-pattern [Beckers et al. 2013].

This is quite similar to what Jackson [Jackson 2001] proposed for requirements. He defined a meta-model of reoccurring domains, like causal, biddable and lexical domains. These domains are used to define basic requirements patterns, so-called *Problem Frames* [Jackson 2001].

In this section, we show a similar meta-model for context-patterns. We show how we derived it from already existing context-patterns and how it can be used to describe the structural part of a new context-pattern. This section summarizes the results from one previous works of ours [Beckers et al. 2013].

This meta-model has several benefits. First, it forms a uniform basis for our context-patterns, making them comparable. Second, findings and results for one pattern can be transferred to the other patterns via a generalization. Third, the meta-model contains the important conceptual elements for context-patterns. Fourth, it enables us to form a pattern language for context-patterns. However, in this work we focus on the aspects of the meta model which create the basis of a pattern language for context-patterns.

Using this meta-model we empower requirements and software engineers to describe their own context-patterns, which capture the most important parts for understanding the context of a system-to-be. The meta-model was derived in a bottom-up way from the different patterns we described independently for different domains. For the process of deriving the general elements, which then form the meta-model, we started to analyze each context-pattern in isolation. For each element in a context-pattern we discussed what the general concept behind this element is or if it is a general concept in itself. In a next phase we harmonized the conceptual elements by comparing the found elements, merging them if needed and setting up their relations. This way we got a coherent

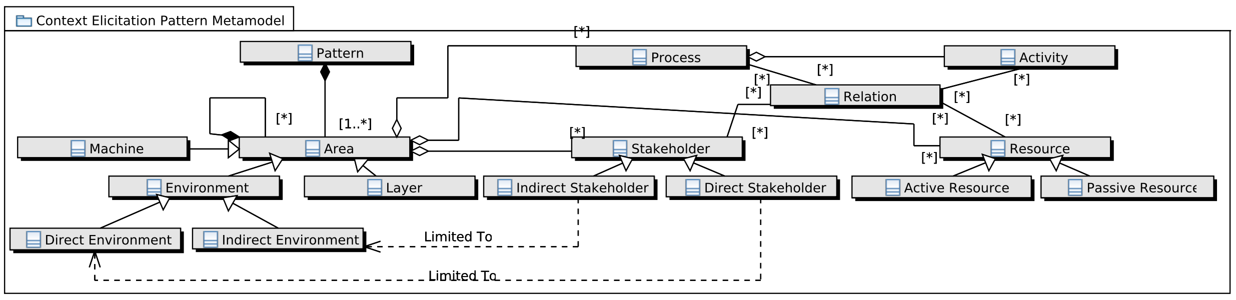

Fig. 4: Context-Pattern Meta-model

set of conceptual elements over all patterns. In the last phase we had to choose which conceptual elements should be part of the meta-model. Finally, we formed the meta-model as depicted in Fig. 4 out of the selected conceptual elements. The meta-model was modeled using the UML notation.

The root element is the `Pattern` itself. Each pattern contains at least one `Area`. In general, an area contains elements of either a technical or organizational view. An area can contain other areas, which do not need to be the same view. An area can concern either a `Machine`, i.e. the thing to be developed, or an `Environment`, which in turn contains elements that have some kind of relation to the machine, or a `Layer`, which encapsulates a set of elements.

The environment can be further refined. There are elements which directly interact with the machine, captured in the `Direct Environment`. And there are elements which have an influence on the system via elements of the direct environment, captured by the `Indirect Environment`.

An element which is part of an `Area` can be a `Process`, a `Stakeholder`, or a `Resource`. A process describes some kind of workflow or sequence of activities. Therefore, it can contain `Activities`. A stakeholder describes a person, a group of persons, or organizational units, which have some kind of influence on the machine. A stakeholder can be refined to a `Direct Stakeholder` who interacts directly with the machine, and an `Indirect Stakeholder` who only interacts with direct stakeholders but has some interest in or influence on the machine. A `Resource` describes some physical or non physical (e.g., information) element which is needed to run the machine or which is processed by the machine and which is not a stakeholder. A resource can be an `Active Resource` with some behavior or a `Passive Resource` without any behavior.

This meta-model has several benefits. First, it forms a uniform basis for our context-patterns, making them comparable. If a method already makes use of one of the patterns, it is now easy to generalize the usage to the elements of the meta-model. This enables one to replace a given used pattern by another one easily. Second, findings and results for one pattern can be transferred to the other pattern via a generalization to the meta-model elements. Third, the meta-model contains the important conceptual elements for context-patterns. Thus, it is helpful to know these elements and search for them in a specific domain when setting up a new context-pattern for a domain. Fourth, it enables to form a pattern language for the context-pattern. The common meta-model eases relating the patterns to each other.

5. RELATING CONTEXT-PATTERNS

For forming a context-pattern language, we investigated the commonalities and differences between the patterns as enumerated in Sec. 2. We also identified how one context-pattern can be used in combination with another context-pattern.

5.1 Relation Types

Based on these insight, we defined relations between our patterns and identified the following kinds of relations:

can refine. The *can refine* relation describes that one context-pattern refines another pattern or parts thereof. For example, one pattern can refer to services, while another describes how these services are composed. Moreover, the semantics or abstraction level of context-patterns differs. Hence, we call the relation *can refine* in order to make it obvious that this may not hold for all possible instantiations of a context-pattern. This problem is caused by the very different detail degree of information available in the early stages of software development.

input. The information contained in the instantiation of one pattern can be the input for another pattern. For example, our law pattern can use the information of other patterns to identify relevant laws. We show how the information in the cloud pattern can be used as scenario description for the law pattern in [Beckers et al. 2012].

Note that we distinguish only these two types. We did not find any additional type while relating our context pattern as described in Sec. 5.3. We explicitly tried to find other relations as described in other pattern languages, for example [Gamma et al. 1994; Schumacher et al. 2006]. While some relation types can be mapped such as complements, which is a kind of can refine relation, other kinds could neither be mapped nor found for our context pattern.

Van Welie and Van der Veer [Welie and Veer 2003] distinguish three fundamental relation types, which can have different variations and flavors:

Aggregation. One pattern is part of another pattern and completes it.

Specialization. One pattern can be derived from (parts) of another pattern and adds more detail or parts of the solution.

Association. Patterns with the same context or the same problem to be solved.

According to these enumeration, our relation types for context patterns only cover the specialization relation type explicitly. But both, the can refine and the input relation, are quite special regarding their specialization properties.

Exploiting a can refine relation always starts at one element of the source pattern, adding more context to this specific element (see Sec.5.3). Additionally, the refinement relation can be bidirectional which is quite uncommon. This is due to the nature of a context which is always bound to the entity one is exploring. And each context contains entities which add more context if inspected in detail.

An input relation is a specialization of a combination of entities of the source pattern. Hence, it does not add context to one entity but to a group of entities.

The aggregation relation is implicitly given by the joint use of patterns (see Sec.5). But the joint use of patterns is not that strict, as the patterns which are used jointly can be used separated. For the association relation, we have no counterpart. The reason is that a context pattern for a domain should be that general that there is no need for further context pattern for this domain.

Noble classified relations between object-oriented design patterns [Noble 1998] and identified the following main relationships:

Uses. A pattern uses another pattern

Refines. A more focused pattern refines a general pattern

Conflicts. Two patterns address the same problem

Our relations for context-patterns map to Nobel's design pattern relations as follows. The *used jointly* and *input for* relations both map to the *uses* relations. For context-patterns it is important to distinguish if two patterns are used together or if they are used in sequence. In a sequence one is used first and the output of the pattern instantiation serve as the input of the other pattern, which is used afterwards. We also have identified the refines relations with the same meaning as Noble's relation. In contrast to Noble's work we do not have a *conflicts* relation,

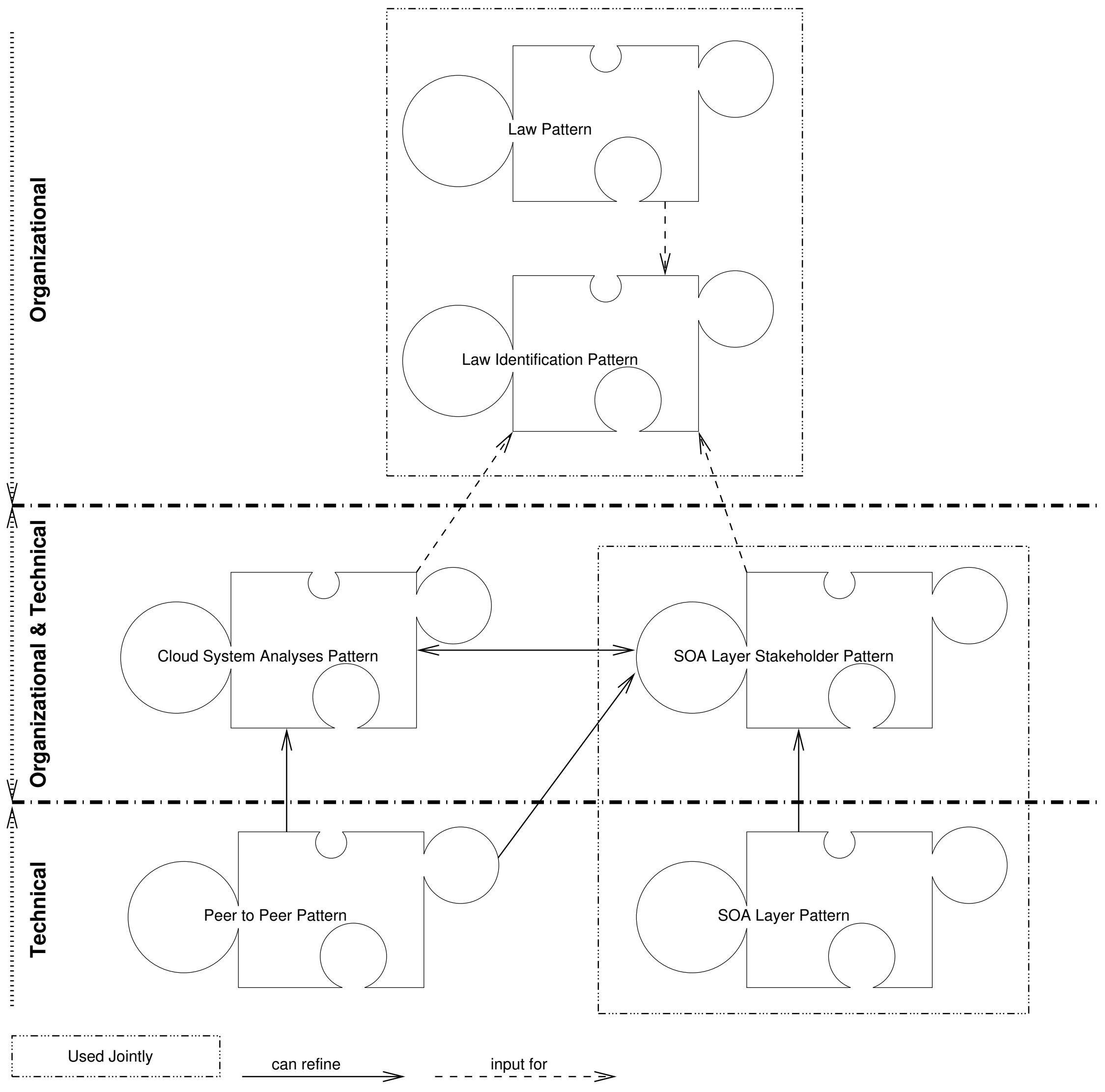
Organizational
Organizational & Technical
Technical
Law Pattern
Law Identification Pattern
Cloud System Analyses Pattern
SOA Layer Stakeholder Pattern
Peer to Peer Pattern
SOA Layer Pattern
Used Jointly
can refine
input for

Fig. 5: Relations between Context-Patterns

because each context requires a separate context-patterns. For instance, a cloud computing scenario requires the cloud system analysis pattern and no other context-pattern such as the SOA Layer pattern can be applied to this particular context.

5.2 Found Relations between Context Pattern

The resulting relations between context-patterns are shown in Fig. 5. In general, we have three groups of context-patterns. Context-patterns which only focus on the technical context, context-patterns that only focus on organisational aspects, and context-patterns which combine those two views. The groups and the information to

	Only in Context-Pattern One (CP1)	same	Mapping		Only in Context-Pattern Two (CP2)
			Element in CP1	Element in CP2	
Area					
Machine			CP1::Element1	**CP2::Element2**	
Environment					
Direct Environment					
Indirect Environment					
Layer					
Process					
Activity					
Relation					
Stakeholder			**CP1::Element4**	**CP2::Element3**	
Indirect Stakeholder					
Direct Stakeholder					
Resource			**CP1::Element8**	CP2::Element11	
				CP2::Element13	
Active Resource			**CP1::Element7**	*CP2::Element2*	
Passive Resource					

Table I. : Pattern Relation Investigation Table Template

which group a pattern belongs are one result of our previous work [Beckers et al. 2013]. In this work we observed that some patterns only describe the indirect environment of the system-to-be (organisational), some only describe the direct environment and the system-to-be itself (technical), and some mix both views. The three groups are indicated in Fig. 5 as layers separated by dot-slashed lines.

The relations are shown using directed arrows. A solid arrow indicates a *can refine* relation, while a dashed arrow indicates an *input* relation. Some of the patterns are used jointly which means that those patterns are usually used together and closely related. For example, the SOA layer stakeholder pattern contains all layers and elements of the SOA layer pattern. The SOA layer pattern only adds the technical relations between the elements, while the SOA stakeholder pattern adds the environments and stakeholders. The can refine relation has a particular effect when it occurs on a pair of joined context-patterns. Namely, the relation always occurs for both joined patterns, but this is not explicitly shown. For example, the SOA stakeholder layer pattern can refine the cloud system analysis pattern, so can the SOA layer pattern. But in a normal case, both SOA patterns will be used jointly to refine a cloud system analysis pattern. Figure 5 shows that the SOA stakeholder layer pattern and SOA layer pattern, as well as the law pattern and law identification pattern, are used jointly. The law pattern is input for the law identification pattern. The SOA layer pattern can refine the SOA layer stakeholder pattern. The cloud system analysis pattern and the SOA layer stakeholder pattern can refine each other. The peer to peer pattern can refine the cloud system analysis pattern and the SOA layer stakeholder pattern. All patterns of the organisational & technical layer can serve as input for the law identification pattern. Figure 5 gives a quick and compact overview of the relations hiding some details we will elaborate in the following.

5.3 Tables for Finding Relations between Context-Pattern

For investigating the relation between two context-patterns we used so-called *pattern relation investigation table*s. The template for such a table is shown in Fig. I. A pattern relation investigation table compares two given context-pattern. In the *first column* of such a table the general element types are stated (like *Area* in the second row) as defined by the meta-model discussed in Sect. 4. Therefore, a row of a pattern relation investigation table contains the refined elements (like *CP1::Element1* in the third row) which correspond to the given general element type. The columns contain the corresponding elements of the two patterns under investigation. This way, a relation investigation table shows the commonalities and difference of two patterns and allows to derive the type of relation between two patterns. Table V illustrates an example.

Some of the refined elements only occur in one of the patterns at hand. Hence, they are added to the columns *Only Context-Pattern One (CP1)* or *Only Context-Pattern One (CP2)* (see Table Fig. I). These elements are not of relevance when two context-patterns are used jointly. For example, for the meta-model type *Relations*, the cloud system analysis pattern and the SOA layer stakeholder pattern are completely disjointed (see Tab. V row *Relation* and columns *Only. . .*).

Some of the refined elements have the same semantic in both of the patterns. They are added to the column *same*. Later, when one has the instances of the two patterns at hand, all refined elements of the same column should be synchronized. This means, for example, that adding one instance of such a refined element to one context-pattern instance results in adding the instance of the refined element to the other pattern. For example, the indirect stakeholders domain and legislators are the same for the cloud system analysis pattern and the SOA layer stakeholder pattern. Hence, when adding an instance of a legislator to the cloud system analysis pattern instance the same instance must also be added as a legislator of relevance to the SOA layer stakeholder pattern instance (see Tab. V row *Indirect Stakeholders* and column *same*).

Other elements do not have exactly the same semantic in each of the patterns but can be mapped to elements of the other pattern. These mappings are not deterministic in all case. Some elements can be mapped to more than one element of the other pattern (see Table I fifteenth row). In this case, it depends on the actual scenario which element is used for the actual refinement. For example, the direct stakeholder cloud provider can be mapped to an infrastructure service provider in case he plays no central role in the scenario under investigation. If the cloud provider has a more central role, he/she might be one of the organizations or process actors involved in the business processes of the scenario (see Tab. V row *Direct Stakeholders* and column *Mapping*). If an element is written in italics, it means that this element is of another type as indicated by the row (see Table I sixteenth row). For example, the machine of the SOA layer stakeholder pattern can be mapped to the SaaS or software product elements of the cloud system analysis pattern. But SaaS and software product are of the type active resource in the cloud system analysis pattern as it has another point of view on the system-to-be. A bold written element indicates that this element might be the reason for a refinement (see Table I fourth row). Such a mapping can be bidirectional (see Table I twelfth row). For example, one might want to develop a SOA application in the first place. But later he/she is also interested to broaden the context to the cloud domain as the application is deployed in the cloud (see Tab. V row *Machine* and column *Mapping*). A bidirectional relation exists between, for example, cloud customer and process actor (see Tab. V row *Direct Stakeholder* and column *Mapping*). On the one hand, the refinement can start in a SOA context in which one of the process actors uses a cloud solution for (parts) of its IT infrastructure. Thus, when the cloud context is also important for our problem at hand, we use the cloud context-pattern jointly. On the other hand, the refinement can also start in the cloud context and it turns out to be important that the cloud customer is part of a SOA context.

5.4 Steps and Sources for Finding Relations between Context Pattern

The information for filling a pattern relation investigation table stems from three different sources:

Own Experience. The first source are our experiences from using the patterns. As we have used such patterns extensively, we already came across situations where we combined patterns (like in [Beckers et al. 2012]). Such known uses can be refined to the relations captured in the pattern relation investigation table.

Domain Experts. The second source are experts of the different domains. They are often experts for one specific domain, but, from our experience, it is not unlikely that they were involved in cases where two or more domains were important.

Available Documentation. The third source are reports or papers which document a certain combination of domains. Such documentation plays the role of known uses for a combination in our case. A documentation can be a more general description given in a white-paper, a product description, a project documentation, and so forth.

The process of filling a pattern relation investigation table starts with documenting the relations experienced by ourselves. Such relations are mostly supported by known uses of us. Hence, they are marked as reliable. Then, we conduct a mind experiment to find possible relations. A possible relation is mostly supported by a hand made example. Such a relation is marked as unreliable. The reason that we do this mind experiment is that in the next step we talk to domain experts. It speeds up the process and raises the willingness of the experts when the table is prefilled. The domain experts then give feedback about missing relations and relations they can support. For both cases they should give an example. This way, some of the relations marked as unreliable can be turned into reliable relations, or an already reliable relation is strengthened. Finally, we search for available documentation on combining the two domains at hand in general and for the unreliable relations in specific. For each documented relation found this way, we add a reliable relation, strengthen a reliable relation, or promote an unreliable relation. After conducting this step, we remove all remaining unreliable relations.

5.5 Results of the Relation Mining

We describe each aggregated relation between two context-pattern using a template (see Tab. II for an example) that states first the *Direction* of the relation, second the *Relation Type*, third the *Reasoning* why the relationship holds, and forth a reference to the *details* of the relation. Relations between context-patterns can consist of several (sub-)relations. The relation type and direction is derived from relation investigation tables. If an element of a context-pattern of the two patterns in the corresponding investigation table is marked bold, this context-pattern can be refined by the other context-pattern. If there is no element marked bold, then we assume that there is an input relation. The template instance only contains a compact reasoning and explanation of the relation.

SOA Layer Stakeholder Pattern $\leftrightarrow$ Cloud System Analysis Pattern - The SOA pattern and the cloud pattern can refine each other (see Tab. II).

Table II. : Pattern Relation SOA Layer Stakeholder Pattern to Cloud System Analysis Pattern

Direction	SOA to Cloud, Cloud to SOA
Relation Type	can refine
Reasoning	The services deployed in a cloud can be created or composed in a SOA. Hence, the information in the SOA layer stakeholder pattern can be seen as a refinement of the services in the cloud system analysis pattern. In addition, the stakeholders involved in the creation and maintenance of the service can be cloud stakeholders. But it is also possible that the cloud system analysis pattern is used to elicit more information about stakeholders of a SOA layer stakeholder pattern or the deployment of the whole system-to-be.
Details	Tab. V

Cloud System Analysis Pattern $\leftrightarrow$ Peer to Peer Pattern - The P2P pattern can refine the cloud pattern (see Tab. III).

Table III. : Pattern Relation Cloud System Analysis Patter to Peer To Peer Pattern

Direction	P2P to Cloud
Relation Type	can refine
Reasoning	Specific technologies that form the core of the cloud, e.g., the cloud database or the hypervisor, are likely based on P2P-architectures. In addition, the services deployed in the cloud can also be based on P2P-architectures. In both cases the P2P pattern can refine the description of these services or cloud components.
Details	Tab. VI

SOA Layer Stakeholder Pattern $\leftrightarrow$ Peer to Peer Pattern - The SOA patterns can be refined by a P2P Pattern (see Tab. IV).

Table IV. : Pattern Relation SOA Layer Stakeholder Pattern to Peer to Peer Pattern

Direction	P2P Pattern to SOA Pattern
Relation Type	can refine
Reasoning	The services described in the SOA pattern can rely on P2P-architectures. The P2P pattern can be used to create a refined description of these services and also reason if these services can fulfill certain quality requirements, e.g., security. The isolated analysis of services in a SOA can be helpful when services shall be evaluated for the question if they can fulfill certain requirements at all. Hence, the P2P pattern can help excluding certain services from the SOA patterns.
Details	Tab. VII

	Only in Cloud System Analysis Pattern(CSAP)	same	Mapping		Only SOA Layer Stakeholder Pattern (SLSP)
			Element in CSAP	Element in SLSP	
Area					Inner System, Outer System
Machine	Cloud		*SaaS*	**Machine**	
			Software Product		
Direct Environment		Direct Environment			
Indirect Environment		Indirect Environment			
Layer					Business Domain, Business Processes, Business Services, Infrastructure Services, Component-based Service Realization, Operational System
Process					Process
Relation	has, monitors, builtAndCustomizedBy, buildBy, work for, owns, provides, isComplementedb, isBasedOn, partOf, isA				influences, part of, participates in, provides
Indirect Stakeholder		Domain, Legislator	*Cloud Provider*	**Organization**	Shareholder, Asset Provider
			Cloud Customer		
			End Customer		
Direct Stakeholder	Cloud Developer		Cloud Provider	**Infrastructure Service Provider**	Component Provider, Operational Systems Provider
				Organization	
				Process Actor	
			Cloud Customer	**Business Service Provider**	
				Organization	
				Process Actor	
			End Customer	Process Actor	
				Organization	
Resource	Resource, Pool				
Active Resource	Hardware, Software, Service		IaaS	Infrastructure Service	Component
			Cloud Software Stack	Infrastructure Service	
			PaaS	Infrastructure Service	
			Software Product	Business Service	
				Infrastructure Service	
				CRM	
				ERP	
				Database	
				Packaged Applications	
				Legacy Applications	
				Machine	
			SaaS	Business Service	
				Machine	
Passive Resource	Data, Location				

Table V. : Cloud System Analyses Pattern to SOA Layer Stakeholder Pattern Relation Investigation Table

	Only in Cloud System Analysis Pattern(CSAP)	same	Mapping		Only Peer to Peer Pattern (PPP)
			Element in CSAP	Element in PPP	
Area					Services, Peer to Peer Protocol
Machine	Cloud				
Direct Environment	Direct Environment				
Indirect Environment	Indirect Environment				
Layer					Application Layer, Service Layer, Feature Management Layer, Overlay Management Layer, Network Layer
Relation	has, monitors, builtAndCustomizedBy, buildBy, work for, owns, provides, isComplementedb, isBasedOn, partOf, isA				uses, constrains
Indirect Stakeholder	Domain, Legislator				
Direct Stakeholder	Cloud Developer, Cloud Provider, Cloud Customer, End Customer				
Resource	Resource, Pool				
Active Resource	Hardware, Software, Service		**IaaS**	Application	Service Management, Service Messaging, Security Management, Reliability and Fault Resilience, Performance and Resource Management, Resource Discovery, Location Lookup, Routing, Network
			Cloud Software Stack	Application	
			PaaS	Application	
			Software Product	Application	
			SaaS	Application	
Passive Resource	Data, Location				Meta Data

Table VI. : Cloud System Analyses Pattern to Peer to Peer Pattern Relation Investigation Table

	Only in Peer to Peer Pattern (PPP)	same	Mapping		Only SOA Layer Stakeholder Pattern (SLSP)
			Element in PPP	Element in SLSP	
Area	Services, Peer to Peer Protocol				Inner System, Outer System
Machine			*Application*	**Machine**	
Direct Environment					Direct Environment
Indirect Environment					Indirect Environment
Layer	Application Layer, Service Layer, Feature Management Layer, Overlay Management Layer, Network Layer				Business Domain, Business Processes, Business Services, Infrastructure Services, Component-based Service Realization, Operational System
Process					Process
Relation	uses, constrains				influences, part of, participates in, provides
Indirect Stakeholder					Domain, Legislator, Shareholder, Asset Provider, Organization
Direct Stakeholder					Component Provider, Operational Systems Provider, Infrastructure Service Provider, Business Service Provider, Process Actor
Active Resource	Service Management, Service Messaging, Security Management, Reliability and Fault Resilience, Performance and Resource Management, Resource Discovery, Location Lookup, Routing, Network		Application	**Infrastructure Service**	Business Service, CRM, ERP, Database, Packaged Applications, Legacy Applications, Component
				Machine	
Passive Resource	Meta Data				

Table VII. : Peer to Peer Pattern to SOA Layer Stakeholder Pattern Relation Investigation Table

	Only in SOA Layer Pattern (SLP)	same	Mapping		Only SOA Layer Stakeholder Pattern (SLSP)
			SLP	Element in SLSP	
Area					Inner System, Outer System
Machine			*Whole Pattern*	**Machine**	
Direct Environment					Direct Environment
Indirect Environment					Indirect Environment
Layer		Business Domain, Business Processes, Business Services, Infrastructure Services, Component-based Service Realization, Operational System			
Process		Process			
Relation	performed by, relies on, exposes, business relation	participates in			influences, part of, provides
Indirect Stakeholder		Organization			Domain, Legislator,Shareholder, Asset Provider
Direct Stakeholder					Component Provider, Operational Systems Provider, Infrastructure Service Provider, Process Actor, Business Service Provider
Active Resource		Component, Business Service, CRM, ERP, Database, Packaged Applications, Legacy Applications, Infrastructure Service			

Table VIII. : SOA Layer Stakeholder Pattern To SOA Layer Pattern Relation Investigation Table

	Only in Law Pattern (LP)	same	Mapping		Only Law Identification Pattern (LIP)
			Element in LP	Element in LIP	
Area		Classification			
Direct Environment			Law Structure	Core Structure	
Indirect Environment			Context	Context	
Process					Related Process(es)
Activities		Activities, Activity Classifier			
Relation	isA	Avoid, Accomplish, Influence, Entitled To			Classified As
Stakeholder		Person Classifier			
Indirect Stakeholder		Domain, Legislator			
Direct Stakeholder			Addressee	Active Stakeholder	
			Target Person	Passive Stakeholder	
Resource		Subject Classifier	Target Subject	Asset	
Passive Resource	Regulations				

Table IX. : Law Pattern to Law Identification Pattern Relation Investigation Table

	Only SOA Layer Stakeholder Pattern (SLSP)	same	Mapping		Only Law Identification Pattern (LIP)
			Element in SLSP	Element in LIP	
Area	Inner System, Outer System				Classification
Machine			Machine	*Asset*	
Direct Environment			Direct Environment	Core Structure	
Indirect Environment			Indirect Environment	Context	
Layer		Business Domain, Business Processes, Business Services, Infrastructure Services, Component-based Service Realization, Operational System			
Process			Process	Related Process(es)	
Activities					Activities, Activity Classifier
Relation	performed by, relies on, exposes, business relation, participates in				Avoid, Accomplish, Influence, Entitled To, Classified As
Stakeholder					Person Classifier
Indirect Stakeholder		Domain, Legislator	Asset Provider	*Passive Stakeholder*	
			Shareholder	*Passive Stakeholder*	
			Organization	*Passive Stakeholder*	
				Active Stakeholder	
Direct Stakeholder			Component Provider	Active Stakeholder	
			Operational Systems Provider		
			Infrastructure Service Provider		
			Process Actor		
			Business Service Provider		
			Organization		
			Component Provider	Passive Stakeholder	
			Operational Systems Provider		
			Infrastructure Service Provider		
			Process Actor		
			Business Service Provider		
			Asset Provider		
			Shareholder		
			Organization		
Resource					Subject Classifier, Asset
Active Resource			Component	*Asset*	
			Business Service		
			CRM		
			ERP		
			Database		
			Packaged Applications		
			Legacy Applications		
			Infrastructure Service		

Table X. : SOA Layer Stakeholder Pattern To Law Identification Pattern Relation Investigation Table

	Only in Cloud System Analysis Pattern(CSAP)	same	Mapping		Only Law Identification Pattern (LIP)
			Element in CSAP	Element in LIP	
Area					Classification
Machine			Cloud	*Asset*	
Direct Environment			Direct Environment	Core Structure	
Indirect Environment			Indirect Environment	Context	
Process					Related Process(es)
Activities					Activities, Activity Classifier
Relation	has, monitors, builtAndCustomizedBy, buildBy, work for, owns, provides, isComplementedb, isBasedOn, partOf, isA				Avoid, Accomplish, Influence, Entitled To, Classified As
Stakeholder					Person Classifier
Indirect Stakeholder		Domain, Legislator			
Direct Stakeholder			Cloud Provider	Active Stakeholder	
			Cloud Customer		
			End Customer		
			Cloud Developer		
			Cloud Provider	Passive Stakeholder	
			Cloud Customer		
			End Customer		
			Cloud Developer		
Resource			Resource	Asset	Subject Classifier
			Pool		
Active Resource			IaaS	*Asset*	
			Cloud Software Stack		
			PaaS		
			Software Product		
			SaaS		
			Hardware		
			Software		
			Service		
Passive Resource	Location		Data	*Asset*	

Table XI. : Cloud System Analyses Pattern to Law Identification Pattern Relation Investigation Table

SOA Layer Stakeholder Pattern $\leftrightarrow$ Law Identification Pattern - Our SOA pattern can be input for law patterns to identify relevant laws for SOA scenarios.

Table XII. : Pattern Relation SOA Layer Stakeholder Pattern to Law Identification Pattern

Direction	SOA Pattern to Law Pattern
Relation Type	input
Reasoning	The SOA patterns can be used as the input for identifying relevant laws and the creation of legal hierarchies. The hierarchies are essential for the mapping of law patterns to law identification patterns.
Details	Tab. X

Cloud System Analysis Pattern $\leftrightarrow$ Law Identification Pattern - Our cloud pattern can be input for the law pattern. We show an example of how to use this relation in [Beckers et al. 2012].

Table XIII. : Pattern Relation Cloud System Analysis Pattern to Law Identification Pattern

Direction	Cloud to Law Pattern
Relation Type	input
Reasoning	The cloud pattern can be used as the input for identifying relevant laws using the law patterns. Moreover, the creation of legal hierarchies can be based on the cloud pattern. The hierarchies are essential for the mapping of law patterns to law identification patterns.
Details	Tab. XI

6. LESSONS LEARNED

While the process description given in Sect. 5 sounds like a straight forward process, it is not in reality. Some possible relations do not come up during a mind experiment session, but at another point in time. The opportunities to talk to domain experts often drop in by accident. Cross-domain documents are found while searching for other content. This leads to a more or less unstructured, iterative process. But one has to make sure that all three sources of information are considered and reflected in the final pattern relation investigation table thoroughly. The ideal scenario for the process mentioned above would be a sequence of workshops following the process. But the given constraints in time and budget often hinder such an ideal scenario.

Another lesson learned is about the effort for establishing relations. At first, it seems that for adding one additional pattern one has to make (numberOfOldPatterns+1) * (numberOfOldPatterns) / 2 comparisons. This turns out to be the upper bound never reached. The reasons are manifold. First, a new pattern has only to be related to patterns of the same kind and adjacent kinds. We never came across a situation where we were able to relate a technical-only context-pattern with an organizational-only context-pattern. Second, if already existing context-patterns are meant to be used jointly, it is sufficient to relate the new patterns to one of them. The relations are transitive in this case. Third, there are domains which exclude each other at first sight. Nevertheless, the effort for relating context-patterns is noticeable. It is possible to learn from already established relations between existing context-patterns, for example, in means of inspiration for the mind experiments or already found documentation. This way, the effort can be lowered.

The next lesson learned is to wait until a new context-pattern is really stable. Every change of the pattern, when, for example, a new element is added, an element is removed, or the semantic has to be changed, leads inevitably to a rerun of the relation establishing process.

Finally, we acknowledge that we only have one evaluation of our process, so far. In the future, we are planning to conduct further evaluations of the process and share further lessons learned.

7. CONCLUSION

In this work we identified relations between context-patterns using the meta-model described in a previous work [Beckers et al. 2013]. The relations were investigated in detail using relation investigation tables introduced in this work. The result of this work is a pattern language for context-patterns with a detailed description of the relations.

We illustrated our approach by showing context patterns, e.g., patterns that consider specific technologies such as Peer-to-Peer networks, specific types of architectures like cloud computing, and specific domains, e.g., the legal domain.

We can use instantiated patterns as a basis for writing requirements, deriving architectures or structured discussions about a specific domain. In addition, our patterns can be used outside the domain of software engineering, for example for scope descriptions, asset identification, and threat analysis, when building an ISO 27001 [ISO/IEC 2005] compliant Information Security Management System [Beckers et al. 2013].

We contribute the following in this work:

—We defined relation types and analyzed all relations between our existing context-patterns. These relations form the semantics of our pattern language.

—The relations between context-patterns in combinations with the meta-model are our initial basis for a pattern language for context-patterns.

—We compared our patterns and pattern language with different existing definitions for pattern languages

8. ACKNOWLEDGEMENTS

We thank our shepherd Christopher Preschern for fruitful discussions and constructive feedback. This research was partially supported by the EU project Network of Excellence on Engineering Secure Future Internet Software Services and Systems (NESSoS, ICT-2009.1.4 Trustworthy ICT, Grant No. 256980), the German Research Foundation (DFG) under grant number HE3322/4-2, and the Ministry of Innovation, Science, Research and Technology of the German State of North Rhine-Westphalia and EFRE (Grant No. 300266902 and Grant No. 300267002).

REFERENCES

ALEXANDER, C. 1977. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.

BECKERS, K., COTE, I., FASSBENDER, S., HEISEL, M., AND HOFBAUER, S. 2013. A pattern-based method for establishing a cloud-specific information security management system. *Requirements Engineering*, 1–53.

BECKERS, K. AND FASSBENDER, S. 2012. Peer-to-peer driven software engineering considering security, reliability, and performance. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES) - 2nd International Workshop on Resilience and IT-Risk in Social Infrastructures (RISI 2012)*. IEEE Computer Society, 485–494.

BECKERS, K., FASSBENDER, S., AND HEISEL, M. 2013. A meta-model approach to the fundamentals for a pattern language for context elicitation. In *Proceedings of the 18th European Conference on Pattern Languages of Programs (Europlop)*. ACM, –. Accepted for Publication.

BECKERS, K., FASSBENDER, S., AND HEISEL, M. 2014. A meta-pattern and pattern form for context-patterns. In *Proceedings of the 19th European Conference on Pattern Languages of Programs (Europlop)*. ACM, –. Accepted for Sheperding.

BECKERS, K., FASSBENDER, S., HEISEL, M., AND MEIS, R. 2012. Pattern-based context establishment for service-oriented architectures. In *Software Service and Application Engineering*. LNCS 7365. Springer, 81–101.

BECKERS, K., FASSBENDER, S., KÜSTER, J.-C., AND SCHMIDT, H. 2012. A pattern-based method for identifying and analysing laws in the field of cloud computing compliance. In *Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*. LNCS Series, vol. 7195. Springer, 256–262.

BECKERS, K., FASSBENDER, S., AND SCHMIDT, H. 2012. An integrated method for pattern-based elicitation of legal requirements applied to a cloud computing example. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES) - 2nd International Workshop on Resilience and IT-Risk in Social Infrastructures(RISI 2012)*. IEEE Computer Society, 463–472.

BECKERS, K., KÜSTER, J.-C., FASSBENDER, S., AND SCHMIDT, H. 2011. Pattern-based support for context establishment and asset identification of the ISO 27000 in the field of cloud computing. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*. IEEE Computer Society, 327–333.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. Wiley.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley.

ELORANTA, V.-P., KOSKINEN, J., LEPPÄNEN, M., AND REIJONEN, V. 2014. *Designing Distributed Control Systems: A Pattern Language Approach*. Wiley.

FABIAN, B., GÜRSES, S., HEISEL, M., SANTEN, T., AND SCHMIDT, H. 2010. A comparison of security requirements engineering methods. *Requirements Engineering – Special Issue on Security Requirements Engineering 15,* 1, 7–40.

FERNANDEZ, E. B. AND PAN, R. 2001. A Pattern Language for Security Models. In *8th Conference of Pattern Languages of Programs (PloP)*.

FOWLER, M. 1996. *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

FOWLER, M. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

HAFIZ, M., ADAMCZYK, P., AND JOHNSON, R. E. 2012. Growing a pattern language (for security). In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. Onward! '12. ACM, New York, NY, USA, 139–158.

HENNEY, K. 2005. Context encapsulation - three stories, a language, and some sequences. In *EuroPLoP* (2010-03-02), A. Longshaw and U. Zdun, Eds. UVK - Universitaetsverlag Konstanz, 379–414.

ISO/IEC. 2005. Information technology - Security techniques - Information security management systems - Requirements. ISO/IEC 27001, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC).

JACKSON, M. 2001. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley.

LUA, E. K., CROWCROFT, J., PIAS, M., SHARMA, R., AND LIM, S. 2005. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials 7*, 72–93.

NIKNAFS, A. AND BERRY, D. M. 2012. The impact of domain knowledge on the effectiveness of requirements idea generation during requirements elicitation. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*. 181 –190.

NOBLE, J. 1998. Classifying relationships between object-oriented design patterns. In *Proceedings of the Australian Software Engineering Conference*. ASWEC '98. IEEE Computer Society, Washington, DC, USA, 98–107.

OTTO, P. N. AND ANTÓN, A. I. 2007. Addressing legal requirements in requirements engineering. In *RE*.

PAUWELS, S. L., HÄBSCHER, C., BARGAS-AVILA, J. A., AND OPWIS, K. 2010. Building an interaction design pattern language: A case study. *Computers in Human Behavior 26,* 3, 452 – 463.

SCHUMACHER, M., FERNANDEZ-BUGLIONI, E., HYBERTSON, D., BUSCHMANN, F., AND SOMMERLAD, P. 2006. *Security Patterns: Integrating Security and Systems Engineering*. Wiley.

SCHÜMMER, T. AND LUKOSCH, S. 2007. *Patterns for Computer-Mediated Interaction*. Wiley.

WELIE, M. V. AND VEER, G. C. V. D. 2003. Pattern languages in interaction design: Structure and organization. In *Proc. Interact '03, M. Rauterberg, Wesson, Ed(s). IOS*. IOS Press, 527–534.

WINN, T. AND CALDER, P. 2003. A pattern language for pattern language structure. In *Proceedings of the 2002 Conference on Pattern Languages of Programs - Volume 13*. CRPIT '02. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 45–58.

ZDUN, U. 2007. Systematic pattern selection using pattern language grammars and design space analysis. *Softw. Pract. Exper. 37,* 9, 983–1016.