A Meta-Model Approach to the Fundamentals for a Pattern Language for Context Elicitation

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

It is essential for building the right software system to elicit and analyze requirements. Requirements define what right is, without them a checking if the right software was built is impossible. Writing requirements that can achieve this purpose is only possible if the domain knowledge of the system-to-be and its environment is known and considered thoroughly. We consider this as the context problem of software development.
In the past, we tackled this problem by describing common structures and stakeholders for several different domains. The common elements of the context where obtained by from observations about the domain in terms of standards, domain specific publications and implementations. Whenever a system-to-be is within the context of a domain already described by a context elicitation pattern, one can use this pattern to describe the context by instantiation. But the description of the structure of a context elicitation pattern, especially in terms of its static structure, was not aligned. This inhibits relating context elicitation patterns to form a patter language. Also describing newly observed pattern is difficult for inexperienced pattern creators without any guidance.
We present these patterns, show how we used them to construct our meta-model, and give an example how to describe a context elicitation pattern using the meta-model.
We propose a meta model for describing context patterns. The meta model contains elements, which can be used to structure and describe domain knowledge in a generic form. These context patterns can afterwards be instantiated with the domain knowledge required for software engineering. We presented a number of context patterns for different areas of domain knowledge in the past. This work is based on these existing patterns, which we abstracted into a meta-model. We present our context patterns, show how we used them to construct our meta-model, and provide an example of how to describe a context elicitation pattern using our meta-model. We contribute this meta model as a basis for a pattern language for context elicitation.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Evaluation/ methodology*; H.1.2 [**Models and Principles**]: User/Machine Systems—*Human Information Processing*; I.5.1 [**Pattern Recognition**]: Models—*Neural Nets*

General Terms: Human Factors

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

ACM Reference Format:

Beckers K., Faßbender S. 2013.A Meta-Model Approach to the Fundamentals for a Pattern Language of Context Elicitation jn 2, 3, Article 1 (May 2010), 28 pages.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 17th Conference on Pattern Languages of Programs (PLoP). EuroPLoP'18, July 10-14 2013, Kloster Irsee, Bavaria, Germany. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7

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]. Several requirements engineering methods exist, e.g., for security. Fabian et. al [Fabian et al. 2010] concluded in their survey about these 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 Jacksons 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 build and requirements 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.

In this paper, we develop and present a meta-model for building patterns for considering domain knowledge during requirements engineering. We consider different kinds of domain knowledge, e.g., technical domain knowledge. Therefore, we use a bottom up approach, starting with a set of previously and independently developed context elicitation patterns.

The ultimate aim of this paper is to identify the common concepts, which are part of the already obtained context elicitation pattern, and aggregate them to a meta-model of elements one has to talk and think about when describing a new context elicitation pattern.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*. In this work we show a similar meta-model for context elicitation. We show *how* we derived it form already existing context elicitation pattern, and how it can be used to describe the structural part of a new context elicitation pattern.

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

Note that there are different views on the term *pattern language* and a *meta-model* is an important part of it. According to Fowler [Fowler 1996; 2002] a pattern language indeed is about the relations between patterns. Hafiz et al [Hafiz et al. 2012] agree and elaborate that an enumeration of pattern is just a pattern catalog. Both, Hafiz et al. and Fowler, basically adopt the view of Alexander [Alexander 1978] towards patterns and pattern languages. In contrast, Jackson [Jackson 2001] calls his domain-types and interfaces between already a language for expressing problem frames (which are kind of patterns [Jackson 2001]). The frames themselves are not related by Jackson. To be precise, Jackson never uses the term "pattern language", but a language for expressing problem frames, which are in turn patters is a close definition.

Those different views can be aligned in some way. Fowler states that a defined "form" [Fowler 1996] or "pattern structure" [Fowler 2002] is essential for expressing a pattern. He refers further to the work of Alexander. For Fowler a pattern structure should consist of a "sketch" [Fowler 2002], which is a structural and graphical description, and a textual description of behaviour and relations. A very similar understanding of how to describe a pattern can be found in Hafiz et al. [Hafiz et al. 2012] and Fernandez et al. [Fernandez and Pan 2001]. This understanding is in line with what Jackson calls a language to describe a problem frame (pattern) [Jackson 2001]. Hafiz refines the structural and graphical description further to a common vocabulary and syntax [Hafiz et al. 2012]. The textual

description additionally needs a grammar and behavioural elements. Hence, our meta-model is not a full pattern language. It "only" defines the vocabulary and syntax for describing pattern. But to form a pattern language it is necessary to have a common way to describe patterns [Fowler 1996; 2002]. Hence, we did the first step towards a pattern language and will complete the missing steps in future work.

Using this meta-model we empower requirements and software engineers to describe their own context elicitation patterns, which capture the most important parts for understanding the context of a system-to-be. Note, that the difference between our meta model and Tolendano's [Toledano 2002] meta-pattern is that he abstracted existing patterns into meta-pattern, while we want to create a meta model as a basis for a pattern language for context elicitation.

In addition, patterns produced with our meta-model are scalable, because it is possible to attach diagrams to the patterns, e.g., UML activity diagrams that explain interactions between different stakeholders. These diagrams can support different views and levels of granularity. All these diagrams are based upon our patterns. traceability links with the elements in the attached diagrams and the created patterns.

The domain knowledge elicited using our patterns is not limited to software engineering. We showed in previous work, e.g., [Beckers et al. 2013] that the knowledge can also support the establishment of security standards of the ISO 27000 family of standards [ISO/IEC 2009].

The rest of the paper is organized as follows. In the next section we show several, previously and independently described, patterns which were the input for out bottom method to build the meta-model. We show the meta-model and its development based on these pattern in Sect. 4. We explain how to use the meta-model in Sect. 5. Section. **??** illustrates how our patterns can support secure software engineering methods. Security is only an example of how our patterns can support quality focused software engineering processes. Section **??** presents related work and Section 6 concludes the paper.

2. DEFINING CONTEXT-PATTERN

Requirements define what properties and functionality a software should have. It is impossible to assess the quality of a software without requirements. Moreover, writing requirements is only possible if the domain knowledge of the system-to-be and its environment is known and considered thoroughly, 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 *how* to elicit domain knowledge correctly for effective requirements elicitation [Niknafs and Berry 2012]. Moreover, several requirements engineering methods exist for security. [Fabian et al.] concluded in their survey about these methods that it is not yet state of the art to consider domain knowledge. We address these problems by describing common structures and stakeholders for several different domains in so-called *context-patterns* for 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 (Sect. **??**) that considers requirements engineering from the point of view of a machine in its environment.

A context-pattern consists of the following parts:

Method. A method contains a sequence of steps. Each step is described using well defined activities, inputs, and outputs. Inputs are descriptions of the required artifacts to perform the activities of the method step. Activities are descriptions of the processing of all inputs into outputs. Outputs are the desired results of the activities of this step. A context-pattern has to contain a method that describes how to use the context-pattern.

Graphical pattern. Our context-patterns do not enforce considering the machine meaning the thing we are going to build explicitly, but demand a description of its environment in graphical form. This environment contains domain knowledge. In particular, any given environment considers certain elements, e.g., stakeholders or technical elements. Moreover, we believe that every environment of a software engineering problem can be divided into parts that have direct physical contact with the machine and parts in the environment that have an effect on the machine without physical contact, e.g., laws. These relations between the environment and the machine have to be part of the graphic, as well. A context-pattern has to contain at least one graphical pattern. We use a UML-based notation for our graphical patterns that uses, e.g., stick figures for actors, but we also use notation elements that are not part of the UML such as rectangles that symbolize an environment and all elements in the rectangle belong to this environment. We explain the concepts that we use in all our graphical patterns in Chapter **??** in detail. This is part of the effort to create a pattern language for context-patterns. Nevertheless, we show in Sect. **??** that a mapping from our graphical patterns to a model in strict UML notation is possible.

Templates. Templates contain additional information about elements in the graphical pattern. For example, a graphical pattern contains a graphical figure of a stakeholder and a corresponding template for the stakeholder can contain, e.g., the motivation of the stakeholder for using the machine and the relations to other stakeholders. Templates provide the means to attach further information to the graphical pattern. The reason for adding this refinement in a template and not in the graphical pattern is not to overload the graphical pattern with too many elements. Templates are optional elements of context-patterns, because not all graphical pattern require a refinement.

In addition, it is possible to attach diagrams to our context-patterns, e.g., UML activity diagrams that refine interactions between different stakeholders of a context-pattern (Chapter **??** for an example). These diagrams can support different views and levels of granularity. All these diagrams are based upon our patterns. Hence, we can apply consistency checks and traceability links with the elements in the attached diagrams and the created patterns.

Moreover, the domain knowledge elicited using our patterns is not limited to software engineering. We showed in previous work, e.g., [Beckers et al. 2013] that the knowledge can also support the establishment of security standards of the ISO 27000 family of standards [ISO/IEC 2009].

3. A CATALOG OF CONTEXT PATTERN

3.1 Peer-to-Peer System Analysis Pattern

Aligning software systems to meet requirements is hard, which is even more difficult when a Peer- to-Peer (P2P)-based software system shall be developed. For example, the effects of churn, the random leaving or joining of peers in the system, can cause data loss. If a requirement exists stating that no data loss shall occur in the system, then churn presents a challenge that has to be considered for this requirement. A software engineer can design a countermeasure, if she is aware of the challenge. This, however, is difficult, because numerous challenges that are caused by attributes of the P2P protocol or even the network layer.

We present a pattern-based method to identify existing challenges in a P2P-based software system. An initial pattern considers all layers of a P2P architecture and offers more detailed patterns for, e.g., P2P protocols. The instantiation of these patterns enables an analysis of the systemâĂŹs challenges and reveals the information in

which layer each challenge originates. An extensive description of the context-pattern can be found in [Beckers and Faßbender 2012].

3.2 Graphical 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 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.

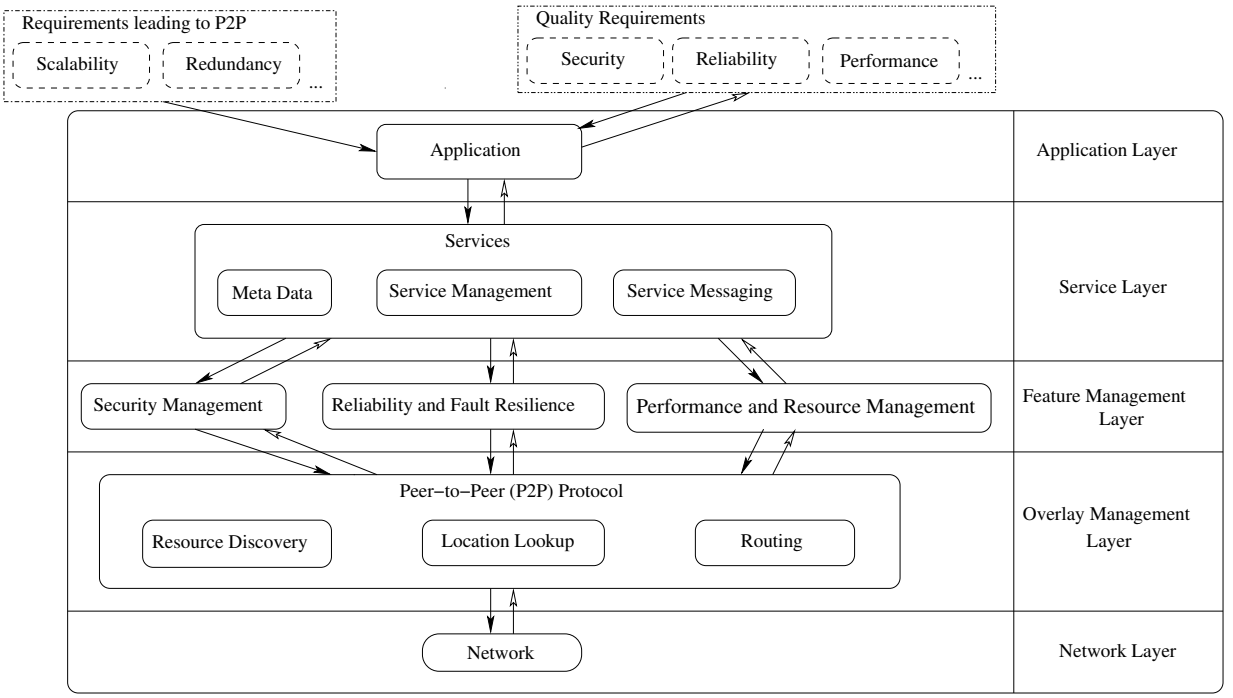

Fig. 1: P2P pattern

3.3 Templates

We created P2P templates for the layers of a P2P system, e.g., the network layer. For details we refer to [Beckers and Faßbender 2012].

3.4 Method

Step 1: Instantiate P2P-Pattern -

We illustrate our method in Fig. 2, which consists of instantiating the P2P pattern and assess the feasibility of the quality requirements for the P2P system. The steps are described in the following.

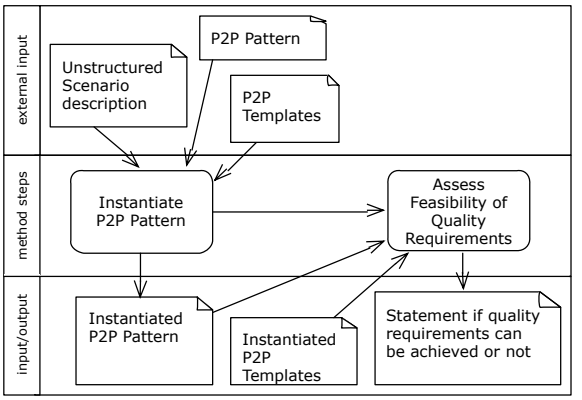

Fig. 2: Method for P2P Pattern Usage

Input:	Unstructured Scenario Description P2P Pattern and Templates
Output:	Instantiated P2P Pattern and Templates

Peer-to-Peer (P2P) architectures design distributed systems, in which identical software works on every peer. These systems are distributed without any centralized control or hierarchical organization that form a self organizing overlay network on top of the Internet Protocol (IP). An Overlay is a network which is built on top of one or more existing networks. This adds additional properties to the underlying network, e.g., more efficient search of data or adding locality information to peers. It provides a communication infrastructure for all peers in the P2P architecture. P2P systems can be clients and servers at the same time. They provide access to their resources by other systems and support resource sharing on an internet scale. This requires fault tolerance, self-organization, and significant scalability properties. One of the obstacles a P2P system has to overcome is Churn, the joining and leaving of peers in a P2P architecture without prior notification. Two fundamentally different types of P2P systems exist. Structured P2P systems organize peers via an algorithm, which leads to an overlay with specific properties. They are often based upon a distributed hash table. Unstructured P2P systems organize their peers in a random graph in a flat or hierarchical manner (e.g., Supernodes exist that outrank normal nodes). They are based upon techniques like flooding, random walks etc. Hence, P2P architectures are fundamentally different from stand alone or client server architectures. The pattern contains the layers of a P2P architecture, shown in Fig. 1. The instantiation of the pattern begins with one or more requirement(s) that lead to the decision to use a P2P protocol. For example, scalability or redundancy requirements can result in this decision. The pattern is derived of a survey of existing P2P systems. The requirements and the layers of a P2P system are connected with arrows. Arrows with a black arrowhead present the collection of information through the P2P layers, getting more detailed on every level. The white arrowheads present challenges that are derived from all the information collected before.

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. Metadata describes what the service offers. For example, content storage using P2P technology. Service messaging describes the messages and protocols services use to 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 these aspects are for maintaining the robustness of a P2P system. We renamed the resource management from the original architecture from [Lua et al.] into performance and resource management, because the quality of the access and distribution of resources is the main performance property of P2P architectures. The Overlay Management Layer is concerned with peer and resource discovery and routing algorithms. We aim to explain these in terms of software engineering and not network engineering. 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.

Step 2: Assess Feasibility of Quality Requirements -

Input:	Instantiated P2P Pattern and Templates
Output:	Statement that clearly describes if and why the quality requirements can or cannot be achieved.

We consider the impact of each challenge on each quality requirement. The challenges are investigated considering the chosen technologies for each of the layers of the P2P system. The analysis is based on the instantiated P2P-Pattern. We use this information to determine if the quality requirements are achievable ([Beckers and Faßbender 2012] for details).

3.5 Service-oriented Architecture Pattern

Our Service-oriented Architectures (SOA) pattern concerns eliciting domain knowledge for SoA. A detailed description of the context-pattern can be found in [Beckers et al. 2012].

3.6 Graphical Patterns

A SOA spans different layers [Beckers et al. 2012], which form a pattern on a SOA with technological focus, as depicted in Figure 3 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 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, such as 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 3 at 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 3 (bottom) spans the SOA layers that form the machine. The business processes describe the behavior 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 gray parts of Figure 3 (bottom). The light gray 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 for example something that exits in the environment independently of the machine or other entities. The dark gray part in Figure 3 (bottom) spans the indirect environment. It comprises all entities not related

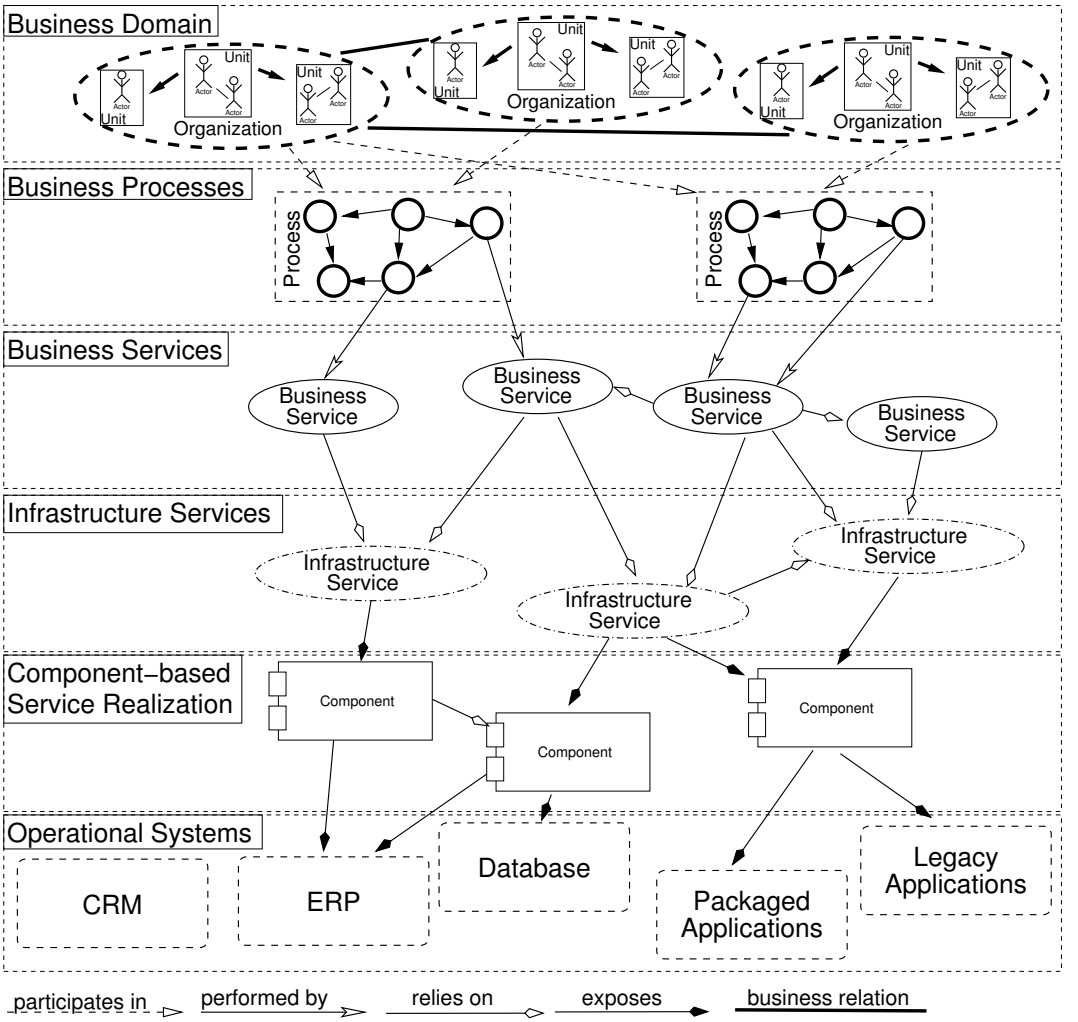

Fig. 3: SOA Layer Pattern

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 organizations. Some entities of the Indirect Environment influence one or more organizations. 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 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 environment. In Figure 3 (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].

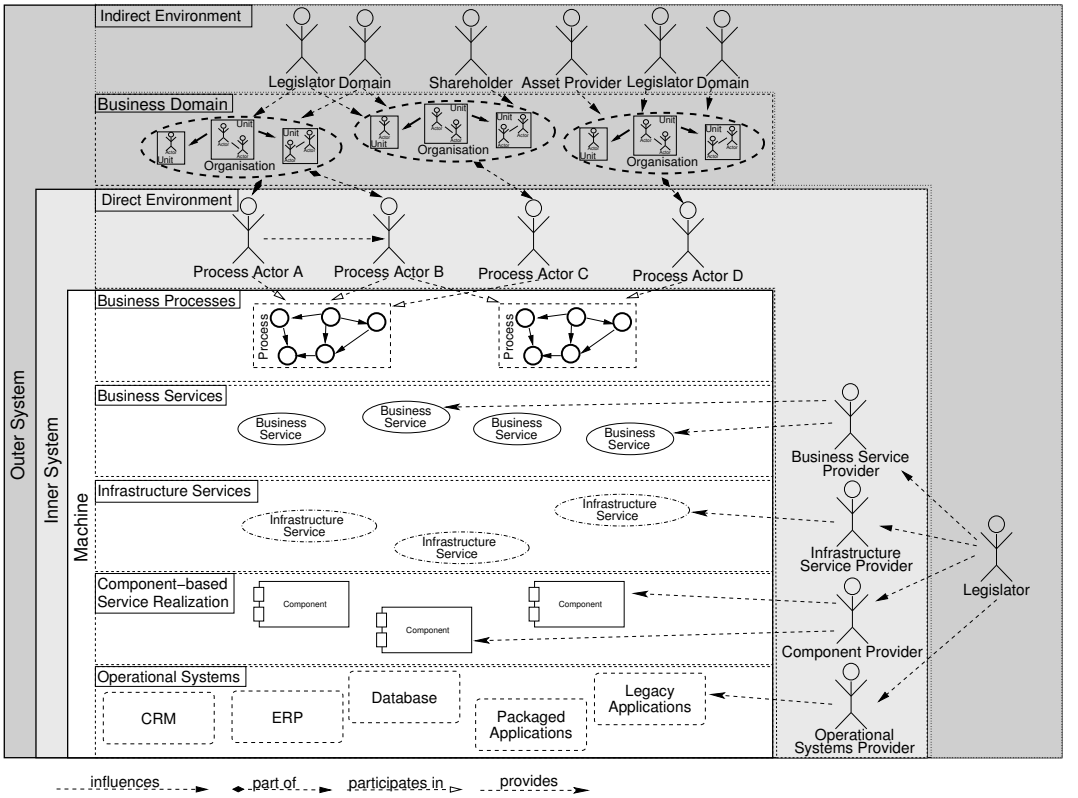

Fig. 4: SOA Layer Stakeholder Pattern

3.7 Templates

We created templates for direct and indirect stakeholders similar to the ones described in Sect. 3.15. For details we refer to [Beckers et al. 2012].

3.8 Method

Step 1: Information Structuring -

We illustrate our method for SOA knowledge elicitation in Fig. 5 and explain its steps in the following.

Input:	Unstructured Scenario Description SOA Layer Pattern
Output:	Instantiated SOA Layer Pattern

For structuring the information necessary to design a SOA according to the SOA layer pattern, we suggest a meet in the middle procedure. The reason is that the business services and infrastructure services layers are intertwined with the business elements. Therefore, we need information both about the technology-related and the business-related layers. Furthermore, our method provides validity checks for the relations between the different layers. The external input for all steps of this phase is the unstructured scenario description. The SOA Layer Pattern is an additional external input for the first step Describe Organizations. It is instantiated layer by layer in separated steps, based on the Unstructured Scenario Description. In the step Describe Organizations, we have to collect all relevant organizations. For each such organization, we have to collect statements about this organization,

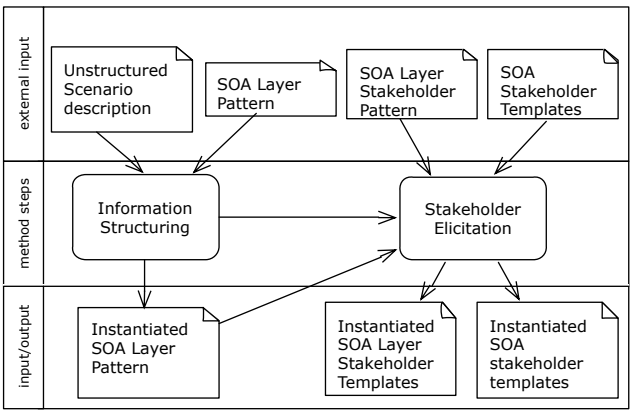

Fig. 5: SOA Knowledge Elicitation Method

describing it further. Next, we have to analyze these statements if they describe business relations between organizations. Last, we have to check for inconsistencies. For example, we have to ensure that no organization is isolated. Being isolated means that there are no business relations to other organizations. In case we find an isolated organization, this organizations is either not of relevance for our scenario, or we are missing important information.

In the next step Describe Choreography And Coarse Grained Processes, we have to structure the interaction between organizations. Additionally, we structure the available information about internal processes of organizations. The input for this step is the partly instantiated SOA Layer Pattern. We start with the organizations and their choreography. The choreography describes the interaction between the organizations. We recommend to document the choreography using an appropriate notation. For example, UML and SoaML collaboration diagrams can be used. UML and SoaML activity diagrams are of use for more detailed interaction and internal process descriptions. The described processes do not have to be complete, but processes and process steps already mentioned or explained in the scenario description should be captured by such diagrams. Next, we have to ensure the coherence of the SOA Layer Pattern instance for finishing the step Describe Choreography And Coarse Grained Processes. The interactions described by the choreography should reflect the business relations found for the organizations. Moreover, the detailed process descriptions must be coherent at the points of transitions between organizations. For the step Describe Operational Systems, we look for statements mentioning IT systems already used within one of the organizations. The operational systems found have to be analyzed for those, which are to be replaced, and those, which should be wrapped in the new SOA. Only the latter kind of operational system should be added to the SOA Layer Pattern instance. At least one component should be described in the step Describe Components for each operational system. Whenever for an operational system there is no statement about a component, which should be re-used, the information is missing in the Unstructured Scenario Description, or the operational system is unnecessary. Note that components can exist, which are not part of an operational system, but are already mentioned in the scenario description. But to be sure not to miss any relevant operational system, for each component mentioned in the Unstructured Scenario Description, we have to check if it exposes such a system. Up to this point, we have structured the business and the technical parts of the description. Next, we close the gap between those parts. We search for statements, which describe business services, for the step Describe Business Services. We add those services to the business service layer. For each business service, we have to check if there is a corresponding process step in the processes described in step Describe

Orchestration And Coarse Grained Processes. If such an activity is missing, it should be added. Additionally we have to check if the business service at hand directly exposes a component, which is already part of the SOA Layer Pattern instance. Last, we have to check whether the business service at hand is atomic or a composition of other business services. If it is a composition, the business services used to compose it have to be added to the business service layer, too. The procedure for Describe Infrastructure Services is almost the same. One difference is that infrastructure services are mapped to business services instead of activities within the process, and that they can be orphans. It is not necessary that an infrastructure service is used by an already known business service. The reason is that infrastructure services may provide a more general functionality.

Step 2: Stakeholder Elicitation -

Input:	Instantiated SOA Layer Pattern SOA Layer Stakeholder Pattern and Templates
Output:	Instantiated SOA Layer Stakeholder Pattern and Templates

In this phase, we elicit the stake- holders of our SOA. Therefore, we inspect each element of the SOA Layer Pattern instantiated in the previous phase. First, we instantiate the direct system environment. We start with the organizations given in the SOA Layer Pattern Instance. For each process related to an organization, we identify the process actors, which act on behalf of the organization in this particular process. There has to be at least one process actor for each organization-process-relation. For all process actors, we have to instantiate the corresponding Direct Stakeholder Templates. Finally, we have to establish the relations between associated process actors. Next, we inspect each business service, infrastructure service, component and operational system, whether there are already known provider(s) or not. When the providers are already known, we instantiate Direct Stakeholder Templates for them and add them and the corresponding relations to the SOA Stakeholder Pattern instance. Further, we instantiate the indirect system environment. We also start with the organizations. We analyze for each organization at hand, if there are relevant legislators, domains, shareholders and asset providers. For each identified indirect stakeholder, we instantiate the corresponding Indirect Stakeholder Template, and we add the indirect stakeholder and their relations to SOA Stakeholder Pattern instance. We repeat this procedure for all providers we find in the direct system environment.

3.9 Law Pattern

Commonly, laws are not adequately considered during requirements engineering. Therefore, they are not covered in the subsequent system development phases. One fundamental reason for this is that the involved engineers are typically not cross-disciplinary experts in law and software and systems engineering. Hence, we present in this section a context-pattern for identifying laws and regulations including a method to systematically consider laws in the requirements engineering process. For our method we chose the German law as the binding law.

3.10 Graphical Patterns

Based on the structure of laws (details can be found in [Beckers et al. 2012]), we define a *law pattern* shown in Fig. 6. The pattern consists of three parts: the dark grey part represents the *Law Structure*, the light gray part depicts the *Classification* to consider the specialization of the elements contained in the *Law Structure* in related laws or sections, and the white part considers the *Context*. The *Context* part of the law pattern contains the *Legislator(s)* defining the jurisdiction, and the *Domain(s)* clarifying for which domain the law was established.

As it is necessary to know in which context and relation a law is used, we introduce *Regulation(s)*, which are *Related To* the section at hand. *Regulation(s)*, *Legislator(s)*, and *Domain(s)* can be ordered in hierarchies, similar to classifiers. For instance, Germany is part of the EU and consists of several states.

Figure 7 shows our law identification pattern. The structure is similar to the law pattern in Fig. 6 to allow a matching of instances of both patterns. In contrast to the legal vocabulary used in the *Law Structure* of our law pattern, the wording for the elements in the dark gray colored *Core Structure* of our law identification pattern is

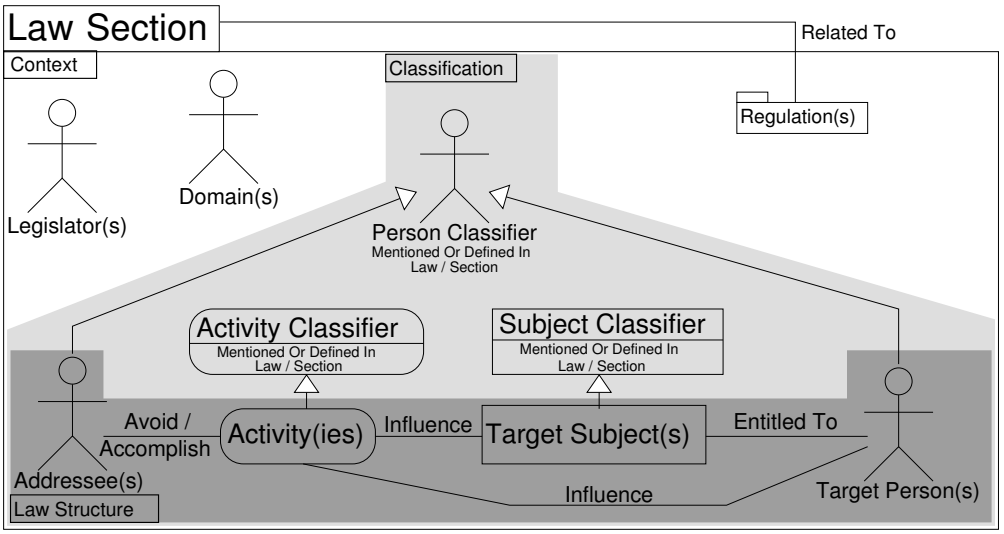

Fig. 6: Law Pattern

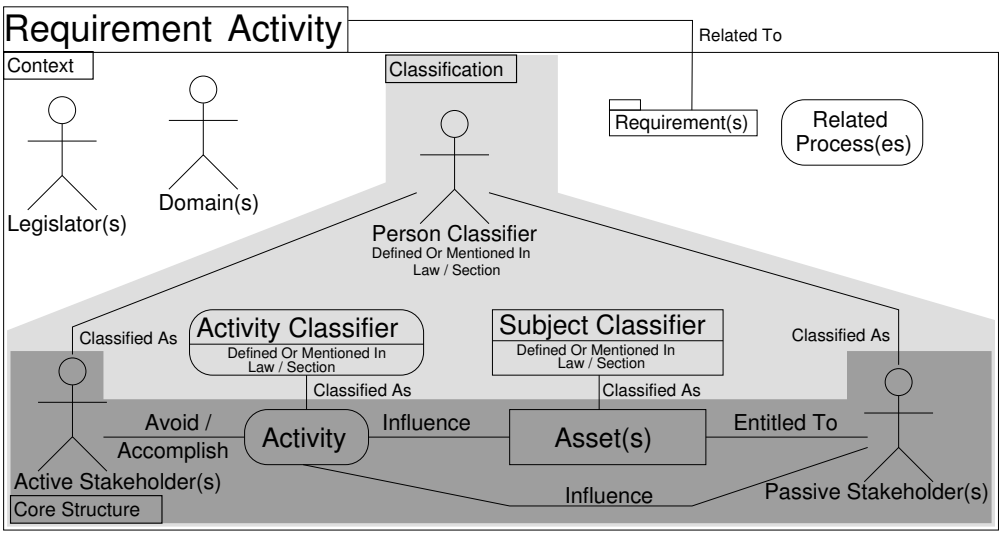

Fig. 7: Law Identification Pattern

based on terms known from requirements engineering. For example, the element *Asset(s)* in our law identification pattern represents the element *Target Subject(s)* in our law pattern.

Our law identification pattern takes into account that requirements are often interdependent (*Requirement(s)* in the *Context* part). Given a law relevant to a requirement, the same law might be relevant to the dependent requirements, too. Furthermore, the pattern helps to document similar dependencies for a given *Activity* using the *Related Process(es)* in the *Context* part.

3.11 Templates

We did not define templates for this context-pattern. Templates are optional in context-pattern (Sect. 2).

3.12 Method

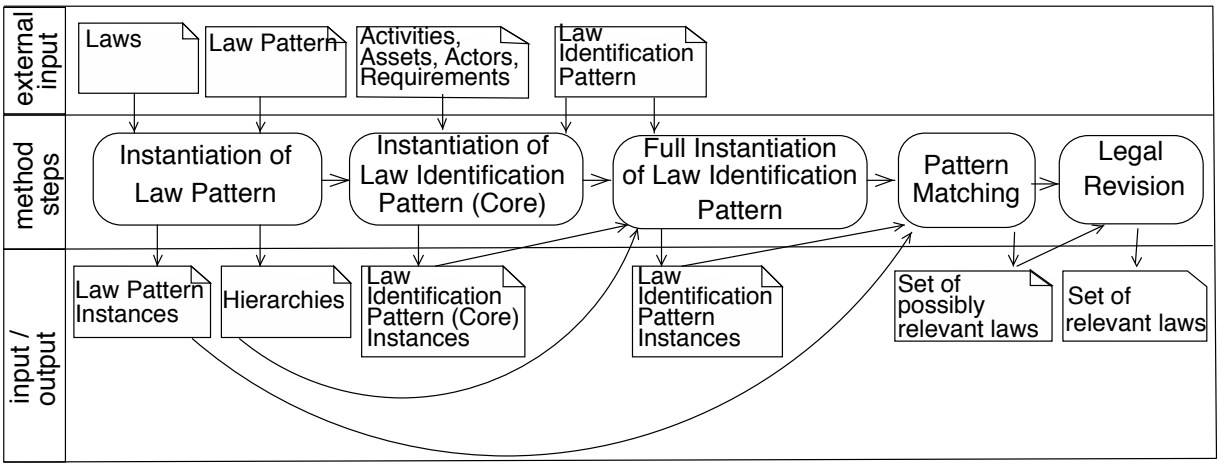

Fig. 8: Law Identification Method

Our general process for identifying relevant laws consists of five steps, which is depicted in Fig. 8. For this process law experts and software engineers have to work together for the necessary knowledge transfer. Step one can be done alone by legal experts and for step two only software engineers are needed. But in step three and four both groups are needed to bridge the gap between the legal and technical world. The last step can be accomplished alone by legal experts.

Step 1: Law Pattern -

Input:	Laws
Output:	Instantiated Law Patterns

In the first step we instantiate law pattern with the relevant laws for our scenario. We now describe the instantiation process for our law pattern using Section 4b BDSG as an example. We explained the importance of this particular law in the beginning of this section. Section 4b BDSG regulates the abroad transfer of data. The resulting instance is shown in Fig. 9. Our method starts based on the first sections of the law to be analyzed. These sections are self-contained, i.e. they define all necessary elements of our *Law Structure*. Additionally, the *Legislator(s)* and *Domain(s)* can be instantiated according to the considered law (e.g. *Germany* and *General Public* in the *Context* part). Given a section of a law not yet captured by our law pattern, we identify and document the related laws and sections referred to by the given section (e.g. *BDSG Sec. 1* in the *Context* part). Then, we search for the *Law Structure* directly defined in this section. In Section 4b BDSG, we find *Abroad Transfer*, and we use it to instantiate *Activity(ies)*. *Addres((s)*, *Target Subject(s)*, and *Target Person(s)* are not defined in Section 4b BDSG. Therefore, related sections defining these terms have to be discovered. In our example, we find *Private Bodies* for the *Addres((s)*, *Personal Data* for the *Target Subject(s)*, and *Individual* for the *Target Person(s)* in Section 1 BDSG (according to *BDSG Sec. 1* in the *Context* part). We arrange these specializations in the appropriate parts of the hierarchies in Fig. 10. The classifier is instantiated with the parent node of the corresponding hierarchy, which is for instance *Transfer* for *Abroad Transfer*.

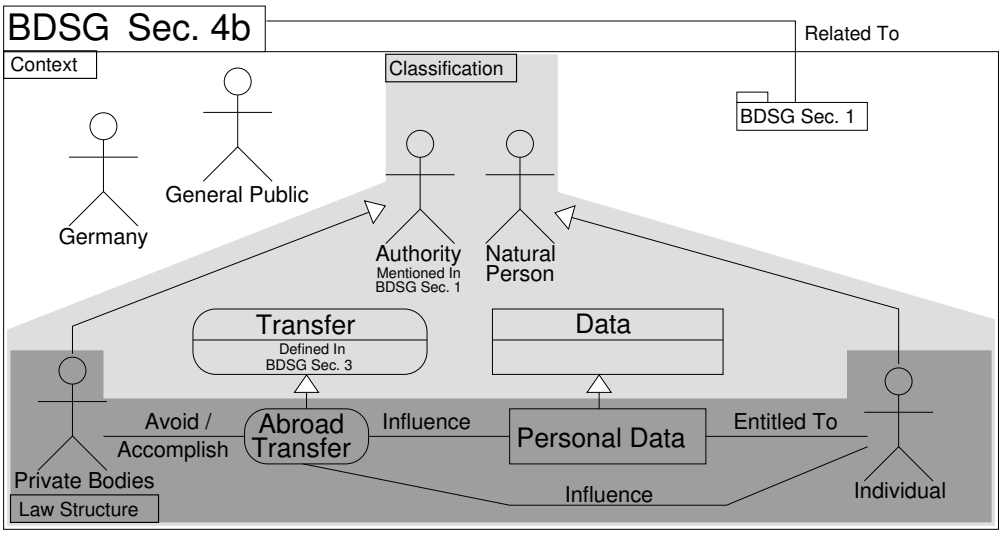

Fig. 9: Law Pattern Instance

Step 2: Law Identification Pattern -

Input:	Scenario Description and Requirements
Output:	Partially Instantiated Law Identification Patterns

Identifying relevant laws based on functional requirements is difficult, because functional requirements are usually too imprecise, they contain important information only implicitly and use a different wording than in laws. To bridge between the gap of wording and to facilitate the discussion between requirements engineers and legal experts, we define a *law identification pattern* to support identifying relevant laws.

As our example in Fig. 11 shows, we select *Hulda* as the cloud provider, then we choose the functional requirement *Scalable Data Storing*. One of the activities associated with this requirement is the activity *Store Distributed*, which refers to the asset *Customer Data* of the *Bank Customer*. Moreover, we instantiate the elements *Legislator(s)* and *Domain(s)*. In our example in Fig. 11, we include the legislators *Germany*, *US*, *EU*, and the domain *Finance*. In addition, we discover the related requirement *Cloud API* and the process *Offering Data Storing*, and document them in the instance of our law identification pattern.

Step 3: Establishing the Relation between Laws and Requirements -

Input:	Partially Instantiated Law Identification Patterns
Output:	Completely Instantiated Law Identification Patterns

To instantiate the *Classification* part, legal expertise is necessary. According to the *Core Structure* of the instance of our law identification pattern and the hierarchies built when instantiating our law pattern, legal experts classify

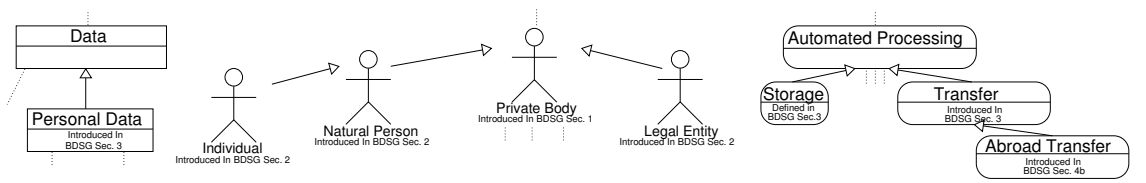

Fig. 10: Examples for Person (left), Subject (middle), and Activity (right) Legal Hierarchies

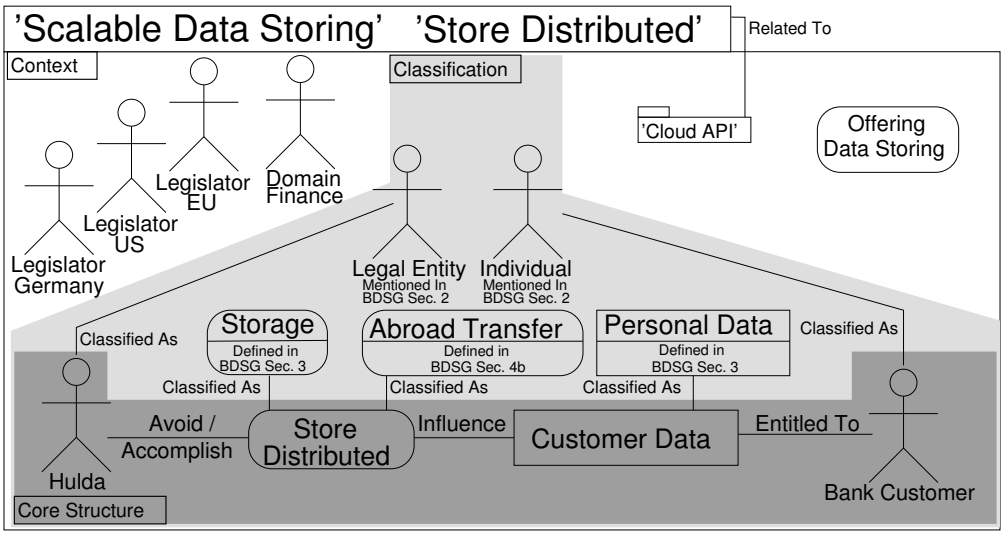

Fig. 11: Law Identification Pattern Instance

the elements of the *Core Structure*. For example, the activity *Store Distributed* is classified as *Abroad Transfer* based on a discussion between the legal experts and software engineers.

Step 4: Deriving relevant Laws -

Input:	Instantiated Law Identification Patterns, Instantiated Law Patterns
Output:	Set of possibly relevant Laws

The identification of relevant laws is based on matching the classification part of the law identification pattern instance (light gray part) with the law structure and classification part of the law pattern instance (light and dark gray parts), and thereby considering the previously documented hierarchies. If all elements match, the law is identified as relevant. For example, we find direct matches in the law pattern instance in Fig. 9 for the elements *Abroad Transfer*, *Personal Data*, and *Individual* contained in the law identification pattern instance shown in Fig. 11. *Hulda* is classified as *Legal Entity* and the only element that does not directly match with *Private Bodies* in the law structure of Section 4b BDSG. In this case, the hierarchy in Fig. 10 helps to identify that *Legal Entity* is a specialization of *Private Bodies*, and thus, we identify Section 4b BDSG as relevant. Finally, we check for all laws identified to be relevant if *Legislator(s)* and *Domain(s)* are mutually exclusive. In our example, the legislator *Germany* contained in *Context* of the law pattern instance depicted in Fig. 9 can be found in *Context* of the law identification pattern instance shown in Fig. 11. The domain *General Public* in the law pattern instance can be considered as a generalization of the domain *Finance* in the law identification pattern instance. The resulting set of laws relevant for the given development problem serves as an input for the next step.

Step 5: Legal Revision -

Input:	Set of possibly relevant Laws
Output:	Set of relevant Laws

This last step covers the identification and specification of requirements based on laws identified to be relevant by our method, e.g., using existing methods such as the one from [Breaux and Antón].

3.13 Cloud System Analysis Pattern

We present our cloud system analysis pattern in Sect. 3.14 that helps to systematically describe cloud computing scenarios and identify assets in these scenarios (Sects. 3.14, 3.15, and 3.16). We illustrates the instantiation of the pattern using an online banking service example (Sect. 3.17) and report on tool support (Sect. **??**). This context-pattern is based on the background presented in Sect. **??** on page **??**.

3.14 Graphical Pattern

The *cloud system analysis pattern* shown in Fig. 12 provides a conceptual view on cloud computing systems and serves to systematically analyse stakeholders and technical cloud elements. The notation used to specify the pattern is based on UML[1] notation, i.e. the stick figures represent roles, the boxes represent concepts or entities of the real world, the named lines represent relations (associations) equipped with cardinalities, the unfilled diamond represents a *part-of* relation, and the unfilled triangles represent generalization.

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 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. Typically, the *Indirect System Environment* is a significant source for compliance and privacy requirements.

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* (Sect. **??** on page **??**). The boxes *Pool* and *Service* in Fig. 12 are hatched, because it is not necessary to instantiate them. Instead, the specialized cloud services such as *IaaS*, *PaaS*, and *SaaS* and specialised *Resources* are instantiated. The

[1] Unified Modeling Language: `http://www.omg.org/spec/UML/2.3/`

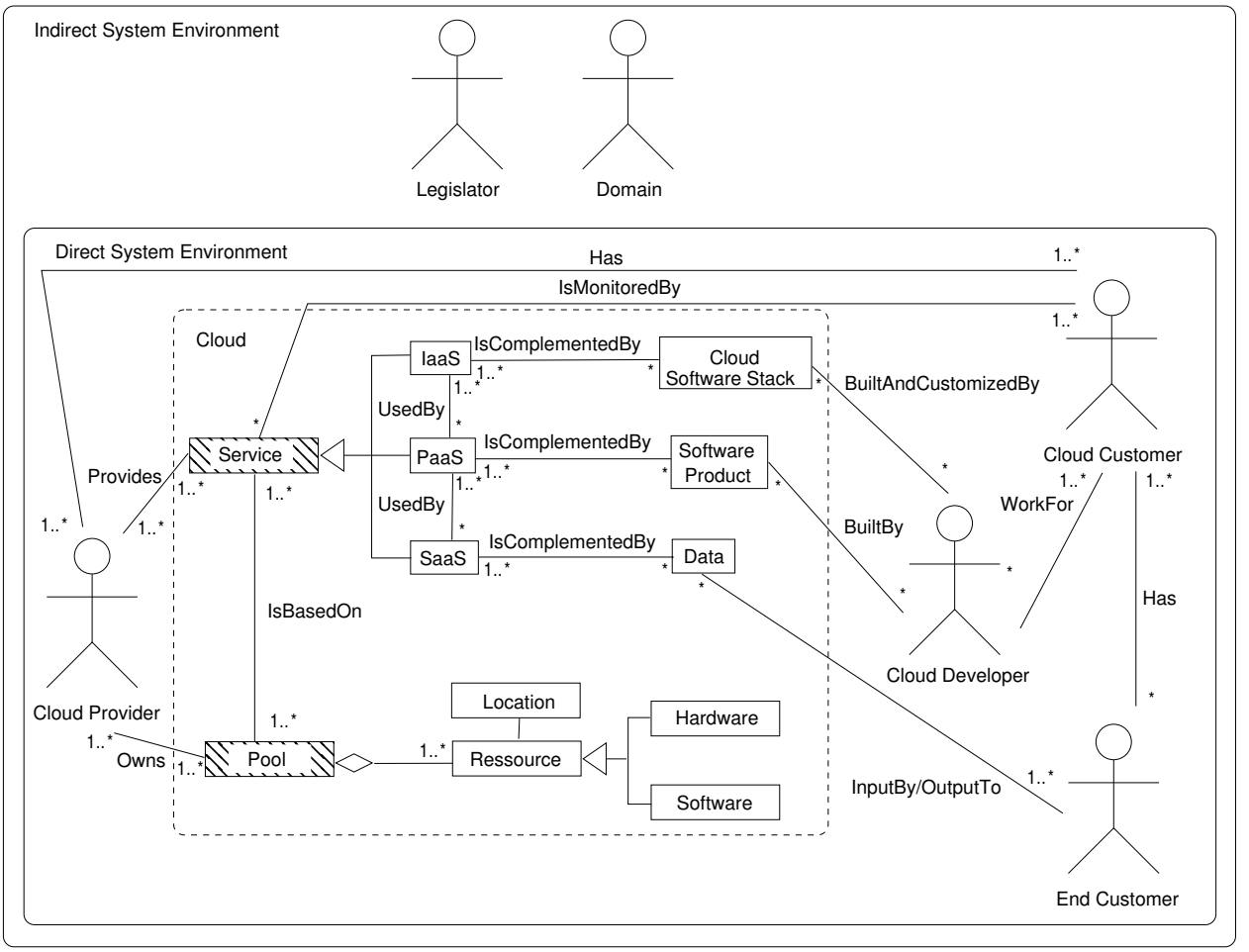

Fig. 12: Cloud System Analysis Pattern

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 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* in- 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*.

3.15 Templates

We accompany this cloud system analysis pattern by templates to systematically gather domain knowledge about the direct and indirect system environments based upon the stakeholders' relations to the cloud and other stakeholders. The first template serves to describe stakeholders contained in the direct system environment:

Name. State the identifier of the stakeholder or group of stakeholders, e.g., company name or group of end customers.

Description. Describe the stakeholder informally, e.g., whether the stakeholder is a natural or a legal person.

Relations to the cloud. Describe the inputs and outputs represented as a relation (line from this stakeholder to the cloud) between the stakeholder and the cloud, e.g., the kind of data or software.

Motivation. State the motivation of the stakeholder for using the cloud based on the previously elicited relations to the cloud, e.g., business goals such as profit and cost reduction.

Relations to other direct stakeholders. For each relation (line from this stakeholder to another direct stakeholder), name the kind of dependency between the stakeholders, e.g., controlled by contract, served by, indirectly influenced by customer-demand.

Assets. Document the already known assets of this stakeholder. Note that these assets are not represented in the graphical cloud pattern and just added to the list of assets created in Step 2 of our method.

Compliance and Privacy. Document the already known compliance and privacy laws as well as regulations.

The second template serves to describe stakeholders contained in the indirect system environment:

Name. (direct stakeholder template.

Description. (direct stakeholder template.

Relations to other stakeholders. For each relation from this stakeholder to another direct or indirect stakeholder (no line explicitly shown), name the kind of dependency between the stakeholders, e.g., protected by, controlled by law, implement laws.

Motivation. State the motivation of the stakeholder for having any kind of indirect relation to stakeholders of the direct or indirect environment or technical cloud elements, e.g., protect privacy of citizens or enforce concrete laws of an economic community.

Compliance and Privacy. Identify relevant compliance and privacy laws and regulations for the cloud scenario, e.g., German Federal Data Protection Act (BDSG) and the German Law on Monitoring and Transparency in Businesses (KonTraG).

3.16 Method

We illustrate our method in Fig. 13, which consists of two steps. The context establishment of a cloud scenario and the asset identification for this scenario. The steps are explained in the following.

Step 1: Context Establishment -

Input:	Unstructured Scenario Description, Cloud System Analysis Pattern and Templates
Output:	Instantiated Cloud System Analysis Pattern and Templates

We now generally explain the process of instantiating our cloud system analysis pattern. The instantiation of the direct cloud environment is done first and the indirect environment afterwards. The reason for this sequence is that we need to know the stakeholders using the cloud and software type running in the cloud to determine the relevant rules and regulations relevant for instantiating the indirect cloud environment.

(1) Instantiate the direct system environment

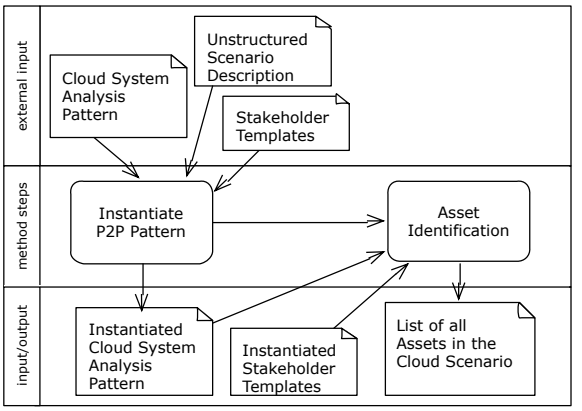

Fig. 13: A Method for Cloud Context Elicitation

(a) State the instantiations of the cloud stakeholders of the direct system environment, e.g., which company is the cloud provider.
(b) Moreover, the stakeholders defined by the cloud system analysis pattern can be complemented by further stakeholders.
(c) For each direct stakeholder instantiate the direct stakeholder template.

(2) Instantiate the cloud
(a) Provide a functional description of the software offered by the cloud. Define the service layer, e.g., SaaS the software is located in, the in- and output of the service(s), and the connections to the stakeholders of the direct system environment.
(b) The *Data* might be analyzed more precisely using, e.g., class diagrams. This helps to apply asset identification and analysis techniques in the second step of this method.
(c) Provide the geographical location(s) of the cloud.
(d) State the deployment scenario of the cloud (private, public, hybrid).
(e) State the technical implementation behind the system, e.g., the required applications in the cloud software stack to provide an SaaS offer.

(3) Instantiate the indirect system environment
(a) Determine the relevant domains (e.g. finance, medical, insurance) for the cloud scenario by considering the outsourced processes of the cloud customer and select the relevant legislators (e.g. Germany, US). Such legislators or jurisdictions are relevant, where the resources are located, i.e. where the data is physically stored and processed, and where users, providers and cloud customers are based.
(b) In addition to the indirect stakeholders covered by the cloud system analysis pattern, further indirect stakeholders can be added.
(c) For each indirect stakeholder instantiate the indirect stakeholder template.

In our example in Fig. 14, the domain is the *Domain Finance*, and the relevant legislators are *Germany*, *EU*, and *US* (since the data centers of *Hulda* are located in Germany as well as in the US).

The cardinalities contained in the pattern can be instantiated in a restricting way only. When instantiating the cloud system analysis pattern, one also fills in the corresponding templates.

Step 2: Identify Assets -

Input:	Instantiated Cloud System Analysis Pattern and Templates
Output:	List of Assets

We explain how to identify assets in the scope of the organization at an appropriate level of detail based on the ISO 27005 standard [ISO/IEC 2008]. We distinguish between *primary assets* and *supporting assets*. Primary assets are *business processes*, *activities* and *information*. Supporting assets are *hardware*, *software*, *network*, *personnel*, *site* or *organization's structure*. The input for the identification of assets are the scope and boundaries identified in the context establishment activity

The cloud system analysis pattern helps identifying the primary and supporting assets by considering the instantiated boxes and the associations between the direct stakeholders and the cloud. The associations indicate the flow of information into and out of the cloud and therefore help to analyze the information assets processed and stored in the cloud.

Primary asset information comprises *vital information*, *personal information*, *strategic information* and *high-cost information*. Vital information is relevant for running the organization's business, personal information comprises personal data or privacy relevant data, strategic information is required for achieving business goals, and high-cost information is information whose gathering or processing requires a long time or high acquisition cost.

3.17 Example

Step 1: Context Establishment

We illustrate our pattern-based support for security standards, using the example of a bank offering an online-banking service for their customers. This bank plans to source out the affected IT processes to reduce costs and scale up their system for a broader amount of customers. Customer data such as account number, amount, and transaction log history are stored in the cloud, and transactions like credit transfer are processed in the cloud. The bank authorizes the software department to design and build the cloud-specific software according to the interface and platform specification of the cloud provider. Figure 14 shows an instance of our cloud system analysis pattern regarding our online banking service example.

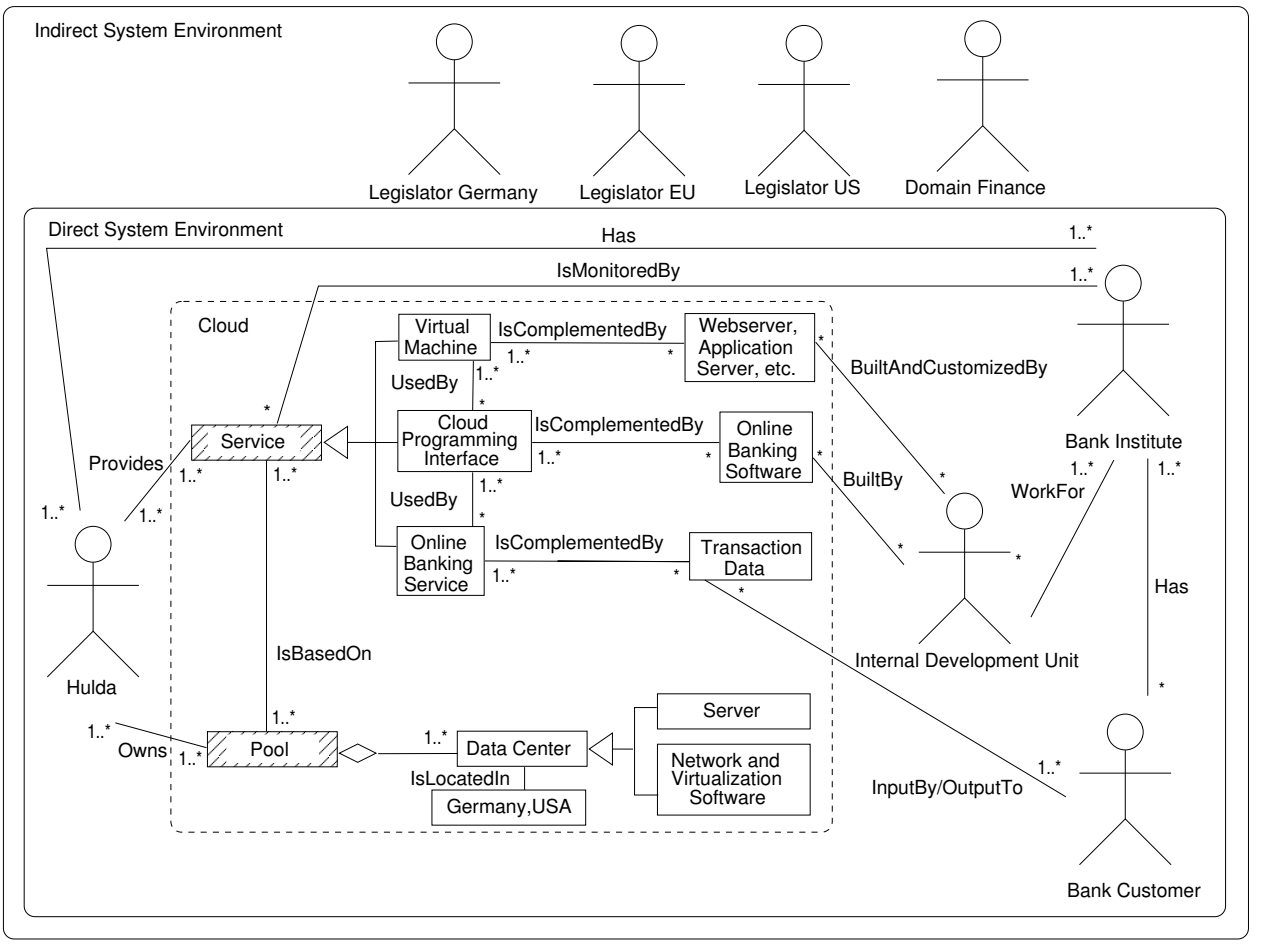

Fig. 14: Concrete Cloud Computing System for Online Banking Service

(1) Instantiate the direct system environment
In our example in Fig. 14, we assume that the cloud provider is *Hulda*, the cloud customer is the *Bank Institute*, the end customers are *Bank Customers*, and the cloud developer is the *Internal Development Unit* of the *Bank Institute*.

(2) Instantiate the cloud
In our example in Fig. 14, we consider the *Online Banking Service* as an SaaS offer, and the data is instantiated with *Transaction Data*. The cloud of the provider *Hulda* is public, and the data centers are located in Germany as well as in the US.

(3) Instantiate the indirect system environment
In our example in Fig. 14, the domain is the *Domain Finance*, and the relevant legislators are *Germany*, *EU*, and *US* (since the data centres of *Hulda* are located in Germany as well as in the US).

We present an example of an indirect stakeholder template instance:

Name. Legislator Germany

Description. The *Legislator Germany* represents all German laws relevant for this cloud scenario.

Motivation. The German laws try to control the risks of companies (*Hulda* and *Bank Institute*) and to protect the privacy of the *Bank Customers* by regulating disclosure of personal data.

Relations to other stakeholders. Controlled by law: The laws have to be obeyed by all stakeholders of the *Direct System Environment*.

Compliance and Privacy. The following regulations might be considered:
—Privacy protection: e.g., BDSG
—Risk management: e.g., German Stock Corporation Act (AktG)

We present an example of an direct stakeholder template instance, as well:

Name. Bank Customer

Description. The *Bank Customer* uses the online banking service of the *Bank Institute*.

Motivation. The *Bank Customer* wants cheap and secure financial transactions via the bank‘s online banking service.

Relations to the cloud. Financial data is *InputBy* the *Bank Customer*. Financial data is personal information that the *Bank Institute* acquires from the *Bank Customer*. The information is required by the *Bank Institute* for billing purposes.

Relations to other stakeholders. Has: *Bank Institute* as SaaS provider

Assets. Financial data, data related to the person

Compliance and Privacy. The following regulations might be considered:
—Privacy protection: BDSG Section 3, Section 4, Section 9, Section 11
—Risk management: AktG Section 91, Section 93

Step 2: Asset Identification

The cloud system analysis pattern instance in Fig. 14 helps identifying the primary and supporting assets. For instance, *Transaction Data* is related to the *Bank Customer* in Fig. 14. Thus, *Transaction Data* that can be refined to more concrete assets such as account number or account balance can be identified as a primary information asset. The owner of that asset is the *Bank Customer*. We categorize the assets in our example as follows: *Transaction Data* as personal information due to the relation to the *Bank Customer*. Here, privacy related data, e.g., account balance, are exchanged with the cloud. Further, billing data can be regarded as vital information for the *Cloud Provider* to run its business. In addition, *Transaction Data* can also be classified as vital information, because it is essential for running the organizationâĂŹs business.

The cloud in Fig. 14, surrounded by the dashed line, structures the cloud system and the instantiated boxes represent supporting assets. Different refinements of the supporting asset software can be found: *Virtualisation Software* as well as the *Online Banking Software*. The latter is deployed into the cloud by the *Internal Development Unit*, which could be wrongly determined as the asset owner. Using the associations in our pattern, it can be analyzed that the *Internal Development Unit WorkFor* the *Bank Institute*, which is therefore a more appropriate owner of the *Online Banking Software*, because it has contracted the software.

The personnel type consists of all the groups of people involved in the information system as for example *users*, *developers* or *decision makers*. The stakeholder in the *Direct System Environment* as well as the *Indirect System Environment* support the refinement of this supporting asset in the standard. We consider the location of elements in the scope, as well. We have to identify the location of the *Data Centers*. This means for our example that personal data processed or stored in data centers of the US do not comply to German privacy laws, and therefore this fact has to be documented here.

		General Concept												
		Layer	Stake-holder	Process	Active Re-source	Relation	Environ-ment	Indirect Environ-ment	Indirect Stake-holder	Direct Environ-ment	Direct Stake-holder	Ma-chine	Area	Re-source
SOA Layer Pattern Element	Business Organizations	x												
	Organization		x											
	Business Processes	x												
	Process			x										
	Business Services	x												
	Business Service				x									
	Infrastructure Services	x												
	Infrastructure Service				x									
	Component-based Service Realization	x												
	Component				x									
	Operational Systems	x												
	CRM				x									
	ERP				x									
	Database				x									
	Packaged Applications				x									
	Legacy Applications				x									
	Participates In					x								
	Performed By					x								
	Relies On					x								
	Exposes					x								
	Business Relation					x								
Stakeholder SOA Pattern Element	Outer System						x							
	Indirect Environment							x						
	Legislator								x					
	Domain								x					
	Shareholder								x					
	Asset Provider								x					
	Inner System						x							
	Direct Environment									x				
	Process Actor										x			
	Business Service Provider										x			
	Infrastructure Service Provider										x			
	Component Provider										x			
	Operational Systems Provider										x			
	Machine											x		
	Influences					x								
	Part Of					x								
	Provides					x								
Conceptual Element	Layer												x	
	Stakeholder													
	Process													
	Active Resource													x
	Relation													
	Environment												x	
	Indirect Environment						x							
	Indirect Stakeholder		x											
	Direct Environment						x							
	Direct Stakeholder		x											
	Machine												x	
	Area													
	Resource													

Table I. : Analysis of the SOA Layer Pattern and the Stakeholder SOA Pattern Elements

The organization's structure contains for example *subcontractors* of the organization. Here, the *Direct Stakeholder Environment* could help identifying them. The *Cloud Provider* as well as the *Internal Development Unit* can be documented here, as well.

4. DERIVING A META-MODEL FOR PATTERN-BASED CONTEXT ELICITATION

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 elicitation pattern in isolation. For each element in a context elicitation pattern we discussed, what the general concept behind this element is or if it is a general concept in itself. Therefore, we set up a table containing the elements of the current pattern to be analyzed as rows and the conceptual elements as columns. For each concept found we checked if this concept was already covered in the table or not. In the case it was already covered we only added a cross to the table. In the case it the concept was not covered by a conceptual element, we added a now column. After iterating over the elements of the pattern, we did a second step by adding the found conceptual elements as rows and analyzing for each of them if the could be further generalized in a reasonable way or not. This way we found also new conceptual elements. Hence, we had to do the second step several times. If nothing new was found, we finished the analysis. This way, we obtained the conceptual elements, which were candidates for the meta-model.

	Type	technical	technical	Technical, organizational	technical, organizational	organizational	organizational
	Pattern	P2P Pattern	SOA Layer Pattern	Stakeholder SOA Pattern	Cloud Pattern	Law Identification Pattern	Law Pattern
Meta-model Element	Pattern	x	x	x	x	x	x
	Areas	x	x	x	x	x	x
	Machine	x	x	x	x		
	Environment		x	x	x	x	x
	Direct Environment		x	x	x	x	x
	Indirect Environment		x	x	x	x	x
	Layer	x	x	x			
	Process		x	x		x	x
	Activity		x	x		x	x
	Stakeholder		x	x	x	x	x
	Direct Stakeholder		x	x	x	x	x
	Indirect Stakeholder			x	x	x	x
	Resource	x	x	x	x	x	x
	Active Resource	x	x	x	x		
	Passive Resource				x	x	x
	Relation	x	x	x	x	x	x
uncovered Elements	Requirements	x				x	
	Requirements leading to P2P	x					
	Requirements Influenced by P2P	x					

x = contains element x = contains element implicit

Table II. : Overview of Elements of the Context Elicitation Pattern and their relation to the Meta-model

Table I shows the result of this phase for the SOA pattern. In this case, we analyzed two pattern in conjunction, because the stakeholder SOA pattern reuses many elements of the SOA layer pattern.

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 set of conceptual elements over all patterns.

In the last phase we had to chose, which conceptual elements should be part of the meta-model. Table II shows the conceptual elements and in which of the patterns a corresponding element exists. Additionally, we selected for each pattern those elements which were not explicitly part of the pattern and checked if the missing element is an implicit part of the pattern. The patterns were also tagged with the information if there is a technical or organizational view provided or a combination of both. This is important to consider, because there might be elements, which only occur in one of the views. Those elements might be excluded by just looking at the pure occurrence number, because they can only occur in a subset of the pattern. But those elements might be nevertheless important to capture aspects which are special for a view.

The general rule to include an element into the meta-model or not, was to add every element with an occurrence greater than three, which means the element occurs in more than the half of the patterns. In case of a view specific element an occurrence of greater than two was sufficient, because the number of patterns associated with a view was four. Every element with an occurrence of two was subject to be discussed. The occurrence of an element was calculated only considering the explicit occurrence in a pattern.

We had to discuss the conceptual elements requirement and machine. For the machine element it seemed that it is only part of patterns which mix-up the technical and the organizational view. So the first reason to include them is that for eliciting the context of a software problem the most usual pattern is one which mixes the technical and organizational view. This reason was supported by the experiences of the authors and context elicitation patterns, which are currently developed and studied, but which are not published yet. A second reason was the fact that the patterns with a more technical view contain the machine implicitly. For example, for the SOA Layer pattern (see Fig. 16) the machine is not an explicit model element, but the extension to the Stakeholder SOA pattern (see Fig. 17) shows that elements of the SOA layer pattern directly relate to the machine. We could not find similar evidence for the requirement element. Moreover, we think that the requirement is part of the phases which follow

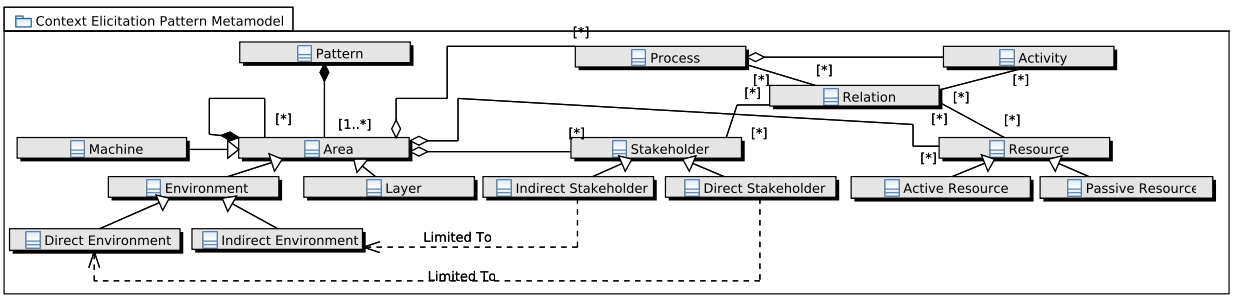

Fig. 15: Context Elicitation Pattern Meta-model

the context elicitation. For the P2P pattern we only added them for visualization means. Law Identification patterns are used in an iterative way. Thus, they are applied after the ideal context without legal restriction is elicited. Hence, this is a very specific case, which one can not generalize. As result, the requirement element is excluded and the machine element added to the context elicitation meta-model.

Finally, we formed the meta-model as depicted in Fig. 15 out of the selected conceptual elements. The meta-model was modeled using the UML notation.

The root element is the `Pattern` itself. Each pattern consists of at least one `Area`. In general, an area contains elements of the same kind, view or level. An area can contain other areas to split it up and make it more fine-grained. An area can be the `Machine`, i.e. the thing to be developed, or an `Environment`, which contains in turn elements that have some kind of relation to the machine, or a `Layer`, which encapsulates elements of the same hierarchy level.

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 material or immaterial 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 elicitation 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 elicitation patterns. Thus, it is helpful to know this elements and search for them in a specific domain when setting up a new context elicitation pattern for a domain. Fourth, it enables to form a pattern language for the context elicitation pattern. The common meta-model eases relating the patterns to each other.

5. APPLICATION OF THE META-MODEL

After the definition of the meta-model we instantiated it for each of our context elicitation patterns. thus, we aligned all of the patterns to the same foundation making them comparable. Additionally, when integrating context elicitation

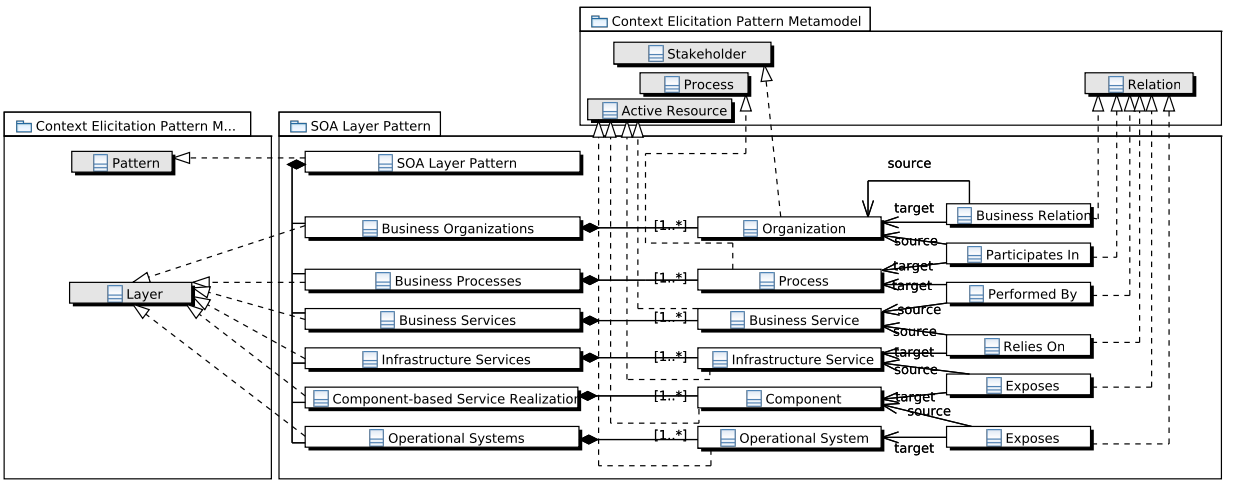

Fig. 16: SOA Layer Pattern Metamodel

patterns into requirements engineering methods this can be done in general only referring to the context elicitation meta-model.

An example of the result of the application of the meta-model can be seen in Figures 16 and 17. Figure 16 shows the meta-model of the SOA layer pattern. The root *pattern* element is the *SOA Layer Pattern* itself. *Business Organizations*, *Business Processes*, *Business Services*, *Infrastructure Services*, *Component-based Service Realization*, and *Operational Systems* are instances of the *Layer* element and part of the *SOA Layer Pattern*. *Organization* is an instance of the *Stakeholder* element. *Process* is a *Process*. *Business Service*, *Infrastructure Service*, *Component*, and *Operational Service* are instances of the *Active Resource*. *Business Relation*, *Participates in*, *Performed By*, *Relies On*, and *Exposes* are *Relations*.

As the stakeholder SOA pattern is an extension of the SOA layer pattern, it re-uses elements of the SOA layer pattern as shown in Fig. 17. It adds the *Stakeholder SOA Pattern* as new *Pattern* instance. *Environment* instances are the *Outer System* and the *Inner System*. The *Indirect Environment* and the *Direct Environment* of the stakeholder SOA pattern are instances of the *Indirect Environment* respective *Direct Environment* of the context elicitation meta-model. The Stakeholder SOA pattern adds a organizational view to the technological focus of the SOA layer pattern. Hence, there is a *Machine* explicitly. As the pattern is stakeholder centric, it basically adds *Stakeholder* and their *Relations*. *Legislator*, *Domain*, *Shareholder*, and *Asset Provider* are instances of the *Indirect Stakeholder* element. For *Direct Stakeholder* is instantiated as *Process Actor*, *Business Service Provider*, *Infrastructure Service Provider*, *Component Provider*, and *Operational Systems Provider*.The new *Relations* added are *Influences*, *Part Of*, *Participates In*, and *Provides*.

These two examples of the application of the meta-model show that the meta-model is sufficient for instantiating context elicitation pattern. Each pattern could be described using the meta-model without any problems. This way we prove that the generalization we did for forming the meta-model was reasonable.

6. CONCLUSIONS

We have presented a first step for creating a pattern language for context patterns, which provide a structured means for eliciting domain knowledge. This step is creating a meta model for context patterns. We illustrated our approach by showing context patterns, e.g., patterns that consider specific technologies like Peer-to-Peer networks, specific types of architectures like cloud computing, and specific domains, e.g., the legal domain. All of these patterns relate to our meta model.

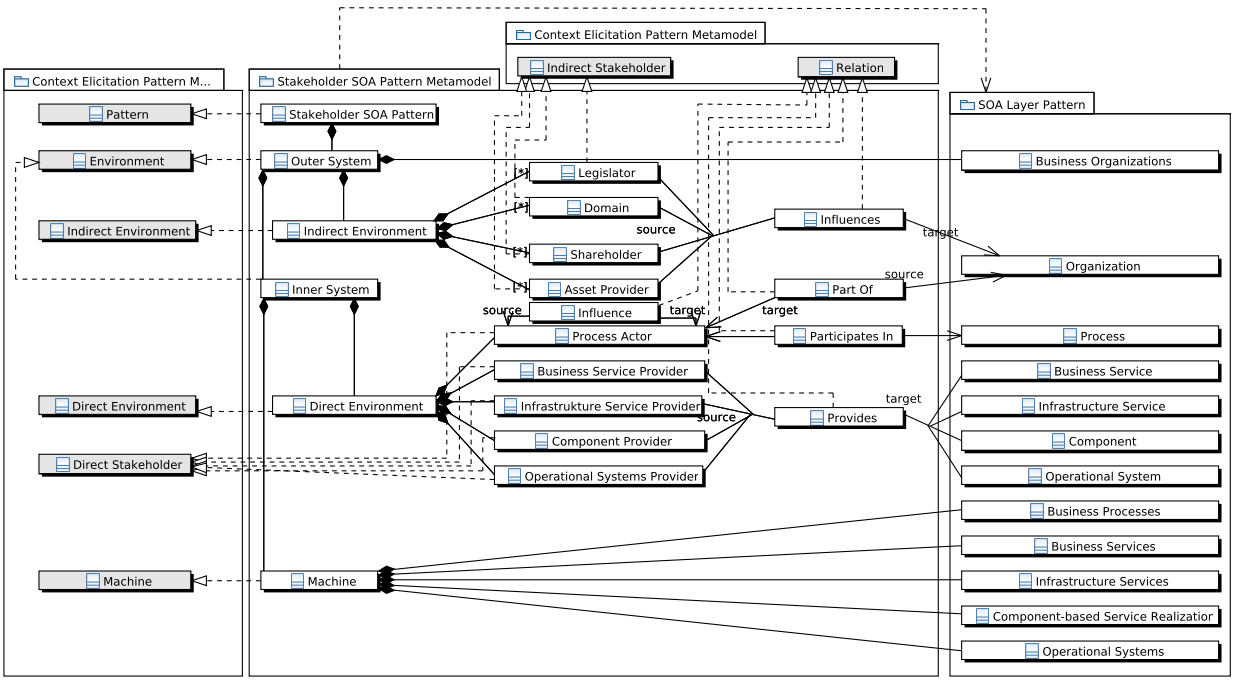

Fig. 17: Stakeholder SOA Pattern Metamodel

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].

Our approach offers the following main benefits:

—A meta-model for describing context patterns for various kinds of domain knowledge. This enables
  —comparing different context elicitation patterns
  —transferring information between context elicitation patterns instances
  —a guided description of new pattern using common elements and terms
  —forming a pattern language for context elicitation

—The patterns can be accompanied by further diagrams to support different views and detail levels to support scalability

—The patterns can be integrated into existing software development processes in order to improve context elicitation activities

—The patterns are useful beyond software engineering. For example, they can be applied to create the documentation for implementing security standards e.g. ISO 27001 [ISO/IEC 2005].

In the future, we will extend our method to provide support for creating textual requirements patterns and derive software architectures. We will also provide the means for consistency checks of instantiated patterns and requirements. For example, it will be possible to check if stakeholder in the domain knowledge pattern instance is considered in software requirements.

7. ACKNOWLEDGEMENTS

We thank our shepherd Hugo Ferreira for fruitful discussions and constructive feedback, as well as the members of our workshop Veli-Pekka Eloranta, Frank Frey, Carsten Hentrich, James Noble, Daniel Sagenschneider, Dietmar Schütz, Michael Weiss, Uwe Zdun and Christian Köppe for helpful discussions.

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) 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. 1978. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

BECKERS, K., CÃŤŤÃĽ, I., FAÃ§BENDER, 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., 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 analyzing laws. In *Proceedings of the International Working Conference onRequirements Engineering: Foundation for Software Quality (REFSQ)*. LNCS 7195. Springer, 256–262.

BREAUX, T. D. AND ANTÓN, A. I. 2008. Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering 34,* 1, 5–20.

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.

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.

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).

ISO/IEC. 2008. Information technology - security techniques - information security risk management. ISO/IEC 27005, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), Geneva ,Switzerland.

ISO/IEC. 2009. Information technology - Security techniques - Information security management systems - Overview and Vocabulary. ISO/IEC 27000, 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.

TOLEDANO, M. D. D. 2002. Meta-patterns: Design patterns explained. Tech. rep.

EuroPLoP'18, July 10-14 2013, Kloster Irsee, Bavaria, Germany. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7