# An integrated Method for Pattern-based Elicitation of Legal Requirements applied to a Cloud Computing Example

Kristian Beckers,Stephan Faßbender
*paluno - The Ruhr Institute for Software Technology – University of Duisburg-Essen, Germany*
*Email: {firstname.lastname}@paluno.uni-due.de*

Holger Schmidt
*ITESYS - Institut für technische Systeme GmbH, Germany*
*Email: {h.schmidt}@itesys.de*

***Abstract*—Considering legal aspects during software development is a challenging problem, due to the cross-disciplinary expertise required. The problem is even more complex for cloud computing systems, because of the international distribution, huge amounts of processed data, and a large number of stakeholders that own or process the data. Approaches exist to deal with parts of the problem, but they are isolated from each other.**

**We present an integrated method for elicitation of legal requirements. A cloud computing online banking scenario illustrates the application of our methods. The running example deals with the problem of storing personal information in the cloud and based upon the BDSG (German Federal Data Protection Act). We describe the structure of the online banking cloud system using an existing pattern-based approach. The elicited information is further refined and processed into functional requirements for software development. Moreover, our method covers the analysis of security-relevant concepts such as assets and attackers particularly with regard to laws. The requirements artifacts then serve as inputs for existing patterns for the identification of laws relevant for the online banking cloud system. Finally, our method helps to systematically derive functional as well as security requirements that realize the previously identified laws.**

***Keywords*-law, security, requirements engineering, software architecture**

## I. Introduction

Eliciting legal requirements for a software system and aligning it to be *compliant* is a difficult task. In order to accomplish this task the identification and analysis of relevant laws, is considered to be difficult, because it is a cross-disciplinary task in laws and software and systems engineering ([1]). Pattern-based approaches capture the knowledge of domain experts for re-use. Hence, we proposed a pattern-based approach for identifying and analyzing laws in our earlier work [2]. However, the identification and analysis of a relevant law alone is not sufficient for software engineers. They require a structured method that uses this approach to derive software requirements and further implementable software specifications.

We explain our method via considering compliance in the field of *cloud computing systems* (or short *clouds*). A PriceWaterhouseCoopers study from 2010 reveals that identifying compliance requirements is a significant challenge for compliance management in clouds.[1] As running example we use a software system for an online-banking service. For describing this example, we re-use an existing *pattern for analyzing clouds*, which is complemented by templates to elicit knowledge about the different stakeholders contained in the pattern ([3]). The pattern and especially the stakeholder templates are the basis for functional requirements engineering and the identification of activities, which serve as input for our law identification and analysis method.

## II. Background

### A. Cloud Computing Systems

According to the *National Institute of Standards and Technology (NIST)* cloud computing systems can be defined by the following properties [4]: the cloud customer can acquire resources of the cloud provider over *broad network access* and *on-demand* and pays only for the used capabilities. Resources, i.e., storage, processing, memory, network bandwidth, and virtual machines, are combined into a so-called *pool*. Thus, the resources can be virtually and dynamically assigned and reassigned to adjust the customers' variable load and to optimize the resource utilization for the provider. The virtualization causes a location independence: the customers generally have no control or knowledge over the exact location of the provided resources. The resources can be quickly scaled up and scaled down for customers and appear to be unlimited, which is called *rapid elasticity*. The pay-per-use model includes guarantees such as availability or security for resources via customized *Service Level Agreements (SLA)*. The architecture of a cloud computing system consists of different service layers and allows different business models: On the layer closest to the physical resources, the *Infrastructure as a Service (IaaS)*, are provided pure resources, for instance virtual machines,

[1] http://www.pwc.de/en/prozessoptimierung/trotz-einiger-bedenken-der-virtuellen-datenverarbeitung-gehoert-die-zukunft.jhtml

where customers can deploy arbitrary software including an operating system. Data storage interfaces provide the ability to access distributed databases on remote locations in the cloud. On the *Platform as a Service (PaaS)* layer, customers use an API to deploy their own applications using programming languages and tools supported by the provider. On the *Software as a Service (SaaS)* layer, customers use applications offered by the cloud provider that are running on the cloud infrastructure.

### B. Relevant Laws for Clouds

In our running example we chose the German law as the binding law. For simplicities sake, we focus in our running example on relevant compliance regulations for privacy. We only explain the laws and regulations in detail that we use in the example. In 1995, the European Union (EU) adopted the *Directive 95/46/EC* on the processing of personal data that represents the minimum privacy standards that have to be included in every national law. Germany implements the European Privacy Directive in the *Federal Data Protection Act (BDSG)*. According to *Section 1 BDSG* all private and public bodies that automatically process, store, and use personal data have to comply with the BDSG. Moreover, the EU law as well as *Section 4b BDSG* forbid sharing data with companies or governments in countries that have weaker privacy laws.

## III. A Requirements Elicitation Method for Clouds

We describe a requirements elicitations method specifically designed to take clouds and laws into account in Fig. 1. The arrows are annotated with inputs for the different steps. The oval symbol denotes steps in the method that require the presence of a legal expert, while the steps denoted in rectangles only require software engineers.

In the next sections we will describe the following steps in detail. The first step is to elicit a structured environment description. Therefore we instantiate a cloud system analysis pattern and the corresponding templates. The pattern and templates will be explained in Sect. III-A. The instantiated pattern and templates are the input for the identifying and specifying functional requirements. Next, the method advises to specify activities based upon the functional requirements and it also advises to identify and refine assets of stakeholders. The information collected until this point is accumulated in direct stakeholder template instances (Sect. III-B). In the fifth step we use the information in the instantiated direct stakeholder templates to identify relevant laws via using law analysis patterns (Sect. III-C). The resulting set of relevant laws for the cloud computing system and the instances of the direct stakeholder templates are used to identify and specify relevant requirements for the software (Sect. III-D). The result of the method is a set of

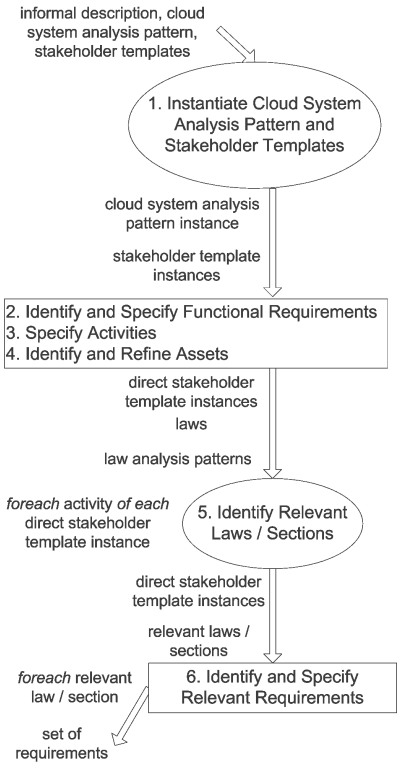


Figure 1. Requirements Analysis Method for Clouds

requirements derived from the legal analysis of the cloud-system-to-be.

### A. Step 1: Instantiate Cloud System Analysis Pattern and Stakeholder Templates

The starting point of our method is an informal problem description. The goal of this initial step is to collect and structure knowledge about the envisaged cloud, especially about the involved stakeholders. For this purpose, we introduced a *cloud system analysis pattern* and *stakeholder templates* in a previous work. in [3]. The pattern and the templates provide a conceptual view on cloud computing systems, and they serve to systematically analyze stakeholders and requirements.

This method is specific for cloud computing, because we use a cloud system analysis pattern. However, this analysis pattern can be replaced with a pattern for a different kind of Information and Communication (ICT) system, e.g., a Service Oriented Architecture (SOA). Hence, the method is adaptable to numerous ICT systems, because the application to a specific system is only determined by the input in the initial step.

### B. Steps 2–4: Process Functional Requirements, Activities, and Assets

In this section, we consider classical functional requirements engineering tasks based on the instantiated cloud

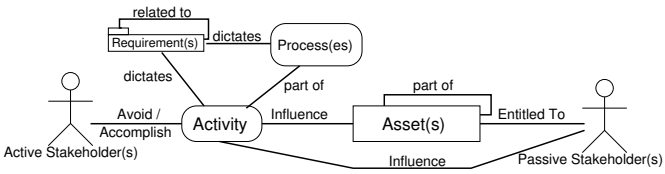


Figure 2. Relations between the elicited elements

system analysis pattern and the corresponding stakeholder templates. More specifically, we accomplish the following tasks for each stakeholder of the direct system environment.

First we identify the functional requirements of the system to be developed. Based on these requirements we derive the activities performed to fulfill the requirements. Last we identify the assets bound to the requirements and activities. For these steps we use existing approaches.

The instances of our cloud analysis patterns and templates are the basis to derive *functional requirements*. Here, given approaches such as the problem decomposition approach by [5] or an approach based on use cases can be applied.

The third step covers the discovery of activities based on functional requirements, e.g. using UML[2] behavior diagrams such as activity or sequence diagrams. This represents best practice in many object-oriented development methods such as the one by [6].

In the fourth step assets are identified based on functional requirements and corresponding activities. Again, we reuse given approaches here, e.g. the one by [7], which makes use of UML activity diagrams to identify assets and threats, or the *misuse case* approach by [8].

Important beside the functional requirements, activities, and assets themselves, are the relations between them and other already elicited information. Fig. 2 shows this relations. First of all, a *Requirement* can be related to other *Requirements* and dictates a certain behavior of the machine. A behavior 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 *Assets* which is entitled to the *Passive Stakeholder*. In addition *Assets* can be related to each other, e.g. one *Asset* is part of an other *Asset*. All these relations have also to be discovered and documented.

Using the results of the artifacts and relations generated in the previous steps, we extend the direct stakeholder template instances initially developed in the first step by new rows "Functional requirements", "Activities", and "Assets".

### C. Step 5: Identify Relevant Laws / Sections

The goal of this step is to bring together legal experts and software and system developers to identify relevant laws and to detect dependent laws. We execute this step for each activity of each instantiated direct stakeholder template. The output of this step are laws that are relevant for the given development problem. Note that each law identified as relevant should be checked by a legal expert.

[2]Unified Modelling Language: http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

Table I
STRUCTURE OF LAW RULES TAKEN FROM PREVIOUS WORK

| | | |
|---|---|---|
| **Addressee(s)** | has (have) to comply to the law. | |
| **Facts of the case** | **Activity(ies)** | describe(s) actions that an addressee has to follow or avoid to be compliant. |
| | **Target subject(s)*** | describes impersonal subjects that are objectives of the activity(ies). Subjects can be material, such as a product, or immaterial, such as information. |
| | **Target person(s)*** | are directly influenced by the activity(ies) of an addressee, or have a relation to the target subject(s). |
| **Legal consequence** | defines the consequence for an addressee, e.g. the punishment when violating the section. | |

A * next to an element of the structure means the element is optional.

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.

To bridge this gap we developed law patterns and a general process for law identification which relies on these patterns. We analyzed, how judges and lawyers are supposed to analyze a law, based upon legal literature research. These insights lead to a basic structure of laws and the contained sections. We give a short description of the results of this analysis in the following. The full discussion can be found in a previous work [2].

A law document is structured into *sections*. Each section defines a legal aspect of the law and contains several statements. These statements are *dictates of justice*. The basic elements of a dictate of justice are shown in Table I. A dictate of justice is divided into the *facts of the case*, the setting which is regulated, and the *legal consequence*, the resulting implications of the setting [9, p. 7]. Furthermore, a dictate of justice has also an *addressee(s)*. The legal method called *subsumption* contains a further refinement of the facts of the case [10, p. 260-64]. This refinement results in the basic elements *activities*, *target subjects*, and *target persons* [9, p. 23-31].

Dictates cannot be analyzed in isolation. All of them have relations to other dictates (or even laws). Thus, relations between laws, sections and dictates of justice are of fundamental importance. They are arranged in a hierarchy, which

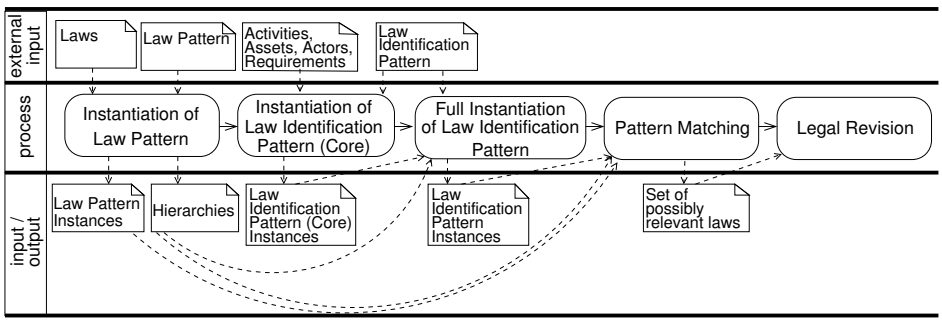


Figure 3. Law Identification Process

is not always free of conflicts [10, p. 255]. A special part of these relations is the terminology used within a jurisdiction. This terminology is organized as hierarchical tree where the terms and notions of the more general dictates of justices are refined by subsequent dictates of justice.

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 the gap of the wording and to facilitate the discussion between requirements engineers and legal experts, we defined a *law identification pattern* (discussed in detail in a previous work [2] ) to support identifying relevant laws based on the early steps of our method presented as already described. We especially use the laws captured with the law pattern presented and the knowledge collected using the stakeholder templates.

The procedure for identifying relevant laws consists of five steps as depicted in Fig. 3. The first step is to set up a database of all laws which might be of relevance for a scenario. Therefore, laws have to be analyzed and stored in the structure of the law pattern. Thus, they are stored as pattern instances. This step is not needed if such a database already exists. The second step uses information from software requirements and their context to instantiate the core structure and the context of the law identification pattern. Instances of the cloud analysis pattern contain parts of the relevant information for this instantiation. Third, the relation between laws and software requirements has to be established to prepare the identification of relevant laws for the given software. Hence, a mapping between the terms and notions of the software requirements to legal terms and notions is derived. Fourth, the law pattern instances and law identification pattern instances have to be matched. This results in a set of laws which might be of relevance for the software. The resulting laws are only possibly relevant, because we use the subsumption method. Fifth, the found laws are the basis for further investigations.

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 legal and technical world. The last step can be accomplished alone by legal experts.

### *D. Step 6: Identify / Specify Relevant Requirements*

The last step in the method is to integrate the instructions from the laws into a given software engineering process. We distinct between functional requirements and non-functional requirements. Functional requirements describe "what the system does" [11, p. 119] and non-functional requirements describe global requirements, e.g., reliability and maintainability on the system-to-be [11]. In our view security requirements are non-functional requirements. Software engineers refine these requirements into software specifications. These are implementable requirements.

The content of laws can be translated into functional and non-functional requirements or a restriction to a functional requirement or even be part of the software specification. For instance, the appendix to BDSG Section 9 demands specific methods, e.g., access control. These have to be part of the software specifications. While other laws shall be transformed into non-functional requirements, e.g., Section 17 TKG demands confidentiality of information. This would have to be transformed into a security requirement. Further laws simply demand a specific functionality that shall be transformed into a functional requirement. For example, the appendix to BDSG Section 9 also demands that the passing on of personal information has to be controlled. This leads to a functional requirement that states that all transmissions of personal information have to be documented. In addition, a law might provide different options to deal with a situation. These would have to be considered as well.

However, the restrictions that a law imposes on a system changes the envisioned software. The software engineer or his/her employer has to decide if the changed system ,or at least a functionality of it, is still useful. This might lead to the decision to not implement the system or functionality.

We have to distinguish between the different kinds of requirements or specification that instructions from laws in

order to achieve a seamless integration of the instructions from laws into a given software engineering process. Hence, we propose a method that can help decide how to translate the instructions from laws into requirements, when they are captured in law patterns. We assume that a significant number of demands from laws have to be translated into security requirements. That is the reason why we focus on these in our method.

According to [12], a security requirement is typically a confidentiality, integrity or availability requirement. It refers to a particular piece of information, the *asset*, that should be protected, and it indicates the *counter-stakeholder* against whom the requirement is directed. A *stakeholder* is an individual, a group, or an organization that has an interest in the system under construction. Furthermore, the *circumstances* of a security requirement describe application conditions of functionality, temporal, spatial aspects, or the social relationships between stakeholders. Hence, circumstances have relations to functional requirements, stakeholders, etc., which shall be considered in the system-to-be.

In oder to determine if the instruction from the law can be transformed, we propose to try to instantiate the instruction as a security requirement. We define preconditions for each of the steps of the instantiation and give advise of how to check if these preconditions are fulfilled. The method is iterative and if one precondition fails, the method terminates and the following preconditions are not checked anymore. If one of the preconditions fail, the instruction cannot be a security requirement. Afterwards we present a method for determining if the instruction is a further functional requirement or a technical measure that has to be integrated into the software specification.

For each instantiated requirement activity pattern matching at least one instance of a law paragraph pattern do:

Instantiate stakeholder

**Precondition:** The stakeholder has to be an stakeholder in the sense of security requirements.

**Determination:** Check if the stakeholder has an interest in the system under construction.

Describe the stakeholder and his/her interest to the system. Use the descriptions from the law patterns, cloud system analysis pattern and the templates. Consider the information in the instantiated law paragraph and requirement activity patterns the direct and indirect stakeholders can be distinguished. The addressee is a direct or an indirect stakeholder, and the target person is a direct stakeholder. The legislator and the domain are indirect stakeholders.

Instantiate asset

**Precondition:** The asset has to be an asset in the sense of security requirements and it has to be owned by the stakeholder.

**Determination:** Check if the asset is some piece of information or hardware or software of the stakeholder and if it should be protected with respect to confidentiality, integrity, or availability.

Describe the asset of the stakeholder and the protection requirements in terms of confidentiality, integrity, or availability.

Instantiate counter-stakeholder

**Precondition:** A counter-stakeholder has to exist.

**Determination:** Check if a counter-stakeholder exists, who threatens the confidentiality, integrity, or availability of the asset.

Describe the counter-stakeholder and the threat he/she presents to the asset in terms of confidentiality, integrity, or availability. In contrast to stakeholders, laws and therefore the instantiated law paragraph and requirement activity patterns do not define counter-stakeholders. These have to be derived using a method for thread analysis, e.g., misuse cases by [8].

Instantiate circumstance

**Precondition:** The circumstances of the security requirement have to be related to functionalities, stakeholders, or aspects of the system.

**Determination:** Check if stakeholder, asset, counter-stakeholder are related to the system-to-be.

Describe the relations the stakeholder, asset, counter-stakeholder have to existing functional requirements or to other stakeholders, assets, counter-stakeholders etc.. Typically, security requirements are considered in the context of functional requirements. Therefore, the functional requirement that is the source of the activity the instantiated requirement activity pattern refers to is the basis for specifying the security requirement.

When all the preconditions are true, the instruction from the law we were looking at can be translated into a security requirement. if this is not the case, we have to determine if the instruction in the law has to be translated into a functional requirement or has to be integrated into the specification of the software. We propose a method of exclusion. Hence, we have to decide if the law prescribes a mechanism or a requirement for the system-to-be. The difference is that a requirement just states a problem that the system shall address, e.g., the system has to store some information. In this case the software engineer has to find a mechanism that implements the requirement. On the other hand, if a law demands access control to certain types of data it is already a mechanism that solves the problem, e.g., that the data has to be kept confidential.

## IV. Example: Cloud Online Banking

We describe the application of our method using an online banking cloud example in this section.

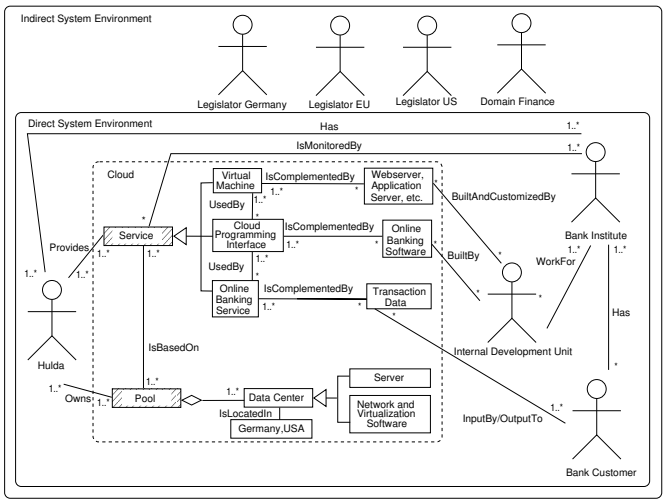


Figure 4. Concrete Cloud Computing System for Online Banking Service

### A. Instantiation of the Cloud System Analysis Pattern and Stakeholder Templates

We describe our method via an example of a cloud customer, a financial institute (bank) that wants to offer services like online banking for its customers. We instantiate the cloud analysis pattern presented in a previous work according to this running example. The pattern describes the cloud in a *Direct System Environment* and the *Indirect System Environment*. The *Direct System Environment* contains stakeholders and other systems that directly interact with the cloud through associations, e.g. the *Bank Customer*. Moreover, associations between stakeholders in the *Direct* and *Indirect System Environment* exist, but not between stakeholders in the *Indirect System Environment* and the cloud. For example, the *Legislator Germany* is part of the *Indirect System Environment*. We instantiate the direct system environment first.

The cloud provider is a company called Hulda. The main goal of the cloud provider is to maximize profit by maximizing the workload of the cloud. Therefore hers/his subgoals are to increase the number of customers and their usage of the cloud, i.e. the amount of data as well as the number and frequency of calculation activities they source out into the cloud. Fulfilling security requirements is only an indirect goal to acquire customers and convince them to increase the subset of processes they source out. The pool of the cloud is instantiated with Hulda's data centers that consist of servers, network, and virtualization software. The data center's are located in the Germany and the U.S.. The cloud customer is instantiated with the bank institute that plans to source out the affected IT processes to the cloud 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. We instantiate the cloud developer with an internal software development unit of the bank. This unit develops software for online banking in the cloud. In order to do this the internal development unit installs and configures web- and application servers in virtual machines. The resulting cloud programming interface is the foundation for the online banking software. The bank customer uses the cloud to conduct his/her financial business.

We derive the indirect stakeholders required for this scenario based upon the instantiation of the Direct System Environment. The cloud is located in Germany and the USA. This is the reason for the instantiation of the indirect stakeholders legislator Germany and US. Germany is a member of the European Union and the legislator EU describes a set these regulations. The financial institute is subject to regulations of the financial regulations. Hence, we instantiate the domain finance as another indirect stakeholder.

We supplement the 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. When instantiating the cloud system analysis pattern, one also fills in the corresponding templates. These templates contain for example the following information: a description of a stakeholder, a stakeholder's motivation for using the cloud, relations to the cloud and to other stakeholders, and the assets of a stakeholder. Detailed information about stakeholder templates are in a previous work.

### B. Processing of Functional Requirements

Right after describing the setting by instantiating pattern and templates for the cloud, we have to elicit the functional requirements, activities, and assets for this setting. For example, we can use the problem based approach of [5]. We derive the first requirements from the environment description, which is in our case build up by the cloud analysis pattern and its templates. Now we can establish the context diagram. By refining the context diagram to the problem diagrams we enrich existing functional requirements and discover related and previously unknown functional requirements. Possible examples for requirements are *Scalable Data Storing* and *Provide Cloud API*. The phenomena within the context and the problem diagrams can be transformed to activities, which we can arrange in processes using a activity diagram. An example for a phenomenon could be *Store Distributed*. This phenomenon can be directly transformed into the activity *Store Distributed*. There are more phenomena bound to the requirement *Scalable Data Storing*, which we arrange in the process *Offering Data Storing*. After defining the activities we discover the related assets. In our case the *Customer Data* is processed within the activity *Store Distributed*.

When all artifacts are known and the relations inbetween them are discovered, we have to document them. We do so

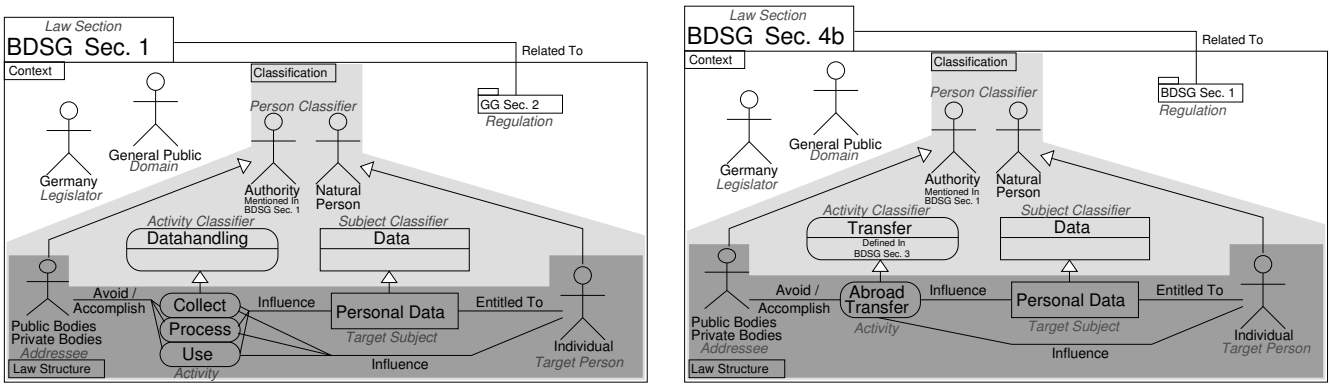


Figure 5. Law Pattern BDSG Sec. 1 (left) and BDSG Sec. 4b (right)

Table II
ACTIVE STAKEHOLDER TEMPLATE HULDA

Name Hulda

Functional Requirements

| Requirement | Related Requirement(s) |
|---|---|
| Scalable Data Storing | Provide Cloud API |

Activities

| Activity | Related Processes | Related Asset(s) | Related Requirement(s) | Related Stakeholders |
|---|---|---|---|---|
| Store Distributed | Offering Data Storing | Customer Data | Scalable Data Storing | Hulda |

Assets

| Assets | Related Asset(s) | Related Stakeholder(s) |
|---|---|---|
| Customer Data | Part of Financial Data | Bank Customer |

by adding the sections *Functional Requirements*, *Activities*, and *Assets* to the existing direct stakeholder templates. For example we add the following information to the template of Hulda in Tab. II (we only show artifacts we discussed in this section)

### C. Identifcation of Relevant Laws / Sections

After the elicitation of functional requirements, activities, and assets the desired behavior of the system to be implemented is clear. Now we have to ensure that this behavior is compliant. Therefore we have to identify relevant laws and sections. The overall procedure is described in Section III III-C.

#### Law Pattern Instantiation

Based on the previously discussed structure of laws, we defined a *law pattern*. The law pattern itself is discussed in detail in a previous work. For our running example we instantiate BDSG Sec. 1 and BDSG Sec. 4b. The resulting law pattern instances are shown in Fig. 5. The light grey words near to a element of in an instance refer to the type of the element in the original pattern. For example in Fig. 5 on the left, the light grey words *Activity Classifier* near to *Datahandling* indicate that the *Datahandling* element is an instantiation of *Activity Classifier* element in the original pattern.

We now describe the instantiation procedure for our law pattern. Our process 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*. Note that the analysis and instantiation of a law and its sections has to be done only the first time a law is related to a software project. For the next projects the instantiation of a law is reusable.

In our case the first section to be analyzed is BDSG Sec. 1. We start with the *Context Part* (depicted as white area in Fig. 5 on the left hand). The *Legislator(s)* and *Domain(s)* can be instantiated according to the considered law (e.g. *Germany* and *General Public* in the *Context* part.). *Legislator(s)* and *Domain(s)* are always defined for a whole law. Thus they are same in all law pattern instances of a law. Given a section of a law not yet captured by our law pattern, we additionally identify and document the related laws and sections referred to by the given section (e.g. *GG Sec. 2* in the *Context* part. GG Sec. 2 is a fundamental dictate of justice, which defines the right of informational self-determination.). Then, we search for the *Law Structure* (depicted as dark grey area in Fig. 5) directly defined in the section at hand. For Sec. 1 BDSG, we find *Collect, Process and Use*, and we use it to instantiate *Activity(ies)*. We also find *Public and Private Bodies* for the *Addressee(s)*, *Personal Data* for the *Target Subject(s)*, and *Individual* for the *Target Person(s)*.

At last we instantiate the *Classification* part (light grey area in Fig. 5 on the left hand). When classifying *Addressee*, *Activity*, *Target Subject*, and *Target Person*, we parallel build up hierarchies for *Person Classifier*, *Activity Classifier*, and *Subject Classifier*. An example for parts of those hierarchies is shown in Fig. 6. In the case of Sec. 1 BDSG, these hierarchies are empty. So we add *Datahandling*, *Data*, *Authority* and *Natural Person* to the corresponding hierarchies.

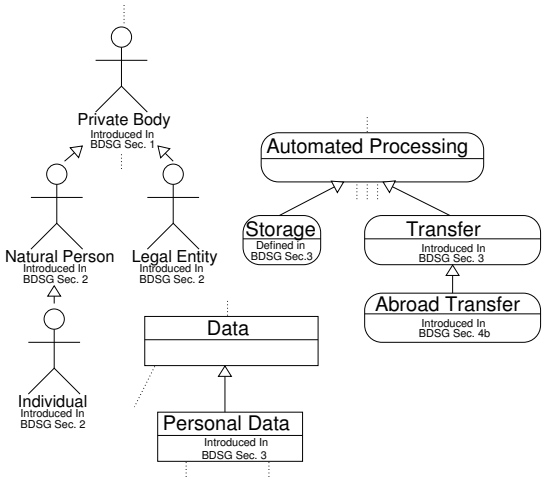


Figure 6. Hierarchies for Person (bottom), Subject (upper right), and Activity (upper left)

As elements of the *Law Structure* part can be classifier for subsequent sections, we also add these elements to the hierarchies, e.g. *Private Bodies*. Note that adding a new element for a hierarchy can result in reorganizing the corresponding hierarchy. For example, when adding *Natural Person* it is a specialization of *Authority*. Hence we add a direct inheritance relation from *Natural Person* to *Authority*. But when adding *Private Bodies*, *Natural Person* is moved down in the hierarchy by removing its direct inheritance relation to *Authority* and adding a direct inheritance relation to *Private Bodies*. The classifiers in the *Classification* part of a law pattern instance are always the parent elements of the *Law Structure* elements in the hierarchies.

The next sections BDSG Sec. 2 and BDSG Sec. 3 are definition sections, which we use to update our hierarchies (partly shown in Fig. 6). For example, BDSG Sec. 3 adds *Transfer* and *Storage* as specialization for *Process* to the activity hierarchy as shown in Fig. 6.

Next we instantiate BDSG Sec. 4b as an example for an incomplete dictate of justice. Given this section 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 (white area in Fig. 5 on the right hand). Then, we search for the *Law Structure* (dark grey part in Fig. 5 on the right hand) directly defined in this section. In Sec. 4b BDSG, we find *Abroad Transfer*, and we use it to instantiate *Activity*. *Addressee*, *Target Subject*, and *Target Person* are not defined in Sec. 4b BDSG. Therefore, related sections defining these terms have to be discovered. In our example, we find *Public and Private Bodies* for the *Addressee*, *Personal Data* for the *Target Subject*, and *Individual* for the *Target Person* in Sec. 1 BDSG (according to *BDSG Sec. 1* in the *Context* part). At last we classify the elements of *Law Structure*. We arrange the elements, which occur for the first time, in the appropriate parts of the hierarchies in Fig. 6. Than a classifier of the *Classification* part (light grey are in Fig. 5 on the right hand) is instantiated with the parent node of the corresponding hierarchy, which is for instance *Transfer*, defined in Sec. 3 BDSG, for *Abroad Transfer*.

*Instantiation of Law Identification Pattern (Core)* For identifying laws relevant for the online banking service example, we consider the requirements, activities and assets, documented in the direct stakeholder template instances of the instantiated cloud system analysis pattern as described in Sect. IV-B. As our example Fig. 7 shows, we select the template instance of the direct stakeholder *Hulda*, then we choose the functional requirement *Scalable Data Storing* (row "Functional Requirements" in the *Hulda* stakeholder template instance). One of the activities associated with this requirement is the activity *Store Distributed* (row "Activities" in the *Hulda* stakeholder template instance), which refers to the asset *Customer Data* (row "Assets" in the *Hulda* stakeholder template instance) of the *Bank Customer*. Moreover, we instantiate the elements *Legislator(s)* and *Domain(s)* according to the instantiated cloud system analysis pattern. In our example Fig. 7, 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. So far, the instantiation process can be performed by a software engineer.

*Full Instantiation of Law Identification Pattern* 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 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.

*Pattern Matching* The identification of relevant laws is based on matching the classification part of the law identification pattern instance (light gray part) with the law

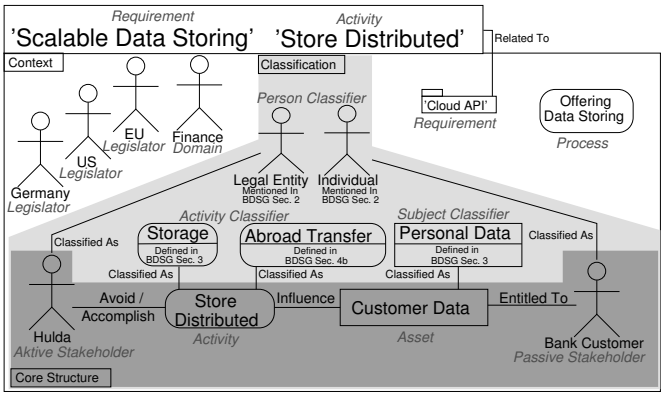


Figure 7. Law Identification Pattern Instance

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 depicted in Fig. 5 on the right hand for the elements *Abroad Transfer*, *Personal Data*, and *Individual* contained in the law identification pattern instance shown in Fig. 7. *Hulda* is classified as *Legal Entity* and the only element that does not directly match with *Public or Private Bodies* in the law structure of Section 4b BDSG. In this case, the hierarchy in Fig. 6 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. 5 on the right hand can be found in *Context* of the law identification pattern instance shown in Fig. 7. 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 step 6 of our requirements elicitation method for clouds presented in Sect. III.

### *D. Identifcation / Specification of Relevant Requirements*

In Sect. IV-D we describe the method of how to bridge the gap between identifying relevant laws for a cloud computing software system and the software development process. We execute the process in the following for the Law Identification Pattern Instance presented in Fig.7 and the previous section. We analyze if the law can be also translated into a security requirement from the point of few from the stakeholder Hulda.

Instantiate stakeholder
: Hulda is a stakeholder of the cloud online banking system, because he/she accomplishes distributed storage in the cloud computing system.

Instantiate asset
: The customer data is not an asset in the sense of a security requirement to Hulda, because he/she stores data for a fee in the cloud and the integrity, confidentiality and availability of the data is not of primary concern to the stakeholder. The reason is that he/she is not the owner of the data. These demands could be part of an contractual obligation of the stakeholder. However, we do not consider these here. The restricted activity in question is the abroad transfer of the the distributed storage. The method terminates at this step, because the asset is not an asset in the sense of a security requirement.

The law identification pattern instance cannot be translated into a security requirement. The next part of the method is to decide if the abroad transfer can be translated into a functional requirement or has to be integrated into the software specification. Abroad transfer is not a specific mechanism. It is rather a restriction on a functional requirement that the cloud software system should store data distributed. The requirement has to be modified into: The cloud software system should store data distributed, but the distribution has to be restricted to European Union. At this point the software engineer or his/her employer has to decide if the system-to-be is still useful with this restriction.

We also investigate a further part of the law identification pattern instance in Fig. 7. We analyze via our method if the abroad transfer for the bank customer is translated into a security requirement, because we can list one example for each of the required instantiations.

Instantiate stakeholder
: The bank customer is a stakeholder of the cloud online banking system, because the bank customer uses the online banking software in the cloud.

Instantiate asset
: The cloud costumer transfers data to the cloud and the cloud stores the data distributed. This data is an asset to the bank customer and he/she want the data to be confidential from, e.g., other users of the online bank. The bank customer also wants the data to be available and he/she also wants its integrity preserved.

Instantiate counter-stakeholder
: A possible counter-stakeholder are the other customers of the cloud banking system. He/she could get access to the data of the cloud customer, because they both have access to the cloud data storage. Hence, the confidentiality of the data can be violated.

Instantiate circumstance
: The store distributed functional requirement is related to the bank customer's data stored in the cloud.

For simplicity's sake we do not illustration the entire security requirement including all counter-stakeholders and circumstances. These security requirements are not the primary intend of the law. However, eliciting these requirements at this point of the software development is useful for creating a trustworthy cloud system.

## V. Related Work

Breaux et al. [13] present a framework that covers analyzing the structure of laws using a natural language pattern. This pattern helps to translate laws into a more structured restricted natural language and then into a first order logic. In contrast to our work, the authors of those approaches assume that the relevant laws are already known and thus do not support identifying legal texts. Their approach does not allow one to find dependent law sections. The approach

also has the drawbacks of formal logic analysis of laws, which will be discussed in Sect. VI.

Siena et al. describe in one publication [14] the differences between legal concepts and requirements. They model the regulations using an ontology, which is quite similar to the natural language patterns described in the approaches mentioned before. So the resulting process to align legal concepts to requirements and the given concepts are quite high level and cannot directly applied to a scenario.

## VI. Conclusions and Future Work

We presented a *an integrated method for pattern-based elicitation of legal requirements*, which is applied to a cloud computing online banking example.

The novelty about our method is that several pattern based approaches are re-used to support the structured elicitation of knowledge of a software system and its environment, to use the aquired knowledge to identify and analyze relevant laws, and finally to refine these into a set of software engineering requirements. So far approach only cover a part of the path from the software description to the requirements.

Our method comprises the main benefits:

- Re-using a systematic pattern-based identification and analysis of laws and the detection of dependent laws
- Operationalize the gathered knowledge into software requirements
- Re-using cloud-specific context and stakeholder analysis based on patterns
- Ease the burden of interdisciplinary work between legal experts and software engineers

In the future we will approach the validation of our approach in different ways. We are seeking contact to experts in the area of law and computer science in order in all ways. The first research question we want to address is if the approach arrives at the same conclusion as a court of law for a given cloud-computing-based case. We will instantiate patterns for the information given in this case and check if we arrive at the same conclusion the court did.

The second research question we want to address is if the method is valid for numerous given requirements engineering approaches. We aim to use different requirements engineering approaches within the method for a given cloud computing system and compare the compatibility with our patterns and derive needed interface or improvements for our patterns to fit the approach for numerous requirements engineering approaches.

We are also planning to address a third research question, that inqures if the pattern matching in our approach is sufficient to find dependent laws. In order to address this question we would need to instantiate at least two complete laws that have multiple dependencies.

We also aim to work on tool support for our approach, e.g. to store, load, and search for laws once they have been fitted to our law patterns.


## Acknowledgment

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).


## References


[1] C. Biagioli, P. Mariani, and D. Tiscornia, "Esplex: A rule and conceptual model for representing statutes," in *ICAIL*. ACM, 1987, pp. 240–251.

[2] K. Beckers, S. Faßbender, J.-C. Küster, and H. Schmidt, "A pattern-based method for identifying and analyzing laws," in *Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, ser. LNCS. Springer, 2012, pp. 256–262.

[3] K. Beckers, J.-C. Küster, S. Faßbender, and H. Schmidt, "Pattern-based support for context establishment and asset identification of the ISO 27000 in the field of cloud computing," in *ARES*. IEEE Computer Society, 2011, pp. 327–333.

[4] P. Mell and T. Grance, "The NIST definition of cloud computing," Working Paper of the National Institute of Standards and Technology (NIST), 2009.

[5] M. Jackson, *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[6] J. Cheesman and J. Daniels, *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[7] E. B. Fernandez, D. L. la Red M., J. Forneron, V. E. Uribe, and G. Rodriguez G., "A secure analysis pattern for handling legal cases," in *Latin America Conference on Pattern Languages of Programming (SugarLoafPLoP)*, 2007, http://sugarloafplop.dsc.upe.br/wwD.zip.

[8] G. Sindre and A. L. Opdahl, "Capturing security requirements through misuse cases," in *Proceedings of the Norwegian Informatics Conference (NIK)*, 2001.

[9] G. Beaucamp and L. Treder, *Methoden und Techniken der Rechtsanwendung*, 2nd ed. C.F.Müller, 2011.

[10] K. Larenz, *Methodenlehre der Rechtswissenschaft*, 5th ed. Springer, 1983.

[11] I. Summerville, *Software Engineering*, 8th ed. Addison-Wesley, 2007.

[12] B. Fabian, S. Gürses, M. Heisel, T. Santen, and H. Schmidt, "A comparison of security requirements engineering methods," *Requirements Engineering – Special Issue on Security Requirements Engineering*, vol. 15, no. 1, pp. 7–40, 2010.

[13] T. D. Breaux and A. I. Antón, "Analyzing regulatory rules for privacy and security requirements," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 5–20, 2008.

[14] A. Siena, A. Perini, and A. Susi, "From laws to requirements," in *RELAW*. IEEE Computer Society, 2008, pp. 6–10.