Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Semantic Querying of Distributed Pattern and Solution Repositories

Christoph Krieger

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Dr. h.c. Frank Leymann

Supervisor: Michael Falkenthal, M.Sc.

Commenced: December 4, 2017

Completed: June 4, 2018

Abstract

In 1977 Christopher Alexander and his colleagues published their pioneering idea about pattern languages consisting of linked patterns, which are linked human-readable documents that describe proven solutions in a context. Since then patterns and pattern languages have been established in various disciplines, constantly new patterns are created and added to pattern languages. Hence, pattern languages the can be seen as living network of patterns. However contrary to the idea of pattern languages as living networks, most of the pattern languages that exist today are documented in static documents such as books, papers or journals that are hard to change. Moreover, with increasing amount of printed documents, it becomes difficult for pattern users to manual select patterns that suits their use case.

We propose a concept of using Semantic Web Standards to describe information about patterns, concrete solutions and their relations in form of linked data on the web. This linked data is a machine-readable representation of pattern and solution languages that can be shared across software systems. Further, we present the concept of an IT-based pattern and solution repository that abstracts the technologies of the semantic web and allows users to publish information about patterns, concrete solutions and their relations in form of RDF data. Furthermore, the pattern repository provides functionalities to retrieve RDF data that describes information about patterns, concrete solutions and their relations from distributed source and visualize the information in human readable documents.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

This chapter gives an introduction for the thesis. First it presents the general background of the thesis. Second it clarifies the motivation of the thesis and states the problem relevance. Subsequent it defines the different research goals of the thesis. Finally it describes the structure of the thesis.

1.1 General Background

In 1977 Christopher Alexander and his colleagues [Ale77] published their pioneering idea of a pattern language consisting of linked patterns, which are human readable documents that describe proven solutions to recurring problems in a context. Alexander et al. define the characteristics of a pattern as follows: "Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution." [Ale79]. Alexander describes the structure of a pattern language as a network of patterns. This means individual patterns are not isolated, instead they are connected to other patterns which solve problems that often occur in the same context as the original problem. Those interrelations support navigation through the pattern language to collect patterns which are typically used in combination [RS96; Zdu07].

While the pattern language by Alexander et al. focuses on the domain of architecture and urban design, patterns and pattern languages have been established in various disciplines as means to capture proven solutions for frequently recurring problems. Today they can be found in disciplines such as education, emergency situations, and communication techniques. Iba et al. [IM10] documented a set of learning patterns that can be applied by learners to support their learning. Furukawazono et al. [FSMI13] present a pattern language that provide survival guidance in the situation of a catastrophic earthquake. Schhuler et al. [Sch08] present a pattern language that captures knowledge about information and communication techniques to address social and environmental problems collaboratively.

Pattern languages can also be found in more technically disciplines like information technology. The Cloud Computing Patterns by Fehling et al. [FLR+14] capture knowledge and experience in the domain of cloud computing by describing the fundamentals to design, build and manage cloud applications. Hohpe et al. [HW04] documented a pattern language providing guidance to design and document enterprise integration solutions.

Alexander describes pattern languages as a living network of patterns. This refers to the fact that pattern languages typically grow over time as people constantly create new patterns to document their knowledge. Hence, a pattern language is not a static artifact, instead

it is subject to a collaborative process that constantly change the structure of the pattern language. In this ongoing collaborative process of documenting new patterns, also the interrelations between patterns are under a constant change. Pattern languages which have primary been authored isolated from each other may converge over new relations between patterns that are part of the two pattern languages [FBL]. For example, the Remoting Patterns by Völter et al. [VKZ13] has many relations to patterns and pattern languages from related domains such as networking, concurrency and resource management. They aim to act as connector between these other languages when applied to distributed object middleware or distributed application development [ZKV04].

1.2 Motivation and Problem Relevance

Contrary to the idea of pattern languages as living networks, most of the pattern languages that exist today are documented in static documents such as books, papers or journals. Falkenthal et al. [FBL] criticizes that static documents can not sufficiently represent the liveness of pattern languages. Instead they provide static snapshots of knowledge to a certain point in time. Furthermore, Rosengard et al. [RU04] argues, that as the amount of printed documents increases, it becomes difficult for it to be effectively used. Hence, IT-based tool support for automatic organization and retrieval of patterns is needed.

To make pattern documents less static, multiple web-pages and online pattern repositories emerged over the last few years. For example, the web-page that hosts the Cloud Computing Patterns by Fehling et al. [Feh17], or the Pattern Library for Interaction Design [Van08]. However most have different pattern formats as well as different underlying schemas for data representation, which leads to isolated data silos that are not accessible to other software systems. Therefor, it needs a concept for common data formats and exchange protocols to make pattern repositories accessible to computer agents that allow automatic organization and retrieval of patterns.

Additionally, most of the existing online pattern repositories are used to publish the contained patterns and do not provide functionalities to author and publish new pattern languages[FBFL14]. As a result, pattern authors need to create their own web-page or online pattern repository to publish their pattern language. However, maintaining a web-page or online pattern repository requires advanced knowledge in information technology. Whereas advanced concepts and technologies of computer science are common in the domain of natural science and engineering [HTT+09], pattern authors and users from non-technical disciplines may be familiar with digital technologies like email and bibliographic databases, but they are rarely familiar with advanced concepts and techniques of computer science.

1.3 Document Structure

The remaining thesis is structured as followed: Fundamental concepts and technology standards that build the foundation of the thesis are presented in Chapter 2. The current research state is presented in Chapter 3. Chapter 4 discusses use cases and requirements that an IT-based system for pattern languages has to fulfill. The concept for an ontological representation of pattern languages and solution languages is presented in Chapter 5. Chapter 6 presents the architecture of an IT-based system for pattern languages and solution languages that leverages Semantic Web technologies to store information about pattern languages and solution languages in common data formats that can be exchanged by applications. The implementation details of a prototype that shows the the feasibility of the concepts are presented in Chapter 7. Chapter 8 completes the thesis with a conclusion and gives an outlook about challenges and improvement potential for future work.

2 Fundamentals

This chapter presents the fundamental concepts and technology standards that build the foundation of this thesis.

2.1 Patterns and Pattern Languages

This thesis uses Christopher Alexander‘s definition of patterns and pattern languages [Ale77; Ale79]. Alexander writes: "Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution" [Ale79]. Subsequent he defines that "As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves. As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant" [Ale79]. This means a pattern is both, a statement in a pattern language and an element that exist in the world. Depending on the discipline those elements may be spatial configurations as described by Alexander, but can also be other things. For example, in the domain of information technology, the elements that it exists in the world could be configurations in program code. In the domain of costumes in films, the elements may be costumes in a wardrobe composed of different articles of clothing.

A pattern language as described by Alexander has a structure of a network. This can be ascribed to the fact that patterns are not isolated solutions to a particular problem, but are interrelated to other patterns. Alexander writes, "the structure of a pattern language is created by the fact that individual patterns are not isolated". Furthermore he states that "the language lives, or not, as a totality, to the degree these patterns form a whole" [Ale79]. Thereby pattern languages can provide a means to combine patterns and to guide users to an overall solution.

2.2 Concrete Solutions and Solution Languages

Patterns abstract and generalizes solution knowledge in an technology and implementation agnostic way, which has the advantage that patterns are applicable for many different use cases. However, as a consequence of the abstraction, the solutions provided by patterns can mostly not be applied directly to specific use cases. Pattern users often need to

spend immense manual efforts in order to create a use case specific implementation of the abstract solution knowledge provided by the pattern. For example, the Gang of Four patterns by Gamma et al.[Gam95] describe various software design patterns for object-oriented programming. However, if for example a Java developer uses those patterns, he needs to transfer the general solution principles described by the patterns to be applicable to his concrete use case, i.e., he has to implement solutions based on the Java programming language. A software architect who has to design a cloud application architecture may use the cloud computing patterns by Fehling et al. [FLR+14] to gain insights about provider supplied cloud offerings, but these patterns are again generic solutions that are independent of concrete vendor products and do therefore not provide provider specific implementation details. To provide a means to reuse existing concrete implementations of the abstract solution knowledge presented by patterns, Falkenthal et al. [FL17] present the concept of *Concrete Solutions*. Concrete solutions are linked to patterns and provide use-case specific implementations of the abstractly described solution documented by the pattern. Moreover, they propose the concept of Solution Languages that structure concrete solutions analogously to pattern languages organize patterns. Thereby the provide user navigation guidance through the set concrete solutions linked to patterns in a pattern language [FL17].

2.3 Semantic Web

The term Semantic Web was coined by Tim Berners-Lee, for an extension of the world wide web, "in which information is given well-defined meaning, better enabling computers and people to work in cooperation" [BHL01]. The World Wide Web Consortium (W3C) provides a stack of technology standards that help to build the Semantic Web. The technology standards of the Semantic Web provide a framework to implement a machine-readable *Web of data* [Con15b]. This section describes the parts of the Semantic Web Standards that are relevant for this thesis, in more detail.

2.3.1 Resource Description Framework (RDF)

The Semantic Web builds on the Resource Description Framework (RDF), which provides a simple data model for publishing and linking data on the web [Con15b]. The intention of RDF is to provide open and interchangeable data format for information of resources on the web, that can be automatically processed by software agents [Con04].

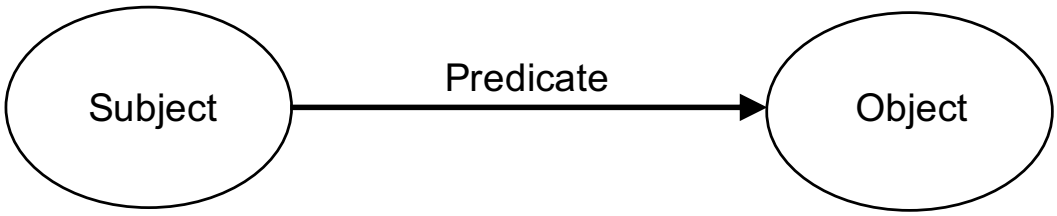

Figure 2.1: Illustration of an RDF triple as node-arc-node link

RDF aims to meet the following goals:

- a simple data model that is easy to process and to manipulate
- formal semantics that provides a reliable basis for arguing the meaning of an RDF expression
- an IRI based vocabulary to define semantics of RDF expressions that is fully extensible
- allow anyone to make statements about any resources that be referenced by an IRI

RDF represents data as a set of triples. Each triple consists of a subject, a predicate, and an object. The subject is an IRI reference or a blank node. The predicate is an IRI reference. The object is an IRI reference, a literal or a blank node. Together they form a node-arc-node link as illustrate in Figure 2.1.

RDF triples allow describing relationships between resources. Thereby the predicate indicates the relationship and the two resources are indicated by the subject and object. A set of such triples forms webs of information about related things, a so-called RDF Graph.

RDF uses IRI references to identify and add semantic meaning to resources and predicates. The IRIs used to encode information ensure that the information about resources and predicates is tied to a unique definition, that everyone can find on the web [BHL01]. Thus, the set of triples that builds an RDF graph can be distributed in different documents on the web and can still be uniquely identified by their IRI. RDF provides a basic IRI based vocabulary that provides semantic meanings for certain IRI references. This vocabulary can easily be extended which allows anyone to define new semantic terms.

An exemplary set of RDF triples encoded in RDF/XML Syntax is shown in Listing 1. There, the terms of the RDF vocabulary defined by the namespace *rdf* are used to identify resources. Further terms of the FOAF vocabulary [Dan14] defined by the namespace *foaf* are used to encode the predicates of the RDF triples with semantic meaning. The resulting RDF Graph describing information about a Person is depicted in Figure 2.2. There, subject and object nodes containing an IRI are represented as ovals, all the predicate arcs are labeled with their IRIs and string literal nodes are represented as rectangles. There, the subject identified by the IRI *https://example.com/MaxMustermann* is related via a predicate

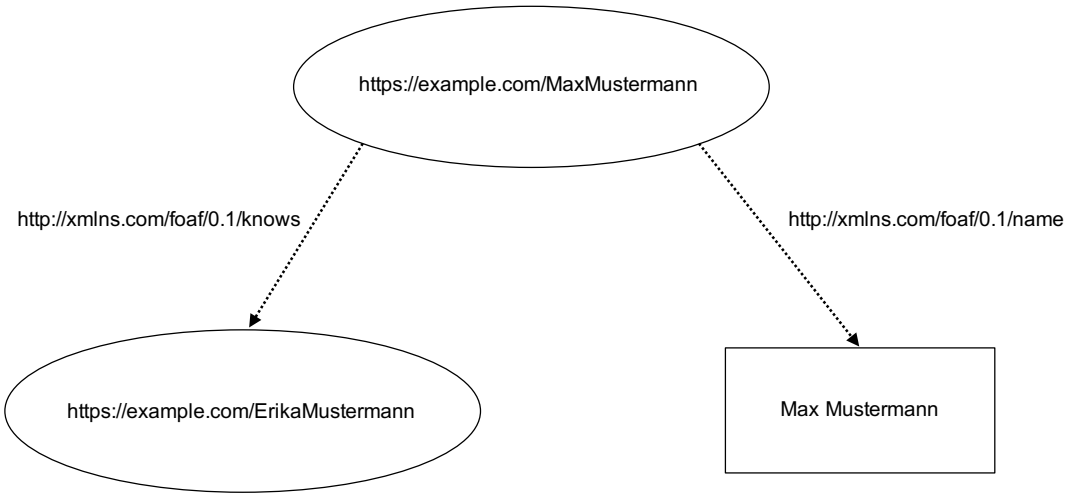

Figure 2.2: A visualization of the RDF Graph described by Listing 1

that is identified by the IRI *http://xmlns.com/foaf/0.1/knows* to an object identified by the IRI *https://example.com/ErikMustermann*. Furthermore, the subject identified by the IRI *https://example.com/MaxMustermann* is related via a predicate that is identified by the IRI *http://xmlns.com/foaf/0.1/name* to an object which is a literal with the value *Max Mustermann*

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:foaf="http://xmlns.com/foaf/0.1/">

 <rdf:Description rdf:about="https://example.com/MaxMustermann">
   <foaf:name>Max Mustermann</foaf:name>
   <foaf:knows rdf:resource="https://example.com/ErikaMustermann"/>
 </rdf:Description>

</rdf:RDF>
```

Listing 1: A set of RDF triples described in RDF/XML syntax

2.3.2 Resource Description Framework Schema (RDFS)

RDF allows expressing knowledge by publishing and linking data. However, different data sets may use different identifiers for the same thing [BHL01]. For example, one dataset may use the identifier *author* and the other *creator*. A program needs to know that these two identifiers actually mean the same. The semantic web provides ontologies to solve this problem. Ontologies are also RDF documents and define concepts and relationships, also called terms, that are used to describe and represent an area of concern. Ontologies provide

a means to classify the terms and characterize possible relationships. Moreover, constraints on using the terms can be defined. In the domain of the Semantic Web, ontologies are also referred as vocabularies. Although there exist no strict distinction between those terms, the word ontology is mostly used for more complex collection of terms [Con15c]. The Semantic Web Standards provide the Resource Description Framework Schema (RDFS) and the Web Ontology Language (OWL) that provides formal semantics to create vocabularies.

The Resource Description Framework Schema (RDFS) provides a data-modeling vocabulary for RDF data and is an extension of the RDF vocabulary [Bri14]. RDFS is a semantic extension of RDF that provides capabilities of describing groups of resources, and relations between those resources. Groups of resources are defined by means of classes. The members of a class are called instances, or individuals. The set of instances that are a member of a class are called the class extension of the class [Bri14]. A class can be defined by defining an instance of *rdfs:class* which itself is a class. In addition, a hierarchy or also called a taxonomy of the classes can be defined by the taxonomic predicate rdfs:subClassOf, which relates a more specific class to a more general class. If a class C_1 is a subclass of a class C_2, then every individual of C_1 is also an individual of C_2. Moreover the rdfs:subClassOf relation is transitive which means if C_1 is a subclass of C_2 and C_2 is a subclass of C_3, then C_1 is also a subclass of C_3 [WMS04]. An example of defining classes and subclasses in RDFS is shown in Listing 2. There, the classes Human, Adult and Child are defined by defining them as instances of *rdfs:Class*. In addition the *rdfs:subClassOf* predicate indicates that the classes Adult and Child are subclasses of the class Human.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:ID="Human"/>

<rdfs:Class rdf:ID="Adult">
  <rdfs:subClassOf rdf:resource="#Human"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Child">
  <rdfs:subClassOf rdf:resource="#Human"/>
</rdfs:Class>

</rdf:RDF>
```

Listing 2: Definition of a RDFS class Human with two subclasses Child and Adult

In RDF, properties describe a relation between a subject resource and an object resource. Similar to the concept of classes and subclasses, RDFS defines the construct of *rdfs:subproperty* to create hierarchies between properties. The *rdfs:subproperty* can be used to describe, that one property is a subproperty of another. If a property P_1 is a subproperty of a property P_2, then all pairs of resources that are related by P_2 are also related by P_1 [Bri14].

The characteristics *domain* and *range* defines that the property links instances belonging to the *domain* to instances belonging to the *range*. Based on this concept, a machine reader could validate if a property is used in a correct context. For example, if a machine reader detects that a property is used to link instances that are not a member of the class extension defined by the domain or range of the property, the machine reader could either ignore the property or mark it with an error. An example of defining a property with domain and range in RDFS is shown in Listing 3. There, the property hasParent is defined. The domain and range define that the property hasParent links instances of the class Child to instances of the class Adult.

```
<rdf:RDF xml:lang="en"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#">

<rdf:Description ID="hasParent">
  <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <rdfs:domain rdf:resource="#Child"/>
  <rdfs:range rdf:resource="#Adult"/>
</rdf:Description>

</rdf:RDF>
```

Listing 3: Definition of a RDF property hasParent with domain Child and range Adult

2.3.3 Web Ontology Language (OWL)

RDFS provides basic semantics to define a machine-readable taxonomy of classes and properties that can be used to describe resources and relations between those resources. However, if machines need to perform useful reasoning task it requires a language that provides formal semantics that goes beyond the basic semantics describe by RDFS [MH04]. The Web Ontology Language (OWL) is an extension of RDFS and has been designed to meet those needs. OWL provides an additional vocabulary with a formal semantics that can be used to represent things, group of things in vocabularies and describe relations between those things. OWL has more ways of expressing meaning and semantics than RDF and RDFS, and therefore goes beyond these languages in its ability to present machine interpretable content on the web [MH04]. For reasons of the number of semantic constructs provided by OWL, this thesis will not discuss all of them in detail. However, this section will provide an overview of the important formal semantics provided by OWL.

OWL extends the vocabulary of RDFS for describing classes and properties. The vocabulary provides semantics to describe classes and their relations in more detail. For example, describing additional property restrictions on classes, in form of cardinality constraints or value constraints. Cardinality constraints describe a constraint on the number of values a property can take. For example, a cardinality constraint could express that for a tennis game the *hasPlayer* property has the cardinality 2. For a football game, the same property would have 22 values. A value constraint defines constraints on the range of a property, in

case the property is applied to that specific class description. Furthermore, OWL provides semantics to describe classes with more advanced class constructors such as intersection, union, and complement [MH04].

OWL also extend the formal semantic to describe properties and their relations. OWL distinguish between two main categories of properties, *object properties* that link individuals to individuals and *datatype properties* that link individuals to data values. Relations to other properties can be defined with concepts such as owl:equivalentProperty that can state that two properties have the same property extension, or *owl:inverseOf* that can be used to define inverse relations between properties. Listing 4 shows an exemplary usage of the *owl:inverseOf* construct. There the object properties hasParent and hasChild are defined. The *owl:inverseOf* construct attached to the hasChild property defines that the hasChild is an inverse of hasParent. Moreover, OWL provides semantics to define cardinality constraints on properties and to define logical characteristics such as symmetry or transitivity.

```
<owl:ObjectProperty rdf:ID="hasParent">
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasChild">
 <owl:inverseOf rdf:resource="#hasParent"/>
</owl:ObjectProperty>
```

Listing 4: The owl:inverseOf construct defines an inverse relation between the properties hasChild and hasParent

2.3.4 SPARQL

In the Semantic Web, the data is stored as RDF files. Just as other databases need a specific query language, the semantic web needs its own RDF-specific query language. The triples of RDF data are queried using SPARQL Query Language for RDF (SPARQL). SPARQL is a W3C standard, that allows to send queries and receive results through different protocols like the Hypertext Transfer Protocol (HTTP) [Con15a].

A SPARQL query contains a set of triple patterns, which is called a *basic graph pattern*. A triple pattern is like an RDF triple except that the subject, predicate, and object may be a variable. The result of the query would be the subgraph of an RDF graph that matches the basic graph pattern [PS+06]. The general form of a SPARQL query to retrieve data is can be defined by the following statements:

- **PREFIX**, which defines namespaces used in the query.
- **SELECT**, which defines the variables that should appear in the result.
- **FROM**, which defines the endpoint of the data graph.
- **WHERE**, which contains the basic triple pattern.

- Modifiers, like **ORDER BY**, **DISTINCT** etc.

Listing 5 shows an example of a SPARQL query to retrieve the names of all persons from a given RDF graph. There the PREFIX statement defines the namespace *foaf*. The SELECT statement defines that the variable ?name should appear in the result. The FROM statement defines *http://www.w3.org/People/Berners-Lee/card* as the endpoint of the data graph. The WHERE state contains the basic graph pattern. The basic graph pattern in this example consists of a single triple pattern with a variable (?person) in the subject position and a variable (?name) in the object position.

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?name
FROM <http://www.w3.org/People/Berners-Lee/card>
WHERE
{
  ?person foaf:name ?name .
}
```

Listing 5: A SPARQL query that defines a graph pattern consisting of a single triple pattern

3 Related Work

This chapter presents related work in the field of pattern language formalizations and pattern language repositories that builds the foundation for this thesis.

In 1977 Christopher Alexander and his colleagues presented the pioneering concept of a pattern language, which was born in the domain of architecture and urban design. They described pattern languages as a network of linked pattern documents, where each pattern describes a common problem and gives a proven solution [Ale77]. The structural organization of the pattern language by Alexander et al. connects patterns that describe the large structures like cities to more fine-grained structures like streets and houses. They show that pattern languages are a powerful mean to solve complex solution as they give a wise guidance of combining different patterns.

Falkenthal et al. criticize [FBL] that contrary to the collaborative idea of living network of patterns, many pattern languages are documented in static documents like books, papers or journal. Furthermore, they state that there is currently no it-based tool available that gives pattern authors "an intuitive means to publish, adapt, and interrelate pattern languages via globally accessible media such as web pages that are linked to each other". Based on Alexander's concept of pattern languages, they discuss how pattern languages can be authored and adapted to implement the idea of living networks of patterns. Moreover, they present a formal notion of pattern languages as node-colored and edge-weighted directed multigraphs, which acts as a foundation of this thesis, to develop a web-based system for authoring and browsing pattern languages. Chapter 4 will discuss this in more detail.

Mullet [Mul] discusses the importance of pattern languages in the field of human-computer interaction design, to enable the application of patterns in combination. However, he criticizes that the vast majority of existing pattern languages do not have the ability to guide users to successful design strategies. Hence, he encourages to start a dialogue on what an effective pattern meta-language could look like.

Zdun [Zdu07] propose an approach to facilitate the selection of patterns based on desired quality attributes and systematic design decisions based on patterns. In this approach, he formalized patterns and pattern relationships in a pattern language grammar. The relationships formalized in the pattern language grammar are annotated with effects on quality goals. Based on the pattern language grammar possible pattern sequences in a pattern language can be derived. Subsequent the quality goal annotations can be used during the selection of sequence steps to indicate how quality goals are affected by design decisions.

Falkenthal et al. [FBB+14a; FBB+14b; FBB+16] state that due to fact that patterns aims to abstract and generalize solution knowledge in a technology and implementation agnostic way, the solutions provided by the patterns can mostly not be applied one-to-one to specific use cases. They refer to this as abstraction gap between patterns and concrete use-cases. As a consequence, users require knowledge about refinement and need to invest additional manual effort, to apply a pattern to a specific use case. In order to close this gap they introduced the concept of pattern refinement to allow the navigation from abstract pattern to Concrete Solutions, which are individual use-case specific realizations of the abstract solution principles describe by a pattern. Thereby they also showed how semantic refinement links can be used to interrelate pattern languages on different levels of abstraction. To provide a means to indicate when to select a specific Concrete Solution, Selection Criteria which are represented as human-readable text or computer interpretable descriptions can be added to the relations between pattern and Concrete Solutions.

Subsequent, based on their former research, Falkenthal et al. [FL17] present an approach to organize concrete solutions into Solution Languages, which provide a means to purposefully navigate through a set of concrete solutions that are linked to patterns of a pattern language. They identify the following three capabilities that a Solution Language need to provide: (i) navigation between concrete solutions, (ii) navigation guidance to find relevant further concrete solutions, (iii) and capabilities to document relevant knowledge about dependencies between concrete solutions, such as knowledge about aggregation steps that are required to combine two concrete solutions. Following they state that the capabilities (i) and (ii) can be realized by semantic links between concrete solutions. To realize (iii) they introduce the concept of a Concrete Solution Aggregation Descriptor (CSAD), which allows annotating a link between concrete solutions with additional information.

Barzen et al. [BL15; BL17] show that patterns are not restricted to natural science, but can also be used to represent knowledge in the humanities. They propose a formalism to collect costumes in movies as concrete solutions and abstract them to costume patterns structured into a costume pattern language. In this formalism, they use domain-specific ontologies in order to define valid properties and values to describe concrete solutions within the domain. They emphasize the power of ontologies to create a unified vocabulary of semantic terms that is available and reusable by others. For example, in the domain of costumes in movies, the ontology captures properties and values of elements that form a costume. Based on the ontology concrete solutions can be detected and grouped into costumes with the same effect. These groups represent the essence of a set of concrete solutions which makes up a costume pattern. Furthermore, they generalize the method that it is also applicable in other domains besides costumes in movies.

Over the past years, a large number of patterns had been described in natural language and documented in printed forms such as books, papers or journals. Rosengard et al. [RU04] criticizes that patterns documented in printed form are ineffective to use and can not be understood by machine agents. Moreover, they state the fact that a consistent representation of patterns is missing. Hence, they introduce the idea of combining software patterns with ontological representations, to create a knowledge base that can be shared between agents.

Dietrich et al. [DE05; DE07] present a similar approach. They argue that patterns present knowledge that is shared by communities, which makes them by nature distributed and inconsistent. Based on this idea, they use the web ontology language OWL to define an object-oriented design ontology, called ODOL[1]. The ontology defines the concepts needed to describe design patterns in a formal and machine-readable format. They implemented a Java based prototype[2] that uses the ontology to detect pattern instances in software artifacts. Moreover, the prototype can be used to extract patterns from code and document them in an RDF file. Their overall vision is to create a web of patterns which they define as a system of loosely linked ontologies and design pattern descriptions that can be shared between communities. However, they also admit that adequate tool support, such as pattern authoring kits to create and publish new patterns, is still missing.

The vocabulary of terms provided by ODOL ontology is limited to the domain of object-oriented design patterns. Di Martino et al. [DE13] extends the ODOL ontology with the concept of pattern categories, represented as OWL class. This class gives a means to categorize patterns in a hierarchical structure by adding arbitrary pattern families as subclasses. As proof of work, they extended the ontology with terms that could be used to describe Cloud Patterns. Furthermore, they criticize that common terms existing in the textual description of patterns have no occurrence in the ODOL ontology. For example, terms that describe the intent or consequences of patterns, but also terms that provide a means to semantically link patterns. In order to express this information, they use the pattern template described by Gamma et al.[GHJV93] as a base to extend the ontology by a series of classes and properties that are useful to identify, compare and relate patterns. In subsequent work [DEC15], based on ODOL they propose a semantic representation of cloud services, patterns and appliances. Based on the semantic representation they propose an automatic reasoning procedure that allows automatic discovery of cloud services, and appliances. Moreover, it enables the mapping between vendor dependent and agnostic cloud patterns and services.

Online pattern repositories have been proven to be a promising solution for publishing, authoring and browsing pattern languages. Over the last view years, multiple pattern repositories such as the Portland Pattern Repository [Cun+06], the Open Pattern Repository [Hee09], or PatternPedia[FBFL14] have been developed containing pattern languages from various domains and offering diverse functionalities to support pattern users.

Köppe et al. [KISV16], argue that although existing pattern repositories offer various functionalities to support pattern users, none of them fulfill all user needs. In their work, they identify requirements and features for pattern repositories which support all phases of the design pattern life-cycle. To achieve this, they analyzed existing pattern repositories as well as related literature to identify relevant aspects and functionalities required by pattern repositories. Additionally, they conducted a focus group consisting of participants of the Pattern Languages Of Programs Conference (PLoP) 2015 to get insights from experienced pattern authors regarding their thoughts on requirements for pattern repositories.

[1] http://www-ist.massey.ac.nz/Projects/wop/odol.html
[2] http://www-ist.massey.ac.nz/wop/

4 Use Cases and Requirements

This chapter lays out a set of use cases describing user interactions that a pattern repository offering IT-based tool support for authoring and browsing pattern languages and solution languages must provide. In addition, it discusses the different functional and non-functional requirements that need to be fulfilled by the software system developed in the context of this thesis. The requirements and use cases are derived from former research in the area of pattern language repositories as well as from the assignment of tasks formulated by the proposal of this thesis.

4.1 Use Cases

Pattern languages have been proven to be a useful tool for capturing deep domain expertise in all kind of disciplines. As result patterns and pattern languages have developed in various domains, such as education, cinematography and information technology. Over the last view years, a large number of patterns had been described in natural language and documented in various printed forms such as books, papers or journals. However, this static form of documentation contradicts the primary idea of pattern languages as living networks of patterns as described by Christopher Alexander [Ale77]. Moreover, the fact that patterns are distributed over various documents makes it hard for users to find patterns that suit their problem.

Köppe et al. [KISV16] argues that IT-based pattern repositories could provide a promising solution to overcome those problems as they have the ability to provide pattern users various functionalities such as collaborative pattern writing, pattern editing, or browsing of existing pattern languages. In their work, they analyzed existing repositories as well as existing research to identify important aspects and user interactions that an IT-based pattern repository must provide. They identify capabilities such as pattern writing, pattern reviews, and pattern browsing. This thesis uses the identified aspects as a basis to identify relevant user interactions that the IT-based pattern repository must provide.

The user group of an IT-based pattern repository is defined by pattern authors and pattern users. They differentiate by the fact that only pattern authors have write-access to the repository for authoring pattern and concrete solution documents. Pattern users have no write access they can use the repository to browse existing pattern and solution languages.

The following major use case from the perspective of a pattern author are identified and formulated, as Figure 4.1 illustrates: (i) Author pattern documents, (ii) Author Concrete Solution Documents, (iii) Link Documents , (iv) Community Collaboration.

(i) **Author Pattern Documents:** Pattern authors can create and edit pattern documents. Thereby pattern authors can define the content of the pattern document with various digital information such as text, pictures, or sketches.

(ii) **Author Concrete Solution Documents:** Patten authors can create and edit concrete solution documents, which provide information about use case specific implementation details of an abstractly described solution, documented by a pattern.

(iii) **Link Documents:** Pattern authors can link pattern documents and/or concrete solution documents, in order to create a browsable pattern and solution languages. Moreover, they can describe the relation type, the direction of the relation, and provide arbitrary additional documentation in form of digital information such as text, pictures, or sketches.

(iv) **Community Collaboration:** Pattern authors can collaborate on authoring pattern documents and concrete solution documents. Pattern authors can write pattern and concrete solution documents together, make change requests on existing pattern and concrete solution documents or review proposed changes. Thereby they can use various collaboration functionalities such as review functionalities, communication functionalities, or versioning.

The following major use case from the perspective of a pattern user are identified and formulated, as Figure 4.2 illustrates: (i) Browse Pattern Languages and Solution Languages, (ii) Provide Feedback.

(i) **Browse Pattern Languages and Solution Languages:** Pattern users can browse pattern languages and solution languages by following the semantic links contained in pattern documents and solution documents. Moreover, they can filter a given pattern language or solution language based on selection criteria to receive only the subset of pattern documents or concrete solution documents that match their specific use case.

(ii) **Provide Feedback:** Pattern users can provide feedback on existing pattern documents and concrete solution documents. They can use rating mechanisms to review the quality of existing pattern or concrete solution documents. Moreover, they can share their experience about using pattern or concrete solution documents with the authors and provide possible ideas for improvement in form of textual feedback.

4.2 Requirements

The functional and non-functional requirements that an IT-based pattern repository has to meet, are derived from both, the assignment of tasks formulated by the proposal of this thesis, as well as former research in the domain of pattern languages and pattern language repositories. Table 4.1 shows a list of functional requirements (FR). A list of non-functional requirements (NFR) is shown in Table 4.2. This section will discuss these requirements in detail.

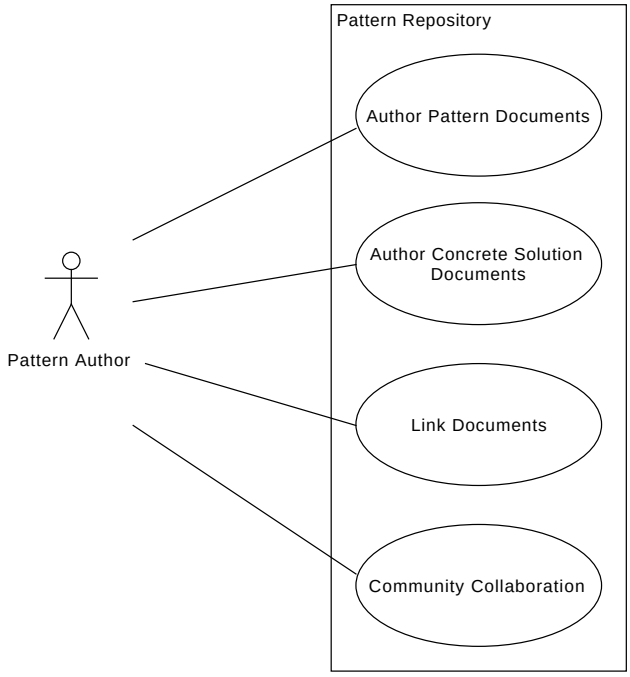

Figure 4.1: Use cases from the perspective of a pattern author on a software based pattern repository

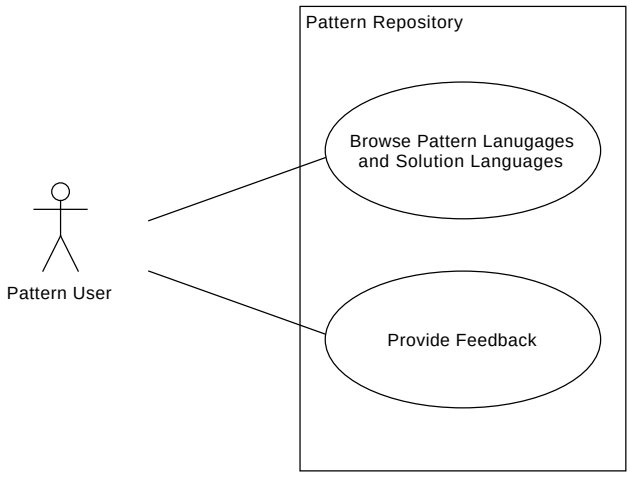

Figure 4.2: Use cases from the perspective of a pattern user on a software based pattern repository

#	Description
FR1	Support Semantic Links in Pattern and Concrete Solution Documents
FR2	Allow to Annotate Semantic Links with Additional Informations
FR3	Support Pattern Languages Consisting of Distributed Pattern Documents
FR4	Support Solution Languages Consisting of Distributed Concrete Solution Documents
FR5	Support Querying of Pattern and Concrete Solution Documents
FR6	Provide Collaboration Functionalities
FR7	Support Versioning of Pattern and Solution Languages

Table 4.1: List of functional requirements

#	Description
NFR1	Usability
NFR3	Compatibility
NFR3	Interoperability

Table 4.2: List of non-functional requirements

FR1: Support Semantic Links in Pattern and Concrete Solution Documents

Patterns are typically not isolated instead they are linked to related patterns, which provide pattern users navigation guidance through a set of patterns that may be relevant in the same context. This navigable network of patterns is called a pattern language. In the pattern language described by Alexander et al. all relationships between patterns had the same semantic meaning. However, in many other domains, link types with different semantic meaning are needed to describe the structure of a pattern language. Thereby the semantic meanings as well as the amount of types depends on the domain of the patterns and can vary between different pattern languages. For example the remoting pattern language by Zdun et al. [ZKV04] define semantic link types such as *implies*, *may use*, and *requires*. The cloud computing pattern language by Fehling et. al [FLR+14] uses semantic links types such as *see also* and *consider after* to interrelate patterns.

The software system should provide means to define semantic links that are not only human readable but can also be processed by machines. The proposal of the thesis specified that Semantic Web Standards should be used to define the semantics of relationships in form of machine-readable metadata. Therefore, the developed software system must provide a means to define semantic relationships between pattern documents and solution documents in form RDF data. Moreover, it must provide a means to process the information provided by the RDF data and visualize it in a human-readable format.

FR2: Allow to Annotate Semantic Links with Additional Informations

In many cases, a semantically typed link is not sufficient to describe the relationship between patterns and/or concrete solutions. Instead, the link needs to provide more detailed knowledge about a relationship. For example, if two patterns are connected via the relationship *alternative*, additional information attached to the relationship can describe use-cases in which one of the alternatives is more suitable than the other. Additional information attached to a semantic link *can be aggregated with* that relates two concrete solutions may provide documentation in form of concrete manual aggregation steps that need to be done. Falkenthal et al. [FBL] suggest to extract that information from the pattern/ concrete solution documents itself, which makes it easier to maintain pattern and solution languages as new relations can be created without rewriting existing documents.

The software system needs to support those concepts and provide a means to annotate semantic links with additional documentation about the relationship. Moreover, it needs to provide a means to author this information in separate documents that can be authored independently from the pattern/ concrete solution documents they interrelate.

FR3: Support Pattern Languages Consisting of Distributed Pattern Documents

A pattern language is not a static artifact, but it is subject to a collaborative process that constantly changes the structure of the pattern language. In this ongoing collaborative process of documenting new patterns, also the interrelations between patterns are under a constant change. Pattern languages which have primary been authored isolated from each other may converge over new relations between patterns that are part of the two pattern languages [FBL]. For example, the Remoting Patterns by Völter et al. [VKZ13] has many relations to patterns and pattern languages from related domains such as networking, concurrency, and resource management that are documented in other places such as books or journals. Those pattern languages are not documented in a central place and had been written by different authors. However, from the perspective of a pattern reader, the pattern languages converge to one as there exist relations between patterns of the different pattern languages. Hence the network of patterns that form a pattern language is not limited to a central location such as books, web pages or pattern repositories. Instead the network of a pattern language can be distributed over various locations.

The developed software system must provide a means to create pattern languages that consist of pattern documents that are not stored centrally, but are distributed over the web and authored by different authorities. Moreover, the software system needs to provide a means to retrieve the distributed pattern documents that form a pattern language in order to provide pattern users a means to browse through the pattern language.

FR4: Support Solution Languages Consisting of Distributed Concrete Solution Documents

Concrete solutions are linked to patterns and provide use-case specific implementations of knowledge described by a pattern [FL17]. Numerous concrete solutions can be linked to one pattern. For example, the Observer Design Pattern described by the Gang of Four Patterns [GHJV93], could be linked to an existing concrete solution implemented in Java and also to an existing concrete solution implemented in C#. Moreover, these two concrete solutions may be authored by different software programmers and are stored in different places on the web. Therefore, analogous to pattern languages the software system must provide a means to create solution languages that consist of concrete solution documents that are distributed over the web and authored by different authorities. Moreover, the software system needs to provide a means to retrieve the distributed concrete solution documents that form a pattern language in order to provide pattern users a means to browse through the concrete solution language.

FR5: Support Querying of Pattern and Concrete Solution Documents

In many cases, the user is only interested in a subset of patterns and/or concrete solutions contained in a pattern and concrete solution language. For example, a software architect that want to host his software application in the cloud may search for patterns describing different types of cloud offerings. A project manager that want to gather knowledge about building a Software as a Service (SaaS) offering may also be interested in cloud computing patterns, but he searches for a different subset of cloud computing patterns. Further, a user that already selected specific patterns that suit his use-case are may want to query use-case specific concrete solution documents based on the patterns he selected. Having a large network of patterns makes it challenging for the user to go through all the existing patterns and concrete solutions and select the ones that suit his problem. Therefore, the software systems need to allow automated selection of suitable pattern documents and solution documents based on given input parameters.

FR6: Provide Collaboration Functionalities

Although Pattern Languages and Solution Languages can be authored by individuals they are most likely subject to a collaborative process in the pattern community [KISV16]. Therefore, an IT-based pattern repository needs to provide functionalities that ease the collaborative authoring of pattern languages and solution languages:

- The content in the pattern repository can be authored by one or more authors that have write-access.
- Pattern authors have the ability to propose changes that can then be reviewed by other pattern authors.

- Users can provide feedback and make suggestions for improvements. For example, a make-up artist who applied a pattern for costumes in films may share her feedback with the authors of the pattern.

4.2.1 FR7: Support Versioning of Pattern and Solution Languages

In collaborative projects version control is an important function as it provides a means to keep track of changes in the projects. Version control keeps track of what files have been changed, the specific changes in the files, and the contributor who made the changes. Additionally, it provides functionalities to revert specific changes in order to recall a prior version. To ease community collaboration, the software system needs to support version control for pattern documents and solution documents.

NFR1: Usability

Pattern languages are used in a wide variety of disciplines. Many of them are non-technical like the patterns in the domain of architecture and urban design for building and planning by Christopher Alexander [Ale77]. Barzen et al. show how patterns can be used in the domain of costumes in films to solve the recurring problem of achieving effects in films by using clothes [BL15]. The authors of such pattern languages have a deep expertise in their domain, but their knowledge in information technology is often limited. Thus, the software system should enable the authoring of pattern languages without advanced knowledge in information technology.

NFR2: Compatibility

The proposal of the thesis specified that software system should be implemented as a browser-based application. This ensures a high degree of compatibility as the system functions universally with various systems. Moreover, it prevents users from installing any software on their computers.

NFR3: Interoperability

To support the development of a distributed living network of patterns it is important to support the growth of the ecosystem around it. It should be easy to develop additional software systems that interact with the existing pattern and solution languages. Moreover, it should be possible to exchange information between software systems.

5 Concept

This chapter proposes an ontological representation of pattern languages and solution languages that uses Semantic Web Standards to describe machine-readable information about patterns and concrete solutions and their interrelations. The chapter is structured as follows, first it describes the general concepts of using Semantic Web Standards to represent pattern languages and solution languages. Subsequent it shows how Semantic Web Standards can be used to create a vocabulary that defines specific concepts of pattern languages and solution languages in form of machine readable metadata. Furthermore this chapter describes how this vocabulary can be used to create a machine-readable network of linked data that describes informations about patterns, concrete solution and their interrelations.

5.1 A Semantic Web of Pattern Languages and Solution Languages

This section describes the general concept of a Semantic Web of Pattern and Solution Languages. This section describes how Semantic Web Standards can be used to create a vocabulary that defines the concepts and relationships, needed to describe and represent pattern languages and solution languages, in a machine readable form. Additionally it describes how instances of those concepts and relationships can be created which can be used to form instances of pattern languages and solution languages consisting of machine readable linked data contained in documents that are distributed on the web.

Authoring pattern languages and solution languages is a collaborative process. Pattern languages and solution languages consist of knowledge that is shared across a community and grows over time as different pattern authors contribute new knowledge. As a consequence this knowledge is by nature distributed and inconsistent [DE07]. This motivates the use of the Semantic Web Standards to represent information of pattern languages and solution languages on the web, because they are inherently designed for machine readable, open and distributed representation of knowledge. In the following we show how the Semantic Web Standards can be used to create a formal and machine processable representation of pattern languages and solution languages, a Semantic Web of Pattern and Solution Languages.

By using Semantic Web Standards to represent knowledge about patterns, concrete solutions and their interrelations we can meet the following objectives:

- In the context of *Support Semantic Links in Pattern and Concrete Solution Documents* (FR1) and *Allow to Annotate Semantic Links with Additional Informations* (FR2). The semantic web standards meets the objective of a formal and machine processable representation of resources and their interrelations on the web in form of linked data. Due to this fact, the semantic web standards can be used to describe informations of pattern, concrete solutions and their relations in form of linked data on the web, which provides a formal and machine processable representation of pattern languages and solution languages.
- In the context of *Support Pattern Languages Consisting of Distributed Pattern Documents* (FR3) and *Support Solution Languages Consisting of Distributed Concrete Solution Documents*, the semantic web standards enable distributed representation of knowledge without a single point of control. This knowledge can be shared and reused across software systems. In order to achieve consensus between authority, vocabularies that define and classify concepts in form of machine-readable schema can be created and shared over the web. Moreover, the standards provide common data formats on the web that are compatible with existing standard web technologies.
- In the context of *Support Querying of Pattern and Concrete Solution Documents*, the semantic web standards provides its own query language SPARQL, which can be used to programmatically retrieve specific RDF data from an RDF graph.

Three of the cornerstones of the semantic web standards are the Resource Description Language (RDF), Resource Description Framework Schema (RDFS), and the Web Ontology Language (OWL), which are designed to represent knowledge about things, group of things and their relations in a machine readable fashion. While RDF builds a standard data model that uses triples in form of subject, predicate and object to express binary relationships between resources that are identified by IRIs, RDFS and OWL are build to create vocabularies that define semantic terms that can be used to encode RDF data with semantic metadata that describe the resources.

The semantic terms contained in an OWL vocabulary describe concepts and relationships that can be used to describe and represent an area of concern. Thereby, OWL provide a means to classify the terms, define possible relationships, as well as possible constraints on using those terms [Con15c]. In the past years different vocabularies have been published, for example the vocabulary of the Dublin Core metadata Initiative [Con08] that provides common terms to describe digital resources such as images, videos, web pages and also physical resources such as books, articles, or DVDs. Another example is the FOAF vocabulary [Dan14] that provide semantic terms to describe persons and their relations to other persons and objects.

One of the big advantages of OWL is its open world assumption, which means that description of resources are not limited to a single file or scope instead they can be extended by other vocabularies, which allows that knowledge can be easily added [WMS04]. For

example, a vocabulary that describe basic terms to describe patterns and relationships between patterns could be extended by a vocabulary that defines additional terms such as terms to describe patterns in the domain of cloud computing.

Vocabularies can be published in form of RDF documents on the world wide web to be shared within the community. Published vocabularies can then be imported into other RDF documents to describe the contained resources in a machine readable way by creating instances of terms defined in the vocabulary. The general concept is illustrated in Figure 5.1. There, a vocabulary defines terms that can be used to describe facts about cloud computing patterns. The RDF data illustrated in Figure 5.1 is written in XML syntax however the documents can also represent machine readable data in other RDF syntax, such as N3, turtle, or RDFa which allows to embed RDF data into Hypertext Markup Language (HTML). The vocabulary defines the class *Pattern* and its subclass *CloudComputingPattern*. Additional it defines the properties *patternName*, *alternative*, and *seeAlso*. The terms defined by the vocabulary are used in the illustrated RDF documents to describe machine readable information about instances of the class *CloudComputingPattern*. The RDF document_A contains RDF data that defines a CloudComputingPattern with the IRI "http://ccp.com/docA#public-cloud". The RDF data describes that this CloudComputingPattern has the property patternName of 'Public Cloud' and the relationship alternative to another CloudComputingPattern referenced by the IRI of "http://ccp.com/docB#private-cloud". As resources are defined by IRIs, the resources that are linked to each other form a network of linked data that can be browsed by a machine-reader, although the data is distributed over the web. Based on the semantically linked knowledge represented in form of an RDF graph, the SPARQL query language can be used to analyze the linked knowledge by writing queries against it. For example, on the RDF graph build by the RDF data in Figure 5.1 a SPARQL query may retrieve all objects that are linked via the predicate ccp:alternative to the subject "http://ccp.com/docA#public-cloud".

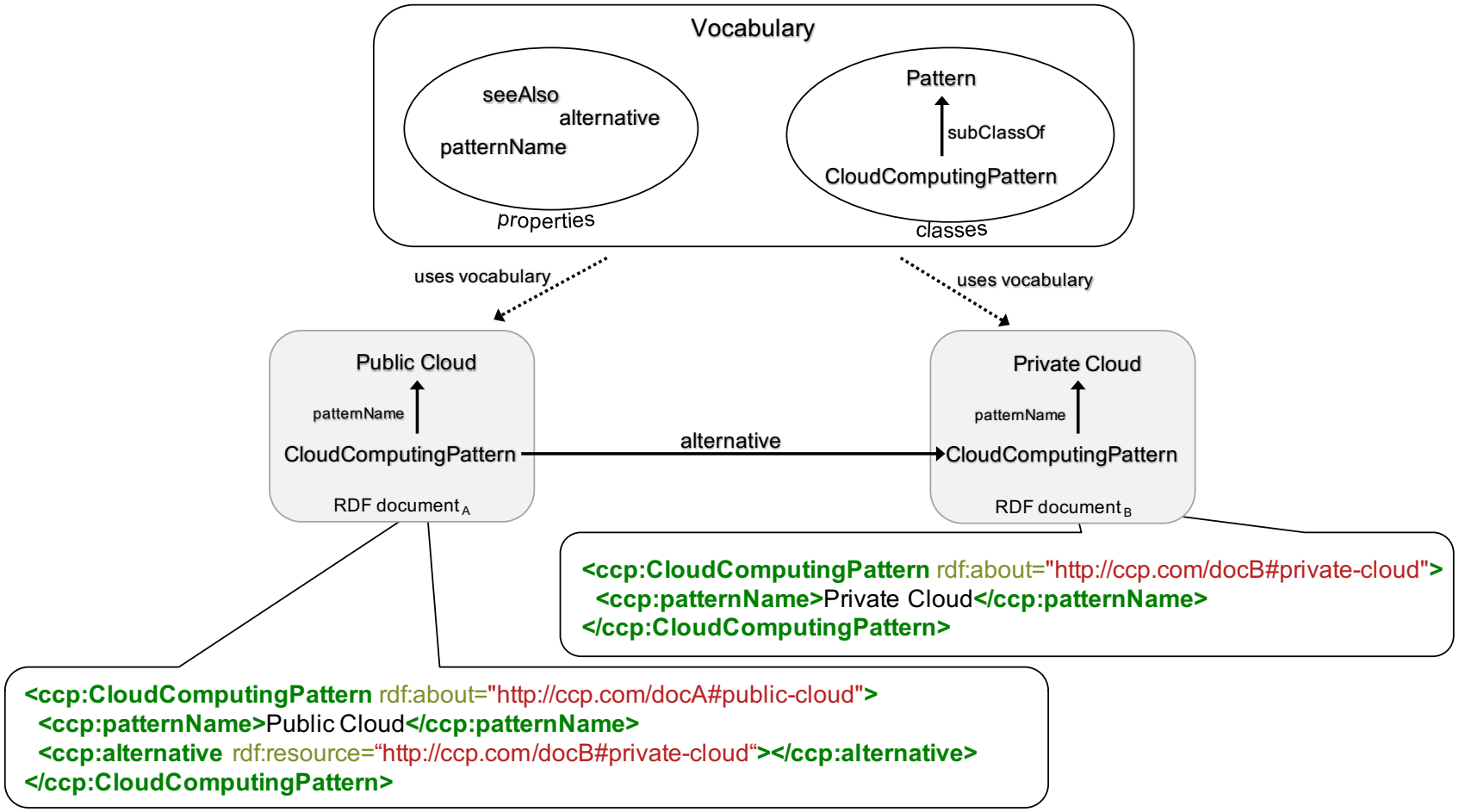

Figure 5.1: The terms defined by a vocabulary are used to encode RDF data in documents with machine readable semantic metadata

5.2 Ontological representation of Pattern Languages

This section describes how Semantic Web standards can be used to describe concepts of pattern languages in form of machine readable metadata. Thereby it describes how OWL can be used to define a vocabulary with the concepts and relationships needed to describe and represent pattern languages in form of machine readable metadata. Additionally it describes how instances of those concepts can be created to describe instances of pattern languages consisting of machine readable distributed documents.

5.2.1 Semantic classification of pattern types

Today patterns and pattern languages have been established in various disciplines such as information technology, education, or architecture. Structuring patterns into pattern languages categorizes patterns by means of pattern types, e.g the patterns in a cloud computing pattern language are cloud computing patterns, the patterns in a object-oriented software design pattern language are object-oriented software design patterns, or the patterns of a urban design pattern language are urban design patterns. As the relevance on the different patterns contained in the pattern language may differ depending on the profession of the pattern language reader as well as his specific use case, patterns contained in a pattern language are often again subdivided into categories to support pattern readers by finding relevant patterns for their use case. For example, a software architect or developer using the cloud computing patterns by Fehling et al. [FLR+14], may want to use cloud offerings to host his application may use the pattern language to learn about cloud offerings and to find out which cloud offerings meet the application specific requirements. An IT infrastructure manager who want to build a Platform as a Service (PaaS) offering, may search for different patterns, those that provide knowledge about cloud properties that need to be fulfilled by PaaS offerings. This categorization of patterns creates a semantic classification of the patterns into pattern types. Moreover it creates a hierarchy between those pattern types.

In OWL this semantic classification of pattern types can be defined by the concept of OWL classes. Based on those classes, instances of patterns can be created that are individuals of a class. If an individual is a member of a class, it tells a machine reader that it falls under the semantic classification of that class. In addition, a hierarchy or also called a taxonomy of the classes can be defined by the taxonomic predicate *rdfs:subClassOf*, which relates a more specific class to a more general class. If a class C_1 is a subclass of a class C_2, then every individual of C_1 is also an individual of C_2. Moreover the *rdfs:subClassOf* relation is transitive which means if C_1 is a subclass of C_2 and C_2 is a subclass of C_3, then C_1 is also a subclass of C_3 [WMS04]. Listing 6 shows an excerpt of an exemplary vocabulary that defines the two classes *CloudComputingPattern* and *CloudApplicationArchitecturePattern*. Moreover it relates the class *CloudApplicationArchitecturePattern* via the predicate *rdfs:subClassOf* to the class *CloudComputingPattern*, which defines that the class *CloudApplicationArchitecturePattern* is a subclass of the class *CloudComputingPattern*.

```
<owl:Class rdf:ID="CloudComputingPattern">
        <rdfs:label>Cloud Computing Pattern</rdfs:label>
        <rdfs:comment>This class represents those things which are Cloud Computing Patterns</rdfs:comment>
</owl:Class>

<owl:Class rdf:ID="CloudApplicationArchitecturePattern">
   <rdfs:label>Cloud Application Architecture Pattern</rdfs:label>
   <rdfs:comment>This is a subclass of Cloud Computing Pattern</rdfs:comment>
   <rdfs:subClassOf>
      <owl:Class rdf:about="#CloudComputingPattern"></owl:Class>
   </rdfs:subClassOf>
</owl:Class>
```

Listing 6: Excerpt of an exemplary vocabulary that defines that the class CloudApplicationArchitecturePattern is a subclass of the class CloudComputingPattern

5.2.2 Navigation through pattern languages

Patterns are typically not isolated nuggets of advice, instead they are linked to related patterns, which allow pattern users to navigate from one pattern to other patterns that may be relevant in the same context. This navigable network of patterns is called a pattern language. Whereas the initial pattern language by Christopher Alexander uses a single implicit link type between patterns, in many other domains different link types are needed to describe the structure of the pattern language and to provide precise navigation guidance to find relevant further pattern [FBL]. For example in the pattern language on cloud computing [FLR+14], patterns are referenced with link types such as *see also* and *consider after*. The remoting pattern language by Zdun et al. [ZKV04] describe dependencies between patterns via different link types such as *implies*, *may use*, and *requires*. Links in existing pattern languages are mostly represented as textual references in printed documents or as simple hyper-links on webpages. The semantic meaning of such links can be understand by humans, but they can not be processed by machine-readers.

Referring to the mentioned examples we introduce the concept of semantic links between pattern languages that provide a navigation guidance through pattern languages, by supporting machine readers to decide if related patterns are relevant or not. In OWL, semantic links between resources can be defined by object properties that link individuals of classes. Similar to classes the characteristics of object properties can be defined in a vocabulary. The example shown in Listing 7, defines property for the relationships *alternative* and *considerAfter*. The *domain* and *range* defines that the property links individuals belonging to the class CloudComputingPattern. The defined properties can then be used to express semantic links between individuals of CloudComputingPattern. An example of the concept is illustrate in Figure 5.2. There RDF data contained in distributed documents describe individuals of the class CloudComputingPattern. The individuals are semantically linked via object properties. The individual PublicCloud contained in document P_a is linked via the object property *seeAlso* to the individual SaaS contained in document P_b. Moreover,

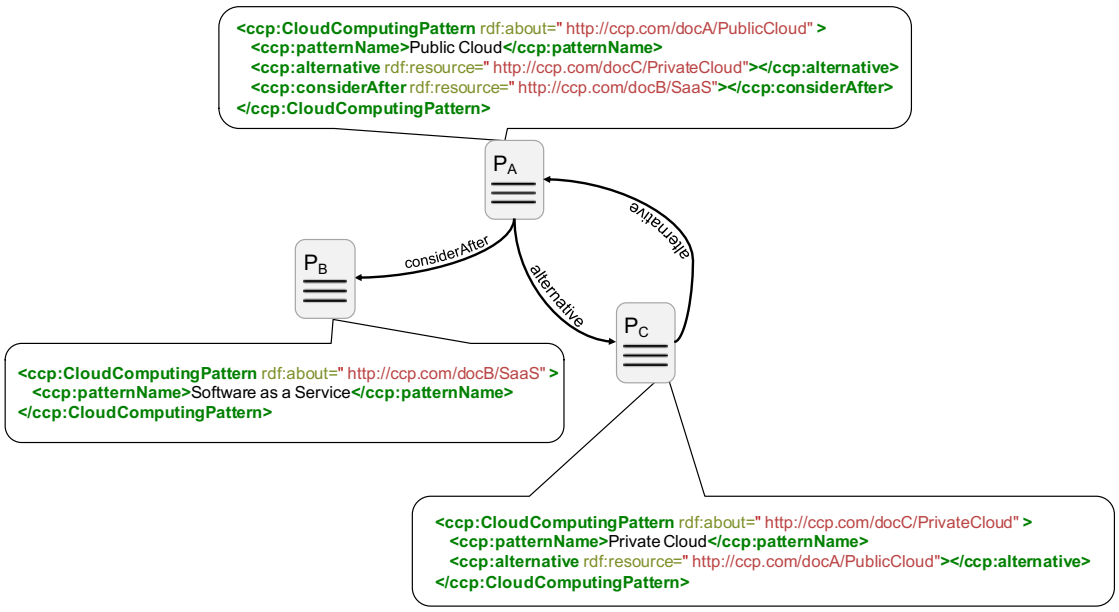

Figure 5.2: Pattern Individuals contained in different documents are semantically linked via object properties

the individual Public Cloud is linked via the object property *alternative* to the individual PrivateCloud described in document P_c. In addition the individual PrivateCloud is also linked via the object property alternative to the individual PublicCloud.

```
<owl:ObjectProperty rdf:ID="alternative">
     <rdfs:domain rdf:resource="#CloudComputingPattern"/>
     <rdfs:range rdf:resource="#CloudComputingPattern" />
</owl:ObjectProperty>
 <owl:ObjectProperty rdf:ID="considerAfter">
     <rdfs:domain rdf:resource="#CloudComputingPattern"/>
     <rdfs:range rdf:resource="#CloudComputingPattern" />
</owl:ObjectProperty>
```

Listing 7: Definition of two object property alternative and considerAfter to link individuals of the class CloudComputingPattern

Pattern Relation Descriptor

Falkenthal et al. [FBB+14b] argues that in some cases it is important that links between patterns not only describe how those patterns are related by means of a semantic keyword but also provide more detailed knowledge about the relationship to describe how the relation is defined in the specific case. For example, if two patterns are interrelated with the semantic link *can be combined with*, additional information can be provided to the user

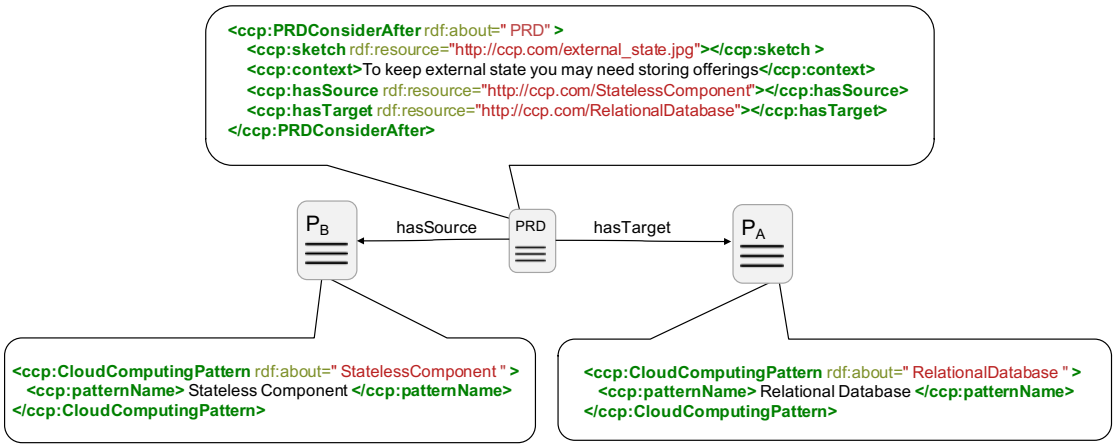

Figure 5.3: A Pattern Relation Descriptor contained in a separate document, describes a relationship between the two pattern individuals Stateless Component and Relational Database

that describe all required steps to actually combine those patterns. Today, the information about relations to other patterns is mostly documented in pattern documents themselves. Falkenthal et al. criticizes that this makes it hard to maintain the pattern language, as changes may require rewrites in the related patterns. Therefore they suggest to extract these information from the patterns themselves and attach it directly to the links that interrelate the patterns.

In order to extract the additional documentation about the relationship between two patterns from the pattern documents itself into a separate document, we introduce the concept of a *Pattern Relation Descriptor* (PRD). The Pattern Relation Descriptor describes the a semantic relation between two patterns as well as additional information about the relationship in a separate document. In OWL, we describe this concept by means of an OWL class PatternRelationDescriptor. In addition more specif types of PRDs can be described that are subclasses of the generic class PatternRelationDescriptor. For example a subclass PRDAlternative could classify instances PRDs that describe the relationship alternative.

An individual of the class PatternRelationDescriptor is always linked to two individuals of the class Pattern and acts thereby as connector between two individuals of the class Pattern. The direction of the relationship can be defined by object properties *hasSource* and *hasTarget*, whose domain is represented by PatternRelationDescriptor and range is represented by Pattern. Information about the relationship can be linked to an PRD individual via the concept of owl properties. This information can contain literal values that are linked via datatype properties but also external resources such as images or graphics that are linked via object properties. The information provided by PRD can be aggregated by a machine reader that parses the information and presents it in a human readable format. For example, the additional information about a relationship could be presented as a tooltip that appears if a user hovers over a link contained in digital pattern document.

An example of a Pattern Relation Descriptor is depicted in Figure 5.3. There the individual PRD of the class PRDConsiderAfter describes a relationship between the individuals Stateless Component and Relational Database. The object properties *hasSource* and *hasTarget* express that the direction of the relationship is from the individual StatelessComponent to the individual RelationalDatabase. Furthermore, the PRD provides additional information that describes the context of the relationship in human readable form and provides a link to a resource that illustrates the concepts.

5.3 Ontological representation of Solution Languages

In order to guarantee that pattern and pattern languages are applicable for many different use-cases, patterns abstract and generalize solution knowledge in a technology and implementation agnostic way. As a consequence, the solutions provided by patterns can mostly not be applied one-to-one to specific use cases.

Falkenthal et al. [FL17] criticizes that this lack of guidance about how to implement a pattern in a concrete use-case specific context leads to immense manual effort and reimplementation of already existing solutions. To overcome this problem they introduced the concept of *Concrete Solutions*. Concrete solutions are linked to patterns and provide use-case specific implementations of the abstractly described solution documented by the pattern. The solution knowledge provided by a concrete solution depends on the domain of application. For example in the domain of software development a concrete solution could provide code, which can be directly used by programmers to be integrated in their software application. For example a Java developer faced with the problem of implementing the Observer Design Pattern described by the Gang of Four Patterns [Gam95] could reuse an existing concrete solution of the Observer Design Pattern written in Java. Depending on the domain concrete solutions are not necessarily program code or other digital artifacts, instead they can also be represented as tangible artifacts. For example, in the domain of costumes in movies as described by Barzen et al. [BL15] a concrete solution could be a costume in a wardrobe.

5.3.1 Concrete Solution Descriptor

Patterns may be linked to multiple concrete solutions, each providing a concrete implementation of the pattern for a specific use-case. For example the Observer Design Pattern, could not only be linked to an existing concrete solution implemented in Java, but also to existing concrete solutions implemented in other programming languages such as, JavaScript, Hypertext Preprocessor (PHP), and C#. A superhero costume pattern may be linked to various existing costumes in a wardrobe such as Superman costume, a Batman costume, and an Iron man costume. Since many existing concrete solutions can be linked to a pattern, a pattern user need guidance to select a proper concrete solution that can be applied in the context of his use-case.

To provide a means to select proper concrete solutions we introduce the concept of a *Concrete Solution Descriptor* (CSD). A Concrete Solution Descriptor acts as connector between a pattern and a Concrete Solution that needs to be linked to a pattern. They provide a means to guide the selection of a proper Concrete Solution by providing additional information that is not contained in the Concrete Solution itself. Those information may contain different selection criteria as well as information about the location of the Concrete Solution. For example, the information about the location of the concrete solution artifact could be expressed through an Internationalized Resource Identifier (IRI) [M D05]. The resource identified by the IRI can be anything that has an identity [BFM98], for example a digital artifact such as an electronic document, an image, or a video but also tangible objects such as a book in a library or a costume in a wardrobe. In addition, the Concrete Solution Descriptor provides an abstraction level that provides a consistent means to represent this information independent from the representation form of the linked concrete solution artifact.

Figure 5.4 illustrates how a Concrete Solution Descriptor can be described in form of RDF triples in order to provide the information in a machine readable format. There the individual AwsELB of the class ConcreSolutionDescriptor describes properties about a concrete solution artifact. The *css:hasPattern* object property describes that the concrete solution artifact is related to the pattern ElasticLoadBalancer. The object property *ccp:hasCSArtifact* describes the location of the concrete solution artifact in form of an IRI. Moreover additional selection criteria are provided in form of datatype properties to describe the platform of the concrete solution can be applied to as well as the format the solution is written in. Furthermore it provides a short description about the intend of the concrete solution artifact in human readable format. Due to the fact that the data is described in form of RDF triples, the SPARQL query language can be used to programmatically retrieve Concrete Solutions based on given query parameters. For example a software system that implements SPARQL functionalities could be used to retrieve all concrete solution artifacts for a Elastic Load Balancer that are implemented for AWS.

5.3.2 Guide Navigation through Concrete Solutions

The sum of all concrete solutions that are linked to patterns of a pattern language builds the solution space of the pattern language [FL17]. The concrete solutions of a solution space are linked to patterns, but they are not linked to each other. As a consequence, if a user selects a concrete solution he has no guidance to navigate through the set of all further relevant concrete solutions. Instead navigation is only possible on the level of patterns by navigating through the pattern language. This can be time consuming and frustrating, especially if users have their conceptual solutions already defined and want quickly browse through the available concrete solutions. Moreover if users want to combine multiple concrete solution they face the problem, that an understandable documentation is missing on how to actually combine these concrete solutions [FL17].

To solve this problem, Falkenthal et al. [FL17] propose, the concept of Solution Languages that provide a means to organize concrete solutions analogously to pattern languages

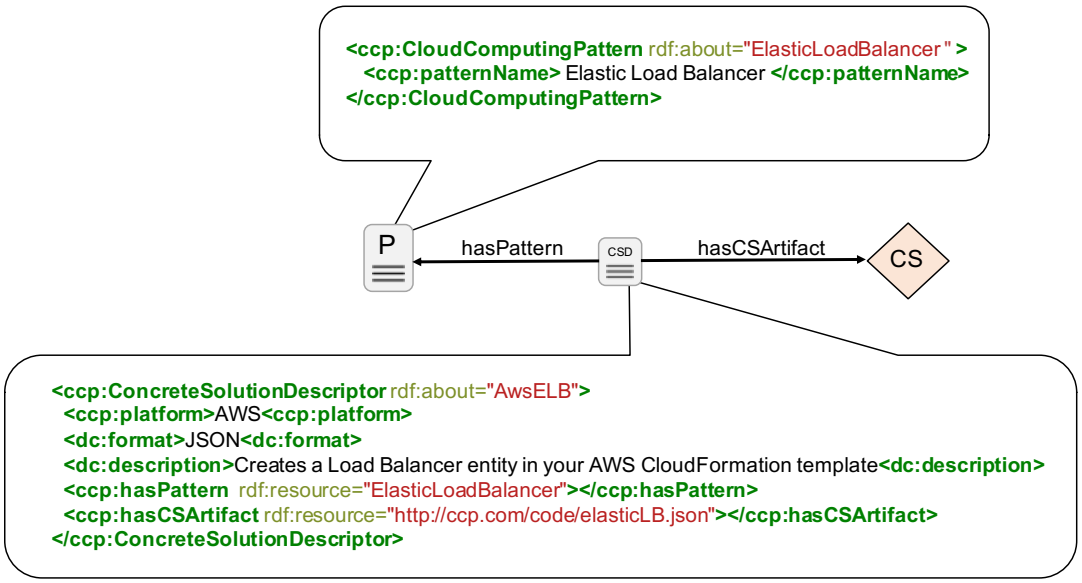

Figure 5.4: A Concrete Solution Descriptor (CSD) described using RDF acts as connector between a pattern (P) and a concrete solution artifact (CS) and provides important information about the concrete solution in a machine processable format

organize patterns. They identify the following three capabilities that need to be provided by a Solution Language: (i) navigation between concrete solutions, (ii) navigation guidance to find relevant further concrete solutions, (iii) and capabilities to document relevant knowledge about dependencies between concrete solutions, for example knowledge about aggregation steps that are required to combine two concrete solutions. Following they state that the capabilities (i) and (ii) can be realized by semantic links between concrete solutions. Based on a semantic link a user can decide if a specific further concrete solution is relevant for his use-case or not. For example a semantic link between two concrete soltutions could indecate that those are *alternatives*, it also indicate that the concrete solutions are different *variants*, for example two concrete solutions written in different programming languages. A semantic link may also indicate that two individual concrete solutions *can be aggregated with* or *can not be aggregated with*.

Whereas (i) and (ii) can be enabled by the use of semantically typed links between concrete solutions, the capabilities provided by the concept of semantic link are sufficient to enable (iii). Therefor, Falkenthal et al. introduce the concept of a *Concrete Solution Aggregation Descriptor* (CSAD). They describe the concept CSAD as follows: "CSADs are the means to add arbitrary documentation about how to aggregate concrete solutions to a Solution Language." [FL17]. However, important details about dependencies between concrete solutions are not limited to aggregation details. For example, if two concrete solutions are different variants the user is not interested in aggregation details instead he may need

information details about the existing differentiators. Therefor we extend this concept by a more general concept of a Concrete Solution Relationship Descriptor (CSRD).

A CSRD allows to annotate a link between concrete solutions with additional information that describe important details about the dependency of two concrete solutions. This concept is analogously to the concept of the PRD on the level of pattern described in Section 5.2. For example, the information contained in a CSRD could provide textual documentation that describe the different working steps that are required to combine the individual concrete solutions. It could also provide other forms of useful documentation such as pictures for a visual instruction of the individual working steps, or a sketch to illustrate the artifact that forms as a result of the aggregation. The content provided by the CSRD depends on the domain of the concrete solutions, e.g to guide the aggregation of two concrete solutions that are programming code, the CSRD could provide additional code snippets that need to be added and could provide detailed description about manual working steps required for the aggregation. In the domain of urban design the content of a CSRD may describe how to combine artifacts like street trees, plants and flowers, which are in this case concrete solutions, to create urban parks in cities.

The information provided by the CSRD is not limited to aggregation details, but could contain arbitrary important information details about a dependency between two concrete solutions. In case of two existing variants of concrete solutions the CSRD could provide documentation details about the existing differentiators. For example, a CSRD describing the dependency between an existing Superman costume and a Batman costume that are both variants of concrete solution of a superhero pattern in a pattern language for costume in movies, may differentiate the variants by providing a list of comics these costumes are applicable.

In OWL this concept can be described by an OWL class ConcreteSolutionRelationDescriptor. Subclasses of this class could define more specific types of ConcreteSolutionDescriptors. For example, an subclass CSRDAggregation could define the specific subset of CSRD that describes the relationship can be aggregated with between two concrete solutions. Based on the semantic classification of CSRD types a machine reader could select specific instances of CSRD based on given parameters. For example, a machine reader could select all instances of CSRD that describe the relationship *can be aggregated with* between a concrete solution CS_1 and other CS that exist in the concrete solution space of a pattern language. Object properties can be defined to describe which concrete solutions are related by the CSRD. For example a object property *aggregates* could link all instances of concrete solutions to CSRD that are part of the aggregation. Moreover additional information could be described by means of datatype properties and object properties that are linked to the instance of a CSRD. Figure 5.5 depicts an example of an CSRD described using RDF and OWL. For sake of simplicity the RDF data contained in the documents that describe the CSDs are not depicted. The described CSRD is an instance of the OWL class CSRDAggregation. The object property *ccp:aggregates* links the CSRD to two instances of CSD. Additional information is linked to the CSRD by means of datatype properties. The datatype property *ccp:preCondition* is used to describe preconditions that need to be fulfilled for the aggregation. The datatype

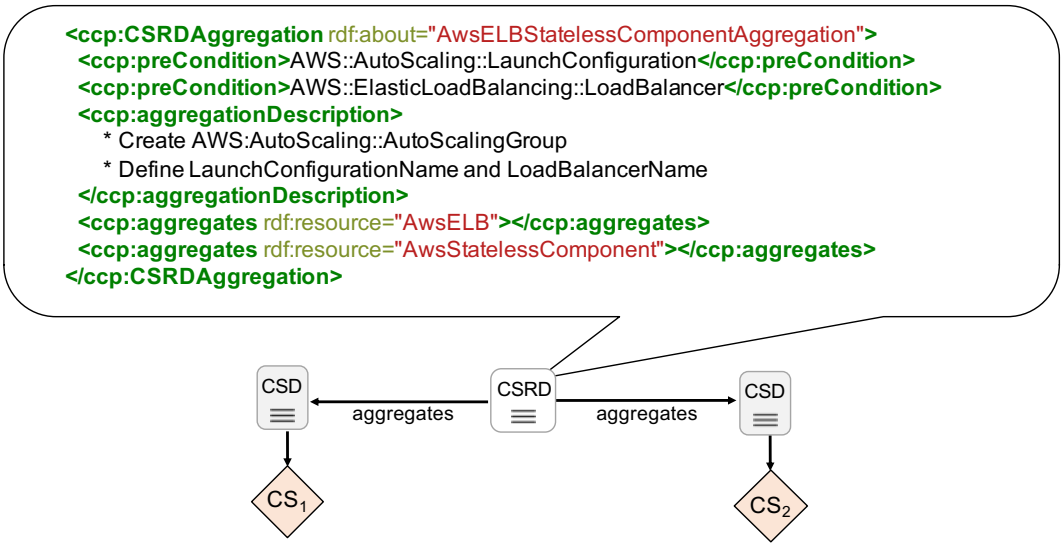

Figure 5.5: A Concrete Solution Relation Descriptor (CSRD) described using RDF acts as connector between two Concrete Solution Descriptors (CSD) and provides important information about their relationship in a machine processable format

property *ccp:aggregationDescription* describes the different manual working steps needed for the aggregation.

Figure 5.6 illustrates the overall view of the concepts described in this chapter. For sake of simplicity, the RDF data contained in the documents is not illustrate in Figure 5.6. There, each document contains RDF data that describes machine readable informations about patterns, PRD, CSD, CSRD and their relations to other resources on the web. Together, the form a machine processable network of linked data, which describes a pattern language and solution language.

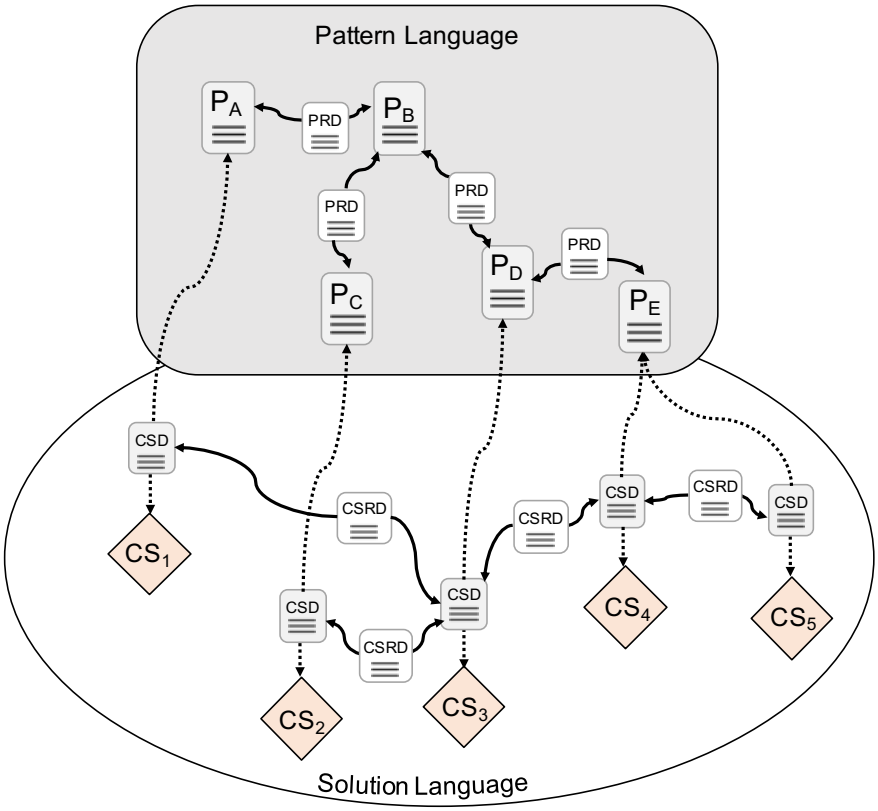

Figure 5.6: Distributed RDF documents describe information about pattern, concrete solutions and their relations in form machine readable linked data that represents information about a pattern and solution language

6 A Semantic Pattern and Solution Repository

This chapter describes the architecture of an IT-based pattern and solution repository for publishing, authoring and browsing pattern and solution languages on the semantic web. The pattern repository allows users to publish information about patterns, concrete solutions and their relations in form of RDF data on the web. Furthermore, the pattern repository provides functionalities to retrieve RDF data that describes information about patterns, concrete solutions and their relations from distributed source and visualize the information in human readable documents.

In the following the software system will be referred to as **Se**mantic **Pa**ttern and **So**lution **Re**pository - *SePaSoRe*

In the context of *usability* (NFR1) *SePaSoRe* considers the fact that authors and users of pattern and solution languages often have deep expertise in their domain but little in the domain of information technology. Hence, they are likely not familiar with the concepts of the Semantic Web Standards, nor do they know how to operate a server that stores the RDF documents. *SePaSoRe* aims to abstract the technologies of the Semantic Web Standards that are used to describe information about pattern, concrete solutions, and their relations. This allows users with little knowledge in information technology to apply the concepts of the semantic web to describe and share knowledge in form of pattern and solution languages.

6.1 Architecture

The block diagram shown in Figure 6.1 uses the Fundamental Modeling Concepts (FMC)[1] notation [KGT05] to represent a model of the static compositional structure of the *SePaSoRe*. The overall architecture of *SePaSoRe* can be divided in the following major building blocks, (i) a GitHub repository as data hosting platform, (ii) the *SePaSoRe* Client running as a single-page application in the browser, and (iii) the *SePaSoRe* Service hosted as a stateless service. In the following these building blocks will be discussed in detail.

[1] http://www.fmc-modeling.org/

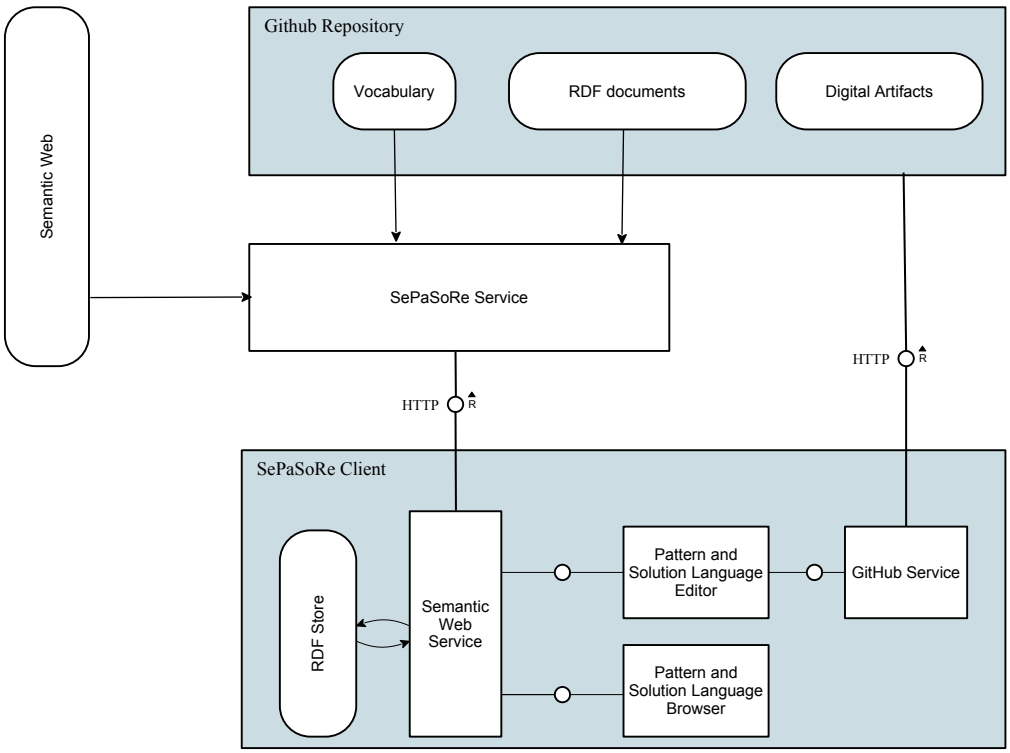

Figure 6.1: Architecture of *SePaSoRe*

6.1.1 *SePaSoRe* Client

To guarantee a high degree of platform compatibility (NFR2) the *SePaSoRe* Client is a single-page application that runs entirely in the browser. It provides capabilities to edit and browse pattern languages in the Semantic Web of Pattern and Solution languages. The client consist of (i) a Pattern and Solution Language Editor, for authoring pattern documents, concrete solution documents, and their relations, (ii) a GitHub Service that interacts with a specified GitHub Repository (iii) a RDF Store, that holds a RDF Graph representation of a given pattern and solution language, (iv) a Semantic Web Service that interacts with the *SePaSoRe* Services and access the RDF Store, and (v) a Browser to visually browse through pattern languages and solution languages contained in the RDF Store.

The purpose of the **Pattern and Solution Language Editor** is to provide a graphical user interface (GUI) that abstract the Semantic Web Standards, which are used as data model for pattern and solution languages in the Semantic Web of Pattern and Solution Languages. It enables user with little knowledge of the RDF Syntax to create and edit information about patterns, concrete solutions, and their interrelations in form of RDF data. In order to provide an easy-to-read and easy-to-write plain text formating syntax for pattern documents,

we decided to use Markdown[2]. A user can edit the content of a pattern via a Markdown editor that is provided by the GUI. Additionally a user can create semantic relations to existing pattern documents via a select box menu. Afterwards the Pattern Editor parses the information to create a valid RDF representation of the pattern.

In the context of *Support Querying of Pattern and Concrete Solution Documents* (FR5) an **RDF Store** is used as database for the RDF Graph representation of the pattern language. The RDF Store supports the SPARQL query language for the storage and retrieval of triples. With the RDF Store we can run SPARQL queries against the RDF graph to retrieve the subset of triples that matches given query parameters. The **Semantic Web Service** is used to interact with the RDF Store. It provides functionalities to retrieve and process RDF data.

The purpose of the **Pattern and Solution Language Browser** is to provide a GUI to visually browse the pattern and solution language which is stored as an RDF graph in the **RDF Store**. The information contained in the RDF triples are parsed and visualized in human readable form. The triples that contain information about relationships between patterns and/or concrete solutions are visualized as hyperlinks that can be used to navigate to related documents. In the context of *Allow to Annotate Semantic Links with Additional Information* (FR2), additional information about the relation is visualized in a pop-up that appears if a user hovers over the hyperlink.

In the context of *Support Querying of Pattern and Concrete Solution Documents* (FR5) the GUI of the Pattern Language Browser provides functionality to search specific patterns based on given parameters. In consideration of *usability* (NFR1) the GUI abstracts the SPARQL query language that is used for retrieval of triples contained in the **RDF Store**.

6.1.2 GitHub Repository

In the context of *Provide Collaboration Functionalities* (FR6) and *Support Versioning of Pattern and Solution Languages*, we decided to use GitHub repositories as a hosting platform for the files that contains data of pattern and solution languages. This files can be RDF documents that contains information in form of RDF triples, but also other files that provide informations about patterns and concrete solution, such as images, or programming code. In this way *SePaSeRo* can utilize the version control and collaboration functionalities provided by GitHub.

In addition, GitHub Pages[3] is used to host the *SePaSeRo* Client directly from the GitHub repository. This means there is no need for users to set up any database or server. Moreover, due to the fact that GitHub offers free public repositories the services of the *SePaSeRo* can be provided without any costs for users. Furthermore, tenant specific access permissions on the repositories can be directly handled by GitHub. For example, a GitHub repository

[2]https://daringfireball.net/projects/markdown/
[3]https://pages.github.com/

owned by an organization restricts write access for members of the organization. In order to provide such tenant specific instances of *SePaSeRo* for a single user or group of users, their need to exist a dedicated GitHub repository for each tenant. To achieve this *SePaSeRo* uses the forking functionality provided by GitHub. A fork of an existing repository creates an copy that is totally independent from the original repository. For example. a build version of the *SePaSeRo* Client can be hosted in a public GitHub repository. To obtain a working copy, users just need to fork the repository and activate the settings for GitHub Pages.

6.1.3 *SePaSoRe* Service

In the context the same-origin policy implemented by current web browsers, JavaScript code that runs in the browser is prevented from making requests against a different origin than the one from which it was served. This contradicts *Support Pattern Languages Consisting of Distributed Pattern Documents* (FR3) and *Support Solution Languages Consisting of Distributed Solution Documents* (FR4) that requires that the information about pattern and solution languages must not be centrally stored in a single repository. Instead the information can be distributed over the web. For example, the information can be distributed over different GitHub repositories owned by different organizations or individuals. This motivated the extraction of specific components into an self-contained hosted stateless service.

The block diagram shown in Figure 6.2 illustrates the compositional structure of the *SePaSoRe* Service, that consists of (i) an RDF Graph Crawler, (ii) a Vocabulary Reader, (iii) a Representational State Transfer (REST)ful HTTP API that provides access to a set of functionalities provided by the *SePaSoRe* Service.

The purpose of the **RDF Graph Crawler** is to crawl the information about patterns, concrete solutions and their relations from distributed sources that together form a pattern and solution language. The Crawler service starts with a list of URLs that reference RDF documents. The pattern crawler retrieves the RDF triples contained in the documents and identifies all objects represented as URL reference and adds them to the list of URLs to visit, called the crawl frontier. This processes is recursively applied to the URLs contained in the crawl frontier until no new URLs are found. The RDF triples found during this process gets combined to form a RDF graph representation of a pattern and solution language.

The purpose of the **Vocabulary Reader** is to read and an analyze an OWL vocabulary which defines pattern types, and other concepts described in Chapter 5, such as types of Pattern Relationship Descriptors, or types of Concrete Solution Descriptors, that can be used to describe pattern and solution languages. Thereby it extracts all classes, properties and their relations described in the vocabulary.

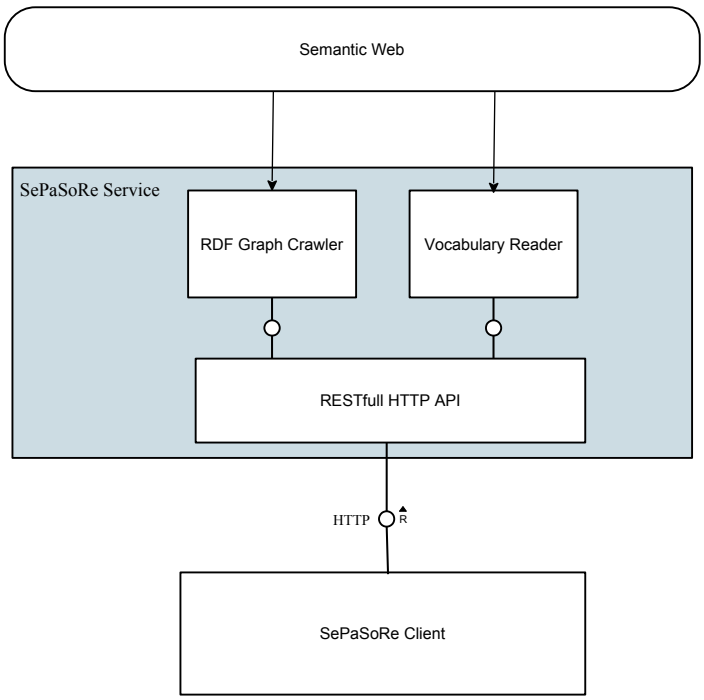

Figure 6.2: Compositional Architecture of *SePaSoRe* Service

7 Prototypical Implementation

This the chapter describes how a prototypical implementation of the **Se**mantic **Pa**ttern and **So**lution **Re**pository - *SePaSoRe*, described in Chapter 7, has been made. Thereby, this chapter also describes how the web ontology language (OWL) has been used to implement a vocabulary that formally defines the concepts of pattern and solution languages described in Chapter 5. In the context of the prototypical implementation a subset of the Cloud Computing Patterns, by Fehling et al. had been used as an exemplary pattern language.

7.1 Exemplary Pattern Language

In the context of the prototypical implementation, a subset of the Cloud Computing Patterns, by Fehling et al. had been used as an exemplary pattern language. This section gives an overview of the properties and structure of this pattern language.

The patterns described in the cloud computing pattern language are all structured according to the same format. For the sake of simplicity the original pattern structure was reduced to the following properties:

- A **Pattern Name**, which is used to identify the pattern
- An **Intent**, which shortly describes the intent of the pattern
- An **Icon**, as a graphical representation
- A **Driving Question**, which states the problem answered by the pattern
- A **Context**, which describes the environment and forces that leads to the problem. This section may also contain references to other patterns.
- A **Solution**, which explains how the pattern solves the problem
- **Related Patterns**, describing interrelations to other patterns.

The pattern language uses the semantic links *see also*, *consider after*, *alternative*, and *known use* to interrelate patterns.

The cloud computing patterns aims to provide a broad range of patterns on how to achieve common cloud related goals. The relevance of the different patterns contained in the cloud computing pattern language may differ depending on the specific use case of the pattern language reader. For example, a software architect or developer that want to use cloud offerings to host his application may use the pattern language to learn about cloud offerings

and to find out which cloud offerings meet the application specific requirements. An IT infrastructure manager who wants to build a Platform as a Service (PaaS) offering, may search for different patterns, those that provide knowledge about cloud properties that need to be fulfilled by PaaS offerings.

In order to support pattern users by finding relevant cloud computing patterns for their use case, the patterns in the cloud computing pattern languages are subdivided into different pattern categories, whereas each category covers a specific area of concern. The patterns contained in the cloud computing pattern language are subdivided into the pattern categories, Cloud computing fundamentals, Cloud offerings, Cloud application architectures, Cloud application management, and Composite cloud applications. This creates a semantic classification of the patterns contained in the pattern language into pattern types. Moreover, it creates a hierarchy between those pattern types.

7.2 *SePaSoRe* Vocabulary

This section describes how the web ontology language (OWL) has been used to create a vocabulary that formally define different types of patterns and related concepts described in Chapter 5, such as types of Pattern Relation Descriptors, Concrete Solution Descriptors and Concrete Solution Relation Descriptors, that can be used to describe pattern languages and solution languages in a machine readable format. In the following this vocabulary will be referred as **Se**mantic **Pa**ttern and **So**lution **Re**pository - *SePaSoRe* vocabulary.

7.2.1 Taxonomy of Pattern Types

The complete taxonomy of pattern types defined by the *SePaSoRe* vocabulary is illustrated in Figure 7.1. There, all ellipses represent classes and all edges represent *rdfs:subClassOf* predicates. The most general pattern type is defined by the class Pattern which is a direct subclass of owl:Thing. All other classes that define pattern types are direct or indirect subclasses of the class Pattern, which tells a machine reader, that all individuals of those classes are also individuals of the class Pattern. Moreover the concept of a general class Pattern that classifies all pattern types, guarantees the extensibility of the knowledge base defined by the vocabulary. The defined pattern types can be extended in other vocabularies by defining new pattern types that are subclasses of the class Pattern. For example a vocabulary that define pattern types in the domain of costumes in films, may extend the pattern types described by the Pattern Language Vocabulary by defining a class CostumePattern as subclass of the class Pattern.

7.2.2 Property Restrictions on Pattern Types

As mentioned in Section 7.1, the cloud computing patterns follow a certain format that must be adhered to. To define the format of pattern types in OWL, we used the concept of

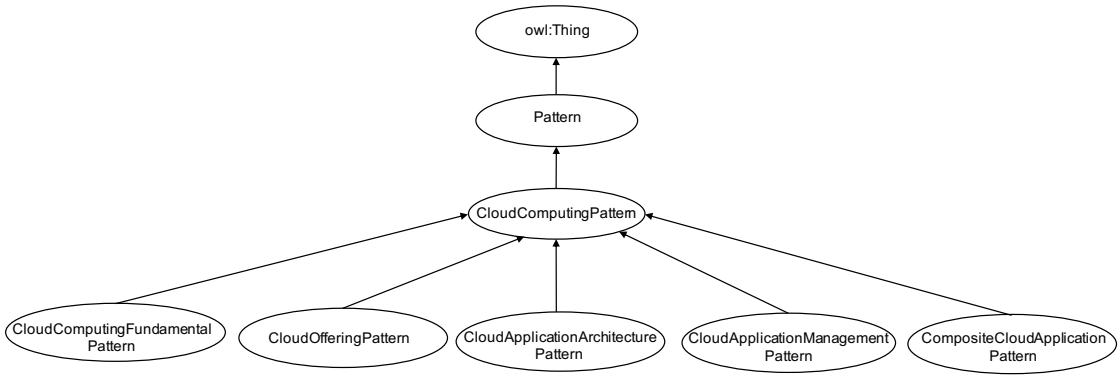

Figure 7.1: Taxonomy of the classes that define pattern types in the *SePaSoRe* Vocabulary, all edges represent rdfs:subCLassOf predicates

property restrictions on classes. Listing 8 shows an excerpt of the vocabulary that defines property restrictions in form of cardinality constraints. For sake of brevity, only a subset of the property restrictions are shown in Listing 8. There, we specify CloudComputingPattern to be a class with exactly one patternName, exactly one intent, and exactly one context. The property constraints defined on classes had been used in the implemented prototype to create a flexible pattern editor in form of a GUI that gets automatically configured based on the pattern type of the instance should be created. For example, the editor for a cloud computing pattern has one input field for the pattern name, one for the intent of the pattern, one for the context, and so forth.

```
<owl:Class rdf:ID="CloudComputingPattern">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#patternName"/>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#intent"/>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#context"/>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    ...
  </owl:Class>
```

Listing 8: Excerpt of the *SePaSoRe* vocabulary, that defines an class CloudComputingPattern and a set of property restrictions in form of cardinality constraints

7.2.3 Pattern Relation Descriptor (PRD)

Figure 7.2 illustrates the taxonomy of PRD types defined by the *SePaSoRe* vocabulary, where all ellipses represent classes and all edges represent rdfs:subClassOf predicates. The *SePaSoRe* vocabulary defines the root class PatternRelationDescriptor. In addition the vocabulary defines subclasses that define specific types of PRDs classified by the semantic relationship described by the PRD. In order to express the direction of the relationship, the vocabulary defines the object properties *hasSource* and *hasTarget*, whose domain is represented by PatternRelationDescriptor and range is represented by Pattern, as well as the object properties *isTargetOf* and *isSourceOf* whose domain is represented by Pattern and range is represented by PatternRelationDesctiptor. In addition, property restrictions in form of cardinality constraints had been used, which define that the class PatternRelationDescriptor has exactly one property *hasSource* and exactly one property *hasTarget*.

7.2.4 Concrete Solution Descriptor (CSD)

To describe the concept of a Concrete Solution Descriptor the vocabulary defines an OWL class ConcreteSolutionDescriptor. Additionally the vocabulary defines the object property *implementsPattern*, whose domain is represented by ConcreteSolutionDescriptor and range is represented by Pattern. In order to link an individual of ConcreteSolutionDescriptor to

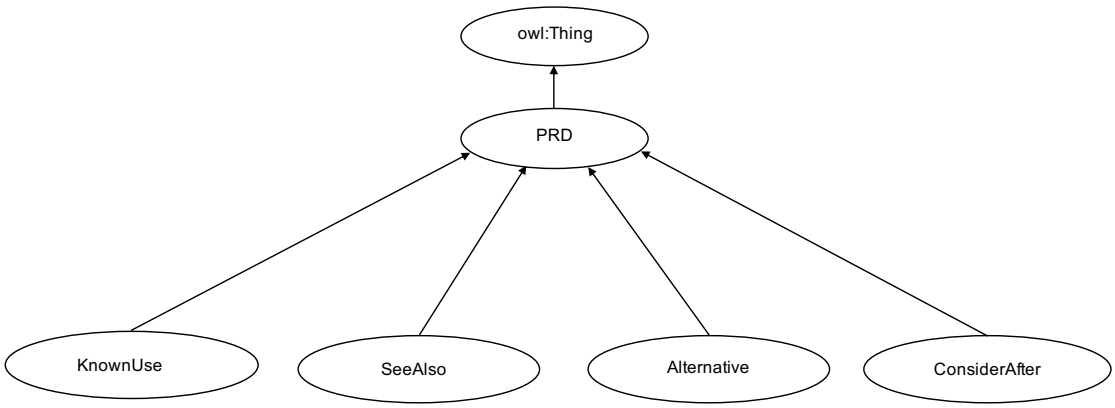

Figure 7.2: Taxonomy of the classes that define types of PRDs in the *SePaSoRe* Vocabulary, all edges represent rdfs:subCLassOf predicates

an concrete solution artifact the vocabulary defines the datatype property *hasCSArtifact*, whose domain is represented by ConcreteSolutionDescriptor and range is represented by xsd:String. We decided to represent the property *hasCSArtifact* as datatype property, due to fact that a concrete solution artifact is most likely not described as an OWL individual. Moreover it must not be a digital artifact, but can also be an physical artifact. The *hasCSArtifact* as defined in the vocabulary provide a means to link any location descriptor of a concrete solution artifact, that can be described in form of a xsd:string, to an ConcreteSoluionDescriptor. In addition, property restrictions in form of cardinality constraints had been used, which define that the class PatternRelationDescriptor has exactly one property *implementsPattern* and exactly one property *hasCSArtifact*.

7.2.5 Concrete Solution Relation Descriptor (CSRD)

Figure 7.3 illustrates the taxonomy of CSRD types defined by the *SePaSoRe* vocabulary. There all ellipses represent classes and all edges represent rdfs:subClassOf predicates. The vocabulary defines the root class CSRD. In addition the vocabulary define the two specific types CSRDAggregation and CSRDAlternative as subclass of the general type CSRD. The purpose of individuals of CSRDAggregation is to describe an aggregation between two concrete solutions. To link the concrete solutions that need to aggregated, the vocabulary defines the object property *aggregatesCS*, whose domain is represented by CSRDAggregation and range is represented by ConcreteSolutionDescriptor. In addition, the vocabulary defines the datatype properties *precondition* and *aggregationDescription*, whose domain is represented as CSRDAggregation and range is represented as xsd:string. Those properties can be used to describe additional information in form of human readable text. For example, they can be used to describe preconditions that need to be fulfilled and different aggregation steps that need to be done.

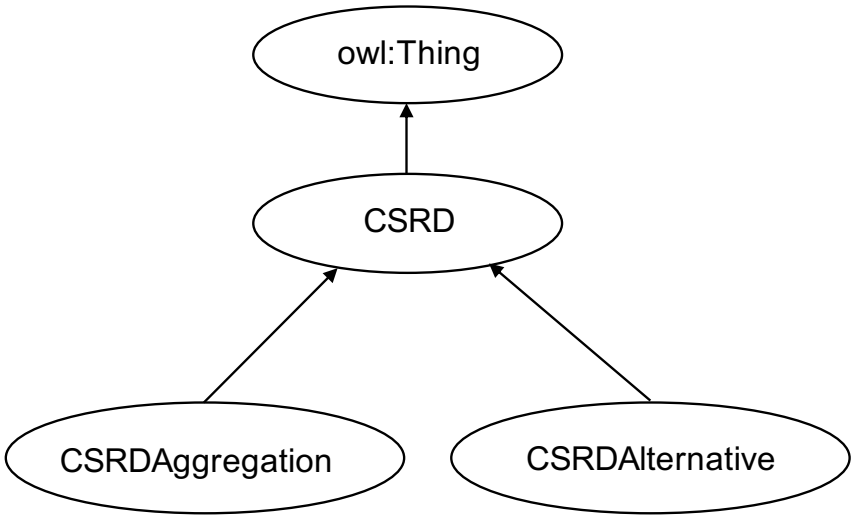

Figure 7.3: Taxonomy of the classes that define types of CSRDs in the *SePaSoRe* Vocabulary, all edges represent rdfs:subCLassOf predicates

7.3 *SePaSoRe*

This section describes the prototypical implementation of *SePaSoRe*. First it describes the technology decisions that has been made to implement the prototype. Subsequent it describes the software architecture of the prototype.

7.3.1 Technology Decisions

This section describes the technology decisions that had been made in the context of the prototypical implementation of *SePaSoRe*. First this section describes the technology decisions made for the prototypical implementation of the *SePaSoRe* Service. Subsequent it describes the technology decisions made for the prototypical implementation of the *SePaSoRe* Client.

SePaSoRe Service

The prototype of the *SePaSoRe* Service had been developed in Java. As an application framework, we decided to use the Spring Framework[1] in order to build a Java-based web application. Further, in the context of rapid implementation of the prototype with minimal configuration efforts, we decided to use Spring Boot[2] to create a standalone,

[1] https://spring.io/
[2] https://projects.spring.io/spring-boot/

self-contained Spring based Application. In the context of build automation and package management, we decided to use Apache Maven[3] as build automation tool, as well as to manage dependencies. In the context of implementing semantic web and linked data functionalities such as retrieving and processing RDF data, we decided to use the Java framework Apache Jena[4]. Apache Jena is composed of different APIs that interact together and provide a rich set of functionalities to process RDF data.

We decided to use the technologies Java, the Spring Framework and Apache Maven because the development team had existing knowledge and experience in using them. Further, we decided to use Apache Jena because it provides stable and rich functionalities for processing RDF data.

SePaSoRe Client

The prototype of the *SePaSoRe* Client had been developed in TypeScript, which is a typed superset of JavaScript[5] that compiles to plain JavaScript. In the context of developing a single-page application that runs in the browser, we decided to use Angular[6] as an web application framework. In the context of providing a convenient and responsive design of the graphical user interface, we decided to use the CSS framework Twitter Bootstrap[7]. In the context of package management, we decided to use npm[8] as a package manager for JavaScript libraries. In the context of providing SPARQL functionalities in the browser, we decided to use the JavaScript library rdfstore-js[9], which is a implementation of an RDF graph store with support of SPARQL query language.

7.3.2 Software Architecture

This section describes the software architecture of the prototype. This section uses the C4 model to describe and communicate the software architecture of the prototype at different levels of detail, starting from a high-level system context into a more detailed container view up to a decomposition of the components into containers [Bro15].

The system context is shown in Figure 7.4. The prototype allows users to browse and edit pattern and solution languages. Thereby it uses a GitHub repository as data store for the create files. We decided to limit the file types in our prototyping to files that are RDF documents that contain informations about patterns, concrete solutions and their relations. The prototype communicates via HTTP with the GitHub API and does create, read, update, and delete (CRUD) operations on the files stored in the repository. Further,

[3]https://maven.apache.org/
[4]https://jena.apache.org/
[5]https://developer.mozilla.org/bm/docs/Web/JavaScript
[6]https://angular.io/
[7]https://getbootstrap.com/
[8]https://www.npmjs.com/
[9]https://github.com/antoniogarrote/rdfstore-js

the prototype reads and interprets RDF data from the semantic web. The semantic web describes the totality of linked data in form of RDF data on the web, this also includes the RDF data hosted on the GitHub repository. Thereby the RDF data that the prototype consumes distinguish between two types. First it reads and analyzes the classes, properties and relations described in the *SePaSoRe* vocabulary. Second it crawls an RDF Graph that describes information about a pattern and solution language in form of linked data.

The containers of the prototype are illustrate in Figure 7.5. The prototype consists of the *SePaSoRe* Service and the *SePaSoRe* Client as described in Chapter 6. The *SePaSoRe* Service has been implemented as a standalone and self-contained Java application based on Spring Boot. Moreover it uses the semantic web framework Apache Jena to read and analyze RDF data. The *SePaSoRe* Client has been implemented as JavaScript and Angular single-page web application. It uses HTTP as a communication protocol to use the functionalities provided by the *SePaSoRe* Service. In the following we will decompose those containers further and describe the major structural building blocks in form of components.

SePaSoRe Service

Zooming into the *SePaSoRe* Service identifies the three components (i) REST Controller, (ii) Vocabulary Service, (iii) Crawler Service as illustrated in Figure 7.6:

The **REST Controller**, which is a Spring REST Controller, implements a RESTful HTTP API. The API provides the following interfaces that exposes functionalities of the *SePaSoRe* Service:

api/getVocabulary expects an URL of an OWL vocabulary as parameter and returns the classes, properties and their relationships as JavaScript Object Notation (JSON) data.

api/getRDFGraph expects a list of URLs of RDF data that act as seed to crawl an RDF graph from distributed sources. It returns the crawled RDF Graph.

The **Vocabulary Service**, which is a Spring Service, implements the functionalities to read and analyze an OWL vocabulary. It uses functionalities provided by the Jena Ontology API to extract all the classes, properties and relations contained in a given OWL vocabulary into a JSON data model.

The **Crawler Service** is implemented as a Spring Service. It uses functionalities of the Apache Jena core RDF API to read an analyze RDF data. It crawls RDF data from distributed sources, that describes information about pattern, concrete solutions and their relations. The crawled graph represents represents a pattern and solution language. To crawl all relevant RDF documents, the crawler proceeds as follows: (i) the Crawler Service starts with a list of URLs that references RDF documents, (ii) the Crawler Service retrieves the triples contained in the documents and identifies all objects represented as URL references that had not been visited yet and adds them to a list of URLs to visit. This process is recursively applied to the list of URLs to visit until no new URLs are found.

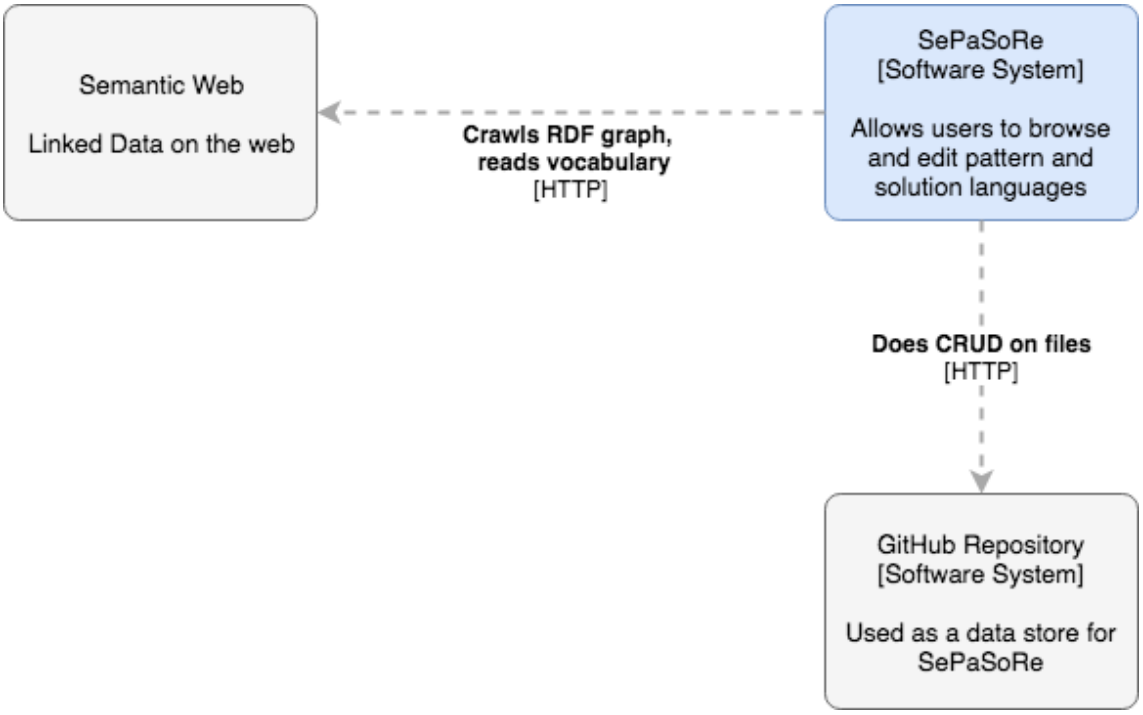

Figure 7.4: System Context diagram of *SePaSoRe*

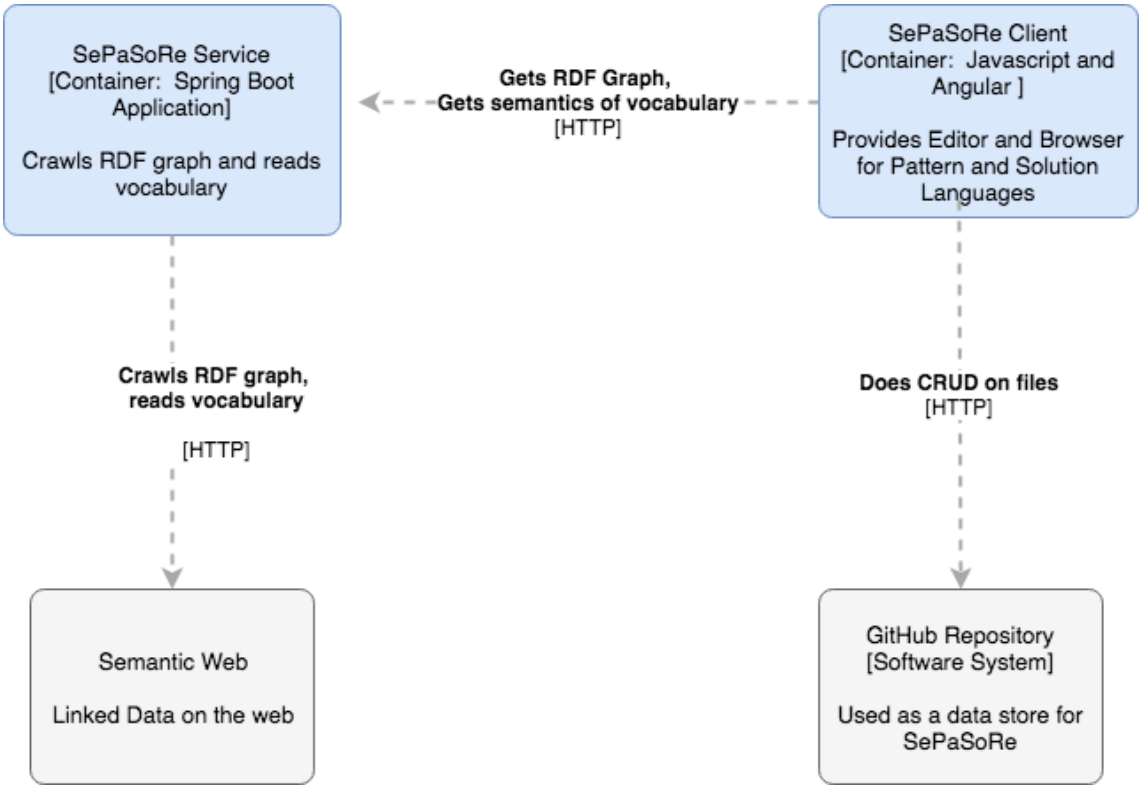

Figure 7.5: Container diagram of *SePaSoRe*

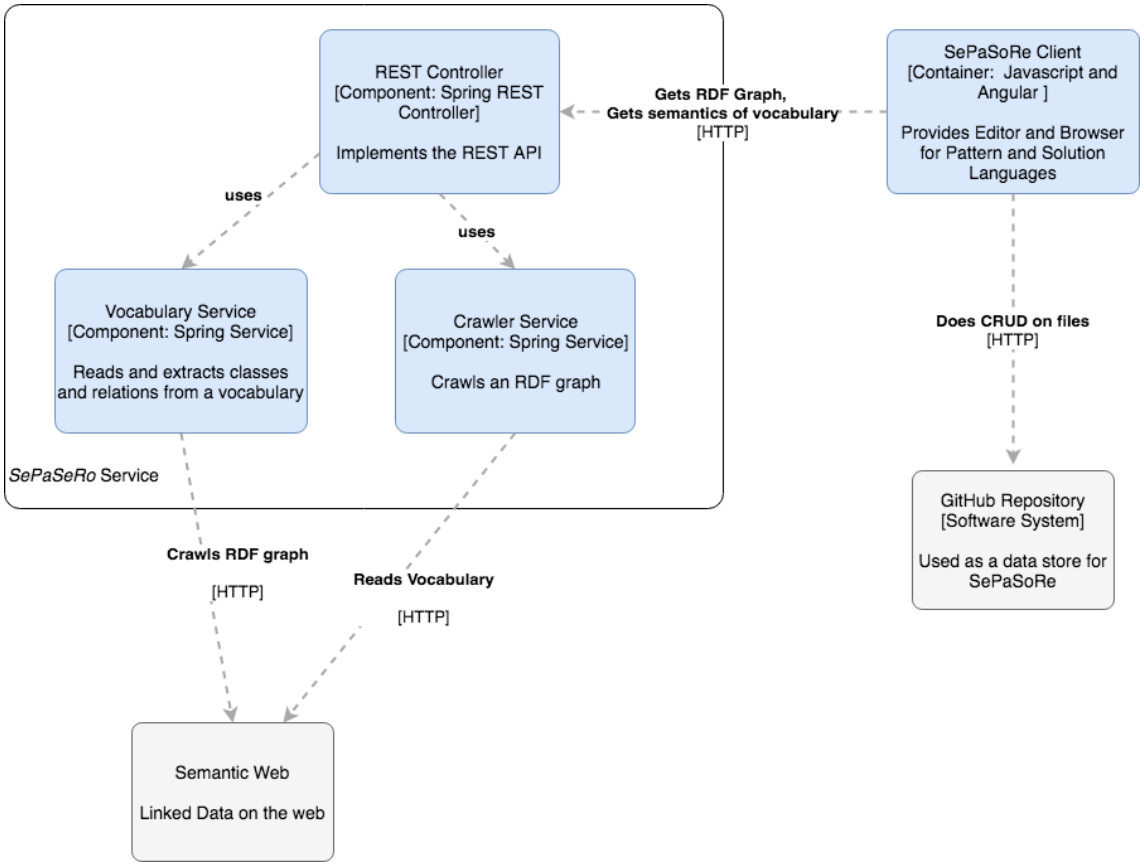

Figure 7.6: Component diagram of *SePaSoRe* Service

SePaSoRe Client

Zooming into the *SePaSoRe* Client identifies the components (i) HTTP Service, (ii) GitHub Service, (iii) RDFStore Service, (iv) Editor Component, (v) Browser Component:

The **HTTP Service** is a Angular Service, which implements functionalities to perform HTTP request against the API of the *SePaSoRe* Service. The functionalities provided by the HTTP Service are used by the Editor Component and Browser Component to retrieve information about RDF data.

The **GitHub Service** implements functionalities to perform HTTP requests against the API provided by GitHub. It implements the functionalities to create, read, update and delete files on a specified GitHub repository.

The **RDFStore Service** implements the logic to query specific data from an RDF graph. It uses the JavaScript library rdfstore-js to store an RDF graph in the browser. Further the library is used to perform SPARQL queries against the RDF graph to retrieve RDF data based on given parameters.

The **Editor Component** implements the logic for authoring and relating pattern and concrete solution documents. It provides its functionalities via a graphical user interface in

which a user can edit the information of the pattern via Markdown editor that provides features for formating and highlighting text. We decided to use Markdown as a markup language due to its easy-to-read and easy-to-write syntax. Moreover it can be directly converted into HTML and embedded into a web page. Relations between pattern and concrete solutions can be create via a graphical user interface, that provides a set of available relation types via a dropdown menu. A search field allows the user to search for the document that needs to be linked. The available pattern types and relationship types that can be create are derived from the classes and properties described in the *SePaSoRe* vocabulary. In addition, the Pattern Component implements logic to transform the pattern or concrete solution document created by a user into individuals of the classes described in the *SePaSoRe* vocabulary that represent and express the information in form of linked RDF triples.

The **Browser Component** implements the logic for visualizing the information contained in the RDF graph, received from the *SePaSoRe* Service. It processes the RDF data, extracts the information and creates human browsable pattern and concrete solution documents. Further it provides a graphical user interface that allows users to search patterns based on search parameters. In order to retrieve patterns based on the search parameters, the Browser Component uses the SPARQL functionalities provided by the RDFStore Service.

8 Conclusion and Outlook

The work proposed a concept of using Semantic Web Standards to describe information about patterns, concrete solutions and their relations in form of linked data on the web, which forms a distributed machine-readable representation of pattern and solution languages. Further, the work proposed a concept of an IT-based pattern and solution repository for publishing authoring and browsing pattern languages on the semantic web, introduced as *SePaSoRe*. *SePaSoRe* provides a graphical user interface that abstracts the underlying technologies of the semantic and allows users to publish and browse information about of patterns, concrete and their solutions in form of RDF data.

The use cases and requirements that *SePaSoRe* must provide are derived from former research in the area of pattern language repositories as well as from the assignment tasks as formulated by the proposal of the thesis. *SePaSoRe* should provide a means to define semantic links between pattern and concrete solution documents that are not only human readable but also processable by machines (FR1). Moreover, it should be possible provide detailed documentation about relationships of patterns and concrete solutions, which is directly attached to the relationship (FR2). In the context of implementing the vision of living networks of patterns, patterns and concrete solutions must be not stored centrally but can be distributed over various locations and authored by different people (FR3) (FR4). The fact that a large amount of patterns makes it hard for users to find patterns by hand, motivates the representation of patterns in a machine processable form that allows machine-based retrieval of patterns (FR5).

Chapter 5 argues that the fact that pattern languages and solution languages consist of knowledge that is shared across a community makes them by nature distributed and inconsistent. Therefore, it proposes the use of Semantic Web Standards to create a formal and machine processable representation of pattern languages and solution languages that can be shared and reused across software systems. Thereby the section showed how OWL can be used to create a vocabulary that defines concepts and relationships needed to describe and represent pattern languages in form of machine-readable RDF data.

Chapter 6 shows the architecture of *SePaSoRe* consisting of the *SePaSoRe* Service hosted as a stateless service and the *SePaSoRe* Client running as a single-page application in the browser. In the context of collaboration and version control of documents it proposes to utilize GitHub repositories as data store and hosting platform for the files created by *SePaSoRe*.

In the end Chapter 7 describes how a prototypical implementation of *SePaSoRe* has been made. In the context of the implementation an OWL vocabulary, introduced as *SePaSoRe* vocabulary, has been created. The *SePaSoRe* vocabulary formally define different types

of patterns and related concepts that had been described in Chapter 5, such as types of Pattern Relation Descriptors, Concrete Solution Descriptors, and Concrete Solution Relation Descriptors, that can be used to encode RDF data with semantic metadata.

Currently the a build version of the *SePaSoRe* Client and the *SePaSoRe* vocabulary is hosted on a GitHub repository. A single instance of *SePaSoRe* Service is hosted in the cloud. As future work, research should be done, to show, if the functionalities of the *SePaSoRe* Service could be also implemented by the *SePaSoRe* Client. This would allow running the software entirely in the browser.

Further future work could test the functionalities of *SePaSoRe* in real-world use cases to gather feedback about the functionalities from the pattern community.

Bibliography

[Ale77] C. Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977 (cit. on pp. 13, 17, 25, 29, 35).

[Ale79] C. Alexander. *The timeless way of building*. Vol. 1. New York: Oxford University Press, 1979 (cit. on pp. 13, 17).

[BFM98] T. Berners-Lee, R. Fielding, L. Masinter. "RFC 2396: Uniform resource identifiers (URI): Generic syntax, August 1998." In: *Status: Draft Standard* (1998) (cit. on p. 46).

[BHL01] T. Berners-Lee, J. Hendler, O. Lassila. "The semantic web." In: *Scientific american* 284.5 (2001), pp. 34–43 (cit. on pp. 18–20).

[BL15] J. Barzen, F. Leymann. "Costume Languages as Pattern Languages." In: *Proceedings of PURPLSOC (Pursuit of Pattern Languages for* (2015), pp. 88–117 (cit. on pp. 26, 35, 45).

[BL17] J. Barzen, F. Leymann. "Patterns as Formulas: Patterns in the Digital Humanities." In: (2017) (cit. on p. 26).

[Bri14] D. B. Brian McBride R.V. Guha. *RDF Schema 1.1*. 2014. URL: https://www.w3.org/TR/rdf-schema/ (cit. on p. 21).

[Bro15] S. Brown. "Visualise, document and explore your software architecture." In: *Coding the Architecture* (2015) (cit. on p. 63).

[Con04] T. W. W. W. Consortium. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. 2004. URL: https://www.w3.org/TR/rdf-concepts/ (cit. on p. 18).

[Con08] T. W. W. W. Consortium. *Mikael Nilsson, Andy Powell*. 2008. URL: http://dublincore.org/documents/dc-rdf/ (cit. on p. 38).

[Con15a] T. W. W. W. Consortium. *Query*. 2015. URL: https://www.w3.org/standards/semanticweb/query (cit. on p. 23).

[Con15b] T. W. W. W. Consortium. *Semantic Web*. 2015. URL: https://www.w3.org/standards/semanticweb/ (cit. on p. 18).

[Con15c] T. W. W. W. Consortium. *Vocabularies*. 2015. URL: https://www.w3.org/standards/semanticweb/ontology (cit. on pp. 21, 38).

[Cun+06] I. Cunningham et al. "Portland pattern repository." In: *http://c2. com/ppr/* (2006) (cit. on p. 27).

[Dan14] L. M. Dan Brickley. *FOAF Vocabulary Specification*. 2014. URL: http://xmlns.com/foaf/spec/ (cit. on pp. 19, 38).

[DE05] J. Dietrich, C. Elgar. "A formal description of design patterns using OWL." In: *Software Engineering Conference, 2005. Proceedings. 2005 Australian*. IEEE. 2005, pp. 243–250 (cit. on p. 27).

[DE07] J. Dietrich, C. Elgar. "Towards a web of patterns." In: *Web Semantics: Science, Services and Agents on the World Wide Web* 5.2 (2007), pp. 108–116 (cit. on pp. 27, 37).

[DE13] B. Di Martino, A. Esposito. "Towards a common semantic representation of design and cloud patterns." In: *Proceedings of International Conference on Information Integration and Web-based Applications & Services*. ACM. 2013, p. 385 (cit. on p. 27).

[DEC15] B. Di Martino, A. Esposito, G. Cretella. "Semantic representation of cloud patterns and services with automated reasoning to support cloud application portability." In: *IEEE Transactions on Cloud Computing* (2015) (cit. on p. 27).

[FBB+14a] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. "Efficient pattern application: validating the concept of solution implementations in different domains." In: *International Journal on Advances in Software* 7 (2014) (cit. on p. 26).

[FBB+14b] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. "From pattern languages to solution implementations." In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. 2014, pp. 12–21 (cit. on pp. 26, 43).

[FBB+16] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, H. Schulze, L. Ostwestfalen-Lippe. "Leveraging pattern application via pattern refinement." In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC)*. 2016, pp. 38–61 (cit. on p. 26).

[FBFL14] C. Fehling, J. Barzen, M. Falkenthal, F. Leymann. "PatternPedia-collaborative pattern identification and authoring." In: *Proceedings of PURPLSOC (Pursuit of Pattern Languages for Societal Change). The Workshop*. 2014, pp. 252–284 (cit. on pp. 14, 27).

[FBL] M. Falkenthal, U. Breitnebücher, F. Leymann. "The Nature of Pattern Languages." In: (cit. on pp. 14, 25, 33, 42).

[Feh17] C. Fehling. *Cloud Computing Patterns*. 2017. URL: http://www.cloudcomputingpatterns.org/ (cit. on p. 14).

[FL17] M. Falkenthal, F. Leymann. "Easing Pattern Application by Means of Solution Languages." In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. 2017, pp. 58–64 (cit. on pp. 18, 26, 34, 45–47).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8 (cit. on pp. 13, 18, 32, 41, 42).

[FSMI13] T. Furukawazono, S. Seshimo, D. Muramatsu, T. Iba. "Survival Language: A Pattern Language for Surviving Earthquakes." In: *Proceedings of the 20th Conference on Pattern Languages of Programs*. PLoP '13. Monticello, Illinois: The Hillside Group, 2013, 30:1–30:13. ISBN: 978-1-941652-00-8. URL: http://dl.acm.org/citation.cfm?id=2725669.2725705 (cit. on p. 13).

[Gam95] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995 (cit. on pp. 18, 45).

[GHJV93] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design patterns: Abstraction and reuse of object-oriented design." In: *European Conference on Object-Oriented Programming*. Springer. 1993, pp. 406–431 (cit. on pp. 27, 34).

[Hee09] U. van Heesch. *Open Pattern Repository*. 2009. URL: http://www.cs.rug.nl/search/ArchPatn/OpenPatternRepository (cit. on p. 27).

[HTT+09] T. Hey, S. Tansley, K. M. Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*. Vol. 1. Microsoft research Redmond, WA, 2009 (cit. on p. 14).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004 (cit. on p. 13).

[IM10] T. Iba, T. Miyake. "Learning Patterns: a pattern language for creative learning II." In: *Proceedings of the 1st Asian Conference on Pattern Languages of Programs*. ACM. 2010, p. 4 (cit. on p. 13).

[KGT05] A. Knöpfel, B. Gröne, P. Tabeling. "Fundamental modeling concepts." In: *Effective Communication of IT Systems, England* (2005) (cit. on p. 51).

[KISV16] C. Köppe, P. S. Inventado, P. Scupelli, U. Van Heesch. "Towards extending online pattern repositories: supporting the design pattern lifecycle." In: *Proceedings of the 23rd Conference on Pattern Languages of Programs*. The Hillside Group. 2016, p. 15 (cit. on pp. 27, 29, 34).

[M D05] M. S. M. Duerst. "RFC 3987: Internationalized Resource Identifiers (IRIs)." In: (2005) (cit. on p. 46).

[MH04] D. McGuinness, F. van Harmelen. *OWL Web Ontology Language Overview*. 2004. URL: https://www.w3.org/TR/owl-features/ (cit. on pp. 22, 23).

[Mul] K. Mullet. "Structuring pattern languages to facilitate design." In: Citeseer (cit. on p. 25).

[PS+06] E. Prud, A. Seaborne, et al. "SPARQL query language for RDF." In: (2006) (cit. on p. 23).

[RS96] F. B. R. M. H. Rohnert, P. S. M. Stal. "Pattern-oriented software architecture: A system of patterns." In: *Jogn Wiley & Sons* (1996) (cit. on p. 13).

[RU04] J.-M. Rosengard, M. F. Ursu. "Ontological representations of software patterns." In: *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. Springer. 2004, pp. 31–37 (cit. on pp. 14, 26).

[Sch08] D. Schuler. *Liberating voices: A pattern language for communication revolution*. MIT Press, 2008 (cit. on p. 13).

[Van08] M. Van Welie. "Pattern Library for Interaction Design." In: *Website]. URL: http://www. welie. com* (2008) (cit. on p. 14).

[VKZ13] M. Völter, M. Kircher, U. Zdun. *Remoting patterns: foundations of enterprise, internet and realtime distributed object middleware*. John Wiley & Sons, 2013 (cit. on pp. 14, 33).

[WMS04] C. Welty, D. L. McGuinness, M. K. Smith. *OWL Web Ontology Language Guide*. 2004. URL: https://www.w3.org/TR/owl-guide/ (cit. on pp. 21, 38, 41).

[Zdu07] U. Zdun. "Systematic pattern selection using pattern language grammars and design space analysis." In: *Software: Practice and Experience* 37.9 (2007), pp. 983–1016 (cit. on pp. 13, 25).

[ZKV04] U. Zdun, M. Kircher, M. Volter. "Remoting patterns: design reuse of distributed object middleware solutions." In: *IEEE Internet Computing* 8.6 (2004), pp. 60–68 (cit. on pp. 14, 32, 42).

All links were last followed on March 17, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature