An Ontology-based Approach for Software Architecture Recommendations

Full Paper

Manoj Bhat
Technische Universität München
manoj.mahabaleshwar@tum.de

Klym Shumaiev
Technische Universität München
klym.shumaiev@tum.de

Andreas Biesdorf
Siemens AG – Corporate Technology
andreas.biesdorf@siemens.com

Uwe Hohenstein
Siemens AG – Corporate Technology
uwe.hohenstein@siemens.com

Michael Hassel
Siemens AG – Corporate Technology
michael.hassel@siemens.com

Florian Matthes
Technische Universität München
matthes@tum.de

Abstract

The design and development of sustainable software systems require software architects to consider a variety of architectural solutions and their trade-offs. With the frequent introduction of new architectural methods and software solutions, as well as, due to time-to-market constraints faced by software architects, considering even a subset of alternative architectural solutions during the decision-making process is a challenge. In this paper, we propose a recommendation system that automatically annotates architectural elements in software architecture documents and then proposes a) alternative architectural solutions for the annotated elements and b) concrete software solutions to realize an architectural design decision. These annotations and recommendations are derived from the knowledge captured in a publicly available cross-domain ontology. The evaluation of the recommendation system indicates that our approach can effectively support software architects to consider alternative architectural solutions while making architectural design decisions.

Keywords

Architectural design decisions, software architecture recommendations, cross-domain ontology.

Introduction

Software design is complex and knowledge intensive (Babar 2009). During the design process, software architects take diverse architectural design decisions (ADDs) that have a far-reaching impact on the functional and non-functional properties of software systems. These ADDs include the selection of architectural styles, design patterns, software components, and also the underlying software stack and IT infrastructure. These decisions result in the architectural blueprint of the system. Hence, software architecture is considered as a set of architectural design and ADDs (Jansen and Bosch 2005).

The ISO 42010 standard (ISO 2011) describes a meta-model to capture ADDs. An ADD addresses stakeholders' concerns and may raise new concerns. An ADD is made by an architect by considering multiple alternative solutions and by analyzing the trade-offs between their strengths and weaknesses (Zimmermann et al. 2007). To capture the architectural knowledge (AK), variants of the ISO 42010 model have been successfully applied in research as well as in industries (Babar et al. 2009). These models form the core of AK management (AKM) tools that enable architects to manage AK within an organization.

One of the major challenges in the domain of AKM is to provide adequate tool support for architects to consider alternative architectural solutions during the decision-making process (Alexeeva et al. 2016). Furthermore, the challenge of identifying alternative solutions to address a specific concern is non-trivial. Due to architects' involvement in multiple projects within an organization, they find it difficult to explore and analyze even a subset of alternative architectural solutions (Hohpe et al. 2016). Moreover, due to

time-to-market constraints faced by architects, it is practically difficult for them to keep pace with the emergence of new architectural methods and technologies on a daily basis. Therefore, AKM tools should ease the process of alternative architectural solution identification, their prioritization, and documentation by providing recommendations to architects within their working environment.

To support architects in managing AK, several ontology-based approaches have been proposed in the past (Akerman and Tyree 2006; Ameller and Franch 2011). One of the critics of using ontologies proposed in these approaches for reasoning and recommendations is that they rely on populating and maintaining the ontologies (Bense et al. 2016), that is, creating instances of concepts and their relationships. On the other hand, over the past several years, broad general-purpose cross-domain ontologies have evolved at a rapid pace (for e.g., Dublin Core, Yet Another Great Ontology, DBpedia) through open community-driven efforts. For instance, the DBpedia ontology (Mendes et al. 2012) is a cross-domain ontology, which is based on the most commonly used infoboxes within Wikipedia articles. It contains more than 685 classes, 2795 different properties, and over 4.2 million instances[1]. Even though these ontologies were not created with the focus on AKM, they capture a wide variety of architectural methods, reference architectures, design patterns, and software systems.

Herein, we propose a novel approach embodied in a recommendation system that uses the DBpedia ontology to support architects during the decision-making process. Within this approach, we first describe how to automatically annotate natural language text with architectural elements. Second, we demonstrate how to generate two types of recommendations: 1) software solutions to realize an ADD and 2) alternate architectural solutions (architectural style, pattern, and software technology). Finally, we evaluate and discuss how both these types of recommendations can support architects in the decision-making process.

Related Work

In his seminal work (Kruchten 2004), Kruchten proposes an ontology to capture the ADD as a first-class concept. He introduces the taxonomy of ADDs, its attributes, as well as its relationship to concepts such as requirements, defects, design, and implementation elements. The benefit of such an ontology is that it preserves the complex graphs of interrelated design decisions and supports use cases such as reasoning and recommendations to support software architects during the decision-making process.

Moreover, there exists a large body of knowledge that uses ontologies to manage AK (Akerman and Tyree 2006; Ameller and Franch 2011; Ramaiah et al. 2007). For instance, Ameller and Franch (2011) present an ontology named Arteon for AK representation. This ontology aims to provide the building blocks of architectural views, frameworks, and elements to build the structural aspects of a software architecture. It should be noted that Ameller and Franch (2011) describe only the concepts within the Arteon ontology and consider the population of individuals (instances) as part of their future work.

Akerman and Tyree (2006) propose an ontology-based approach to support software development with the focus on ADDs and its associated concepts such as architectural assets and stakeholder concerns. Within this approach, authors present the process of creating the concepts, individuals, and relationships to support different case studies. However, authors do not provide detailed information on the number of individuals and their relationships in the ontology and this ontology is not publicly available for reuse.

The aforementioned works solely focus on capturing the concepts and their relationships in an ontology. They do not focus on the populating individuals within an ontology. We believe that in the AKM domain, reasoning and recommendation use cases is feasible only if the ontology captures the concepts of the domain as well as contains a significant number of individuals along with their relationships. Furthermore, unless an ontology evolves by incorporating new architectural methods and software solutions, the applicability of ontology-based AKM tools will be limited. With this regard, Ramaiah et al. (2007) propose a novel semi-automatic approach for populating the ontology. The concepts related to software architecture and their instances are automatically identified and extracted by parsing the Wikipedia articles into an ontology. Such an ontology complements the approach proposed in this paper and can be used as an alternative to the DBpedia ontology which is also derived from Wikipedia.

[1] http://wiki.dbpedia.org/services-resources/ontology. Last visited on 10.01.2017.

Furthermore, several approaches have also been proposed to guide architects with AK. Van der Ven and Bosch (2013) propose a model to reason about ADDs by analyzing version management repositories. Esfahani et al. (2013) describe a framework that supports architects to explore the architectural solution space under uncertainty and in making best possible choices. However, as discussed in the recent state of the art literature review (Capilla et al. 2016), only a few AKM tools partially support AK reasoning.

To the best of authors' knowledge, none of the existing approaches focuses on using existing broad cross-domain ontologies to support architects during the decision-making process. In following sections, we propose a recommendation system that uses the DBpedia ontology to recommend alternative architectural solutions that could be considered by architects for analysis and documentation purposes.

Software Architecture Recommendation Approach

We build our approach on our previous work on a meta-model based framework for architectural knowledge management named AMELIE (Bhat et al. 2016). AMELIE stands for Architecture Management Enabler for Leading Industrial softwarE. It provides a platform to support architects during a project's architecture lifecycle. The extensible architecture of this platform allows creating new recommendation services for the end-users. In this paper, we focus only on two specific recommendations that rely on using cross-domain ontologies for architectural decision support.

Type I is the recommendation of software solutions to support an ADD. The variety of software solutions to support an ADD is not always available to software architects. Guiding an architect during the design phase through recommendations of available software solutions will improve the possibility that an architect will consider multiple alternatives before making an ADD. For instance, if an architect realizes, that under project constraints most of the available software solutions to implement an ADD are proprietary, then (s)he can better re-negotiate with stakeholders before making the decision.

Type II is the recommendation of alternative solutions (architectural styles, patterns, and technologies) related to an ADD. Contrary to the previous type, where the aim is to recommend software solutions for higher-level concepts, here, alternative solutions belonging to the same abstraction level are proposed. During the process of architecture documentation, providing relevant alternatives allows architects to rationalize and reason about the ADDs over the available alternatives.

To prove the validity and to evaluate our approach, we have implemented the approach in a prototype. To enable researchers to repeat the results, we describe our approach in four consecutive phases below.

Phase 1: Automatic Annotation of Architectural Elements

During this phase, architects use a client[2] (web-based editor or Microsoft Word with a tailored plug-in) for writing software architecture documents. The client automatically tracks changes and submits written text to the application server. The first task of the server is to annotate text received from the clients. To this end, we use the Unstructured Information Management Architecture (UIMA) framework (Apache 2016). The annotator component within the application server comprises of three sub-components.

1) Sentence annotator: The input text received from the client is broken down into sentences using the sentence annotator. The result, which is a list of sentences, is then passed to the DBpedia annotator. This is indicated in Figure 1 using Steps 1.2 and 1.3.

2) DBpedia annotator: For annotating text with the concepts in the DBpedia ontology, we reuse the plug-in named DBpedia Spotlight implemented by Daiber et al. (2013). The DBpedia Spotlight performs two main tasks: phrase spotting and disambiguation (Step 1.4 in Figure 1). In the first step, the phrase-spotting algorithm identifies phrases that should be linked to DBpedia entities in the given text. These phrases are identified using a string-matching algorithm (Aho-Corasick 1975). In the second step, the identified phrases are scored using the TF-IDF weights and cosine similarity measure (cf. Daiber et al. 2013). All the phrases with a similarity score above a configurable threshold value are included in the result. The evaluation results presented in the following sections are based on a threshold value of 0.7.

[2] https://amelietor-9f8c3.firebaseapp.com/

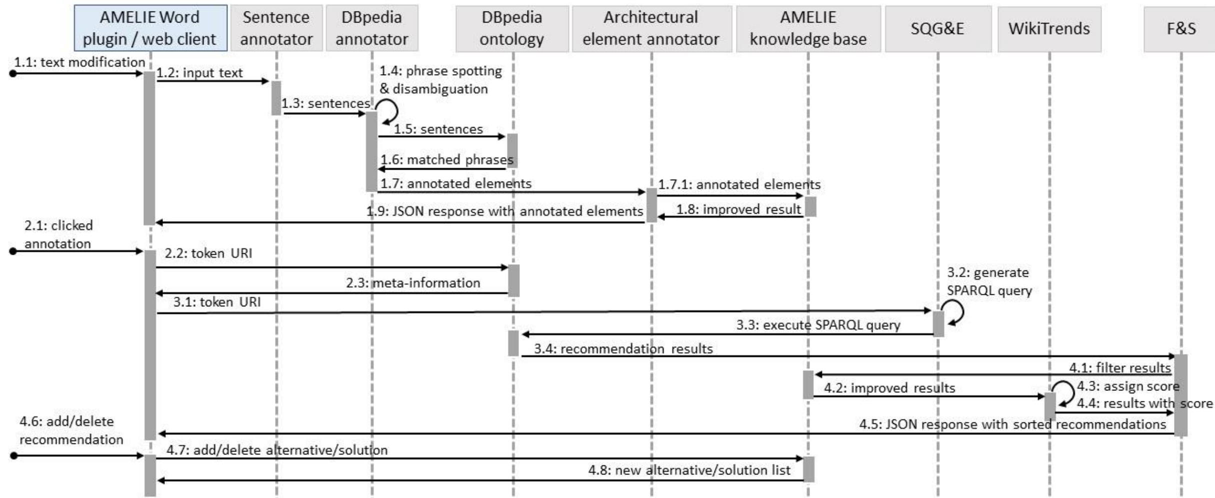

Figure 1. Sequence diagram illustrating an ontology-based software architecture recommendation approach

Since, we also propose a mechanism to prioritize the results, varying the threshold between 0.6 to 0.7 did not affect the results. The result from this annotator is further processed by the Architectural element annotator to generate a JavaScript Object Notation (JSON) array of annotations (Step 1.7 in Figure 1).

3) Architectural element annotator: To improve the precision of the annotated text results, we implemented an architectural element annotator, which restricts annotations received from the DBpedia annotator to those that correspond to an architectural element. A bag of words consisting of tokens captured in the knowledge base (KB) is used for filtering irrelevant annotations. As discussed in (Bhat et al. 2016), architects can update concepts within the KB at run-time. If the user marks specific annotations as irrelevant, then this annotator filters out such annotations. Alternatively, if the user adds a custom annotation, then the corresponding annotations are added to the result. This is indicated by Steps 1.7.1 and 1.8 in Figure 1. In the current prototype, basic string matching algorithm identifies such concepts. It should be noted that, since this component reflects an organization-specific artifact that can be maintained to improve the performance, we do not consider this component during the evaluation.

The final response is sent back to the client as a JSON array (Step 1.9 in Figure 1.). The client application shown in Figure 2 uses this information for highlighting the annotated text. A color-coding scheme easily distinguishes different types of annotated elements (for example, Genre, Software, and Concept).

Phase 2: Retrieve Meta-information for the Annotated Architectural Elements

If the user is interested in a specific annotated architectural element (indicated by a mouse-click in the client), a new request along with the URI of the annotated element is sent to the application server. The look-up component in the server uses the URI to retrieve element's properties from the DBpedia ontology. The properties include *dbo:abstract*, *dct:subject*, *owl:sameAs*, *rdfs:comment*, *dbo:wikiPageID*, and *dbo:wikiPageExternalLink*. These properties within the DBpedia ontology (*dbo*) are derived from the upper ontologies including Dublin core (*dct*), Web Ontology Language (*owl*), Resource Description Framework Schema (*rdfs*), and Simple Knowledge Organization System (*skos*). The properties of the requested element are compiled into a JSON object and sent back to the client. In Figure 1, Steps 2.1, 2.2, and 2.3 reflect the above workflow. The meta-information related to the selected element is displayed on the right side of the user interface (UI) under the meta-information tab as shown in Figure 2.

Phase 3: Recommendation Support

To provide recommendations related to the architectural elements including architectural styles, patterns, and software solutions, we first manually analyzed the concepts and relationships between concepts within the DBpedia ontology. Understanding these concepts and their relationships is necessary for realizing the two specific types of aforementioned recommendations. We identified a subset of concepts

Figure 2. User interface of the recommendation system

within the DBpedia ontology, which are necessary for formulating the look-up queries (presented in Appendix A). Figure 3 presents the necessary concepts and their relationships.

skos:Concept - A fundamental element of the SKOS ontology that allows one to assert that a resource is a concept. A relationship (object property) R *rdf:type* C, indicates that the resource R is a type of concept C. Examples of concepts include “architectural patterns”, “software design patterns”, and “web application framework”. Furthermore, each concept can be hierarchically structured using the *skos:broader* relationship. For example, the concept “patterns” is a broader concept of “software design patterns”.

TopicalConcept is a class defined in the DBpedia ontology. It is the base class for concepts such as “genre”, “academic subject”, “mathematical concept”, “standard”, etc.

Genre is a subclass of TopicalConcept and allows one to capture the genre of a specific object. Instances of *Genre* include “relational database”, “graph database”, “service-oriented architecture”, etc. An instance of *Genre* is realized by multiple software solutions. That is, *Genre* has an inverse relationship with Software through the “is *dbo:genre* of" relation. Furthermore, a *Genre* can be related to a higher-level *Concept* through the *dct:subject* relationship. For example, *Genres* such as “graph database" and “column-oriented database” are related to the *Concept* “database models”.

Work is a class defined in the DBpedia ontology. It captures generic properties such as “creationYear”, “developer”, and “productionCompany” which are relevant for different kinds of work including “software”, “artwork”, “article”, and “book”.

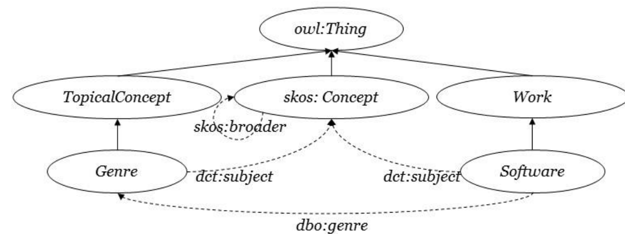

Figure 3. Subset of the DBpedia ontology

Software is a subclass of *Work* and has properties such as release date, programming language, license, and operating system. Software has two object properties, namely, *dbo:genre* and *dct:subject*. The domain of *dbo:genre* relation is *Software* and its range is *Genre*. For instance, the software “Neo4j” belongs to the “graph database” Genre. Similarly, the domain of *dct:subject* relation is *Software* and its range is *skos:Concept*. For example, “Neo4J” has a *dct:subject* relation with the “NoSQL” Concept.

We refer to the aforementioned concepts and their relationships for formulating the look-up queries and providing recommendations to software architects. The look-up queries are formulated using the SPARQL query language (Prud’Hommeaux et al. 2008) and executed by the SPARQL query generator and executor (SQG&E) component (cf. Figure 1). To deliver the two aforementioned recommendations, we designed the queries described in the following sub-sections.

Type I: Recommend Software Solutions to Realize an ADD

To support the recommendation of software solutions, two different SPARQL queries are used to handle the relevant types -- *Genre* and *Concept*. If the annotated architectural element is an instance of *Genre*, then the query in Listing 1.A is executed. We do not illustrate the scenario with the instance of *Concept*, as it results in a similar SPARQL query. In the SPARQL query, **variable* is replaced by the respective element in the client (for example, http://dbpedia.org/resource/Relational database). As shown in Figure 2, recommendations related to the selected element are shown on the right panel of the UI.

The query in Listing 1.A is executed against the DBpedia ontology using the Apache JENA framework (Jena 2015). This query aggregates the results of three sub-queries wherein each sub-query extracts software resources for a given *Genre*. For instance, the results for the *Genre* "Relational database" include the URIs of *Software* such as "SQLite", "MySQL", "MariaDB", etc. Each URI is further processed to retrieve meta-information including description and license and sent back to the client application.

Sub-query 1: Retrieves *Software* resources related to a *Genre* (e.g., "Relational database") using the *dbo:genre* relationship. The result of the Select statement is a set of URIs of the software resources (indicated by *?software*). **Sub-query 2:** This sub-query focuses on "*Genre dct:subject Concept*" and "*Software dct:subject Concept*" relationships (cf. Figure 3). For a *Genre*, first, all the *Concepts* are retrieved and then the *Software* resources related to each of the *Concept* are extracted. **Sub-query 3:** This sub-query ensures that the *Software* resources corresponding to similar *Genres* are extracted. For a *Genre*, first, its related *Concepts* are retrieved using the *dct:subject* relationship. Subsequently, for each of the *Concept*, all the *Genres* and their corresponding *Software* resources are retrieved. This sub-query can also be extended using the *skos:broader* relation for a wider set of results.

Type II: Recommend Alternative Architectural Solutions Related to an ADD

To retrieve alternative architectural solutions (architectural styles, patterns, and technologies) from the DBpedia ontology, the SPARQL query in Listing 1.B comprising of two sub-queries is executed. **Subquery 1:** If the annotated element is an instance of *Genre* then the first Select clause of the SPARQL query in Listing 1.B is applicable. For instance, if the annotated element is "Relational database" (a *Genre*), then the output of the query includes "Key-value database", "Column-oriented DBMS", and "Document-oriented database" as alternatives. Note that the **variable* is replaced by the annotated element in the SPARQL query. The query first checks if the resource is of type *Genre*, then it retrieves all its associated *Concepts* using the *dct:subject* relation. Next, using the inverse relationship it retrieves all the other *Genres* of the identified *Concepts*. **Sub-query 2:** This sub-query returns alternative solutions for concrete software solutions. It navigates the graph structure of the DBpedia ontology to retrieve S*oftware* resources belonging to the same *Genre*. That is, if the annotated element is "MongoDB", then the result (*?alternative*) comprises of "CouchDB", "OrientDB", and "ArangoDB".

The workflows involved for both the recommendation types are indicated in Figure 1 using Steps 3.1 to 3.4. To improve the precision of recommendations related to alternative styles and design patterns, we composed another SPARQL query with a predefined filter. The logic of this query is similar to the previous queries, that is, it involves graph navigation using the *dct:subject* object property. This query is shown in Listing 1.C. For instance, corresponding to an architectural style such as "multi-tier architecture" the retrieved results include "service-oriented architecture", "microservices", and "entity component system". This illustrates a look-up query that restricts the results using a regular expression.

Phase 4: Filtering and Sorting Recommendation Results

Corresponding to the recommendations for each annotated element, a confidence score is maintained in the AMELIE KB. To address the cold-start problem in our recommendation system, we generate a confidence score using the Wiki Trends[3] ratings. An average of the trend score for the past five years on a monthly basis is computed and assigned to the confidence score. The results are then sorted by a descending confidence score before presenting to the users. Steps 4.1 to 4.4 in Figure 1 reflect the current

[3] http://www.wikipediatrends.com/

Documents	(#) Manual annotations	(#) Automatic annotations	(#) Irrelevant annotations	(%) Precision	(%) Recall	F-score
Doc 1	29	24	2	91.66	75.86	0.83
Doc 2	39	35	3	91.42	82.05	0.86
Doc 3	31	33	8	75.75	80.64	0.78
Doc 4	49	47	4	91.49	87.75	0.84

Table 1. Results - Annotation of architectural elements

workflow. Furthermore, as shown in Figure 2, users also have an option to either accept or reject recommended results corresponding to each annotated element. Subsequently, the confidence score is incremented or decremented in the KB. Recommendations with a confidence score below a configurable threshold value are filtered out and not shown to the users. Furthermore, we also use the KB consisting of annotated elements, recommendations, and confidence scores as a cache for improving the performance of the recommendation system. Again, it should be noted that we only consider the system generated confidence score during the evaluation. However, the recommendations and its ordering can be improved using such a preference model within organizations.

Evaluation

To evaluate our approach, we conducted an empirical study to directly observe and quantitatively analyze the results of the recommendation system. As part of the evaluation setup, we selected four chapters with the following headings “Technical/IT infrastructure” and “Technical Decisions” from four different software architecture documents (aligned to a company-specific blueprint) produced as part of the large ongoing software engineering project in the data analytics domain. All documents are used for internal and external communication by four different sub-teams (7-10 people) out of a total number of people (more than 100 people) involved in the projects. The evaluation was carried out in two phases.

Phase 1: Evaluation of Architectural Elements Annotator

Two software architects with more than five years of experience manually annotated the documents. Each architect independently color-coded all four documents. After that, in a shared meeting these architects merged their annotations. Annotations that were not accepted by both the architects during the meeting have been excluded and the final number of manual annotations has been counted.

To get the count of automatic annotations, the same four parts of software architecture documents were subsequently uploaded to the AMELIE web client and the input text was annotated by the DBpedia annotator component. Some of the annotated architectural elements include “N-tier”, “client/server”, “middleware”, “Java”, “C Sharp”, “Hadoop”, “Junit”, and “MySQL”. Table 1 shows the comparison of results between the manual and automatic annotation of architectural elements in the documents. Overall, 148 **non-repetitive** architectural elements were manually color coded by experts. In comparison, the DBpedia annotator with a precision (fraction of automatically retrieved architectural elements that are relevant) of 87.58 %, recall (fraction of relevant architectural elements that were successfully retrieved) of 81.57 %, and F-score (harmonic mean of the precision and recall (Rijsbergen 1979)) of 0.84 automatically annotated 139 **unique** architectural elements. The results of Doc 3 (cf. Table 1) are least accurate since this document contains terms related to organizational names and acronyms used in different context, the precision is comparatively low (75.75 %). Moreover, since the document contains architectural terms used specifically within the organization (e.g., custom communication protocol, internal software system names) the recall is also low (80.64 %).

Nevertheless, the above results indicate that the general architectural elements can be extracted from documents using the publicly available DBpedia ontology. Subsequently, other useful meta-information including description and link to the Wikipedia article that are related to the annotated elements can also be extracted and presented to architects during the software architecture documentation process.

	Correct recommendations	Total recommendations	Precision (%)
Type I	228	283	80.56
Type II	306	379	80.73

Table 2. Results - Recommendation of software and alternate solutions

Annotated concept	Software solutions
Web container	Apache Tomcat, Apache Geronimo, Enhydra, GlassFish, JBoss Application Server, Jetty
File system	IBM General Parallel File System, Extended file system, File Allocation Table, Amazon S3
Object-oriented	Smalltalk, Ruby, Java, C++, C Sharp, Visual Basic.NET, Objective-C, TypeScript, Lava
Relational database	PointBase, TimesTen, FrontBase, DatabaseSpy, MaxDB, MySQL, MariaDB, WebScaleSQL
NoSQL	EXist, BaseX, Neo4j, Elliptics, LevelDB, FoundationDB, DocumentDB, C-treeACE

Table 3. Results - Software solutions

Phase 2: Evaluation of the Recommendation System

The two architects mentioned before manually evaluated the results corresponding to the recommendation of software solutions (Type I) and alternate architectural solutions (Type II). The evaluation was carried out using the web client. For the evaluation, architects considered only the top ten recommendation results. This restriction was applied not only due to the impracticability of evaluating all the results in a considerable amount of time but also due to the lack of user interest in the hits positioned below the tenth position (cf. Caphyon. 2016). Furthermore, since it is not feasible to enumerate all relevant recommendations for each of the annotated element, we have not considered the recall score but only the precision at ten (P@10). The results of both the recommendations are shown in Table 2.

Corresponding to Type I recommendations, 29 architectural elements were annotated as either a *Genre* or a *Concept*. Some of these high-level concepts include “Web container”, “File system”, and “Relational database”. Table 3 shows a subset of the recommended software solutions for realizing some of the ADDs. The overall precision of the Type I recommendations is 80.56%. This result positively supports our first hypothesis that software solutions to realize an ADD can be extracted from the DBpedia ontology.

For Type II recommendations, all the annotated concepts were analyzed. These concepts, for instance, included “Apache Tomcat”, “Java”, “PostgreSQL”, and “Maven”. Some of these recommendations are shown in Table 4. The precision of Type II recommendation result is 80.73%. During the evaluation, it was observed that the recommendations related to annotated concepts representing software systems were better as compared to concepts related to architectural styles or design patterns. This is mainly because software technologies are considerably well maintained in the DBpedia ontology as compared to some of the higher-level concepts. Particularly, in the DBpedia ontology concepts such as “Object-oriented”, “Hypertext Transfer Protocol”, or “Unified Modeling Language” belong (*rdf:Type*) to the concept *Thing*, which is a broad concept. Furthermore, these concepts also inherit properties from other broad concepts through the *dct:subject* relationship. Hence, the results related to these elements are typically incorrect. Due to such scenarios, it becomes necessary to limit the results of the SPARQL queries to avoid performance issues in the recommendation system. However, the drawback of enforcing such a limitation is a decreased accuracy of the recommendations.

The aforementioned results positively support our second hypothesis. That is, alternatives for ADD (especially technology related ADD) can be extracted from the DBpedia ontology. However, further investigation is required to extract recommendations related to architectural styles and design patterns.

Limitations and Threats to Validity

In this section, we present four identified limitations of our approach and subsequently propose possible solutions to addresses these limitations as part of our future work.

Limitation 1: In our approach, we do not consider the context for the recommendations. The context must be identified based on how users interact with the system. If the user accepts an annotated element,

Annotated concept	Alternative solutions
N-tier	Service-oriented architecture
CORBA	D-Bus, Internet Communications Engine, DCOM, XPCOM, IBM System Object Model
HDFS	Google File System, Ceph, Amazon S3, Amazon S3, NFS, Microsoft Azure's file system
Teradata	Salesforce.com, Oracle Exadata, MonetDB, HPCC, Scriptella, Sybase IQ, Apatar
Junit	CsUnit, TestNG, Selenium, Mockito, JTriger
Model-view-controller	Pipeline, Presentation-abstraction-control

Table 4. Results - Alternative architectural solutions

then this confirmation must be used for the recommendations that follow within the same document. For instance, if the user up-votes the term "Java" then the context of the programming language must be set a priori. That is, if the term "JUnit" follows within the document, then the alternative software solutions should correspond to Java. This can be accomplished by exploring the *dbo:programmingLanaguage* relationship between "work" and "programming language" in the DBpedia ontology.

Limitation 2: The relationships indicated by the dotted arrows in Figure 3 represent many-to-many relationships. In the SPARQL queries, we do not ignore the concepts that could lead to false positives. Prioritizing as well as removing some of the related concepts before executing the SPARQL queries will improve the recommendations. By identifying semantic similarity of concepts in the domain and range of a relationship, it is possible to provide a ranking mechanism to investigate the associated concepts.

Limitation 3: The accuracy of the recommendations relies on the DBpedia ontology itself. For instance, it is possible that a newly introduced software solution is not updated in the DBpedia ontology and hence is not reflected in the recommendations. A feedback loop from the users of the system should address this limitation, by adding new concepts to the DBpedia ontology directly.

Limitation 4: For the evaluation, only four software architecture documents were considered. We realize that the corpus is not large enough and the recommendation system needs to be evaluated with a larger number of software architecture documents as part of the future work.

Conclusion and Future Work

In this paper, we proposed an approach for reusing the knowledge captured in the broad cross-domain DBpedia ontology to a) highlight architectural elements in software architecture documents, b) recommend alternative architectural solutions, and c) recommend software solutions for realizing an ADD. The core benefit of our approach lies in using an already existing ontology that is well-maintained through open-community driven efforts.

On decision-making theories, Herbert Simon (1996) pointed out that "Most theories of decision making start out with a given set of alternatives and then ask how to choose among them. In design, by contrast, most of the time and effort is spent in generating the alternatives, which aren't given at the outset." Our approach aligns with the aforementioned issue and generates a set of alternatives from the existing knowledge. These alternatives should further support the decision-making process. Furthermore, Herbert Simon also suggested, "The idea that we start out with **all** the alternatives and then choose among them is wholly **unrealistic**". Limitation 3 in the previous section emphasizes this point. Nevertheless, we believe that suggesting even a subset of alternative architectural solutions might encourage software architects to reason about the selection of architectural choice compared to its alternatives.

At last, though our initial evaluation of the recommendation system together with our industry partner indicates the feasibility of our approach, further investigation with larger corpus of software architecture documents would strengthen our claims. Hence, with the feedback loop through the user preference repository, the recommendations can be further improved. As future work, we plan to address the limitations documented above and to share our experiences as part of the lessons learned.

REFERENCES

Aho, A. V., and Corasick, M. J. 1975. "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, 18(6), pp. 333-340.

Akerman, A., and Tyree, J. 2006. "Using ontology to support development of software architectures," *IBM Systems Journal* (45:4), IBM, pp. 813–825.

Alexeeva, Z., Perez-Palacin, D., and Mirandola, R. 2016. "Design decision documentation: a literature overview," in *Software Architecture: 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28--December 2, 2016, Proceedings 10*, pp. 84–101.

Ameller, D., and Franch, X. 2011. "Ontology-based architectural knowledge representation: structural elements module," in *International Conference on Advanced Information Systems Engineering*, pp. 296–301.

Apache, UIMA. 2016. "Unstructured information management applications," *URL: http://uima.apache.org*.

Babar, M. A. 2009. "Supporting the software architecture process with knowledge management," in *Software Architecture Knowledge Management*, Springer, pp. 69–86.

Babar, M. A., Dingsøyr, T., Lago, P., and van Vliet, H. 2009. *Software architecture knowledge management*,

Springer.

Bense, H., Benjamin, G., Hoppe, T., Hemmje, M., Humm, B. G., Schade, U., Schäfermeier, R., Paschke, A., Schmidt, M., Haase, P., Siegel, M., Vogel, T., and Wenning, R. 2016. "Emerging trends in corporate semantic web," *Informatik-Spektrum* (39:6), pp. 474–480 (doi: 10.1007/s00287-016-0998-x).

Bhat, M., Shumaiev, K., Biesdorf, A., Hohenstein, U., Hassel, M., and Matthes, F. 2016. "Meta-model based framework for architectural knowledge management," in *Proceedings of the 10th ECSA Workshops*, p. 12.

Caphyon. 2016. "CRT study," *URL: https://www.advancedwebranking.com/cloud/ctrstudy/*

Capilla, R., Jansen, A., Tang, A., Avgeriou, P., and Babar, M. A. 2016. "10 years of software architecture knowledge management: practice and future," *Journal of Systems and Software* (116), Elsevier, pp. 191–205.

Daiber, J., Jakob, M., Hokamp, C., and Mendes, P. N. 2013. "Improving efficiency and accuracy in multilingual entity extraction," in *Proceedings of the 9th International Conference on Semantic Systems*, pp. 121–124.

Esfahani, N., Malek, S., and Razavi, K. 2013. "GuideArch: guiding the exploration of architectural solution space under uncertainty," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 43–52.

Hohpe, G., Ozkaya, I., Zdun, U., and Zimmermann, O. 2016. "The software architect's role in the digital age," *IEEE Software* (33:6), pp. 30–39 (doi: 10.1109/MS.2016.137).

ISO, M. 2011. "ISO/IEC/IEEE Systems and software engineering -- architecture description," *ISO/IEC/IEEE 42010:2011(E)*, pp. 1–46 (doi: 10.1109/IEEESTD.2011.6129467).

Jansen, A., and Bosch, J. 2005. "Software architecture as a set of architectural design decisions," in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pp. 109–120.

Jena, A. 2015. "A free and open source Java framework for building semantic web and linked data applications," *Available online: jena.apache.org/ (last accessed on 28 April 2015)*.

Kruchten, P. 2004. "An ontology of architectural design decisions in software intensive systems," in *2nd Groningen workshop on software variability*, pp. 54–61.

Mendes, P. N., Jakob, M., and Bizer, C. 2012. "DBpedia: a multilingual cross-domain knowledge base," in *LREC*, pp. 1813–1817.

Prud'Hommeaux, E. et al. 2008. "SPARQL query language for RDF," *W3C Recommendation* (15).

Ramaiah, M. S., Prabhakar, T. V, Rambabu, D., and others. 2007. "ArchVoc--Towards an ontology for software architecture," in *Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent, 2007. SHARK/ADI'07: ICSE Workshops 2007. Second Workshop on*, p. 5.

Rijsbergen, C. J. 1979. "v.(1979)," *Information Retrieval* (2).

van der Ven, J. S., and Bosch, J. 2013. "Making the right decision: supporting architects with design decision data," in *European Conference on Software Architecture*, pp. 176–183.

Zimmermann, O., Gschwind, T., Küster, J., Leymann, F., and Schuster, N. 2007. "Reusable architectural decision models for enterprise application development," in *International Conference on the Quality of Software Architectures*, pp. 15–32.

Appendix A

```
SELECT DISTINCT ?software WHERE {{
 SELECT ?software WHERE {
 ?software dbo:genre *variable .
 ?software ns:type dbo:Software }
} UNION { SELECT ?software WHERE {
 *variable dct:subject ?concept .
 ?software dct:subject ?concept .
 ?software ns:type dbo:Software }
} UNION { SELECT ?software WHERE {
 *variable dct:subject ?concept .
 ?genre dct:subject ?concept .
 ?genre ns:type dbo:Genre .
 ?software dbo:genre ?genre .
 ?software ns:type dbo:Software }}}
```

(A)

```
SELECT DISTINCT ?alternative WHERE {
*variable dct:subject ?concept .
?alternative dct:subject ?concept .
FILTER(regex(?concept, ``pattern'',
``i''))}
```

(C)

```
SELECT DISTINCT ?alternative WHERE {
{
 SELECT ?alternative WHERE {
 *variable ns:type dbo:Genre .
 *variable dct:subject ?concept .
 ?alternative dct:subject ?concept .
 ?alternative ns:type dbo:Genre }
} UNION {
 SELECT ?alternative WHERE {
 *variable ns:type dbo:Software .
 *variable dbo:genre ?genre .
 *variable dct:subject ?subject .
 ?alternative dbo:genre ?genre .
 ?alternative dct:subject ?subject .
 ?alternative ns:type dbo:Software }
} }
```

(B)

Listing 1. (A) SPARQL query for retrieving software solutions (B) SPARQL query for retrieving alternative solutions (C) SPARQL query for retrieving alternative architectural styles and design patterns