A Semantic Web Based Approach for Design Pattern Detection from Source Code

Samad Paydar, Mohsen Kahani
Web Technology Lab., Dept. of Computer Engineering
Ferdowsi University of Mashhad
Mashhad, Iran
samad.paydar@stu-mail.um.ac.ir, kahani@um.ac.ir

***Abstract*—Design patterns provide experience reusability and increase quality of object oriented designs. Knowing which design patterns are implemented in a software is important in comprehending, maintaining and refactoring its design. However, despite the interest in using design patterns, traditionally, their usage is not explicitly documented. Therefore, a method is required to reveal this information from some artifacts of the systems (e.g. source codes, models, and executables). In this paper, an approach is proposed which uses the Semantic Web technologies for automatically detecting design patterns from Java source code. It is based on the semantic data model as the internal representation, and on SPARQL query execution as the analysis mechanism. Experimental evaluations demonstrate that this approach is both feasible and effective, and it reduces the complexity of detecting design patterns to creating a set of SPARQL queries.**

Keywords-design pattern; semantic web, source code, ontology, software analysis

I. INTRODUCTION

Design patterns describe common solutions for common design problems, and provide experience-reusability in the context of object oriented design. They lead to the development of more reliable, reusable, maintainable and comprehensible designs. Since their successful description and categorization in [8], they have attracted great attention in the domain of object oriented development.

Despite the interest in using design patterns, in most cases, their usage is not explicitly documented. Explicit specification of this information is important in comprehending, maintaining and refactoring an existing design. Different approaches have been presented for detecting the design patterns implemented in an existing design [9], [11].

Recently, the Semantic Web technologies (e.g. RDF and Ontologies) have been extensively used in the knowledge engineering domain. They make it possible to represent knowledge of a certain domain in a machine-processable format, and enable machines to provide more support for humans involved in the domain. In software engineering domain, the Semantic Web technologies have been used for different tasks [26] like software testing [27], software product line management [28], component selection [14], and configuration management [25].

This paper proposes a Semantic Web based approach for detecting design patterns from Java source code. The main elements of the approach are: an ontological representation of source code information, and a query-based mechanism for accessing and analyzing this information. Despite its simplicity, the experimental results demonstrate that the proposed approach can effectively reduces the cost and complexity of the design pattern detection task.

The rest of the paper is organized as follows: In Section II the related works are briefly reviewed. Section III describes the proposed approach. The experimental evaluations are discussed in Section IV. Finally, Section V concludes the paper.

II. RELATED WORK

Binkley [24] describes *source code analysis* as a process in which information of a program is automatically extracted from its source code or from another artifact which is generated from source code. He identifies three main components for source code analysis: *"the parser, the internal representation, and the analysis of this representation"*.

This paper deals with design pattern detection from source code, which is a source code analysis task. Therefore, it is interesting to discuss the related works in terms of the 3-component theoretical space proposed by Binkley [24].

A. Design pattern detection

Here, some of the works on detecting design patterns from source code are briefly described. A more comprehensive review of related methods is presented in [23].

Generally, there is a parser component in these methods which reads the source code and transforms it into a specific internal representation. Then, the analysis is performed by executing a special search method which checks the internal representation against a set of pre-defined descriptions of patterns.

In [1], the entry/exit listing of the object invocations is used as the internal representation to represent the extracted execution traces. Further, dynamic analysis techniques are employed to analyze the internal representation and find the patterns. Pattern Description Language (PDL) is used for specifying the design pattern descriptions.

DP-Miner [9] uses XML format as the internal representation, and the analysis is performed by a procedure which processes the matrices generated from the internal representation to find pattern instances.

In [12], a special type of graph called Refactoring Pattern (ReP) Graph is used as the internal representation and the analysis is performed in terms of a particular inference over the ReP graphs.

Tsantalis et al. [2] propose an approach which uses a matrix-based internal representation for storing inter-class relations of the input software. Design patterns are then detected by specialized similarity measurement between matrices of the input software and those of the design patterns descriptions.

A disadvantage of these methods, from the point of view of source code analysis, is that generally the internal representation and the analysis mechanism are both tailor-made for the specific task of design pattern detection, and they cannot be effectively reused for another task, i.e. code smell detection from source code. Further, these works are mostly based on a black-box view, since the logic of the analysis is hardcoded in a tool. This makes it hard to understand, verify and modify the detection mechanism.

B. Semantic Web Enabled Software Engineering

There are some works on utilizing the Semantic Web technologies to access or integrate information of different software artifacts [26]. Hartig et al. [14] address the problem of component selection in component-based software development. Their automated approach is based on using an ontology for describing software components and their dependencies. Having a repository that contains ontological descriptions of different components, they provide an algorithm for finding the best solution for the component search problem.

Alnusair and Zhao [15] similarly address the component search problem by developing some ontologies for describing source code elements and components. They provide different search methods (e.g. signature-based, keyword search) which utilize SPARQL query execution on the background.

Schueger et al. [16] present an approach which employs NLP techniques and a number of heuristics to automatically estimate quality attributes (e.g. certainty and reproducibility) of bug reports. Information of bug reports and their qualities are then represented using an ontology. As a result, it is possible to use SPARQL queries for searching over bug reports based on their quality attributes.

Durao et al. [17] presents an ontology-based approach for source code search. It uses ontologies and classification techniques to classify source code into different domain categories (e.g. GUI, IO, and Network). It also provides a semantic search facility for searching stored source code. A similar approach for semantic search over source code is presented by Keivanloo et al. [18].

Tappolet [19] presents a discussion and roadmap towards using ontologies for creating a general framework to tackle with three problems of effective software analysis: data representation, inter- and intra-project repository integration. Ideas mentioned in [19] are realized in [13] and [20].

The most similar work to the current paper is presented by Tappolet et al. [20], which is the extended version of [13]. In [20] the potential applications of the semantic web for software analysis is discussed. Their proposal includes a set of ontologies for representing source code, bug tracking and version control systems. In addition, they discuss the possibility of using iSPARQL [20] and SPARQL-ML [22] for implementing software analysis tasks in terms of queries.

They report experiments on using their proposed approach for five software analysis tasks: software evolution analysis, computing source code metrics, detection of code smells, defect and evolution density, and bug prediction.

Although the work of this paper has similarities with [20], it is different in the sense that, first, it evaluates a new task which is not considered in [20], i.e. design pattern detection. Second, while Tappolet et al. [20] evaluate applicability of their approach, this work evaluates also its effectiveness, by comparing it with another existing approach. Such a comparison is not considered by Tappolet et al [20]. Finally, this work doesn't have the limitation of [20], since it covers statement-level information of the source code, and generates much richer representations.

III. PROPOSED APPROACH

With regard to the Binkley's model [24], the proposed approach uses the RDFizer as the parser, the semantic data model as the internal representation, and SPARQL query execution as the analysis component.

The RDFizer parses the source code to gather interesting information, and represents this information using an ontology which describes different concepts of Java source code along with their relations to each other. The resulting RDF specification is stored in a repository which provides a facility for executing SPARQL queries on the stored RDF triples. The analysis task is performed through executing appropriate queries on the repository.

The main difference of the proposed approach with other design pattern detection methods is that it is based on the semantic data model. This data model has some advantages over ad-hoc formats (e.g. matrix-based representation):

1. RDF [3], along with RDFS [4] and OWL [5] provide an expressive data model based on description logic, powerful enough to clearly express instance data (ABox) and schema structure (TBox). In other words, both the data and the metadata are uniformly represented [6].
2. Well-defined semantics and formalism of these languages provides ontological reasoning and inference capabilities [6]. This capability is not inherently provided by other data models, e.g. relational data model, and therefore it must be implemented in the application layer.
3. Data is totally decoupled from the application logic that processes it. Therefore, it can be used by different applications.
4. The graph-based nature of RDF makes it a flexible data structure that can be extended by easily adding new

nodes and edges [7]. It means that both the data and its schema can be incrementally extended.

The main elements of the approach are described in the next sections.

A. Ontology

The underlying ontology is developed by extending the SIMILE Java2RDF ontology[1]. The original version of this ontology describes various structural features of Java source code, e.g. members of a class and parameters of a method, but it does not address the behavioral features, e.g. methods called, or classes instantiated by a method. Therefore it is extended by adding a number of properties and concepts, which for the sake of brevity, only some of them are mentioned below.

- The properties *constructorCount*, *methodCount*, *fieldCount*, are respectively used for expressing the number of the constructors, methods and fields of a class, and the property *parameterCount* is used for the number of the parameters of a method. It is worth noting that these properties are added for two reasons. First, SPARQL 1.0 does not support a SQL-like COUNT aggregate function. Although SPARQL 1.1 provides such a function, but it is not yet ratified as a W3C recommendation. Second, it was practically identified that advantages of explicitly stating this information (i.e. providing richer specifications, making queries simpler, and reducing complexity of query evaluation), is more than its disadvantages (i.e. increasing size of the specifications)
- The properties *calls*, and *isCalledBy*, are respectively used for stating methods called by, and methods calling a specific method.
- The property *instantiates* is added for specifying classes that are instantiated by a specific method
- The property *imports* is added for declaring classes which are imported in a Java source file
- The concepts *IfStmt*, *ThenStmt*, *SwitchCaseStmt*, *TryCatchStmt*, *ForStmt*, *WhileStmt*, *DoWhileStmt* are respectively used to represent instances of *if*, *then*, *switch-case*, *try-catch*, *for*, *while*, *do-while* statements

B. RDFizer

An RDFizer is developed which scans the input source code and generates the corresponding RDF representation based on the ontology mentioned in the previous section. It is implemented as a plug-in for the *Eclipse* IDE, and uses the Eclipse *Abstract Syntax Tree (AST)* API for parsing source code. Therefore it is able to access all the details of the source code, e.g. even the comments.

A small part of the source code of the *java.awt.Point* class, and parts of its RDF specification (in N3 serialization) is illustrated in Fig. 1 and Fig. 2.

[1] http://simile.mit.edu/java#

C. Repository

A *RepositoryManager* tool is implemented in Java which provides simple facilities for: creating a repository, adding RDF to it, and executing SPARQL queries over it. Since it is implemented using *Jena 2* API, it is possible to add inference rules to a specific repository. The *RepositoryManager* uses different *Jena* reasoners, e.g. *GenericRuleReasoner*, to perform reasoning.

IV. Experimental Evaluation

An experiment is conducted to evaluate the applicability of the proposed approach to the task of design pattern detection. The experiment is performed on a system with Intel Core2 Duo processor (2.26, 2.27 GHz), 4GB RAM, and 64-bit Windows Vista operating system.

The description of the experiment is as follows: Nine design patterns have been selected from different *GoF* categories as the subject patterns. Source code of the input software is given to the RDFizer to generate the corresponding RDF representation. The *RepositoryManager* is used to create a repository for storing the resulting RDF. Then, a query is created for the detection of each design pattern. Finally, each query is executed over the repository, and its results are considered as the instances of the corresponding design pattern.

In order to perform this experiment, first a sample dataset is prepared, and then the required queries are created. These are described next.

```
public class Point extends Point2D
   implements java.io.Serializable {
   public int x;
   public int y;
   public Point getLocation() {
          return new Point(x, y);
   }   …
}
```

Figure 1. A small part of the java.awt.Point source code.

```
obj:java_awt_Point          rdfs:label          "Point" ;
    a    java:Class ;
    java:modifier   java:public ;
    java:abstract   "false"^^xsd:boolean  ;
    java:extends     obj:java_awt_geom_Point2D ;
    java:constructorCount     "1"^^xsd:integer  ;
    java:method       obj:java_awt_Point_getLocation .
obj:java_awt_Point_getLocation_       rdfs:label
"getLocation" ;
    a     java:Method ;
    java:modifier     java:public ;
    java:abstract     "false"^^xsd:boolean  ;
    java:parameterCount       "0"^^xsd:integer  ;
    java:expression-type      obj:java_awt_Point ;
    java:instantiates      obj:java_awt_Point .
```

Figure 2. An excerpt from the RDF specification of java.awt.Point.

A. Dataset

The source code of three open-source Java projects, *JHotDraw5.1*, *JRefactory2.6.24*, and *JUnit3.7* are used as the input to the RDFizer, and the resulting RDF specifications are stored in a repository. Some statistics about this repository is presented in TABLE I.

B. Queries

Based on the specification of the patterns by *GoF*, each design pattern can be described by a set of statements, each stating some conditions or relations between different entities in the source code. If the ontology that is used for the semantic representation of source code is expressive enough for describing these statements, then it is possible to develop a query, in terms of that ontology, for detecting instances of the design patterns.

It was identified that the RDFizer and the underlying ontology are respectively powerful and expressive enough to enable creation of the required queries. However, since the description of different design patterns is not very strict, and there are variants for each pattern, to cover the most typical variants of a specific design pattern, usually it is needed to create queries which use one or two UNION operators, and hence are not short.

Due to space limitation, only the query associated with the *Decorator* design pattern is shown in Fig. 3 (prefix part is omitted for brevity).

C. Results Analysis

After executing each query on the repository, its results are compared with the results of the approach of Tsantalis et al. [2] which are generated by running the associated tool. TABLE II. reports the results of these two approaches.

As it is shown in this table, the proposed approach has been successful in detecting the instances of the design patterns. In many cases its results are identical to results of [2], while in a few cases there are differences. These cases are manually evaluated by two experts and the results are briefly described here.

The most noticeable difference is associated with detecting *State-Strategy* on *JHotDraw* and *JRefactory*. The proposed approach has detected 11 results on *JRefactory* which have not been detected by [2]. Manual evaluation indicated that they all are correctly detected.

Furthermore, the 3 results that are detected by [2], but not by the proposed approach, are correct too, i.e. the proposed approach has missed them.

TABLE I. SOME STATISTICS ABOUT THE SAMPLE REPOSITORY

Project	RDF Generation Time (ms)	RDF Storage Time (ms)	# of RDF Triples
JUnit	6704	4947	96873
JhotDraw	20985	8284	140877
Jrefactory	80543	18415	333346
Total	108232	31646	571096

```
SELECT DISTINCT ?component ?decorator ?field ?operation
WHERE {
{
?component     rdf:type  java:Interface .
?component     java:method     ?operation .
?operation  rdfs:label     ?operationLabel .
?decorator  java:assignable-to     ?component .
?decorator  java:field        ?field .
?field         java:expression-type      ?component .
?decorator  java:method     ?method1 .
?method1    rdfs:label     ?operationLabel .
?method1    java:calls     ?operation .
?concreteDecorator  java:assignable-to     ?decorator .
?concreteDecorator  java:method     ?method2 .
?method2    rdfs:label     ?operationLabel .
?method2    java:calls     ?method1
} UNION {
?component     rdf:type  java:Interface .
?component     java:method     ?operation .
?operation  rdfs:label     ?operationLabel .
?decorator  java:assignable-to     ?component .
?decorator  java:field        ?field .
?field         java:expression-type      ?component .
?decorator  java:method     ?method1 .
?method1    rdfs:label     ?operationLabel .
?method1    java:calls     ?otherMethod .
?otherMethod java:calls    ?operation .
?concreteDecorator  java:assignable-to     ?decorator .
?concreteDecorator  java:method     ?method2 .
?method2    rdfs:label     ?operationLabel .
?method2    java:calls     ?method1
} UNION {
?component     rdf:type  java:Interface .
?component     java:method     ?operation .
?operation  rdfs:label     ?operationLabel .
?decorator  java:assignable-to     ?component .
?decorator  java:field        ?field .
?field         java:expression-type      ?component .
?decorator  java:method     ?method1 .
?method1    rdfs:label     ?operationLabel .
?method1    java:calls     ?operation .
} }
```

Figure 3. The SPARQL query used for detecting Decorator design pattern.

The two *Prototype* instances which are detected by [2] and missed by the proposed approach, are better to be considered as instances of the *Composite* pattern.

By manually checking the instances which are missed by the proposed approach, it was identified that there are two reasons for this. The first reason is that the current implementation of the RDFizer has still some weaknesses. It does not yet support all the features of Java source code, e.g. generics or static initializers.

The second reason is that the implementation details of the design patterns are not fixed, and there might be slight difference between their documentation and implementation. Some of the [2]'s results that are missed by the proposed queries are due to different interpretation of the implementation aspects of the patterns. Unfortunately it is

not easy to judge [2]'s interpretation because its details are inaccessible for the authors.

Similar to all works in information retrieval domain, it is interesting to evaluate the results in terms of precision and recall. However, in the absence of an exact list of the design pattern instances of the three input projects, it is not possible to precisely measure recall of the proposed approach. It is possible to assess precision by manually checking the results. Such a manual verification was performed and it was shown that none of the detected results are incorrect, although they might differ from how another expert describes their related design patterns.

In order to evaluate the proposed approach in terms of its efficiency, its execution time is compared with that of [2]. The proposed approach includes three steps, and the execution time of these steps is separately calculated as T1, T2 and T3. These steps are:

1. Generating RDF from Java source files
2. Storing the RDF files in the repository
3. Executing the query on the repository

The sum of T1 and T2 is considered as the time required by the proposed approach to prepare the repository. Further, T3 can be considered as the detection time of this approach.

The approach presented in [2] has two main phases:

1. Preprocessing phase including system parsing, inheritance hierarchy detection and construction of system matrices
2. Detection of the design patterns using information gathered in the previous phase.

TABLE III. compares the preparation time of the proposed approach with the preprocessing time of [2]. Further, the detection time of the two approaches on the three sample projects is presented in TABLE IV.

Since [2] detects both State and Strategy design patterns simultaneously therefore their detection time is reported as a single value.

TABLE II. RESULTS OF DESIGN PATTERN DETECTION EXPERIMENT

#	Design Pattern	Project											
		JUnit				JHotDraw				JRefactory			
		A[a]	B[b]	C[c]	D[d]	A	B	C	D	A	B	C	D
1	Singleton	0	0	0	0	2	2	0	0	12	12	0	0
2	Composite	1	1	0	0	4	4	0	0	0	0	0	0
3	Observer	4	4	0	0	9	9	1	1	5	1	4	0
4	Decorator	1	1	0	0	28	28	0	0	0	0	0	0
5	Factory Method	1	0	1	0	6	4	2	0	3	3	0	0
6	Prototype	0	0	0	0	3	5	0	2	0	0	0	0
7	State-Strategy	8	7	1	0	49	49	4	4	45	38	11	3
8	Template Method	1	1	0	0	13	13	0	0	28	29	0	1

a. number of instances detected by the proposed approach

b. number of instances detected by the [2]

c. number of results of the proposed approach which are not detected by [2]

d. number of results of [2] which are not detected by the proposed approach

As shown in TABLE III. the preparation time of the proposed approach is much greater than the preprocessing time of [2]. There are different reasons for this. First, the I/O overhead of creating RDF files in the first step, and then reading and storing them in the repository in the second step increases the preparation time of the proposed approach. Since [2] does not make its intermediate representation permanent, and stores it as matrices in the main memory, it has better performance with regard to the preprocessing time.

As illustrated in TABLE IV. generally, the detection time of the proposed approach is better than that of [2].

This experiment demonstrates that the proposed approach for design pattern detection is both feasible and effective, since it generates good results in comparison with another existing method. Its detection time is also negligible compared to the opponent method. However there is a concern about the great preparation time of the proposed approach, and it might lead to the questions about the usability of this approach.

The answer to this question is that this high cost of preparation must be viewed along with the benefits of the proposed approach. The main point is that once this cost is paid and the repository is prepared, it can be used for performing different analysis tasks like those demonstrated in [20] and the one covered in this paper. Obviously, the approach used in [2] does not exhibit this reusability and application independence.

An important advantage of the proposed approach is that it supports ontological reasoning which can be used to simplify the detection process. This capability is utilized during the experiments. The developed RDFizer initially covered just direct method calls. It did not generate RDF triples for specifying indirect calls. However when creating a query for detecting state-strategy, it turned out that indirect method calls are also important.

There were two possible solutions for this: 1) modifying the RDFizer to generate required triples, and 2) adding a number of Jena rules to the repository so that it automatically infers indirect calls from direct calls.

TABLE III. PREPARATION/PREPROCESSING TIME OF THE APPROACHES

	proposed approach	Tsantalis et al. [2]
	Preparation Time (ms)	Preprocessing Time (ms)
JUnit	11651	140
JHotDraw	29269	312
JRefactory	98958	1513

TABLE IV. DETECTION TIME OF THE APPROACHES

#	Design Pattern	Project					
		JUnit		JHotDraw		JRefactory	
		A[1]	B[2]	A[1]	B[2]	A[1]	B[2]
1	Singleton	10	s	s	0	11	0
2	Composite	29	15	26	15	29	78
3	Observer	23	62	21	249	20	2715
4	Decorator	5	15	5	15	4	78
5	Factory Method	4	0	5	0	6	0
6	Prototype	3	46	4	280	3	2683
7	State-Strategy	19	109	20	280	20	3755
8	Template Method	3	0	2	0	2	0

The second option has been chosen by creating two rules shown in Figure 4. Rule1 says that if a method *method1* calls a method *method2*, and *method2* calls a method *method3*, then the consequence is that *method1* indirectly calls *method3*. It was learned that the capability of supporting reasoning is very important, since it reduces the complexities and costs of performing analysis tasks.

```
[rule1: (?method1 java:calls ?method2), (method2 java:calls
?method3) -> (?method1 java:indirectlyCalls ?method3)]

[rule2: (?method1 java:calls ?method2), (?method2
java:indirectlyCalls ?method3) -> (?method1
java:indirectlyCalls ?method3)]
```

Figure 4. Jena rules defined for detecting indirect calls

V. CONCLUSION

In this paper, a semantic web based approach is proposed for detecting design patterns from Java source code. It uses an RDFizer to generate semantic representation of source code, an RDF repository for storing these representations, and a query-based access mechanism for retrieving information from the source code. As the experimental results demonstrate, this approach is successful in effectively detecting instances of different design patterns in source code.

The main theme of future works is to use the proposed approach for other analysis tasks. For instance it is interesting to investigate to what extent this approach can be used for helping programmers in learning new APIs.

VI. REFERENCES

[1] L. Hu, K. Sartipi, Dynamic analysis and design pattern detection in Java programs, In: Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering, SEKE 2008, San Francisco Bay, 2008, pp. 842-846.

[2] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis, Design pattern detection using similarity scoring. IEEE Transactions on Software Engineering 32 (11) (2006) 896-909.

[3] RDF-Semantic Web Standards, http://www.w3.org/RDF/

[4] RDF Vocabulary Description Language 1.0: RDF Schema, http://www.w3.org/TR/rdf-schema/

[5] OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features/

[6] G. Antoniou, F.V. Harmelen, Web Ontology Language: OWL, In: S. Staab, R. Studer (eds.), Handbook of Ontologies, Springer, 2009, pp. 91-110.

[7] The RDF Advantages Page, http://www.w3.org/RDF/advantages.html

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison Wesley, 1995.

[9] J. Dong, D.S. Lad, Y. Zhao, DP-Miner: design pattern discovery using matrix, In: Proceedings of the 14th Annual IEEE International Conference on Engineering of Computer Based Systems, ECBS'07, Arizona, USA, 2007, pp. 371-380.

[10] A.D. Lucia, V. Deufemia, C. Gravino, M. Risi, A two phase approach to design pattern recovery, In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, CSMR '07, Amsterdam, The Netherlands, 2007, pp. 297-306.

[11] N. Shi, R.A. Olsson, Reverse engineering of design patterns from Java source code, In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, Washington, DC, 2006, pp. 123-134.

[12] T. Sharma, D. Janakiram, Inferring design patterns using the ReP graph, Journal of Object Technology 9 (5) (2010) 95-110.

[13] K. Kiefer, A. Bernstein, J. Tappolet, Mining software repositories with iSPARQL and a software evolution ontology, In: Proceedings of the 4th International Workshop on Mining Software Repositories (MSR), Minneapolis, MA, 2007.

[14] O. Hartig, M. Kost, J.C. Freytag, Automatic component selection with semantic technologies, In: Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, SWESE 2008, Karlsruhe, Germany, 2008.

[15] A. Alnusair, Tian. Zhao, Retrieving reusable software components using enhanced representation of domain knowledge, In: Recent Trends in Information Reuse and Integration, Lecture Notes in Computer Science (LNCS), Springer Verlag. (Accepted, to appear in 2011).

[16] P. Schuegerl, J. Rilling, P. Charland, Enriching SE ontologies with bug report quality, In: Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, SWESE 2008, Karlsruhe, Germany, 2008.

[17] F.A. Durão, T.A. Vanderlei, E.S. Almeida, S.R.L. Meira, Applying a semantic layer in a source code search tool, In: Proceedings of the 2008 ACM symposium on Applied computing, SAC '08, New York, NY, 2008, pp. 1151-1157.

[18] I. Keivanloo, L. Roostapour, P. Schugerl, J. Rilling, Semantic web-based source code search, In: Proceedings of the 6th International Workshop on Semantic Web Enabled Software Engineering, SWESE 2010, San Francisco, CA, 2010.

[19] J. Tappolet, Semantics-aware software project repositories, In: Proceedings of the 5th European Semantic Web Conference, ESWC'08, Ph.D. Symposium, Tenerife, Spain, 2008.

[20] J. Tappolet, C. Kiefer, A. Bernstein, Semantic web enabled software analysis, Journal of Web Semantics 8 (2010) 225-240.

[21] C. Kiefer, A. Bernstein, M. Stocker, The fundamentals of iSPARQL—a virtual triple approach for similarity-based semantic web tasks, In: Proceedings of the 6th International Semantic Web Conference (ISWC), 2007, pp. 295-309.

[22] C. Kiefer, A. Bernstein, A. Locher, Adding data mining support to SPARQL via statistical relational learning methods, In: Proceedings of the 5th European Semantic Web Conference, ESWC'08, Tenerife, Spain, 2008, pp. 478-492.

[23] J. Dong, Y. Zhao, T. Peng, A review of design pattern mining techniques, International Journal of Software Engineering and Knowledge Engineering 19(6), 2009, pp. 823-855.

[24] D. Binkley, Source code analysis: a road map, Future of Software Engineering (FOSE '07), 2007, pp. 115-30, 2007.

[25] H.H. Shahri, J.A. Hendler, A.A. Porter, Software configuration management using ontologies, 3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2007), Innsubruk, Austria, 2007.

[26] H.J. Happel, S. Seedof, applications of ontologies in software engineering, 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)

[27] Y.Wang, X. Bai, J. Li, R. Huang, Ontology-based test case generation for testing web services, ISADS, March 2007.

[28] E. Bagheri, F. Ensan, D. Gasevic, Decision support for the software product line domain engineering lifecycle, Automated Software Engineering Journal 19(3): 335-377, Springer, 2012.