

Enhancing Software Maintenance by using Semantic Web Techniques

David Hyland-Wood¹, David Carrington¹, and Simon Kaplan²

¹ School of Information Technology and Electrical Engineering,
The University of Queensland
Brisbane, Australia 4072

{dwood, davec}@itee.uq.edu.au

² Faculty of Information Technology,
Queensland University of Technology
Brisbane, Australia 4001
s.kaplan@qut.edu.au

Abstract. Software systems and information about them diverge quickly in time, resulting in difficulties understanding and maintaining them. Information about software systems, or system metadata, may include functional and non-functional requirements documentation, metrics, the success or failure of tests and the means by which various components interact or were intended to interact. Various proposals have been made to link software components and their metadata. We propose another such system using Semantic Web techniques to encode system metadata and demonstrate the particular advantages that these techniques offer. Specifically, we show how the use of an OWL-DL ontology of software engineering concepts, RDF encoding of system metadata and SPARQL queries over the resulting RDF graph can be used to enable language-neutral relational navigation of software systems thus facilitating software understanding and maintenance.

1 Introduction

Software maintenance has been defined as, “the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment” [18]. Software maintenance on large systems is responsible for a majority of the overall cost [35]. Software systems and information about them diverge quickly in time, resulting in difficulties understanding and maintaining them. This divergence is typically a consequence of the loss of coupling between software components and system metadata [35].

The key to maintaining complex software systems is to understand them. Victor Basili has said, “Most software systems are complex, and modification requires a deep understanding of the functional and non-functional requirements, the mapping of functions to system components, and the interaction of components.” [3] Many researchers have mapped the complicated relationships between domain knowledge, program components and system metadata such as requirements, metrics and tests

[11, 32, 38, 39]. In particular, the need for relational navigation of all of these components has been recognized [19].

Research has variously focused on the relationship of program components [41], and the recovery of requirements traceability from source code [9, 1]. Unfortunately, the first is insufficient and the second, in its many forms, is “not a trivial process” [24].

Previous attempts to avoid the breakage between system metadata and software components include the top-down techniques of writing documentation first [12, 37] and avoiding external documentation at all [4].

Software engineering researchers are well experienced with semantic techniques. Semantic techniques have been proposed to assist document editing [40], modeling of object-oriented systems [23, 6, 8, 21, 34] and their patterns [31], and programmatic type checking [10].

This paper presents a proposal to facilitate the joining of system metadata with program components via Semantic Web techniques. The Resource Description Framework (RDF) [26], Web Ontology Language (OWL) [25] and the SPARQL RDF Query Language [30] together allow a technique superior to those previously proposed. Specifically, we show how an OWL-DL ontology of software engineering concepts, RDF encoding of system metadata and SPARQL queries over the resulting graph may be used to enable language-neutral relational navigation of software systems thus facilitating software understanding and maintenance.

2 An Ontology of Software Engineering Concepts

Why should an ontology of software engineering concepts (hereafter referred to as SEC) be created in the first place? We certainly could have represented all necessary information in an RDF graph and queried over that graph without using an ontology at all.

We chose to create an ontology primarily to separate software engineering domain knowledge from operational knowledge (software components and system metadata), and to make our domain assumptions explicit. Further, we found that the process of creating the ontology illuminated gaps in our knowledge of the way some object-oriented programming languages were structured. These have been identified as common reasons to use an ontological approach by Noy and McGuinness [27].

It is hoped that an ontology encoding such common characteristics of the software engineering domain will be reused. Publication of the ontology on the Internet in a standard (OWL-DL) format will facilitate its potential reuse [7].

Our SEC ontology describes the relationship between object-oriented software components (programs which contain packages which contain classes, abstract classes and interfaces which contain methods and method signatures). The similarity to the language structure of Java is intentional, but eventual representation of C# and other common object-oriented languages is desirable. The term “object-oriented class” is used throughout this paper to disambiguate such a class from an OWL class, unless the meaning should be clear from context. Relationships captured include, for exam-

ple, that an object-oriented class may implement an interface, extend a super class, contain methods, or have membership in a package.

Software tests, metrics and requirements are also represented in the ontology and their relationships defined to the various software components. Two types of tests are represented, unit tests and integration tests, which extend from a common parent. Tests have results, denote the success or failure of the last run and the datetime of the last run. Tests are associated with software components and are themselves implemented as software components.

Metrics, like tests, are associated with a particular software component. They have values and datetimes when calculated. Descriptions (including units for the calculated metrics) are held in a generic `rdfs:comment` annotation property.

Requirements are associated with multiple software components and can be encoded by one or more object-oriented classes. A particular method may be designated as the “entry point” for the requirement. An entry point provides a clue as to where to begin tracing the implementation in source code. The actual description of a requirement is provided in an `rdfs:comment`.

The key to using the SEC ontology is capturing when information changes. This is done via an object property `lastModifiedAt`, a datetime property that may be used on any software component, test, metric or requirement and denotes when it was last modified. The SPARQL queries discussed in Section 3, Implementation Experiences, rely on this information. Requirements have an additional datetime property to denote when they were last validated (by a human) against the software components that implement them.

A portion of the ontology is presented here in order to explain general design decisions. The portion presented describes software requirements. A requirement, which extends `owl:Thing`, is encoded by one or more object-oriented classes and designates one or more methods as entry points. The object property `requirementEncodedBy` is restricted so that it may only apply to an object-oriented class and the object property `hasEntryPoint` is restricted so that it may only apply to an object-oriented method. In abstract syntax:

```
Class(Requirement partial
  restriction(requirementEncodedBy
    allValuesFrom(OOClass))
  restriction(hasEntryPoint allValuesFrom(OOMethod))
  owl:Thing)
```

A requirement should never be a software component of any type, nor a metric, nor a test. We use OWL’s disjoint operator to achieve this constraint. In abstract syntax:

```
DisjointClasses(
  Requirement
  Metric
  OOSoftwareComponent
  Test)
```

The properties `hasEntryPoint`, `lastValidatedAt` and `requirementEncodedBy` only apply to requirements. The `owl:domain` of each was therefore set to the `Requirement` OWL class. The abstract syntax for `hasEntryPoint` looks like this:

```
ObjectProperty(hasEntryPoint
  inverseOf(entryPointFor)
  domain(Requirement))
```

The inverse properties (in this case, `entryPointFor`) are declared in the same place. Annotation properties (`rdfs:label` and `rdfs:comment`, which provide human-readable metadata) for `hasEntryPoint`, are omitted for brevity. Definitions for other ontological elements are similar. The most complex definition is for `OOClass`, the OWL class representing an object-oriented class.

The entire SEC ontology is available at [14]. It is an initial version and was developed to highlight the use cases described in Section 3, Implementation Experiences.

We acknowledge that there are many areas for refinement. This first version does not cleanly represent Java language restrictions since the maximum cardinality of the `extends` property (used for the relationship between an object-oriented class and its super class) is not restricted to 1. Java allows only single inheritance.

The SEC ontology deviates from other modeling languages, such as UML or OCL, by referring to implementations of methods and separating the concept of unimplemented methods (method signatures). The SEC ontology further separates the concept of interfaces from abstract classes, a distinction which is not made in all object-oriented languages.

The SEC ontology currently allows only one way to denote a relationship between two object-oriented classes. Classes may contain methods, which may use other methods (and hence those methods' parent classes). The direct relationships between the classes could be inferred, but are not explicitly stated. The reliance on methods is probably insufficient to represent all relationships between object-oriented classes, such as uses of classes and methods in class constructors, which are also not currently represented. Such extensions are being considered for a later version of the ontology.

We recognize the arguments for representing all important information, such as requirement, metric and test definitions, as object properties (which are grounded in a URI) instead of relying on `rdfs:comments`. Those representations may change in a later version.

3 Implementation Experiences

The SEC ontology was created with the SWOOP ontology editor [20], whose ontology debugging features [29] proved particularly useful. Ontology creation remains a subtle art and the automated debugging features were instrumental in locating logical inconsistencies and repairing them. The debugging features are themselves reliant on the Pellet OWL DL reasoner [33]. An online version of Pellet [28] was used to validate the ontology's design by performing SPARQL queries against it.

HTML documentation for the ontology was created using the Protégé ontology editor with its OWL plug-in [22] and OWLDoc plug-in [13]. The generated OWLDoc for our SEC ontology is available online at [15].

3.1 Example Data

Example data based on the SEC ontology was developed and is available online at [16]. The example data represents a small portion of a real-world software package (from the .org.jrdf.sparql package of the JRDF project, <http://jrdf.sourceforge.net>). The example data consists of two object-oriented classes that contain four methods between them. They belong to a package, which belongs to a program. Each class has an associated unit test. A simple metric is associated with one of the classes. Each class has a requirement associated with it. OWLDoc is also available for the example data at [17].

The example data was selected because it represented a small portion of a real code base. By developing SPARQL queries that returned useful information from the example data, the validity of the approach was shown.

The example data was loaded into the Redland RDF application framework [5] and SPARQL queries made against it. SPARQL queries were developed to show that properties representing the last modification of components and the last validation of requirements could be updated and that subsequent queries could be used to determine state changes. Queries were developed to show:

1. Whether or not requirements were currently validated against their associated software components;
2. Which requirements required re-validation following a change to the modification datetime on an associated software component;
3. Which tests have failed;
4. Which requirements relate to failed tests; and
5. Which object-oriented classes had associated tests.

These queries should be viewed as representative of the type of useful queries that can be made. The success of these SPARQL queries against real-world data prove that Semantic Web techniques can be used to implement the relational navigation of software collaboration graphs and system metadata described in [19]. SPARQL queries were also run via the online Pellet OWL-DL reasoner. Some syntactic differences were discovered in the Redland and Pellet implementations of SPARQL, but suitable translations were readily obtained.

A portion of the example data is shown as a graph in Fig. 1. At the center, one may see a representation of a Java class from the JRDF project called **DefaultSparqlConnection**. It is of type `OOClass`, which is a designation from the SEC ontology. It has a method called `executeQuery()`. The `hasMethod` relationship is also defined in the SEC ontology. Similarly, it is associated with a requirement called `Req1001` and a unit test called **DefaultSparqlConnectionUnitTest**.

In the full example data set, the relationships shown in Fig. 1 are mirrored by their inverses. That is, a relationship `methodOf` exists from the `executeQuery()` method to its parent class **DefaultSparqlConnection**. Only one of each type of relationship

is shown here for brevity and clarity. The use of inverses in the full data set simplifies some SPARQL queries and is designed to ease their comprehension.

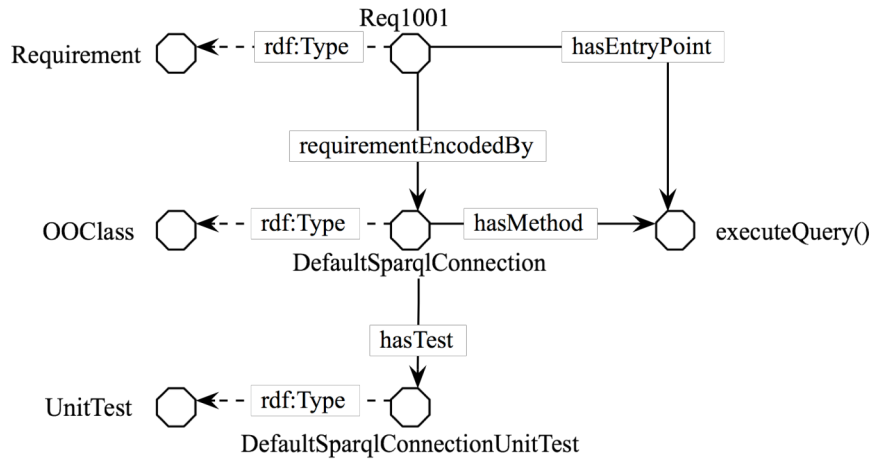


Fig. 1. A portion of the sample data, shown as a graph

3.2 An Example SPARQL Query

A simplistic SPARQL query is shown in the program listing below. This query returns all object-oriented classes in the example data and, if they have tests associated with them, returns them. The result contains two columns (defined in the SELECT clause), one for the classes and the other for the associated tests. SPARQL's OPTIONAL clause is used to ensure that all object-oriented classes are returned in the left column.

It is worth noting that the unit tests themselves appear in the left column. That is correct and due to the fact that the unit tests are also object-oriented classes in their own right. The ontology does not attempt to restrict such usage. Software engineering practitioners may note that tests may themselves have tests.

Example SPARQL query to show all classes in the example data and any associated tests

```

prefix sec:
<http://www.itee.uq.edu.au/~dwood/ontologies/sec.owl#>
prefix rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?class ?test
WHERE {
  ?class rdf:type sec:OOClass .
  OPTIONAL {?test sec:isTestOf ?class}
}
  
```

Table 1. Results of running the example SPARQL query on the sample data

class	test
sec-example: SparqlQueryBuilderUnitTest	
sec-example: SparqlQueryBuilder	sec-example: SparqlQueryBuilderUnitTest
sec-example: DefaultSparqlConnection	sec-example: DefaultSparqlConnectionUnitTest
sec-example: DefaultSparqlConnectionUnitTest	

If results were desired that did not show tests in the left column, a SPARQL FILTER operation could be used to keep tests from appearing, or the query could simply be changed to remove the OPTIONAL keyword thus forcing only classes with tests to be shown.

4 Analysis

The OWL classes and properties that constitute the SEC ontology were analyzed to determine whether this approach could be easily implemented within integrated development environments or project management tools. Our analysis focused on two considerations; which ontological elements (OWL classes and properties) could be automatically generated and whether the remaining elements (which would require manual input) would require a large or small amount of screen space in a user interface.

4.1 Usage Considerations

Of the four fundamental OWL class types in the SEC ontology (software component, test, metric and requirement), information about software components is the simplest to obtain. Software class collaboration graphs may be acquired via parsing of source code. That would provide sufficient information to generate the ontological properties associated with software components shown in Table 2.

The choice of metrics to apply to a software project must be a choice of a developer, which means that the existence of a metric and its description would need to be entered by a human. Presuming an IDE or other software which could calculate such metrics, however, the metric's value and time of calculation could be automatically established.

Tests are similar to metrics, in that their assignment, if not necessarily their creation, is matter of developer choice. Although some tools exist to generate tests, these tools do not generally require the tests to be run. The collection of test results, time run and success/failure status may be automatically generated.

All SEC ontology classes may be provided an indication of when they were last modified. That property could be updated automatically if the modification takes place via a system capable of knowing about it, such as an IDE.

Table 2. Ontology of Software Engineering Concepts properties that may be automatically generated

Concept	May be Automatically Generated
Software Components	extendedBy extends hasMethod hasMethodSignature hasPackage hasPackageMember implementsInterface interfaceImplementedBy lastModifiedAt methodOf methodSignatureOf methodUsedBy packageMemberOf packageOf usesMethod
Metrics	hasMetricValue lastModifiedAt metricCalculatedAt
Tests	hasTestResults lastModifiedAt succeeded testAt
Requirements	lastModifiedAt

Table 3 shows the remainder of the ontological properties in the SEC ontology. They define relationships between ontological elements which may not be automatically generated in all cases. However, this list can be reduced. The SEC ontology includes many inverse properties. If a property is provided by human input, then its inverse may be generated based upon that input, which reduces the information demanding human input to the list shown in Table 4.

Table 3. SEC ontology properties that may not be possible to automatically generate

Concept	Requires Human Assignment	Requires Human Generation
Software Components	entryPointFor encodesRequirement hasMetric hasTest	
Metrics	isMetricOf	
Tests	isTestOf	
Requirements	hasEntryPoint requirementEncodedBy	lastValidatedAt

Table 4. Minimal SEC ontology properties that require input by a human. Compare to Table 3

.Concept	Requires Human Assignment	Requires Human Generation
Metrics	isMetricOf	
Tests	isTestOf	
Requirements	hasEntryPoint requirementEncodedBy	lastValidatedAt

4.2 User Interface Considerations

Note that all information relating directly to software components may be generated based on the inputs shown in Table 4. Only five properties require human input, suggesting that this approach may be implemented in an IDE or project management software system without burdening the end user.

Once a metric for a particular software component is selected, it must be assigned to that component. One may envision a user interface design that minimizes the work to make that assignment, such as selecting a metric via a pull-down menu associated with the software component. Some tools exist to automatically assign metrics to software components, negating the need for human input.

Tests, especially if they are automatically generated, may be automatically assigned to a software component. If combined with metric tools that automatically calculate dependencies, the total number of properties requiring human entry is reduced to just three. If human input is needed, they may be assigned in the same manner as metrics.

Requirements require the majority of human inputs since they are typically written in natural language and dissociated from source code. Not only will a human be needed to create a requirement and assign it to a set of software components, but a human will need to inform the IDE or project management system when the requirement has been fulfilled in software because that is an inherently subjective judgment.

In the best case scenario, an IDE or project management system would be able to manage requirements in an integrated manner, as perhaps via a plug-in architecture such as exists in the Eclipse IDE.

5 Discussion and Conclusions

Software systems and information about them diverge quickly in time, resulting in difficulties understanding and maintaining them. This work has proposed a system using Semantic Web techniques to encode system metadata and demonstrate the particular advantages that those techniques offer. Specifically, we showed how the use of an OWL-DL ontology of software engineering concepts, RDF encoding of system metadata and SPARQL queries over the resulting graph may be used to enable language-neutral relational navigation of software systems thus facilitating software understanding and maintenance.

We created an ontology to separate software engineering domain knowledge from software components and system metadata, and to make our domain assumptions explicit. We designed the SEC ontology to represent generic software engineering concepts in order to facilitate its reuse.

We believe that these techniques can be applied to existing systems (during reengineering, reverse engineering or routine maintenance). The mapping of requirements, metrics and tests to the elements of a software collaboration graph can occur at any time during a software system's life cycle. Naturally, it is hoped that software engineers creating new systems will consider using this technique from the beginning, to avoid the necessity of reverse engineering.

The techniques considered in this paper may be implemented in an integrated development environment or project management tool. They require relatively little human input and would require little in the way of user interface intrusion.

Given that the life span of large software systems is limited by the ability of its maintainers to retain the links between system metadata and program elements, the potential benefit to further study of these techniques seems substantial.

6 Further Work

This work could readily be extended by prototyping the concepts in an existing IDE with a plug-in architecture and conducting usability studies. The impact of this work on the maintenance costs of both new projects and existing projects needs to be studied to measure its effectiveness.

Additional SPARQL queries could be developed to demonstrate useful leading indicators so that a software project may be directed toward maintenance activities that will lengthen its life cycle. For example, metrics calculating the rarely-implemented Maintainability Index [36] could be used to suggest when a particular software component should be considered for refactoring and which requirements that would impact.

The use of an OWL ontology to provide structure and RDF to hold project information lends itself to use over the Internet. Distributed development teams and the tools to support them could be studied using these techniques. A common difficulty during development occurs when one programmer makes a change that negatively impacts another programmer's work. Extension of the methodology described in this paper to increase the visibility of system changes is desirable.

The methodology presented in this paper is designed to be solely reliant upon an RDF data store with a SPARQL query engine. That is, an OWL-DL reasoner is not required. Some effort was made (e.g. by liberal use of inverse properties) to avoid reliance on a reasoner. Nevertheless, using a reasoner has advantages, such as inferring relationships that are not explicitly made and thereby reducing query complexity. For example, an object-oriented class may include methods which it inherits from a super class. A simplistic SPARQL query could miss the methods in the super class if a reasoner were not used. Reasoners also have disadvantages. The analysis of large software projects would result in large RDF graphs and the scalability of reasoners is still being studied. Further study of the best use of reasoners is desirable.

7 Acknowledgements

The authors wish to thank Mr. Paul Gearon of Herzum Software LLC and Mr. Brian Sletten of Bosatsu Consulting, Inc. for their kind suggestions for the improvement of the ontology of software engineering concepts.

David Hyland-Wood's efforts were partially funded by the University of Maryland Information and Network Dynamics (MIND) Laboratory, in turn funded by Fujitsu Laboratory of America College Park, NTT Corporation, Lockheed Martin Corporation and Northrop Grumman Corporation.

Open Source software used in this work included Perl, CVS, JRDF, the SWOOP ontology editor, the Pellet OWL-DL reasoner, the Protégé ontology editor with OWL and OWLDoc plug-ins, and the Redland RDF Store.

References

1. Antoniol, G., Canfora, G., Casazza, G., de Lucia, A., Merlo, E.: Recovering traceability links between code and documentation, Proc. IEEE Transactions on Software Engineering, Vol 28, Issue 10 (2002) 970-983
2. Baader, F., Horrocks, I., and Sattler, U.: Description Logics, in Staab, S., Studer, R. (eds.): Handbook on Ontologies in Information Systems of International Handbooks on Information Systems, Chapter I: Ontology Representation and Reasoning, Springer (2003) 3-31
3. Basili, V., and Mills, H.: "Understanding and Documenting Programs." IEEE Transactions on Software Engineering SE-8, 3 (1982), 270-283.
4. Beck, K., Andres, C.: Extreme Programming Explained : Embrace Change, 2nd ed., Addison-Wesley Professional, New York (2005)
5. Beckett, D.: The Redland RDF Application Framework, <http://librdf.org/> (updated 2006)

6. Berardi, D., Caly, A., Calvanese, D., de Giacomo, G.: Reasoning on UML Class Diagrams, TR-11-2003, Dipartimento di Informatica e Sistemistica, Universita di Roma, La Sapienza (2003)
7. Bontas, E., Mochol, M., Tolksdorf, R.: Case Studies on Ontology Reuse, Proc. 5th International Conference on Knowledge Management (I-Know 2005) (2005)
8. Brockmans, S., Eberhart, A., Volz, R., Löffler, P.: Visual modeling of OWL DL ontologies using UML, Proc. Third International Semantic Web Conference (ISWC 2004) (2004) 198-213
9. Coyle, A.: Simulation of Traceability in the Development of Complex High-Integrity Software-Based Systems. Master's Thesis, School of Information Technology and Electrical Engineering, University of Queensland, St. Lucia, 4072, Australia (2002)
10. Goldman, N.: Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer, Proc. 2nd International Semantic Web Conference (ISWC 2003) (2003)
11. Han, J.: Software Documents, Their Relationships and Properties. Proc. 1994 Asia-Pacific Software Engineering Conference (APSEC'94) (1994) 102-111
12. Holt, P.O.: System Documentation and System Design: A Good Reason for Designing the Manual First, Proc. IEE Colloquium on Issues in Computer Support for Documentation and Manuals, (1993) 1/1-1/3
13. Horridge, M.: OWLDoc, <http://www.co-ode.org/downloads/owl/doc/co-ode-index.php> (2004)
14. Hyland-Wood, D.: An OWL-DL Ontology of Software Engineering Concepts, version 0.1, <http://www.itee.uq.edu.au/~dwood/ontologies/sec.owl> (2006)
15. Hyland-Wood, D.: OWLDoc for an Ontology of Software Engineering Concepts, version 0.1, <http://www.itee.uq.edu.au/~dwood/ontologies/owl/doc/sec/index.html> (2006)
16. Hyland-Wood, D.: Example Data for an OWL-DL Ontology of Software Engineering Concepts, version 0.1, <http://www.itee.uq.edu.au/~dwood/ontologies/sec-example.owl> (2006)
17. Hyland-Wood, D.: OWLDoc for Example Data for an Ontology of Software Engineering Concepts, version 0.1, <http://www.itee.uq.edu.au/~dwood/ontologies/owl/doc/sec-example/index.html> (2006)
18. IEEE Standard Glossary of Software Engineering Terminology. New York, NY: Institute of Electrical and Electronic Engineers (1990)
19. Jarrott, D., MacDonald, A.: Developing Relational Navigation to Effectively Understand Software., Proc. 10th Asia-Pacific Software Engineering Conference (APSEC'03) (2003) 144-153
20. Kalyanpur, A., Parsia, B., Sirin, E., Cuenca-Grau, B., Hendler, J.: Swoop: A 'Web' Ontology Editing Browser, Journal of Web Semantics Vol 4(2) (2005)
21. Kalyanpur, A., Pastor, D., Battle, S., Padget, J.: Automatic mapping of OWL ontologies into Java, Proc. 16th International Conference on Software Engineering and Knowledge Engineering (SEKE) (2004)
22. Knublauch, H., Fergerson, R., Noy, N., Musen, M.: The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications, Third International Semantic Web Conference (ISWC 2004) (2004)
23. Knublauch, H., Oberle, D., Tetlow, P., Wallace, E. (eds.), A Semantic Web Primer for Object-Oriented Software Developers, W3C Working Group Note, <http://www.w3.org/TR/sw-oosd-primer/> (2006)
24. Lormans, M., van Deursen, A.: Reconstructing Requirements Coverage Views from Design and Test using Traceability Recovery via LSI, Proc. 3rd International Workshop on Traceability in Emerging Forms of Software Engineering, ACM Press (2005) 37-42
25. McGuinness, D., van Harmelen, F.: Web Ontology Language (OWL) Overview, W3C Recommendation, <http://www.w3.org/TR/owl-features/> (2004)

26. Manola, F., Miller, E. (eds.): RDF Primer, W3C Recommendation, <http://www.w3.org/TR/rdf-primer/> (2004)
27. Noy, N., McGuinness, D.: Ontology Development 101: A Guide to Creating Your First Ontology. Stanford Knowledge Systems Laboratory, ftp://ftp.ksl.stanford.edu/pub/KSL_Reports/KSL-01-05.pdf.gz (2001)
28. Parsia, B., Sirin, E., Grove, M., Alford, R.: Online OWL Consistency Checker with SPARQL Query, <http://www.mindswap.org/2003/pellet/demo.shtml> (as updated 2006)
29. Parsia, B., Sirin, E., Kalyanpur, A.: Debugging OWL Ontologies, Proc. 14th International World Wide Web Conference (WWW2005) (2005) 633 – 640
30. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF, W3C Candidate Recommendation, <http://www.w3.org/TR/rdf-sparql-query/> (2006)
31. Rosengard, J., Ursu, M.: Ontological Representations of Software Patterns, Proc. KES'04, Lecture Notes in Computer Science, Vol. 3215, Springer Verlag (2004) 31-38
32. Rugaber, S.: The Use of Domain Knowledge in Program Understanding. *Annals of Software Engineering* (2000) 9:143–192
33. Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A Practical OWL-DL Reasoner. Submitted for publication to *Journal of Web Semantics*. <http://www.mindswap.org/papers/PelletJWS.pdf> (2005)
34. Tetlow, P., Pan, J., Oberle, D., Wallace, E., Uschold, M., and Kendall, E.: Ontology Driven Architectures and Potential Uses of the Semantic Web in Software Engineering, W3C Working Draft, <http://www.w3.org/2001/sw/BestPractices/SE/ODA/> (2005)
35. VanDoren, E.: Maintenance of Operational Systems – An Overview, Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/str/descriptions/mos_body.html (1997)
36. VanDoren, E.: Maintainability Index Technique for Measuring Program Maintainability, Carnegie Mellon Software Engineering Institute, <http://www.sei.cmu.edu/str/descriptions/mitmpm.html> (2002)
37. Van Lamsweerde, A., Delcourt, B., Delor, E., Schayes, M.-C., Champagne, R.: Generic Lifecycle Support in the ALMA environment, Proc. IEEE Transactions on Software Engineering, Vol 14, Issue 6, (1988) 720-741
38. Welsh, J.: Software is History! in *A Classical Mind: Essays in Honour of C.A.R. Hoare*, Prentice Hall (1994) 419–430
39. Welsh, J., Han, J.: Software documents: Concepts and Tools. *Software – Concepts and Tools* (1994) 15:12–25
40. Welsh, J., Yang, Y.: Integration of Semantic tools into Document Editors. *Software – Concepts and Tools* (1994)15:68-81
41. Wilde, N.: Understanding Program Dependencies. Technical Report SEI-CM-26, Carnegie Mellon University (1990)