

# Toward a Software Maintenance Methodology using Semantic Web Techniques

David Hyland-Wood  
*MIND Laboratory*  
*University of Maryland*  
*College Park and*  
*School of ITEE*  
*The University of*  
*Queensland*  
*dwood@itee.uq.edu.au*

David Carrington  
*School of ITEE*  
*The University of*  
*Queensland*  
*davec@itee.uq.edu.au*

Simon Kaplan  
*Faculty of Information*  
*Technology*  
*Queensland University of*  
*Technology*  
*s.kaplan@qut.edu.au*

## Abstract

*This paper describes ongoing research to develop a methodology for software maintenance using Semantic Web techniques. We propose the collection of software system metadata which 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. We attempt to record and track changes to the metadata and use it to proactively notify developers of changing requirements and quality measurements which may impact further development. This research uses Semantic Web techniques such as RDF, OWL and SPARQL to encode system metadata and discusses the particular advantages that these techniques offer over other proposals. 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. Proposed extensions to this research are outlined, including the development of a toolset for distributed software maintenance.*

## 1. Introduction

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

[1]. Many researchers have mapped the complicated relationships between software components and system metadata such as requirements, metrics and tests [2, 3, 4, 5]. In particular, the need for relational navigation of all of these entities has been recognized [6]. In this paper we report on progress toward a methodology for capturing and making use of system metadata, coupling it with information regarding software components, relating it to an ontology of software engineering concepts and maintaining it over time in a distributed information environment. Unlike some previous attempts to address the loss of coupling [e.g., 7, 8], our methodology is based on standard data representations and may be applied to existing software systems. The methodology is robust in the sense that most of the required information may be automatically generated from existing sources, thus reducing the need for human input. We briefly present evidence for this in Section III.

Semantic Web techniques include the Resource Description Framework (RDF) [9], Web Ontology Language (OWL) [10] and the SPARQL Query Language for RDF [11]. RDF is a language for representing information about resources, in particular information resources on the World Wide Web. We use it to represent information about software components. OWL is a vocabulary description language for RDF. We use it to define the relationships between software components, metrics, tests and requirements. The OWL and RDF statements form a graph of software components and their metadata with defined relationships between them. Each relationship is assigned a uniform resource indicator (URI) [12] to globally disambiguate them. The SPARQL query language is then used to query

this RDF graph in order to extract useful information regarding the state of the software system.

Section II presents an OWL ontology to describe object-oriented software engineering concepts. Section III provides some early implementation experiences and how it may be used to assist in the maintenance of software systems. Sections IV and V are speculative; they describe proposed research to leverage the initial success of the ontological approach. Together, these sections suggest a way forward in defining a software development process which supports and integrates long-term maintenance.

## 2. An ontology of software engineering concepts

Why should an ontology of software engineering concepts 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 OWL ontology primarily to separate software engineering domain knowledge from operational knowledge (software components and system metadata), and to make our domain assumptions explicit. These have been identified as common reasons to use an ontological approach [13].

Our ontology of software engineering concepts (referred to hereafter as the SEC ontology) describes relationships 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++, C# and other common object-oriented languages is desirable. Relationships captured include, for example, that an object-oriented class may implement an interface, extend a super class, contain methods, and 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. Tests have results, which 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 RDF Schema 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 RDF Schema comment.

A key to creating a graph useful for software engineering queries is capturing when information changes. This is done via an OWL 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. SPARQL queries will 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.

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

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 two ways to denote a relationship between two object-oriented classes; inheritance and usage by child methods. 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 RDF Schema comments. Those representations may change in a later version.

### 3. Early implementation experiences

Example data based on the SEC ontology was developed and is available online at [15]. 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.

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 [16] 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 associated software components;
- 2) Which requirements required revalidation following a change to an associated software component;
- 3) Which tests have failed;
- 4) Which requirements relate to failed tests; and
- 5) Which object-oriented classes have 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 show that Semantic Web techniques can be used to implement the relational navigation of software collaboration graphs and system metadata. 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.

Large software systems will involve large amounts of metadata. Queries of large RDF graphs have been reported to vary in time by (at best)  $O(\log(n))$ , where  $n$  is the number of nodes in the graph [17]. There is thus

sufficient motivation to reduce the amount of human-entered metadata and to ensure that queries of the metadata graphs can occur within a useful time. The latter is assisted by investigating the simplest useful SPARQL queries and by ensuring that the SEC ontology defines the minimal number of metadata relationships necessary.

An initial analysis of the 32 relationships in the SEC ontology suggested that as little as three would require human input, all of which related to requirements. They are `hasEntryPoint`, `requirementEncodedBy` and `lastValidatedAt`. Two additional relationships related to metrics and tests may need to be entered by humans if appropriate tools are not applied (`isMetricOf` and `isTestOf`). Both of those relate metadata to software components. The small amount of human input required suggests that this approach may be implemented in an integrated development environment (IDE) or project management software system without burdening the end user.

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.

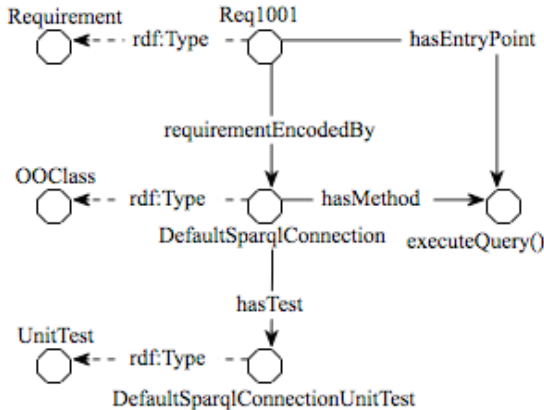
The authors recognize that requirements may change during a project's lifecycle. No cardinality has therefore been assigned to requirements; it is expected that requirements may have versions, or iterations, over time. Requirements in our methodology may be treated as mandatory or optional, variant or invariant, as the needs of a particular project dictate.

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 [18].

A portion of the example data is shown as a graph in Figure 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 Figure 1 are mirrored by their inverses. For example, 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.



**Figure 1. A portion of the sample data**

A simplistic SPARQL query is shown in Figure 2. This query returns all object-oriented classes in the example data and, if they have tests associated with them, returns them. The result, in Table 1, 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.

```
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}
}
```

**Figure 2. Example SPARQL query**

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.

**Table 1. SPARQL query results**

Class	Test
sec-example: SparqlQueryBuilder UnitTest	
sec-example: SparqlQueryBuilder	sec-example: SparqlQueryBuilder UnitTest
sec-example: DefaultSparql Connection	sec-example: DefaultSparql ConnectionUnitTest
sec-example: DefaultSparql ConnectionUnitTest	

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. Toward a software maintenance methodology

A software maintenance methodology is envisioned, such that a developer or team of developers would manually input information about requirements into an IDE, alongside the entry of code. Functionality in the IDE (perhaps implemented as a plug-in) would store software system metadata in an RDF graph and accept SPARQL queries, the results of which could be used to modify user interface elements. Changes in the metadata, including when changes were made to software components (classes and methods), requirements, metrics and tests, would be tracked. Just as existing IDEs allow for the automated running of tests and calculation of metrics, a background process would keep the metadata up to date. SPARQL queries would be run to monitor the results of changes to the metadata.

Developers would be notified via the IDE's user interface when changes required action. Examples would include:

- 1) Change to a software component (a class or method) would require a revalidation of any associated requirements. The names of unvalidated requirements would be displayed to the user until revalidated.
- 2) Failing tests would relate to requirements, as well as to software components via the metadata. Requirements impacted by failing tests could be displayed alongside test results.
- 3) Metrics calculated to be out of desired scope

could be treated as equivalent to failed tests. The names of software components and requirements relating to such metrics could be displayed until the metrics are calculated to be within desired scope.

Software system metadata should be kept in a state sufficient for the above notifications to be made. Each requirement, metric and test should have associated software components. Metrics and tests should be described in metadata sufficiently to be automatically run. Missing metadata should be brought to the attention of the user so that it may be added.

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 [1] could be used to suggest when a particular software component should be considered for refactoring and which requirements that would impact.

This methodology is designed to be reliant solely upon an RDF data store with a SPARQL query engine. That is, an OWL-DL reasoner is not required. OWL reasoners may be used to check the logical consistency of ontologies, determine the logical formalisms required to express ontologies, and infer new statements from existing statements. Scalability concerns lead us to avoid runtime dependence on a reasoner, although one was used during the development of the ontology. Reasoners do not currently scale because they must recompute all inferences following a change to the underlying data. The analysis of large software projects would result in large RDF graphs, thus causing difficulty maintaining the state of inferred statements. The scalability of reasoners is still being studied. Our SEC ontology was therefore designed (e.g. by liberal use of inverse properties) to avoid runtime reliance on a reasoner.

It would be quite helpful to logically infer relationships between software components and their metadata for the purposes of 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. Further study of the best use of reasoners is desirable.

The presence of software system metadata is an

indication of how completely this methodology is being used. Lack of metadata, or of a certain type of metadata, would indicate a condition which should be brought to a user's attention.

Some means of validating existing metadata should be developed. Constraints on the application of ontological elements already indicate some areas where the SEC ontology constrains application. For example, a test must be implemented by a software component and a metric must be associated with a software component. Further refinement of these constraints, and subclassing within the ontology will be necessary to allow a greater degree of automated validation of proper usage. This is particularly true of metrics. A particular metric may be applied at the program, package, class or method level. Creation of a new ontological element in the ontology, which subclasses `Metric` and is bound to relate to a particular type of software component, would assist automated validation.

## 5. Toward distributed maintenance

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 therefore desirable.

The Semantic Web techniques used to describe and manipulate software engineering metadata were designed to be grounded in URIs and used on the World Wide Web. They intrinsically make use of the Web's distributed architecture [19], especially Representational State Transfer (REST) [20]. Given the distributed nature of modern software development teams, is it possible to extend the use of Semantic Web techniques in this context back onto the Web? In other words, can a methodology be created for the distributed development of software which uses Semantic Web techniques in their native environment?

The goals of a Web-based software development environment are to facilitate distributed software development and maintenance by:

- 1) Using the Web's distributed nature to remove the need for centralized repositories, such as revision control systems that are commonly used in existing development environments;
- 2) Facilitating software reuse by (a) providing

URI addressability of code modules, (b) providing versioning of code modules with a formatted URI assignment scheme, (c) indicating a code module's maturity via standardized metadata and (d) providing global searchability of code modules via existing Web search mechanisms;

3) Advising users of code modules of version changes and potential impacts of those changes;

4) Advising code maintainers of the popularity (or lack thereof) of code modules.

Code modules, meaning classes or methods that may be reused, could be published on the Web and assigned URIs. Version control of modules could be handled simply by assigning each version a new URI. A canonical URI for a module could be assigned to point to the newest version at any given time, much the way the World Wide Web Consortium [21] handles URI assignment for documents. A software development system could reference code modules via an "include"-like statement which takes a URI as an argument instead of a file or module name. Such a system of URI-based includes has been partially implemented by Redfoot [22] and an extension to a REST-based microkernel such as NetKernel [23] would be straightforward.

Centralized revision control systems would not be necessary if code is referenced via URI and a URI scheme exists for accessing previous/next/current versions of a code module. The Web becomes the revision control system. Instead of checking edited code modules into a central revision control system, code modules would be published to the Web. Individual authors maintain control over the act of publishing. Existing Web security systems could be used to control the right to publish code modules at particular URLs. The act of publishing code modules on publicly-accessible (and hence publicly-indexable) URIs eases searchability.

Software reuse would be particularly facilitated if code modules were written in cross-platform dynamic (also called "scripting") languages such as Python or Ruby. Dynamic compilation of code modules at run time and abstraction from an underlying hardware platform and operating system make reuse of modules substantially more likely than traditional statically-linked languages.

The software system metadata discussed in this paper could be extended to include comments or even structured metadata relating to a given module's maturity level. Some indication of maturity is necessary for non-trivial code modules and suggests readiness to reuse. We suggest that rich metadata be

developed to represent maturity, given the wide variance of usage among software version numbers.

An RDF graph of software system metadata local to a developer may be supplemented by metadata from published code modules (such as current version numbers and modification dates). The composite graph may be queried to determine which components require review, revalidation or refactoring following or preceding upgrades of modules. The ability for a developer to determine impact in advance of upgrades is powerful. Published metadata regarding available versions of software components could form an objective basis for making upgrade decisions. Leading indicators such as this may serve to reduce post-upgrade bug hunts.

We are used to thinking of the detailed state of software components, tests, metrics and requirements as private information, local to a particular developer and hidden from others. Each developer has their own copy of a project's files and calculates their own system state description. Publication of this information on the Web could allow for subtle but important changes to the way we think about software development and maintenance. For example, since each URL-referenceable code module could have associated version information, documentation on the versions of code modules which constitute a project release could be automatically generated. Such version numbers could also be automatically assigned upon decision to publish.

Semantic Web techniques such as RDF graphs do not require the use of particular metadata. Indeed, developers could choose to override certain published metadata in favor of their own. Overriding of published metadata could be useful for trialing new edits or preparing to publish new versions.

Code maintainers may watch the number and frequency of downloads of their code modules, in much the same way that owners of Web document resources monitor the popularity of their documents.

One potential problem with a Web-based software development environment is that Web resources are controlled by their (distributed) publishers. That means that a code module might disappear from the Web without warning. This issue has impacted other Web content and has historically been dealt with by proxies, local caching, republishing by third parties and public Web archives.

## 6. Conclusion

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.

Long-lived software systems are rarely static. They evolve when they are applied to new problems and subsequently maintained to new requirements. Theories to describe the manner of these changes are in their infancy, as are mechanisms to direct their evolution toward new requirements without causing maintenance failure. Our methodology is an attempt to strengthen the coupling between software components and system metadata to stave off maintenance failure while the mechanisms of software evolvability are studied. Although it is likely that these techniques will be superceded by others as our understanding of software evolution matures, some form of mapping between what a human wants to do and what a software system encodes seems inescapable.

Frederick Brooks, the first to codify software management techniques, laid the blame for failed software projects squarely on a lack of understanding of fundamental requirements: "I believe the hard part of building software to be the specification, design and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation" [24]. Understanding what we want to do remains harder than encoding it. Maintaining a system which encodes something different from what we think it encodes is likely to remain problematic.

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). It may be interesting to apply these techniques to a model-driven architecture considering that they naturally relate system metadata to a model of the software components.

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. Suggestions were made toward both a software maintenance methodology and a distributed software development methodology supportive of maintenance concerns.

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.

## 7. Acknowledgments

David Hyland-Wood's efforts were supported in part by the National Science Foundation (NSF ITR 04-012) via the MIND Laboratory, University of Maryland College Park.

The authors wish to thank Mr. Paul Gearon of Herzum Software LLC, Mr. Brian Sletten of Bosatsu Consulting, Inc., Christian Halaschek-Wiener and Vladimir Kolovski of the MIND Laboratory, University of Maryland College Park for their kind suggestions for the improvement of the SEC ontology.

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.

## 8. References

- [1] VanDoren, E.: Maintenance of Operational Systems – An Overview, Carnegie Mellon Software Engineering Institute, [http://www.sei.cmu.edu/str/descriptions/mos\\_body.html](http://www.sei.cmu.edu/str/descriptions/mos_body.html) (1997)
- [2] Han, J.: Software Documents, Their Relationships and Properties. Proc. 1994 Asia-Pacific Software Engineering Conference (APSEC'94) (1994) 102-111
- [3] Rugaber, S.: The Use of Domain Knowledge in Program Understanding. *Annals of Software Engineering* (2000) 9:143-192
- [4] Welsh, J.: Software is History! in *A Classical Mind: Essays in Honour of C.A.R. Hoare*, Prentice Hall (1994) 419-430

- [5] Welsh, J., Han, J.: Software documents: Concepts and Tools. *Software – Concepts and Tools* (1994) 15:12–25
- [6] Jarrott, D., MacDonald, A.: Developing Relational Navigation to Effectively Understand Software., Proc. 10th Asia-Pacific Software Engineering Conference (APSEC'03) (2003) 144-153
- [7] 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
- [8] 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) 720-7
- [9] Manola, F., Miller, E. (eds.): RDF Primer, W3C Recommendation, <http://www.w3.org/TR/rdf-primer/> (2004)
- [10] McGuinness, D., van Harmelen, F.: Web Ontology Language (OWL) Overview, W3C Recommendation, <http://www.w3.org/TR/owl-features/> (2004)
- [11] Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF, W3C Candidate Recommendation, <http://www.w3.org/TR/rdf-sparql-query/> (2006)
- [12] Berners-Lee, T., Fielding, R. and Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax, IESG Standard, RFC 2396bis, <http://gbiv.com/protocols/uri/rev-2002/rfc2396bis.html> (2005)
- [13] 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](ftp://ftp.ksl.stanford.edu/pub/KSL_Reports/KSL-01-05.pdf.gz) (2001)
- [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.: 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)
- [16] Beckett, D.: The Redland RDF Application Framework, <http://librdf.org/> (updated 2006)
- [17] Wood, D., Scaling the Kowari Metastore, in Dean, M., et al. (Eds.): WISE 2005 Workshops, LNCS 3807 (2005) 193-198
- [18] The Eclipse Community, Eclipse IDE, <http://www.eclipse.org/> (Updated 2006)
- [19] Jacobs, I. and Walsh, N. (eds.): Architecture of the World Wide Web, Volume One, W3C Recommendation, <http://www.w3.org/TR/webarch/> (2004)
- [20] Fielding, R. T.: Architectural styles and the design of network-based software architectures, PhD Thesis, University of California, Irvine (2000)
- [21] World Wide Web Consortium: <http://w3.org/>
- [22] Krech, D.: Redfoot Hypercoding System, <http://redfoot.net/> (Updated 2006)
- [23] NetKernel Open Source Community: Netkernel, <http://www.1060.org/> ((2001-2006)
- [24] Brooks, F.: No Silver Bullet : Essence and Accidents of Software Engineering, *Computer*, Vol. 20, No. 4, Apr 1987 (1987)10-19