Metadata Foundations for the Life Cycle Management of Software Systems

David Paul Hyland-Wood

A thesis submitted for the degree of Doctor of Philosophy at
The University of Queensland in July 2008
School of Information Technology and Electrical Engineering

Declaration by Author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the General Award Rules of The University of Queensland, immediately made available for research and study in accordance with the *Copyright Act 1968*.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material.

Statement of Contributions to Jointly Authored Works Contained in the Thesis

Hyland-Wood, D., Carrington, D. and Kaplan, S. (2008). Toward a Software Maintenance Methodology using Semantic Web Techniques and Paradigmatic Documentation Modeling, IET Software, Special Issue on Software Evolvability, (to appear). – Hyland-Wood was responsible for 95% of conception and 100% of design, drafting and writing. Carrington and Kaplan provided discussion and review of the material.

Hyland-Wood, D., Carrington, D. and Kaplan, S. (2006) Toward a Software Maintenance Methodology using Semantic Web Techniques, Proc. of Second International IEEE Workshop on Software Evolvability 2006. – Hyland-Wood was responsible for 95% of conception and 100% of design, drafting and writing. Carrington and Kaplan provided discussion and review of the material.

Statement of Contributions by Others to the Thesis as a Whole

No contributions by others.

Statement of Parts of the Thesis Submitted to Qualify for the Award of Another Degree

None.

Published Works by the Author Incorporated into the Thesis

Hyland-Wood, D. (2006). RESTful Software Development and Maintenance, Proc. of the Open Source Developers Conference (OSDC) 2006. - Partially incorporated as paragraphs in Chapters V and VII.

Additional Published Works by the Author Relevant to the Thesis but not Forming Part of it

Hyland-Wood, D. (2007). Managing Open Source Software Projects in Belbaly, N., Benbya, H. and Meissonier, R. (Eds.), Successful OSS Project Design and Implementation: Requirements, Tools, Social Designs, Reward Structures and Coordination Methods, (to appear).

Wood, D. (2005) Scaling the Kowari Metastore, in Dean, M., et al. (Eds.): WISE 2005 Workshops, LNCS 3807, pp. 193-198, 2005.

Wood, D., Gearon, P., Adams, T. (2005). Kowari: A Platform for Semantic Web Storage and Analysis, Proc. of XTech 2005.

Wood, D. (2004). The Tucana Knowledge Server version 2.1, Proc. of XML 2004, http://www.idealliance.org/proceedings/xml04/.

Wood, D. (2004). RDF Metadata in XHTML, Proc. of XML 2004, http://www.idealliance.org/proceedings/xml04/.

Hyland-Wood, D., Carrington, D. and Kaplan, S. (2006). Scale-Free Nature of Java Software Package, Class and Method Collaboration Graphs (Tech. Rep. No. TR-MS1286), University of Maryland College Park, MIND Laboratory.

Acknowledgments

Many, many thanks go to my wife Bernadette for her unflagging support over the years of my study. Although her own interest in software is serious, it ends prior to the depths plumbed in this work; yet she offered meaningful advice and encouraged me to continue in spite of obvious incentives to the contrary. She proved her worth continuously as my life partner, my frequent business partner and my best friend.

Thanks are also due to my advisors, Associate Professor David Carrington and Professor Simon Kaplan, for their thoughts, guidance and encouragement.

Professor James Hendler , now at Rensselaer Polytechnic Institute, provided encouragement, guidance and funding during my tenure at the University of Maryland Information and Network Dynamics (MIND) Laboratory Laboratory. I credit Jim for encouraging me to think on the scale of the World Wide Web. He was also responsible for arranging a U.S. National Science Foundation grant (NSF ITR 04-012) that funded a significant portion of this work. The MIND Laboratory was funded by Fujitsu Laboratory of America College Park, NTT Corporation, Lockheed Martin Corporation and Northrop Grumman Corporation. Other MIND Laboratory researchers who deserve specific thanks include Christian Halaschek-Wiener and Vladimir Kolovski.

My long-time collaborators Mr. Paul Gearon of Fedora Commons, Mr. Andrae Muys of Netymon Pty Ltd and Mr. Brian Sletten of Zepheira LLC, deserve thanks for their kind suggestions during the last several years, especially for the improvement of the ontology of software engineering concepts. Brian implemented the new PURL server discussed in Chapter V. David Feeney of Zepheira LLC implemented the second Active PURL prototype as a NetKernel module and refined implementation details with me. Conversations with Dr. Eric Miller of M.I.T. and Zepheira LLC have been invaluable in shaping my thoughts on the deep history and wide applicability of metadata. Dr. Peter Rodgers and Tony Butterfield of 1060 Research Ltd deserve thanks for building 1060 NetKernel and making pre-release versions of it available for my

prototyping.

Open Source software used in this work included Java, Perl, CVS, Subversion, JRDF, the SWOOP ontology editor, the Pellet OWL-DL reasoner, the Protégé ontology editor with OWL and OWLDoc plug-ins, the Mulgara Semantic Store, the Sesame RDF store, the Redfoot RDF library, 1060 NetKernel and the Redland RDF Store.

Abstract

Software maintenance is, often by far, the largest portion of the software lifecycle in terms of both cost and time. Yet, in spite of thirty years of study of the mechanisms and attributes of maintenance activities, there exist a number of significant open problems in the field: Software still becomes unmaintainable with time.

Software maintenance failures result in significant economic costs because unmaintainable systems generally require wholesale replacement. Maintenance failures occur primarily because software systems and information about them diverge quickly in time. This divergence is typically a consequence of the loss of coupling between software components and system metadata that eventually results in an inability to understand or safely modify a system. Accurate documentation of software systems, long the proposed solution for maintenance problems, is rarely achieved and even more rarely maintained. Inaccurate documentation may exacerbate maintenance costs when it misdirects the understanding of maintainers.

This thesis describes an approach for increasing the maintainability of software systems via the application and maintenance of structured metadata at the source code and project description levels. The application of specific metadata approaches to software maintenance issues is suggested for the reduction of economic costs, the facilitation of defect reduction and the assistance of developers in finding code-level relationships. The vast majority of system metadata (such as that describing code structure, encoded relationships, metrics and tests) required for maintainability is shown to be capable of automatic generation from existing sources, thus reducing needs for human input and likelihood of inaccuracy. Suggestions are made for dealing with metadata relating to human intention, such as requirements, in a structured and consistent manner.

The history of metadata is traced to illustrate the way metadata has been applied to virtual, physical and conceptual resources, including software. Best practice approaches for applying metadata to virtual resources are discussed and compared to the ways in which metadata have

historically been applied to software. This historical analysis is used to explain and justify a specific methodological approach and place it in context.

Theories describing the evolution of software systems are in their infancy. In the absence of a clear understanding of how and why software systems evolve, a means to manage change is desperately needed.

A methodology is proposed for capturing, describing and using system metadata, coupling it with information regarding software components, relating it to an ontology of software engineering concepts and maintaining it over time. Unlike some previous attempts to address the loss of coupling between software systems and their metadata, the described methodology is based on standard data representations and may be applied to existing software systems. The methodology is supportive of distributed development teams and the use of third-party components by its grounding of terms and mechanisms in the World Wide Web.

Scaling the methodology to the size of the Web required mechanisms to allow the Web's URL resolution process to be curated by metadata maintainers. Extensions to the Persistent URL concept were made as part of this thesis to allow for curation of URL resolutions. A computational mechanism was defined to allow for the distinction of virtual, physical and conceptual resources that are identified by HTTP URLs. That mechanism was used to represent different types of resources in RDF using HTTP URLs while avoiding ambiguities.

The maintenance of system metadata is shown to facilitate understanding and relational navigation of software systems and thus to forestall maintenance failure. The efficacy of this approach is demonstrated via a modeling of the methodology and two case studies.

Keywords

Software engineering, software maintenance, metadata, digital curation, persistent URL, PURL, ontology, methodology

Australian and New Zealand Standard Research Classifications (ANZSRC)

080309 Software Engineering 50%, 080707 Organisation of Information and Knowledge Resources 30%, 080199 Artificial Intelligence and Image Processing not elsewhere classified 20%

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Definition
AAP	The Association of American Publishers
Active PURL	A PURL that actively participates in the collection and/or creation of metadata describing a Web resource
BASH	Bourne Again Shell
Code	Software source code
CRC	Class-Responsibility-Collaboration cards, a technique in XP
CSS	Cascading Style Sheets
DC	Dublin Core Metadata Initiative
DOI	Digital Object Identifier
DTD	Document Type Definitions. Used for HTML, SGML and XML documents.
EII	Enterprise Information Integration
EXIF	Exchangeable Image File Format (EXIF) metadata for digital camera image formats
FEAST	The the Feedback, Evolution And Software Technology projects conducted by Lehman and the hypothesis of the same name
FLOSS	Free/Libre/Open Source Software
FOAF	Friend of a Friend, an RDF vocabulary for describing inter-personal relationships
GRDDL	Gleaning Resource Descriptions from Dialects of Languages
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated (software) Development Environment
IETF	Internet Engineering Task Force
ISO	International Standards Organization
JAR	Java Archive file format
JRDF	Java Resource Description Framework API
LSID	Life Sciences Identifier
MARC	The Machine-Readable Cataloging system, a de facto library standard
MCF	The Meta Content Framework project

MDL	XML's Meaning Definition Language
NISO	The National Information Standards Organization
NLS	oNLine System, an early hypertext implementation
OASIS	The Organization for the Advancement of Structured Information Standards
OCLC	Online Computer Library Center
OMG	The Object Management Group
OWL	Web Ontology Language, a W3C Recommendation
OWL DL	A version of OWL with the expressiveness of descriptive logics
OWL Full	A version of OWL with the expressiveness of RDF with no computational guarantees
OWL Lite	A version of OWL with limited features (a subset of OWL DL)
OWL-S	OWL for (Web) Services
PICS	Platform for Internet Content Selection, a W3C Recommendation
PIM	Product Information Management systems
POD	Plain Old Documentation, a Perl programming language mechanism
POWDER	The W3C's Protocol for Web Description Resources Working Group
PURL	Persistent Uniform Resource Locator
QNames	Qualified Names, a way to abbreviate URIs in XML
RDF	Resource Description Framework, a W3C Recommendation
RDFa	Resource Description Framework attributes
RDFS	Resource Description Framework Schema, a W3C Recommendation
Rich PURL	A PURL that redirects to RDF metadata describing a Web resource
REST	Representational State Transfer
RSS	Rich (or RDF) Site Summaries, a format for Internet news feeds
SEC	An OWL DL ontology of software engineering concepts
SGML	Standard Generalized Markup Language, an ISO standard
SKOS	Simple Knowledge Organisation System
SOA	Service Oriented Architectures
SOAP	Simple Object Access Protocol, a W3C Recommendation
SPARQL	The SPARQL Protocol and RDF Query Language, W3C Recommendations
SWAMM	Software Agent Maintenance Methodology
TAG	The W3C Technical Architecture Group

TDD	Test Driven Development
UDDI	Universal Description, Discovery, and Integration (UDDI) registry services, an OASIS standard
UKOLN	The U.K. Office for Library and Information Networking
UML	Unified Modeling Language, an OMG standard
URL	Uniform Resource Locator, a URI resolvable on the World Wide Web
URI	Uniform Resource Identifier
W3C	The World Wide Web Consortium
Web	The World Wide Web
WSDL	Web Service Description Language, version 2.0 of which is a W3C Recommendation
XInclude	XML Inclusions
XLink	XML Linking Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XMP	eXtensible Metadata Platform
XP	eXtreme Programming
XPointer	XML Pointer Language
XTM	XML Topic Maps

I. Introduction

"Another flaw in the human character
is that everybody wants to build
and nobody wants to do maintenance."
-- Kurt Vonnegut, Jr. (1922 – 2007)

1.1 Introduction

Software has long been one of the most complex structures created by humans [Brooks 1987] and often the most expensive portion of engineered systems that include it, as foreseen in the 1970s by Barry Boehm [Boehm 1973]. Software maintenance is, often by far, the largest portion of the software lifecycle in terms of both cost and time [Glass 2003, pp. 115-124]. Software maintenance is thus critically important to the economics of modern engineered systems, from phones to cars, industrial furnaces to spacecraft. Yet, in spite of thirty years of study of the mechanisms and attributes of maintenance activities, there exist a number of significant open problems in the field. Software still becomes unmaintainable with time [Jones 2007] (known formally as "maintenance failure" and informally as "bit-rotting").

Maintenance failure is exacerbated by the rapid divergence of software systems and information about them. This divergence is typically a consequence of the loss of coupling between software components and system metadata [Van Doren 1997]. Many researchers have mapped the complicated relationships between software components and system metadata such as requirements, metrics and tests [e.g. Han 1994, Rugaber 2000, Welsh 1994a, Welsh 1994b]. In particular, the need for relational navigation of all of these entities has been recognized [Jarrott 2003].

Lehman was the first to recognize that software evolves during its lifecycle [Lehman 1969]. He later noted that multiple feedback loops that exist within a software development effort, and that those feedback loops influence the process of evolution [Lehman 1996]. Those feedback loops include the injection of multiple (possibly conflicting) requirements and design decisions. Various degrees of programmer understanding of requirements, design decisions and implementation details contribute to other feedback loops. Lehman developed the Feedback, Evolution And Software

Technology (FEAST) hypothesis based on that insight: "As for other complex feedback systems, the dynamics of real world software development and evolution processes will possess a degree of autonomy and exhibit a degree of global stability." [ibid.] Further, in order "to improve real world software processes one must take into account the multi loop feedback structure of such processes." [Lehman 2001b] Lehman and his colleagues explored the boundaries of that hypothesis in two projects during the 1990s: FEAST/1 [Lehman 1996] and FEAST/2 [Lehman1998]. The FEAST projects led to the formulation of "Lehman's Laws" of software evolution [Lehman 2001a].

Lehman's Law's were derived empirically from analyses of traditional, centralized software development practices; some researchers think that they may not apply to other forms of development, such as Free/Libre/Open Source software (e.g. [Godfrey 2000]). Regardless, theories describing the evolution of software systems are in their infancy [Nehaniv 2006]. In the absence of a clear understanding of how and why software systems evolve, a means to manage change is desperately needed.

Lehman suggested that "practical means" for the "mastery" of feedback loops in software require "the use of metrics, models, interpretations that facilitate reasoning about process-feedback-loop structures." [Lehman 2001b] This thesis describes a methodology for software maintenance that provides one such approach to reasoning about software as it evolves.

The methodology presented in this thesis addresses both sociological and technical concerns. Software system requirements are created by humans in natural language and must be mapped to a technical implementation. The mapping of natural languages to software implementations is imprecise at best, thus requiring some sociological components in our methodology. The technical aspects are necessary to encode system information in a manner that facilitates reuse by individuals other than the creator or creators.

Software system information is encoded as structured metadata in our methodology. A working definition and theory of metadata application to software descriptions is developed to ensure that the metadata framework used by the methodology is relevant and architecturally sound.

The methodology involves capturing and making use of software system metadata, coupling it with information regarding software components, and relating it to an ontology of software engineering concepts. The methodology is a paradigmatic description of software to enhance prospects for distributed software maintenance. We discuss theories of software system descriptions in terms of Kuhn's paradigms and the application of metadata to software. We then 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 show how changes to metadata may be recorded and tracked and propose how they may be used to proactively notify developers of changing requirements and quality measurements that may impact maintenance. We show how Semantic Web techniques can be used to enable language-neutral relational navigation of software systems thus facilitating software understanding and maintenance. Finally, we show how software system metadata may be maintained over time in a distributed information environment.

Our methodological approach can be applied to existing software systems. Some earlier attempts to address the loss of coupling between software systems and their descriptions were not able to be applied to existing systems, thus limiting their applicability [e.g., Holt 1993, Van Lamsweerde 1998]. The methodology described in this thesis 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 and obviating the need to re-architect systems for future maintenance activities.

The complexity of modern software systems has long led researchers to suggest the use of automated methods to record, track and leverage information about them [Tjortjis 2001]. Semantic Web techniques have been applied to many aspects of the problem, including a recording of design patterns [Dietrich 2005] and organizational maturity [Soyden 2006]. Happel, et al, prototyped an RDF/OWL-based system to describe software system components and domain knowledge related to them for the purposes of locating components for reuse [Happel 2006]. In a proposal close to ours, Ankolekar discussed navigating both human and technical relationships within a software project to facilitate maintenance, but no evidence of implementations of her ideas has been found in her thesis or subsequent publications [Ankolekar 2004].

1.2 Problem Statement

This research agenda may be framed as a series of questions about the state of software maintenance:

- What elements of system metadata are required during software maintenance?
- How should system metadata be applied to existing software projects? Should it be attached to source code, stored in parallel to source code or are there other alternatives?
- Can the loss of coupling between software components and system metadata be

prevented, or can a software project recover from the loss in order to avoid maintenance failure?

- Can a methodology be defined that facilitates ongoing maintenance without requiring a significant amount of "extra" work during the initial software development phase?

The theoretical claim of this thesis is: Formalized and standardized metadata has been underutilized in software system descriptions, resulting in an inability to efficiently coordinate software maintenance activities. The use of formalized and standardized metadata to describe software systems provides a basis for more efficient software maintenance.

The methodological claim of this thesis is: The maintenance of system metadata is shown to facilitate understanding and relational navigation of software systems and thus to forestall maintenance failure.

1.3 Thesis Overview

The goals of this thesis are:

- To develop a theory of metadata from first principles that may be used to guide the application of metadata to the description of software systems.
- To develop a methodology for software maintenance activities based upon the theory of metadata.
- To demonstrate the application of the developed methodology to existing software systems and show that process improvements result.

To meet those goals, this thesis provides a general classification for applications of metadata to resources, inclusive of trust relationships, on the World Wide Web and a theory of the application of metadata to software source code supported by that classification.

Extensions to the Persistent Uniform Resource Locator (PURL) concept are made to facilitate third party curation of metadata describing Web resources. These extensions include means to computationally disambiguate virtual, physical and conceptual resources identified by Uniform Resource Locators that use the Hypertext Transfer Protocol scheme (HTTP URLs).

Object-oriented software engineering concepts are formally described and related by the development of an ontology in the Web Ontology Language (OWL). This ontology is used to

describe object-oriented software systems and their properties that relate to software maintenance activities.

A methodology for distributed software maintenance (the SWAMM methodology) is then developed. The SWAMM methodology uses a Representational State Transfer architectural style and structured Semantic Web-based metadata descriptions of software, system and requirements components.

Finally, the usefulness of the SWAMM methodology is validated via two enactments of the methodology, a model of the methodology in silico and a survey of the use of the methodology by software maintenance practitioners.

Figure 1-1 summarizes the organization of this thesis. This chapter provides introductory material. Chapter II provides background material on the state of software maintenance research. Some fallacies of software maintenance are discussed in relation to the causes of software maintenance failure.

Chapters III to V develop a theory of metadata as it may be applied to software systems, especially software systems developed, deployed and maintained in a distributed manner. Distributed software development has become much more commonplace during the last two decades due to telecommuting, virtual organizations and the Free/Libre/Open Source Software movement. Distributed development is not (yet?) the norm, but it is sufficiently common to warrant significant study.

Chapter III introduces the history of metadata and various uses of metadata for different types of resource description. Metadata techniques appropriate for software maintenance are developed from first principles and historical context. Three key types of metadata are developed: informational, physical and conceptual. These types of metadata are subsequently used to explain architectural and technology choices for the definition of a software maintenance methodology.

Chapter IV discusses metadata techniques that have been applied to software, especially metadata applied to Internet and World Wide Web resources. Similarities are drawn between World Wide Web resources and software system components that are developed by distributed teams. Best practice techniques for the application of metadata to Web resources are identified and those techniques are used as guidelines for the application of metadata to software systems.

Chapter IV also analyses architectural principles required to distribute and manage metadata. A top-down analysis of requirements for managing metadata is used to suggest Representational State

Transfer (REST) [Fielding 2000] as an appropriate architectural style for addressing metadata describing virtual resources.

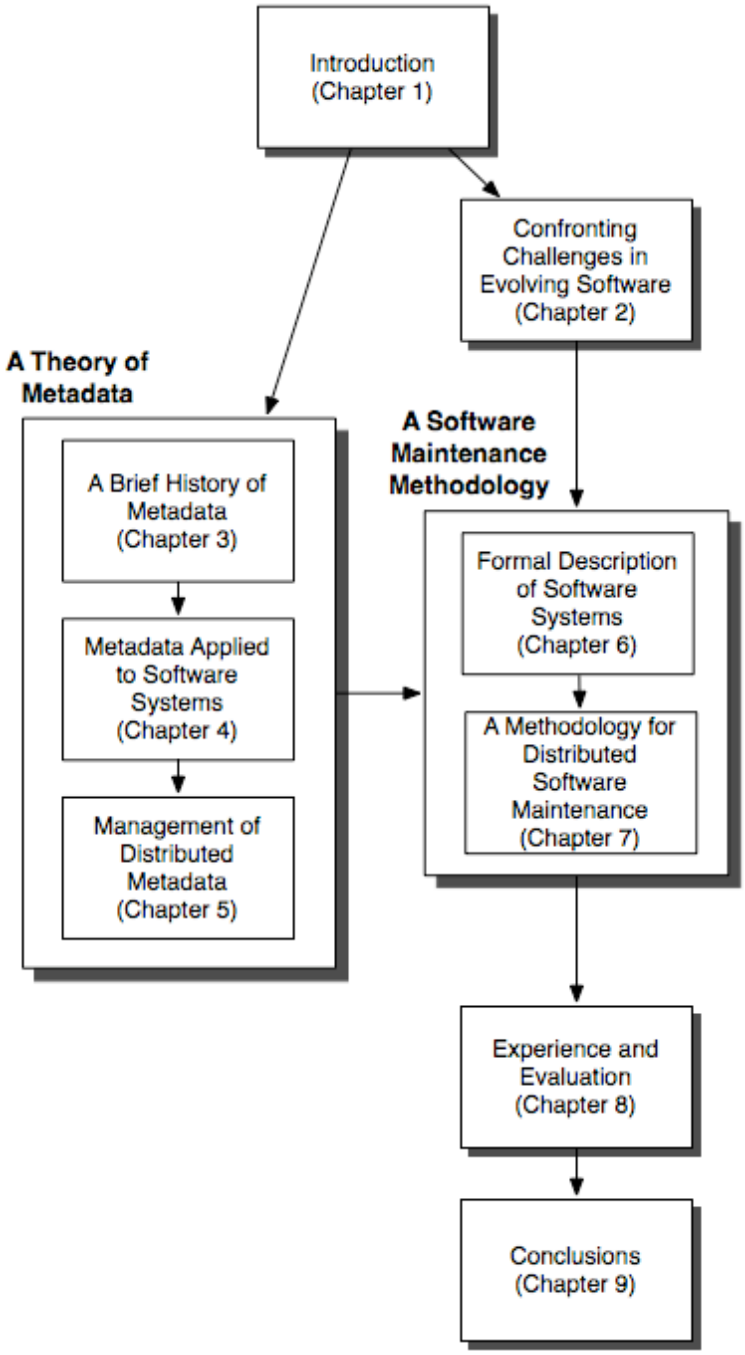


Figure 1-1. Thesis Organization.

Chapter V addresses the need to unambiguously and automatically identify the three types of metadata described in Chapter III (informational, physical and conceptual). Existing persistent identifier schemes are surveyed. Extensions to the Persistent Uniform Resource Locator (PURL)

service were developed as part of this thesis. Those extensions allow for the unambiguous universal addressing of informational, physical and conceptual resources on the World Wide Web and provide a mechanism for the practical grounding of metadata terms needed for software maintenance.

Chapters VI and VII develop a methodology for software maintenance that is based upon formal descriptions of software system metadata. Chapter VI presents semantic components necessary to describe arbitrary software systems and relate them to questions of maintenance. Semantic components include one or more ontologies of software system components and guidelines for semantically describing software systems. The World Wide Web Consortium Recommendations for metadata are used to formalize semantic descriptions. A review of the Resource Description Framework (RDF) [Manola 2004], RDF Schema (RDFS) [Brickley 2004], Simple Knowledge Organization System (SKOS) [Miles 2008] and Web Ontology Language (OWL) [McGuinness 2004] semantics is supplied to demonstrate the minimal semantic operations needed for software maintenance. A sample ontology for object-oriented systems is presented. No claim is made that the descriptions of software in this chapter are the only ones valid nor that the descriptions are even optimal; instead, it is shown that multiple descriptions by different people in different places with different contexts may be used to positive effect. Such variance is a strength of the techniques suggested. The ability to accept and adapt multiple descriptions of a software system is shown to provide resilience in the face of change.

Chapter VII builds upon the information presented in Chapters III-VI to create a methodology for distributed software maintenance. This methodology makes use of sociological mechanisms as well as structured, standardized metadata to facilitate the evolution of existing software systems. Actors and actions within the methodology were modeled to determine the properties of the methodology. The methodology, the model of the methodology and a scalability analysis are presented in Chapter VII.

Chapter VIII presents two case studies performed using the developed methodology. First, a "Hello, World" example (based upon code from the JRDF Open Source software project) is presented in detail to illustrate all aspects of the methodology. Secondly, automated metadata extraction tools are used to generate a metadata description of an existing software project (the Persistent Uniform Resource Locator, or PURL, server). Semantic Web visualization applications are used to show the benefits of using standardized metadata techniques. Queries are developed over the data to demonstrate the usefulness of the methodology. Finally, the results of a survey are

reviewed demonstrating the usefulness of metadata to software maintenance practitioners.

Chapter IX provides a discussion of the strengths and weaknesses of the proposed methodology, presents conclusions and suggests future work.

II. Confronting Challenges in Evolving Software

"Computer Science is the only discipline
in which we view adding a new wing to a building as being maintenance."
-- Jim Horning

"… software is largely a service industry operating under the persistent
but unfounded delusion that it is a manufacturing industry."
-- Eric Raymond

2.1 Introduction

Software maintenance has a checkered past. It has been relatively ignored in favor of tools and techniques for software development. Thus, a comment on the state of the field in 1983 ("Software maintenance has been almost always neglected" [Parikh 1983, pp.8]) sounds much like a comment from 2003 ("… the computer science or software engineering curriculum that contains material on software maintenance is rare indeed." [Glass 2003, pp. 116]). Software maintenance is so under-discussed and under-valued that it was possible for at least one author to publish a book on a software development methodology as late as 2000 without once mentioning maintenance. Recognizing that all may not proceed perfectly during initial development, the author of that methodology stated simply that, "Corrections of errors or added materials are sent to the recipients of the original report with an explanatory letter." [Sandq 2000, page 235] In spite of significant effort by software engineering theorists, such a blithe attitude toward maintenance is not uncommon, especially among consultants and developers able to pass maintenance tasks to others.

The earliest large-scale software systems were naturally reflections of their funding models. Military and large corporations dominated funding and therefore development and use. A smaller percentage of software systems developed in 2008 are developed for military and corporate users due to the existence of software products aimed at consumers. Tools and techniques developed to assist development (such as packaging of code into libraries, use of version control systems, testing and logging systems, integrated development environments and higher-level languages) enabled a broader application of software. Software was applied to a wide range of small and large business

problems and even personal projects throughout the 1990s and 2000s. By that time, software was no longer solely the purview of governments and large corporations. Software maintenance became, and remains, everyone's problem.

There are provable costs to maintenance, such as the number of programmers, their equipment and facilities. Interestingly, the history of maintenance cost estimates shows that they have changed little in spite of radical changes in development tools and techniques. Maintenance today accounts for 40-80% of a software project's total cost, with an average of 60% [Glass 2003, pp. 115]. Earlier estimates were similar (and probably relied upon by Glass): Shooman estimated 30-80% in 1983 [Shooman 1983, pp. 16], in 1988 Yourdon said 50% [Yourdon 1988, pp. 24], Pressman said 50-70% [Pressman 1988, pp. 203], and Shere 60-70% [Shere 1988, pp. 60]. Pressman went on to warn that, "If your numbers indicate substantially less effort, it is probably an indication that much maintenance work is going unrecognized." [Pressman 1988, pp. 203]

Field deployment of software can lead to substantially worse maintenance scenarios, from high cost to loss of material or life. A widely referenced U.S. Air Force project in 1976 was reported to cost $75 per instruction to develop but $4000 per instruction to maintain due to the inaccessibility of the components [Shooman 1983, pp. 484]. Field deployment still causes maintenance failures even though distributed computing techniques and the Internet now allow software to be remotely patched more easily. Field deployment exacerbated resolution of the infamous "Year 2000 bug" because many systems using two-digit year date handling routines were embedded in small devices such as handheld instruments and remote sensing kits [Jones 2006]. An extreme modern example is the onboard software update that caused a battery failure in the Mars Global Surveyor spacecraft in 2007 [Dunbar 2007].

Methodologies have been developed to address obvious failures in the management of software. They have, almost exclusively, focused on development and not on maintenance. Fred Brooks famously compared system development experiences of the 1970s to California's La Brea Tar Pits, where many dinosaurs struggled, died and became fossilized warnings to others: "Large and small, massive and wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty – any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion." [Brooks 1995, pp. 4] Resulting systems were often functional, but also late, expensive and difficult to maintain. Systematic efforts to find and fix each difficulty led to a series of software creation techniques, initially several forms of structured programming. One of the goals of structured programming was to create systems that

were readily modifiable or, in other words, maintainable [Martin 1985, pp. 4-5].

Unfortunately, structured programming, like other methodologies before and since, failed to address aspects of development that would later become critical for maintenance. Jackson Structured Development (JSD), for example, started by describing the structure of each input and output data stream, not by performing analysis of a problem to be solved [Jackson 1983, pp. xii]. Most structured design methodologies emphasized the creation of maintainable systems based upon a combination of up-front design and documenting the code [Yourdon 1988, pp. 24-25]. These techniques are now seen as insufficient due to the constantly changing nature of requirements. Perhaps they were known to be insufficient then, as well. Yourdon said in the same work, "Maintenance programming is a serious problem" and had only experimental restructuring engines to suggest to those in need. His work presumed that his readers were coding from scratch.

The role of documentation, either external or internal to a program, is primarily to assist maintenance efforts. Early and current researchers [e.g. from Boehm 1981 to Wiegers 2001] believe that documentation is the key to maintaining software systems, and regularly admonished practitioners for failing to keep it up to date. Practitioners, in their turn, responded with an unwillingness to contribute to an inherently unreliable medium and a stubborn refusal to ignore schedule pressures. Documentation is generally out of date and incomplete at any stage of the software life cycle [Glass 2003, pp. 123]. Updating documentation is often treated as a chore, often not as an important part of a software deliverable. When documentation is created, it is most often created in isolation from the code itself. Software without adequate documentation is thus created every day, in spite of the continued appearance of new development tools and methodologies. These factors collude to mortgage the future success of a software project and make maintenance progressively harder [Wiegers 2001].

Maintenance of structured code, like its successor object-orientation, was presumed to be easier, partially because the code would be easier to read and partially because programmers were encouraged to document their designs and decisions. Specific guidance was commonly given that in-code documentation was to record solely a developer's intent: "How [a] module works is not described. This is best ascertained by actually reading the code." (emphasis in original) [Martin 1985, pp. 54-57]. Some methodologies, such as Extreme Programming (XP), eschew guidance regarding comments at all and suggest that individual programmers should decide on a personal "style" [Beck 2005, pp. 69]. In spite of this, many managers and programmers have attempted to create commenting styles that described the code and were physically attached to it. Examples

include Brooks' self-documenting PL/1 code [Brooks 1995, pp. 173], Knuth's Literate Programming methodology [Knuth 1992], Larry Wall's Plain Old Documentation (POD), a mechanism for attaching documentation to scripts in the Perl language [Wall 1987], the Java programming language's Javadoc tool [Sun Microsystems 1994] and Dimitri van Heesch's Doxygen source code documentation generator [van Heesch 1997].

Lack of accurate documentation leads maintenance programmers to make changes without fully understanding their potential impact. Software is known to become less reliable over time as successive enhancements during maintenance are made without a full and complete understanding of their impacts on other parts of the system (known as "bad fix injection"). As many as 20% of defect repairs can result in injections of new defects, with the average amount around 7% [Jones 1995]. Perhaps that value suggests progress; Brooks said in his 1975 classic, "Fixing a defect has a substantial (20 to 50 percent) chance of introducing another [Brooks 1995, pp. 242].

Roughly half of all working programmers are now engaged in maintaining existing software and that figure appears to be rising rapidly [Jones 2007]. One may view this trend as a measure of success for the software industry; systems that work are kept, not replaced. It may be time to view software maintenance not as a problem at all, but as a consequence of the malleability of software-based systems. Glass, a proponent of this idea, has suggested making maintenance a "magnet" for experienced programmers [Glass 2006]. That may be difficult to accomplish. Many of maintenance's woes may be laid at the feet of psychology, not engineering. As the Vonnegut quote at the opening of Chapter I nicely summarizes, creative people seem to prefer acts of creation to those of maintenance, and that observation is not limited to software. Learning to navigate someone else's code, using older languages or techniques and rigorously testing is not as fun or as glamorous as seeing a new feature appear in a GUI or seeing a complicated feature work for the first time. Indeed, a common euphemism for "programmer" is "developer", not "maintainer".

A software system must be replaced when maintenance fails. Replacement often requires reverse engineering of the original and modified requirements before a new system may be designed. Reverse engineering becomes necessary because the code and the information about the code have diverged to a degree that the best documentation for the system is the code itself. This is known as a "loss of coupling" between the software components and system metadata [Antoniol 2002]. Such a maintenance failure occurs as a direct result of (a) programmers creating new bugs via "bad fix injection", (b) breaking the documentation by failing to keep it up to date, and (c) failing to migrate away from aging or removed system dependencies. Our software maintenance

methodology addresses (a) and (b) by facilitating understanding of existing code and providing mechanisms for documentation maintenance. Chapter VII introduces those aspects.

The remainder of this chapter defines software maintenance and maintenance failure, presents three fallacies present in the history of software maintenance theory, and lists some open questions in the field. We pose the problems that we address in this thesis, the most general of which is the ongoing comprehension of large software systems for the purpose of facilitating maintenance actions.

2.2 Mechanisms of Software Evolution

Software that is maintained clearly changes, and generally becomes more complex, with time. It may thus be said to "evolve" by the definition of that word. May software be said to evolve in the sense of a Darwinesque evolutionary algorithm? Meir Lehman suggested that the changes to IBM's OS/360 operating system resembled an evolutionary process [Lehman 1969] seven years before the biologist Richard Dawkins first proposed (in 1976) that many aspects of culture, including tools such as software programs, evolve in a Darwinesque manner, at least by analogy [Dawkins 1989]. Software programs, however, differ greatly from biological organisms. An obvious and important difference is that the unit of modularity is not an "individual" [Nehaniv 2006].

Tools in general seem to evolve from dissatisfaction with their form ("form follows failure") [Petroski 2006]. Software's form, as a medium for the creation of tools, seems to follow failure as well [Nehaniv 1997]. Both end users' and developers' dissatisfactions cause software to change [Reitz 2006]. There is, perhaps, a parallel to be drawn with biological evolution, where form follows competitive advantage.

Tight coupling of many software components, present in most systems, is known to inhibit evolvability of those systems [Reitz 2006]. The concept of loose coupling, where a relationship between two entities is made resilient to change by minimizing assumptions about the entities themselves, is a tenant of modern software architectures such as the World Wide Web. The term "loose coupling" and its relationship to architectural resilience was borrowed from organizational studies [Weick 1976]. Similarly, the late binding of entity addressing can further reduce coupling [Fielding 2000]. The concepts of loose coupling and late binding are desired architectural properties for distributed software maintenance.

Their environment impacts biological systems continually, but that impact is limited to phenotype. Only genetic changes, mutations, can be passed to offspring. Software is not as fortunate. It has been suggested that development tools, software documentation and even the state of the software itself impact the evolvability of a software system [Wernick 2006].

Mechanisms of software evolvability thus remain poorly understood. Our immature understanding of how software systems evolve begs the question of how to avoid maintenance failure in existing systems and systems that will be created using existing techniques. Theories of software evolution do not yet provide sufficient answers.

2.3 Defining Software Maintenance

The classic life cycle of software (i.e., the one software engineers adopted from their civil, mechanical and electrical brethren) places maintenance at the end of a series of linear development steps [Pressman 1988, pp. 5-7]. That makes sense for a bridge and it makes sense for a car. It makes sense for computer hardware. It even makes sense for some types of computer software, such as field deployed embedded systems with stable requirements. There was much debate in the 1970s and 1980s whether it was appropriate for software in general. At the heart of the debate was whether the addition of new features should be counted as "maintenance". If a requirement changed, was it "maintenance" to change the implementing code? What if a new requirement was added? During those decades, at the dawn of software engineering research, software maintenance was variously seen as either a separate activity such redesign activities [e.g. Shooman 1983, pp. 16, 484] or not [e.g. Brooks 1995 pp. 242, Parikh 1983 pp. ix, Yourdon 1988 pp. 24]. The latter view eventually came to dominate, with some researchers expressing frustration with the definition but noting its common usage ([e.g. Martin 1983, pp. 4]). The single fact that redesign activities are considered part of software maintenance separates software, conceptually and actually, from every other engineering discipline.

Redesign occurs because requirements change constantly. Changing requirements were seen in the early days of software engineering as an unfortunate, and even avoidable, side effect of poor initial analysis. A study performed in 1976 reported that the top reasons for the high cost of software were "user demands for enhancements" and poor documentation [Shooman 1983 pp. 484-5]. If we adopt the perspective championed by Glass, however, we can see that user demands for enhancement are a feature, not a bug. The total system cost of software may be high, but software

allows us to create systems that simply would not be possible without it. Consider what our systems would look like (and cost!) if they could only use hardware components. Importantly, enterprises in a capitalist system exist within a competitive environment. They must either adapt, or perish. Other consumers of software, such as militaries, also compete. The software that supports them must do the same. Requirements are therefore bound to change constantly.

Researchers in the 1980s attempted to apply concepts of more mature engineering disciplines to software. This thinking by analogy yielded some important insights, but eventually met with failure as the analogy was stretched beyond its limit. Software, as we have already shown, is not analogous to hardware in terms of maintenance. The classic production lifecycle was applied to software in spite of its inapplicability. Pressman, for example, described maintenance as "the reapplication of each of the preceding activities for existing software" [Pressman 1988, pp. 6]. Development of software does not stop. In other words, software development is not constrained by the mere fact of a delivery. Delivery may, and usually does, occur many times in the course of a product.

Fully 60% of maintenance activities relate to user-generated enhancements [Glass 2003, pp. 117]. Coupled with the fact that 60% of software lifecycle costs relate to maintenance we get the so-called 60/60 Rule, one of the few proposed "laws" of software maintenance [Glass 2003, pp. 115-118]. The 60/60 Rule is shown in Figure 2-1. Understanding changes to be made is a major activity during maintenance. 30% of total maintenance time spent on understanding the existing product. [Glass 2003, pp. 115-124 and Shere 1988, pp. 61]. Martin [Martin 1983, pp.4] found that up to 80% of maintenance activities relate to changing requirements.

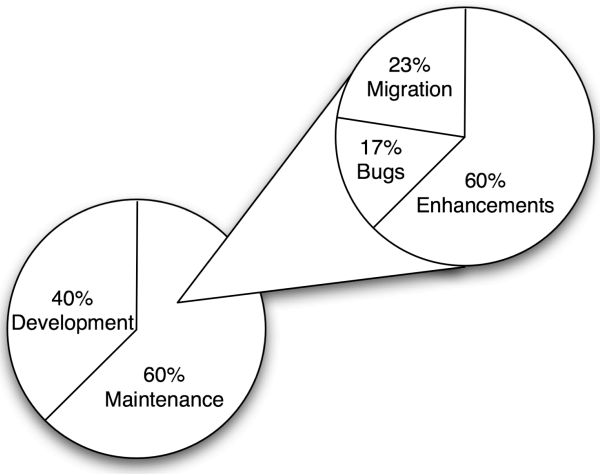


Figure 2-1. The 60/60 Rule of Maintenance Costs.

The 60/60 Rule should cause us to rethink the focus of software engineering research. The tendency to address development activities may not yield the most impressive gains. Boehm's early assertion [Boehm 1981] that proper software engineering discipline can reduce defects by up to 75% may be true, and became the basis for much work toward development methodologies, but so what? A good methodology may reduce bugs (17% of the total maintenance effort), but not address migration or enhancement time or cost at all.

As a software project moves from development activities to maintenance ones, the amount of time spent attempting to understand, trace and redesign (what Glass calls "undesign") the code increases. Testing and debugging time, often related to failures to comprehend a codebase, remain a major factor in spite of the existence of previous tests. Table 1-1 shows the breakdown of developer time spent during development and maintenance.

Table 2-1. Developer Time Spent During Development and Maintenance Activities. (After Glass 2003, pp. 121-122)

Task	Development	Maintenance
Understanding requirements	20%	15%
Design/Undesign	20%	30%
Implementing	20%	20%
Testing and debugging	40%	30%
Documentation		5%

Maintainers were originally thought to be spending the bulk of their maintenance time creating enhancements because requirements were poorly captured. See Figure 2-2. It now seems that maintainers spend the bulk of their time changing requirements specifically because they can [Glass 2006]. Software, as Glass pointed out, is malleable. Users, managers and investors generally see an economic benefit from modifying an existing system that almost meets their current requirements rather than creating a new one.

Judged in that light, Boehm's statement that 20% of defects account for 80% of the work and 20% of the modules account for 80% of the defects [Selby 2007, pp. 3] would appear to be a benefit. Poorly designed areas tend to cluster and may be refactored in one place. New requirements, on the other hand, may or may not cluster, but directly leverage the malleability that is a central property of software.

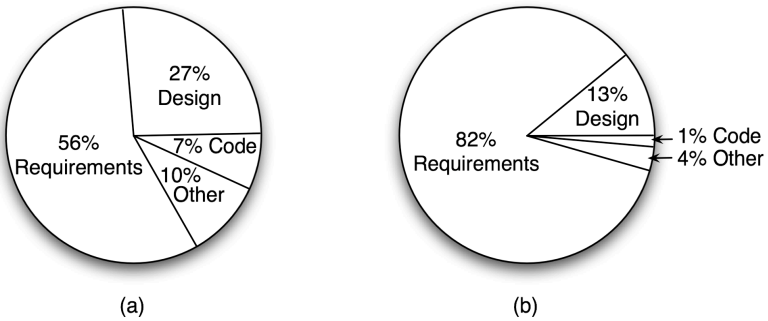


Figure 2-2. (a) The greatest number of bugs occur in requirements specification and design, (b) The greatest cost is incurred in correcting requirements bugs (After Martin 1983, Figure 12-3).

Jones [Jones 2007] has refined the definition of software maintenance with a detailed delineation of twenty-three separate forms of modification. However, he notes that each may be categorized as either defect repair or enhancement. In my opinion, Jones does not change the definition of maintenance; he details it.

Acknowledging that redesign efforts are part of the software maintenance definition, we can still find obvious differences between creating enhancements and repairing defects. Defect repairs, be they the result of bad fix injection, poor requirements gathering or failure to correctly implement a feature, do not require changes to the way the software is managed; the original expectation was one of defect-free operation already. Enhancements, on the other hand, are expected to change the way the software operates, are documented and are tested. Table 2-2 summarizes the key differences.

Table 2-2. Typical differences between enhancements and defect repairs. (Adapted from Jones 2007, Table 1)

	Enhancements	**Defect Repairs**
Funding Source	Clients	Developer
Requirements	Formal	None
Specifications	Formal	None
Inspections	Formal	None
User documentation	Formal	None
New function testing	Formal	None
Regression testing	Formal	Minimal

Clearly, some programmers will perform testing and update documentation (or even

specifications and requirements) after repairing a defect, but the practice is all too uncommon. Table 2-2 should not be read as a criticism of programmer practices, but as recognition of trends in current practice. Thus, the title of the table denotes "typical" differences between enhancements and defect repairs.

The Institute of Electrical and Electronic Engineers (IEEE) and the Association for Computing Machinery (ACM) have jointly defined software maintenance 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" [IEEE Standard Glossary 1990]. We note that the term "delivery" is ambiguous and suggests a classical view of the software lifecycle. "Delivery" in the modern sense can constitute any number of production software releases or updates from conceptualization to retirement of a software product.

2.4 Maintenance Failure

A software product will eventually reach the end of its useful life. The goal of software maintenance is to delay the end of life as long as possible. Accordingly, the end of life for a software product is often a state known as "maintenance failure". Maintenance failure eventually occurs in any software product that encodes requirements that remain extant. Software products whose need is superceded (e.g. a product that calculates wholesale tax following the introduction of a comprehensive goods and services tax) may be removed from service prior to entering maintenance failure.

Maintenance failure may occur for several reasons. The most common are that a product fails to adapt to changes in environment (such as hardware, operating system or necessary libraries) or it fails to adapt to new requirements. The latter occurs when new requirements can no longer be added economically. [Brown 1980 pp. 279, Beck 2005 pp. 137]

Early researchers believed the solutions to maintenance failure were straightforward. After analyzing hundreds of software defects in the 1970s, pioneer researcher Barry Boehm said, "if more time had been spent on validating the design… prior to coding many of the conceptual errors would not have been committed to code." [Selby 2007, pp. 1] Note the connection to Figure 2-2. Boehm missed the connection from constantly changing requirements to constantly adding value by adapting to changing needs.

Shooman suggested in 1983 that poor documentation is the cause of maintenance failure: "If the

program is to be changed several times over a 10-year period, the documentation is more important than code… Obviously, in any sensible project we want both code and documentation." [Shooman 1983, pp. 486] "It would be wise to pay, say, 10 percent more for the development of reliable software with proper documentation. This will more than pay for itself in the long run through reduced maintenance and ease of redesign." [ibid. pp. 16] "The maintenance or modification costs will be strongly related to the quality of the documentation." [ibid. pp. 484] Shooman believed that the problems of software maintenance were understood and that the industry was well along the path toward implementing and fielding "the solution". [ibid. pp. 19-20] Unfortunately, the problems he identified were not solved by a combination of careful development practices and complete documentation. His hopes for the future included, (a) improved languages and tools, (b) increased use of new development methods and (c) program proofs and automatic programming. The first two have certainly happened and had an impact on productivity. The latter has not happened on a large scale. Although some progress has been recently made in the automated verification of general purpose code, we cannot yet judge how these new techniques will change our abilities to create software [Holzmann 2003]. Increased productivity in development has led to greater, not fewer maintenance challenges.

Yourdon postulated in 1988 that large software projects fail in development due to increasing complexities. "A project involving up to 100,000 lines of code is sufficiently complex that neither the systems analyst nor the user is likely to have a clear understanding of exactly what the system is supposed to do, and even if there is a clear understanding, the user is likely to change his mind about some of the requirements." [Yourdon 1988, pp. 3]. This is the same phenomenon we reviewed earlier when discussing bad fix injection; bad fixes are inserted precisely because the developer does not understand the system.

There may be hard limits on our ability as human beings to understand complex systems and those limits are likely to be based in our physiology. Dunbar's Number [Dunbar 1993] is an accepted measure of the number of close interpersonal relationships that may be maintained by a human being. This limit (roughly 150) is thought to be due to a physiological limitation of the human neocortex. Human memory also has known limitations. There are thus limitations to the complexity of a system comprehensible by humans. One may certainly abstract from the details of a complex system to gain a general understanding or choose to understand a subcomponent in perfect detail. The perfect understanding of all components coupled with a perfect understanding of the (often conflicting) intents of system users is simply not possible for very large software systems and arguably rare for smaller software systems. The interpretation of requirements, documentation

and code by multiple users, managers, analysts and programmers of a large software project, each according to their own imperfect understanding of what they are attempting to accomplish, merely adds to the complexity of the system.

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 [Antoniol 2002]. Van Doren discusses an "Evolution/Replacement Phase, in which the system is approaching or has reached insupportability [Van Doren 1997]. The software is no longer maintainable. It has become so 'entropic' or brittle that the cost and/or risk of significant change is too high, and/or the host hardware/software environment is obsolete. Even if none of these is true, the cost of implementing a new requirement is not tolerable because it takes too long under the maintenance environment. It is time to consider reengineering." It is this type of maintenance failure that we strive most to prevent [Beck 2005, pp. 137].

None of these descriptions of maintenance failure address how we might migrate a software project back from the edge of maintainability; we are more likely to accept the inevitable, abandon the attempt and create a new one. A primary goal of this thesis is to provide a means to (slowly, carefully, perhaps even painfully) recover a project's maintainability. The solution is to lower local entropy and, like any closed system, that means we are required to put work into it.

2.5 Three Historical Fallacies

Much progress was made by the early 1980s in understanding the dynamics of software maintenance. With a generation of large-scale system implementations behind them and software engineering research well underway, theorists believed that they understood what caused software to become unmaintainable. Unfortunately, although researchers correctly captured the problems of software maintenance, their solutions were marred by three key fallacies, which we call The Fallacy of Perfect Knowledge, The Fallacy of the Big Round Ball and The Fallacy of Perfect Execution. As we show below, all three were the direct result of thinking by analogy about the practices of older engineering disciplines.

2.5.1 The Fallacy of Perfect Knowledge

The Fallacy of Perfect Knowledge – it is possible to capture complete, non-conflicting

requirements with sufficient attention to detail. Requirements, even when agreed upon in detailed up-front design, will change. It is impossible to know them all in advance. Requirements gathered from more than one source can also result in inconsistencies. Requirements may also mean different things to different people. Differing interpretations may be due to perception, goals or language. In order to create a general-purpose software maintenance methodology, we must accept and even embrace these ideas.

2.5.2 The Fallacy of the Big Round Ball

The Fallacy of the Big Round Ball – requirements don't change appreciably after delivery or can be controlled. Early researchers believed that if requirements could be fully understood before coding began, if the new structured programming techniques were rigidly adhered to, if new systems were documented fully and correctly, there would be no maintenance crisis. Some academics and practitioners took note of the problem of post-delivery changes to requirements and labeled them evil; static requirements yielded more stable systems. Some sought to limit a user's right to request changes (e.g., "Reduce the need for change maintenance by planning for and controlling user enhancements" – one of a list of "solutions to maintenance" [Martin 1983, pp. 13]). Unfortunately, such strict controls have the unintended side effect of making a software system less useful to its end users. Such decisions, often based upon short-term economics, were greatly responsible for the alienation of information technology departments from their user bases in the 1990s and the subsequent development of smaller, often duplicate, software systems within business units during that period.

2.5.3 The Fallacy of Perfect Execution

The Fallacy of Perfect Execution – it is possible to create flawless code with sufficient attention to detail. We need to admit that arbitrary logic is hard to verify in the general case, and hard or impossible to fully test. Drawing an analogy to the bricks and beams used in other construction-related activities, a researcher recently suggested that software is hard to verify because "there are no good, predictable building blocks. The elements out of which programs are constructed, statements, procedures, or objects cannot be composed in a predictable fashion" [Dickerson 2006]. It is also hard to trace programmer intent, especially when requirements change or are inconsistently documented. Bugs will remain part of every software product shipped.

Admitting these three fallacies is tantamount to changing the way we think about the state of software at the end of development. We can come to see delivered software as buggy, likely to

change and with inaccurate documentation. That insight, simple though it may be, encourages us to approach maintenance differently. It encourages us to develop tools and techniques to incrementally refactor software implementations, requirements and documentation.

2.6 Emerging Problems and Approaches

Software currently in production was created using a great variety of techniques and procedures. The software industry continues to struggle with this legacy and is expected to do so for the foreseeable future. We therefore require a way to record information captured about these programs as they are maintained. Failure to do that in a systematic manner leaves us where we are now; at the mercy of the memory of individual programmers assigned to a project.

The size of modern projects is increasing dramatically. In 1988, Yourdon called 10M LOC "utterly absurd" [Yourdon 1988, pp. 2], but 2007 saw the Eclipse Foundation release its Europa coordinated project release with seventeen million source lines of code [http://www.linuxdevices.com/news/NS9334092346.html]. Similarly, an official Microsoft Web log reported that the company's popular Office suite for Macintosh computers measured thirty million source lines of code [http://blogs.msdn.com/macmojo/archive/2006/11/03/it-s-all-in-the-numbers.aspx]. The size of the maintenance domain is increasing.

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." [Basili 1982] Many researchers have mapped the complicated relationships between domain knowledge, program components and system metadata such as requirements, metrics and tests [e.g. Rugaber 2000, Welsh 1994a]. Some have presaged this thesis in their interpretation of software development as a document-oriented process, and have recognized the importance of inter- and intra-document relationships in supporting traceability from requirements to implementation details [Han 1994, Welsh 1994b].

Research has variously focused on the relationship of program components [Wilde 1990], and the recovery of requirements traceability from source code [Coyle 2002, Antoniol 2002]. Unfortunately, the first is insufficient and the second, in its many forms, is "not a trivial process" [Han 1994]. We still require a way to capture and maintain our understanding of a code base that is broadly applicable.

The way we view software for maintenance, as a series of functions to be repaired and added to, is flawed for historical reasons. The traditional view of a computer was a machine to be commanded via a series of instructions, those instructions being collected into functions. The functions were collected into a hierarchy (a program), which led naturally to the concept of development structured around functions. [Jackson 1983, pp. 3-4]. We know today that many programs are not hierarchical, such as those designed with grid, object-oriented, message-oriented and resource-oriented architectures. These more modern logical abstractions for describing computer programs are more Web-like. Some problems of hierarchical thinking are investigated in Chapter III. Chapter IV discusses how Web-like software structures may be described.

Software maintenance and software development, those two states of any software project, need to viewed as the continuum that they are. Some people are already thinking this way. So-called Agile methodologies are claimed to be in a constant state of maintenance. Beck, one of the architects of Extreme Programming, put it this way: "Maintenance is really the normal state of an XP project. You have to simultaneously produce new functionality, keep the existing system running, incorporate new people into the team, and bid farewell to members who move on." [Beck 2005, pp. 135-6] Glass's view of maintenance as a positive outcome of successful projects fits nicely with Beck's model.

Karl Wiegers has gathered a series of principles for approaching requirements maintenance on existing systems [Wiegers 2001]. His collection importantly addresses not only broken code, but also the lack of correct documentation and understanding (of both code and requirements) so common in systems maintained over many years. He suggests retaining the older documentation, even though it is knowingly out of date, and slowly bringing it to relevancy as one addresses each section of code. This is a form of documentation refactorization. This thesis builds upon Wieger's idea in Chapter VI but extend it in accordance with the metadata best practices for virtual information resources described in Chapter IV.

A few researchers have attempted to combine the formal methods of software engineering with formal descriptions enabled by Semantic Web technologies (e.g. [Goth 2007]). Tetlow, for example, claims that some of the benefits of formal methods have not been widely fielded because they are "too abstract for the average developer". The benefit to Semantic Web techniques is that they are unreservedly formal, and yet may be used by developers "in an exceptionally informal way" [ibid.]. The W3C has drafted two documents to date dealing with "Semantic Web-enabled software engineering"; a primer for developers of object-oriented software [Knublauch 2006] and

another describing the use of formal ontologies for the description of software systems [Tetlow 2006]. This thesis builds on some of the ideas of Semantic Web-enabled software engineering in Chapters VI and VII.

III. A Brief History of Metadata

"Knowledge is of two kinds.
We know a subject ourselves,
or we know where we can find information on it."
-- Samuel Johnson (1709 – 1784)

"It depends on what the meaning of the word 'is' is."
-- Bill Clinton (1946 -)

3.1 Defining Metadata

There can be no doubt that the amount of digital information created, stored, transmitted or replicated is huge and increasing rapidly. Metadata describing our digital, or virtual, information is increasing along with it. The wide adoption of the Internet has allowed digital information to be created at such a rate that in 2007 the amount of transient information exceeded the ability to store it [Gantz 2008, pp. 2]. Much of this information, including electronic mail, instant messages, the contents of file systems and Web sites and high-value content such as the holdings of libraries and museums, is described by metadata. Automated collection of metadata, such as in cameras, radio-frequency identification (RFID) systems and Global Positioning System (GPS) receivers, has drastically increased the amount of metadata available.

Metadata is a simple term, but one used in so many ways as to nearly defy description. The Greek *meta* means "with, across or after" and in English the sense of "after", "beyond" or "about" is generally considered proper usage. In its simplest sense, metadata is "data about data"; and that phrase is used routinely in Library Science, Computer Science and publishing industry literature. Unfortunately, that simple definition begs a series of questions: What is the data that the metadata is about? What is the purpose of the metadata? What is the form of metadata? Different communities have answered those questions differently.

We propose to carefully define metadata for use within this work. Metadata is taken by many to describe something else. The "something else" often means a type of resource. The type of

resource and indeed the definition of "resource" itself are almost universally left as an exercise for the reader. We will show that even that degree of broadness is (quite often!) too little. Our definition is meant to be broad and encompassing, for reasons that will hopefully become clear by the end of this chapter.

We first need to define the terms that we use later to define metadata. We use the term "resource" to mean anything referenceable by humans or their constructs. It may be a book, a Web page, a screwdriver, a piece of dirt. We call the special subset of resources that serializes knowledge for later retrieval (not necessarily by humans!) as "knowledge resources". Resources and knowledge resources may be physical (such as screwdrivers and books) or virtual (such as the concept of zero or the Web home page of the British Broadcasting Corporation). Virtual resources are often called "digital" or "electronic", although that characterization only applies to those virtual resources that are publishable on a computer network. Physical resources are often called "non-digital", "concrete" or even "real" (although they are no more "real" than virtual resources in that both types of resources exist and may be used by humans or their constructs by definition). The careful definition of these terms will assist us to define metadata in a meaningful way.

The definition of "resource" used in the remainder of this thesis follows the World Wide Web Consortium (W3C) Technical Architecture Group's decision to define a "resource" as anything that may be referred to by a Uniform Resource Identifier (URI), a globally unique naming scheme. Our definition of "knowledge resource" (alternately, "virtual resource") follows their definition of "information resource" as resources in which "all of their essential characteristics can be conveyed in a message". [Jacobs 2004]

At least two additional types of resources may be identified; physical resources (those that exist physically in the real world, such as my car) and conceptual resources (such as the idea of a car). Physical and conceptual resources may be given URIs, but no representation of them, canonical or otherwise, may be returned in response to the resolution of those URIs. The subtle distinction between knowledge, physical and conceptual resources requires delicate handling when they are all assigned HTTP URLs as identifiers. Chapter V provides a computational mechanism for the distinction of knowledge, physical and conceptual resources that have HTTP URLs. That mechanism is used in Chapters VI and VII to disambiguate types of resources.

The term "metadata" is itself Balkanized. One variously sees "metadata", "meta data", "meta-data" and even "METADATA". The latter was used as early as 1969 by Jack Myers in relation to The Metadata Company, a software provider in the United States still operating as of this writing.

The Metadata Company registered the term as a U.S. trademark in 1986. Variations on the presentation of the term were initially attempts by the users to avoid trademark infringement, but that has given way to wide public usage of "metadata". Current legal opinion in the United States has suggested that "metadata" has entered the public domain due to widespread use, but the situation has yet to be tested in a court of law. The Metadata Company claims continuing ownership of the term "metadata" and prefers "meta-data" to be used by others. [Yanosh 2007]

Computer scientists generally use the term metadata as it relates to a narrow subset of data with which they are dealing. For example, the World Wide Web Consortium's Metadata Activity, a group focused specifically on developing metadata standards for Web and Semantic Web content, defined metadata as "machine understandable information for the web." [Brickley 2001] The Web Services standards (e.g. [Booth 2007]), however, use an entirely different definition and representation for metadata. In both cases, the definitions presume that metadata is structured and that it is descriptive in nature. The W3C standards for metadata do not require it to be put to a particular use.

People describing relational database schemas, enterprise information integration (EII), data warehousing or product information management (PIM) systems have their own requirements for metadata and their own definitions. William Durrell, a database administrator, has defined metadata as "structured, encoded data that describe characteristics of information-bearing entities to aid in the identification, discovery, assessment, and management of the described entities." [Durrell 1985] He assumes that his metadata is structured, that its purpose is description and that it is put to a particular use.

The Extensible Markup Language (XML) community has defined a large number of metadata mechanisms, as have the developers of computer file systems, image formats and geospatial information systems. They all refer to structured metadata for the purpose of describing virtual knowledge resources and generally scope the intended use.

Library scientists, too, have narrowly defined metadata for the subsets of their field of study. Librarians have described knowledge resources using information external to those resources since antiquity. Until recently, though, they never called those descriptions "metadata". Instead, they used the terms catalogues, indexes, references, categorizations. A common name was not assigned to the collection of these techniques. Library theorists in the 1990s began to use the term metadata to transcend both physical and virtual knowledge resources and the particular use of description of those resources. This shift was driven by the fact that library resources, once purely physical, have

become a complicated mixture of physical and virtual forms. The Library Science community is slowly adopting the term metadata to encompass a wider range of meaning than the Computer Science community. This definition from the U.K. Office for Library and Information Networking (UKOLN) is no longer atypical:

> Metadata can be defined literally as "data about data," but the term is normally understood to mean structured data about digital (and non-digital) resources that can be used to help support a wide range of operations. These might include, for example, resource description and discovery, the management of information resources (including rights management) and their long-term preservation. [Day 2007]

Many library scientists have come to regard metadata as being structured, but no longer tied to a particular use or purpose.

The publishing industry has discovered the many uses of metadata for managing the process of creating books and magazines. The U.S.-based National Information Standards Organization (NISO), a non-profit standards body, defined metadata as it applies to publishers as, “structured information that feeds into automated processes ... This definition holds whether the publication that the metadata describes is in print or electronic form.” [Brand 2003] Metadata is seen as structured and put to a particular use, but may be beyond simple descriptions of a resource. The Association of American Publishers (AAP) extended the definition of metadata away from simply describing knowledge resources when it said, “Metadata is data about data. Metadata can describe such elements as the creator (e.g., author bio) and the content (e.g., # of pages, chapter titles). Metadata can be the business rules that have been assigned to govern a transaction (e.g., royalty rates from an author’s contract or the “allow print” rule in the setting of digital rights).” [McCoyd 2000] The APA’s view suggests that metadata may be put to many purposes and may transcend both structure and purpose. Interestingly, this definition suggests that metadata may describe more than one resource (the original resource and the way that resource is later used, such as its sales transactions). We revisit this form of interconnected data reuse later in some detail. The two definitions from the publishing industry are closer to the way librarians view metadata than the way computer scientists typically use the term.

Metadata theorist David Marco has attempted to broaden the definition of metadata to encompass all of these uses. He suggests that metadata be considered "all physical data and

knowledge from inside and outside an organization, including information about the physical data, technical and business processes, rules and constraints of the data, and structures of the data used by a corporation." [Hay 2006] That is certainly an expanded and general definition in comparison to the others. However, why limit metadata to being used by a corporation? Librarians would surely object on more grounds than changing "corporation" to "organization". Also, is it necessary to define metadata as "all data"? Although we recognize the identity relationship (metadata is itself data), there seems to be a useful separation of the term metadata.

We suggest combining a broad definition of structure, a broad definition of intent and a broad definition of use. We define metadata as information used to administer, describe, preserve, present, use or link other information held in resources, especially knowledge resources, be they physical or virtual. This definition is in line with current library science metadata theorists such as Anne Gilliland-Swetland [Gilliland-Swetland 1998]. Table 3-1 summarizes the types of metadata she has defined.

Table 3-1. Types of Metadata and Their Functions (after Gilliland-Swetland 1998 and Caplan 2003)

Type	Definition	Examples
Administrative	Metadata used in managing and administering knowledge resources	• Acquisition information • Rights and reproduction tracking • Location information
Descriptive	Metadata used to describe or identify knowledge resources	• Cataloging records • Finding aids • Specialized indexes
Preservation	Metadata related to the preservation of knowledge resources	• Documentation of physical condition • Documentation of actions taken to preserve
Technical	Metadata related to how a system functions or metadata behaves	• Hardware and software needed • Digitization information, including formats
Use	Metadata related to the level and type of use of knowledge resources	• Exhibition records • Content re-use tracking
Linkage	Metadata describing the relationships between one resource and others.	• Structural relationships such as a book chapter's relationship to a book. • Pointers to objects of similar content

Gilliland-Swetland has listed some guidance regarding metadata. Her points should now sound familiar to our readers: “Metadata does not have to be digital”, “Metadata relates to more than the description of an object”, “Metadata can come from a variety of sources” and “Metadata continues to accrue during the life of an information object or system”. The fact that this guidance was being presented to library scientists in 1998 demonstrates the transition of thinking within that community as it faced challenges from the World Wide Web as a type of reference library. Only minor changes are reflected in Caplan’s view in 2003 [Caplan 2003].

Metadata may also be discussed in relation to its location relative to its underlying resource. For example, a book may contain a table of contents, a title page and an index. Those are all examples of metadata that is attached to its resource. A card catalogue entry in a library, though, is external to the resource that it describes. Table 3-2 lists the five types of metadata locations and provides examples of each. We use the relative locations of metadata to explain major trends in metadata development later in this chapter.

Table 3-2. Locations of Metadata and Their Functions

Location	Definition	Examples
External	Metadata that is part of a different knowledge resource than the one it describes	• Library card catalogue • Search engine index
Attached	Metadata that is part of the resource that it describes	• Book index • Title page
Separable	Metadata that may be (perhaps temporarily) attached and may be removed from a knowledge resource that it describes	• Luggage tags • RFID tags • Labels on hanging folders
Attached/Separable	Metadata attached to a virtual resource and easily copied non-destructively for other purposes.	• HTML Meta tags • Hyperlinks
Implicit	Metadata that is implied by the environment of a knowledge resource that it describes	• Room location • Order in a stack of books

Now that we have a working definition of metadata, we can use it to trace the development of metadata from prehistory to the present. We can use that history to highlight trends in knowledge organization and follow those trends to discover best practice concepts for applying metadata to the

problem of software maintenance.

3.2 Metadata in the Ancient World

The history of metadata is naturally tied to the history of the data it describes. The concept of metadata, however tenuous, appears to predate writing as we think of it. The types of metadata and their locations relative to the content they describe were also developed in prehistory. The remainder of this chapter explains and sets in context the history of metadata as a concept and trace its development to modern times. The way the contextual history of metadata has colored the way metadata is currently applied to software is described and more appropriate ways of handling software metadata are suggested.

Metadata and writing are the products of city-dwelling specialists and they in their turn depend on agriculture and livestock husbandry in order to subsist. A very brief overview of the processes leading to written civilization is thus justified.

Humans appear to have begun processing information into signs and symbols early in their history. The archaeologist Clive Gamble has suggested that the acquisition of information and the passing of that information to offspring is one of two fundamental means by which mammalian parents can assist the spread of their genes (the other being simply having more babies and trusting to luck). He ties the acquisition of information to the means by which it is carried when he states,

> … it is not the bits of information but rather the chunks into which they can be assembled that is significant. To become information these chunks require signs and symbols to provide a code for action. In perceptual terms these can be visual and verbal as well as tactile and olfactory. In evolutionary terms it is the externalization of information which such codes permit and which allow it to serve as a representation of action irrespective of whether it is communicated by language or through material culture." [Gamble 1999, pages 363-364]

In one fell swoop Gamble has provided an evolutionary justification for the early development of both art and writing, notwithstanding their later developments. Art, religious in nature and detailed in execution, was developed many millennia before writing. When Homo Sapiens moved

into Europe and Central Asia roughly 40,000 years ago they were already decorating and illustrating their artifacts. By 32,000 years ago they began creating detailed cave paintings. One theory holds that painting had already developed in open camps, but only the more protected caves were suitable for preservation. It is possible that cave paintings became popular as caves became accessible for the first time due to the increasing rarity of cave bears. By 20,000 BCE cave paintings were abundant throughout Eurasia [Guthrie 2005, pages 23-26].

Fixed settlements were followed by agriculture and, eventually, cities. Our understanding of the earliest fixed settlements are constantly being revised but appear to have occurred in the Levant prior to the invention of agriculture [e.g. Tudge 1998]. The great mythologist Joseph Campbell summarized the spread of early civilizations as he saw it:

> …it may be noted, first, that the arts of grain agriculture and stockbreeding, which are the basic forms of economy supporting the high civilizations of the world, now seem to have made their first appearance in the Near East somewhere between 7500 and 4500 B.C., and to have spread eastward and westward from this center in a broad band, displacing the earlier, much more precariously supported hunting and food-collecting cultures, until both the Pacific coast of Asia and the Atlantic coasts of Europe and Africa were obtained by about 2500 B.C. Meanwhile, in the nuclear zone from which this diffusion originated, a further development took place, circa 3500 to 2500 B.C., which yielded all the basic elements of the archaic civilizations – writing, the wheel, mathematics, the calendar, kingship, priestcraft, the symbolisms of the temple, taxation, etc. – and the mythological themes specific to this second development were then diffused comparatively rapidly, together with the technological effects, along the ways already blazed, until once again the coasts of the Pacific and Atlantic were obtained. [Campbell 1991a, page 135]

Campbell marks himself as a *diffusionist*, arguing that writing (and much else) were developed in the common center (his "nuclear zone") of the Levant and the Fertile Crescent. The diffusion argument is based primarily upon archaeologically supported dates for writing samples (early evidence of agriculture, etc) found around the world. The alternative, *parallel origin*, theory

suggests that writing (agriculture, etc) may have been invented independently in more than one location. The parallel origin theory is based on the presumption that long distance communication, especially to the Americas, could not have taken place in time for diffusion to have occurred [Farb 1969]. The debate has strong arguments on both sides and has yet to be resolved.

The Sumerian civilization in Mesopotamia appears to have seen the first transition from small villages to substantial cities in the fourth millennium BCE. The need for a centralized government to oversee large irrigation projects for agriculture may have provided the impetus. [Lerner 1998, pp. 13-14] By that time, oral traditions and elaborate artwork encompassed motifs that still echo today in Eastern and Western religions [Campbell 1991b, pp. 39-40]. For many years it was assumed that Sumerians, especially the residents of the ancient city of Uruk, were the inventors of writing. This no longer appears to be the case. Early evidence of writing appears in a broad region stretching from northern Syria to western Iran between 8500 and 3000 BCE. The earliest evidence consists of "tokens", small clay objects in a variety of geometric shapes, and coincides with the uptake of agriculture. [Rogers 2005, pp. 81-84] It appears that these tokens represented a promise to pay, perhaps from one farmer to another. Rogers, building on ideas from Denise Schmandt-Bessarat, suggests, "We can imagine two farmers agreeing: 'In exchange for the twenty bushels of grain I am giving you now, you will give me two sheep in the spring; I will keep these two tokens as a record of the agreement.'" These tokens became more abstract with time in order to represent objects other than animals and, in Schmandt-Bessarat's controversial but compelling thesis, initiated writing.

This history is relevant because it appears that metadata was conceived by proto-writers as a convenient adjunct to these early IOUs. Tokens have been found baked into clay "envelopes" (or bullae) presumably to keep them from being altered or separated before the transaction was completed. Scratched into the surface of some of these envelopes by 3500 BCE were clues to the contents: In other words, metadata; in this case the earliest known example of *attached metadata*.

What we would properly call writing appears gradually, at first consisting of a few pictographs. The cuneiform script developed from the memory aides of the tokens around 3500-3400 BCE, into an ideographic representation of roughly 2000 words by the turn of the millennium, until, around 2600 BCE, it appears to represent a serialization of the entire spoken language. Syllables appeared, as did case and tense [Rogers 2005, pp. 90-91]. Metadata initially developed alongside early writing. By 3000 BCE, Assyrians inscribed notations on clay cylinders to denote the material inside [Stockwell 2001, pp. 93].

The Egyptians, too, wrote early and their language and writing style are very different from the Sumerians. The first evidence of Egyptian writing dates to approximately 3500 BCE and reflects the same administrative content as in Sumer [Rogers 2005, pp. 100]. Much earlier writing is supposed with the presumption being that the medium used was papyrus paper. Papyrus, abundant along Egypt's Nile riverbanks, provided a handy but archaeologically fragile medium for early writing. Some scholars have suggested that the Egyptians may have written before the Sumerians [Casson 2001, pp. 1]. The Sumerians' clay tablets, occasionally hardened by fire, survived the long years better than any papyrus. It is not known how the Egyptians organized their vast writings related to the law, religion and surgery. [Lerner 1998, pp. 18] By contrast, the earliest known Chinese writing dates from 1500 BCE, although it was sufficiently mature by then to point to a much earlier origin; possibly by as much as a millennium. [Stockwell 2001, pp. 10].

The degree to which the Sumerians and their Akkadian conquerors (circa 2500 BCE) developed metadata is not known. It is known however that substantial collections of thousands of tablets were stored in royal proto-libraries. Wiegand and Davis thereby presume, "some system by which documents could be distinguished or classed." [Weigand 1994, pp.23] That may be overstating the case. Later resistance to organized metadata could suggest that priestly scribes (whose special knowledge granted them special privilege) might have resisted careful organization as a deterrent to outsiders. This was certainly the case with Socrates, who did not want his words written down, and with the Chinese who resisted indexing into the nineteenth century because they preferred their scholars to be intimately familiar with their subject matter. [Stockwell 2001, pp. 13 and 26] The existence of a priestly class to deal with writing may be inferred from the mythology of the region: The Epic of Zu from Mesopotamia reports the theft of an artifact called the Destiny Tablet from the chief god Enlil by the winged storm god Zu. The parallel Greek myth has Prometheus stealing fire from Zeus. Writing was treated as a magic desirable to obtain. [Weigand 1994, pp. 24]

There has been no way to prove that a librarian function existed in ancient Sumer. Wiegard and Davis again make the presumption: "…given the very existence of book collections and a place where they are housed, (as in the Greek Βιβλιοφελακιον), it is easy to suppose that there were implied acquisition, classification and maintenance tasks." [Weigand 1994, pp. 25] Easy, yes, but accurately? Perhaps the very idea of classification, if not of acquisition and maintenance, came later. The cataloguing of written materials into classifications would certainly constitute metadata.

The oldest known catalogue of written material dates to 2000 BCE. Two clay tablets were found at Nippur in southern Mesopotamia that listed titles (the first line or so) of other tablets.

They apparently represent the holdings of a library. One lists 68 titles and the other 62; 43 of the titles overlap. They possibly list the holdings of the library at different times. Interestingly, the titles listed are works of Sumerian literature, not the earlier administrative works such as bills of sale, inventories and royal distributions. [Casson 2001, pp. 4] It is possible that *implicit metadata*, based upon shelf order or location, was used to find the administrative works, although no direct proof exists. In any case, these two tablets constitute the earliest known example of *external metadata*.

Colophons are another form of *attached metadata*. The oldest known colophons have been found at Hattusas in modern Turkey and date from the seventeenth to the thirteenth centuries BCE. They are inscribed onto the back of clay tablets containing religious invocations. They acted much like a modern title page and held information regarding the author, title, date created. Information on the number of a tablet in a series and any history of repairs might also be indicated. More advanced catalogues have also been found at Hattusas and dating from the thirteenth century. Those catalogues contain the earliest bibliographic entries of any detail. The number of tablets in a work, and even abstracts of content were represented. [Casson 2001, pp. 4-6]

Direct evidence for the existence of dedicated librarians appears in the eighth century BCE. An Assyrian wall painting depicts two scribes. One of the scribes holds a stylus above a tablet, possibly of wood. The other holds a stylus above a leather or papyrus roll. The second scribe, clearly the inferior from anthropologic details of appearance, has been interpreted as collecting information (metadata) from the actions of the first. By the seventh century, there is no doubt: "At Ninevah, in King Ashurbanipal's library, little clay markers called *girginakku* were found on some tablets. The person responsible for these identifying markers was called a *rab girginakku* or "keeper of the markers". [Weigand 1994, pp. 25] Ashurbanipal ruled from 668 to 626 or 627 BCE. [Casson 2001, pp. 9] *Girginakku* are the first known embodiment of *separable metadata*.

Greeks had adopted a version of the Sumerian cuneiform writing system along with their eastern neighbors during the seventeenth to thirteenth centuries BCE, but they used it for only mundane accounting purposes. The great epics of Homer, if they had been developed yet, were being handed down by the prevailing oral tradition and were only later captured in written form. Invasions by Dorian tribes into the Greek heartland in the twelfth century BCE resulted in a collapse of the Greek political system centered around the Mycenaean palaces. The use of Linear B writing collapsed in Greece with the political system that supported dedicated scribe-priests. Literacy came to a halt in Greece for several centuries. [Casson 2001, pp. 17-30] These so-called

Greek Dark Ages provided a cultural reset for the Greek people and, during the tenth century BCE, resulted in two critical innovations: the Greek city-state and the beginnings of rational thought [Vernant 1982, pp. 10]. It was the latter that resulted in a peculiarly Greek view of the world as static, unchanging. That world view made it possible to conceive of capturing and categorizing all knowledge; and that, in turn, would lead to both a Golden Age of Greek literacy and the widespread development and use of metadata. [Stockwell 2001, pp. 13]

The Greek Golden Age produced an explosion in literacy once the Greeks readopted writing by adapting an alphabetic form of script from the Phoenicians. It was suddenly possible for most children to learn to read. More to the point, the removal of the old kings had removed the priestly scribe class and the sense of religious awe surrounding writing. Libraries and the collection of books, however, were not popular until the third century BCE when Greek *gymnasia* began fulfilling roles for the public building both body and mind. [Jackson 1974, pp. 7-9]

The philosopher Aristotle (384 - 322 BCE, Ἀριστοτέλης) and his student Alexander (later to become "the Great", 356 - 323 BCE, Μέγας Αλέξανδρος) assembled some of the first large personal libraries in the fourth century BCE. The Greek historian Strabo said of Aristotle that "He was the first to have put together a collection of books and to have taught the kings of Egypt how to arrange a library." That may not be literally the case. The "kings of Egypt" referred to are generally considered to be the Ptolemaic rulers following the collapse of Alexander's empire in the last years of the fourth century BCE and the "library" is most likely to be the great Library of Alexandria, in a city created by Alexander to avoid the treacherous Nile Delta. Historians disagree whether it was Aristotle or his student Demetrius of Phalerum was responsible for Strabo's first century CE assertion. [Casson 2001, pp. 29] Regardless who taught the kings of Egypt, it seems that Aristotle did produce the first hierarchical categorization system for library organization. Variations on that theme (with relatively minor non-hierarchical adjustments) dominate library categorizations through to the present time. We return to this topic in more detail in Chapter IV.

Aristotle's contemporary and rival Speusippos (408 – 339 BCE, Σπεύσιππος) took over the running of Plato's Academy following their teacher's death. Speusippos was interested in writing down and categorizing his knowledge, especially his knowledge of biology. Strabo tells us that he created the first encyclopedia. [Stockwell 2001, pp. 15] Encyclopedias are particularly interesting to the study of metadata because, as Stockwell puts it:

> It is conceivable to view printed encyclopedias as early attempts to produce hypertext on paper. They were not designed for linear,

> sequential reading, and much of their usefulness depends on indexing, cross-referencing, and various bibliographies that were inserted within the context of the individual articles themselves. [Stockwell 2001, pp. 177]

Encyclopedias, even in the fragments that survive of Speusippos', bring together multiple forms of attached metadata; attached, descriptive, preservation and use (see Table 3-1).

Alexander the Great traveled with part of his Greek library throughout his campaigns in Asia. Interestingly, he discovered first hand proof that not all knowledge could be captured and catalogued. Upon entering the Indus Valley near Taxila he asked to speak with the local philosophers and was taken to meet with Indian holy men sitting naked upon hot rocks. His officers were informed that one could not come to understand philosophy while clothed. They were invited to strip and sit, but refused. [Stockwell 2001, pp. 16] Had they taken away the lesson that all knowledge could not be confined to a book, the great Greek libraries may never have existed. However, paying attention could have led them to a non-Aristotlean method of describing the information they collected.

The Ptolemaic rulers of Egypt, Greeks themselves, intended to make the Library of Alexandria "a comprehensive repository of Greek writings" [Casson 2001, pp. 35]. Works were bought, stolen and copied from sources throughout the Greek world. According to Lerner, "When a shipment of books arrived in Alexandria it was marked with the name of the person who had brought them and then stored in a warehouse until the library staff had time to process the books. When they were brought to the library, the accession staff would label each roll either with its physical origin (where it came from or who had owned it) or its intellectual pedigree (who had edited or corrected the particular copy). This information would enable the cataloguers to determine which versions of a particular work the library owned." [Lerner 1998, pp. 28] The collected metadata was often recorded on a leather tag that was then attached to a roll (see Figure 3-1). The "tags" used today to mark up textual file formats such as HTML and XML are the intellectual descendents of this system.

The first head librarian at Alexandria was Zonodotus (ca. 280 BCE, Ζηνόδοτος). Renowned as the first editor of Homer, he also established the arrangement of rolls in the library. Rolls were placed into rooms by subject and arranged alphabetically by the first letter of the authors' names (the first provable use of *implicit metadata*). Full alphabetization by author's names did not occur until the second century CE. [Casson 2001, pp. 37]

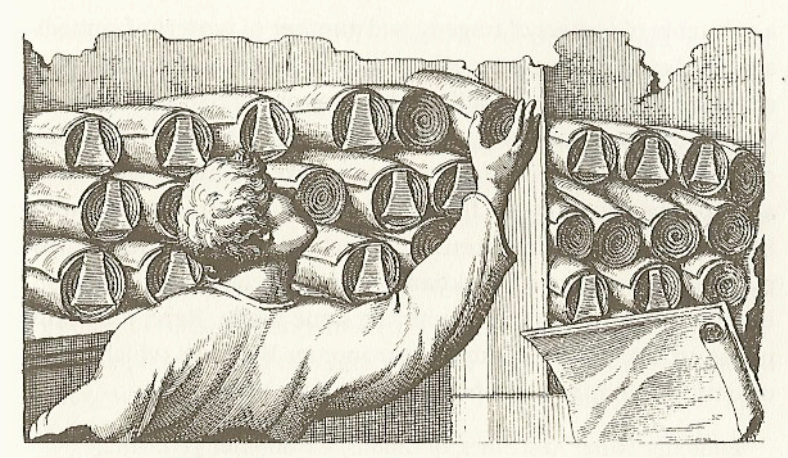


Figure 3-1. Line drawing of a Roman relief showing rolls with literal tags containing metadata (after Casson 2001, Figure 3.1)

Callimachus of Cyrenae (ca. 305 BC- ca. 240 BC, Καλλίμαχος) worked at the Library of Alexandria and began several bibliographic projects. He was a poet (some of his poetry has survived), and may have been head librarian (although this is debated) from 260-240 BCE. His huge bibliographic work, the *Pinakes*, covered all known Greek writers and is considered by many to have served as a catalogue for the massive holdings of the Library. Having grown to a reported 600,000 rolls, the Library certainly needed a catalogue. The *Pinakes* (Greek for "tables" and short for its full title of "Tables of Persons Eminent in Every Branch of Learning and a List of their Writings") filled 120 rolls. Although lost in its entirety, surviving quotes and remnants exist and serve to provide the flavor of its organization. The names of each writer were arranged alphabetically within categories developed by Callimachus. Each entry included a brief biography and a list of their works. At 120 rolls, five times the size of Homer's *Illiad*, the *Pinakes* brought the development of metadata to a new level [Casson 2001, pp. 39-40]. Coupled with a separate shelf list (drawn up in advance of the *Pinakes*, which was derived from it), an author's work could be determined to exist and then located within the library.

The categories of Callimachus should be seen as iteration on Aristotle's and provide a glimpse into Greek thought. The top-level categories were Oratory, History, Laws, Philosophy, Medicine, Lyric Poetry, Tragedy and Miscellany [Harris 1995, pp. 45], or perhaps Medicine should be replaced with Rhetoric [Jackson 1974, pp. 14]. I suspect that the Greek root for both medicine and physics (*phusikē*, meaning a knowledge of nature) has led to translation errors into English. In any case, Callimachus' efforts earned him remembrance as the "father of bibliography". Later librarians at Alexandria carried on his work and, eventually, comprehensive catalogues became standard practice in the libraries of the world.

Callimachus, though, had a problem with his categorization system. He himself wrote poetry and also prose. How did he categorize himself? Were his entries duplicated or cross-referenced or collapsed or simply left out? [Casson 2001, pp. 40-41] There is no way to know his solution to this problem since the *Pinakes* have not survived. It is clear, though, that the problem was introduced by use of hierarchies. We refer to this problem as Callimachus' Quandary. It would not be solved in the general case for twenty-two and a half centuries.

The Library of Alexandria was, by far, the largest and most complete library on Earth for nearly a millennium. It was challenged, but not surpassed, during the Roman era. Its innovative use of metadata to organize and make accessible its huge collection of rolls was its singular success. Its management foreshadowed the management of libraries up to modern times. The fate of the Library of Alexandria is not known in detail, other than it eventually ceased to be. Wars and religious strife took their toll. The burning of the city and scattering of the library's staff during the Egyptian civil war of 89-88 BCE was surely a blow to the Library. Various scholars have suggested that the library was effectively closed after Julius Ceasar's invasion of 47 BCE [Gates 1968, pp. 15] or Aurelian's in 273 CE [Casson 2001, pp. 47]. The library seems to have slowly become less a center of learning during the Roman era and certainly by the time of Christian purges of pagan books in 391 CE. [Jackson 1974, pp. 17] [Harris 1995, pp. 46-47] Perhaps the most likely cause of the great library's death was simply neglect. The Greek view, carried on by the Romans, of the unchanging world left nothing left to do once knowledge was collected. The rolls of papyrus slowly rotted away until they became useless. Eventually they were burned to heat the public baths. [Stockwell 2001, pp. 10] If that was indeed the case, then the slow death of the Library of Alexandria resonates today in our discussion of the maintenance of knowledge and of knowledge systems. One might draw a reasonable comparison with the "bit rot" of software discussed in Chapter I.

What then of the great civilizations of the East? The earliest known writing in East Asia was found in the town of Anyang in central China. It consisted of scratches on "oracle bones" used for divination during the Shang dynasty. The oldest dated to about 1200 BCE, although the date is often misquoted in the literature as 1300 BCE. [e.g. Lovell 2006, pp. 29] A much earlier developmental date may be presumed from the 4500 signs already present on the oracle bones. [Rogers 2005, pp. 30] The writing system was perfected in northern China and spread rapidly. China's early political unity helped to ensure that a single writing system was used throughout the large region. [Diamond 1998, pp. 331 and 413]

China appears to have had libraries early in its history. Thornton reports:

> We read that in 221 B.C. Shih Huang Ti, the founder of the Ch'in Dynasty, ordered the destruction of all books except those on agriculture, divination and medicine, this law being repealed in 190 B.C. In the year 124 B.C. the Emperor Wu Ti established a national university for the study of the restored Confucian classics, and it is apparent that libraries must have existed in this country at a very early period. [Thornton 1941, pp. 13] (NB: The Ch'in Dynasty is now commonly referred to as the "Qin" when using the Pinyin system of Romanization.)

In spite of a Western bias toward Marcus Terentius Varro's work *Imagines*, part of his lost *Disciplinarum libri IX* encyclopedia written in the first century BCE, the earliest known book containing both text and pictures dates from the Western Han dynasty of the Three Kingdoms Period (late second century BCE). Paper was invented in China around the end of that millennium.

Very little, however, was developed in the way of Chinese metadata. Chinese scholars were expected to know their subjects intimately. Indexes and bibliographies ran contrary to the testing system used for the imperial bureaucracy. Some categorization could not be escaped however. A book classification scheme was developed in the first century CE with seven divisions at the top level. Two centuries later, a modification resulted in four top-level categories (classics, philosophy, history and history of literature). The latter scheme may have influenced Francis Bacon's sixteenth century CE three-category scheme (history, poetry, philosophy) which served as the basis for modern classification schemes in the West. [Stockwell 2001, pp. 23]

Metadata can be viewed as an invention of the Near East with substantial refinements by the Greeks. Its spread westward formed the basis for Western libraries. Reasons of culture and governmental authority tended to suppress the use of metadata in the east.

The key elements for organizing content had been defined in antiquity. The Greek/Roman approach to gathering knowledge ended with the Visigothic/Vandal (410 CE) and Ostrogothic (489 CE) invasions of Rome. Fortunately, the lessons of knowledge organization via metadata were retained and reused during the next fifteen hundred years. Further developments tended be those of scale, not of concept, until the invention of the digital computer forced a rethinking of knowledge organization. Figure 3-2 summarizes the development of key metadata concepts developed in ancient times.

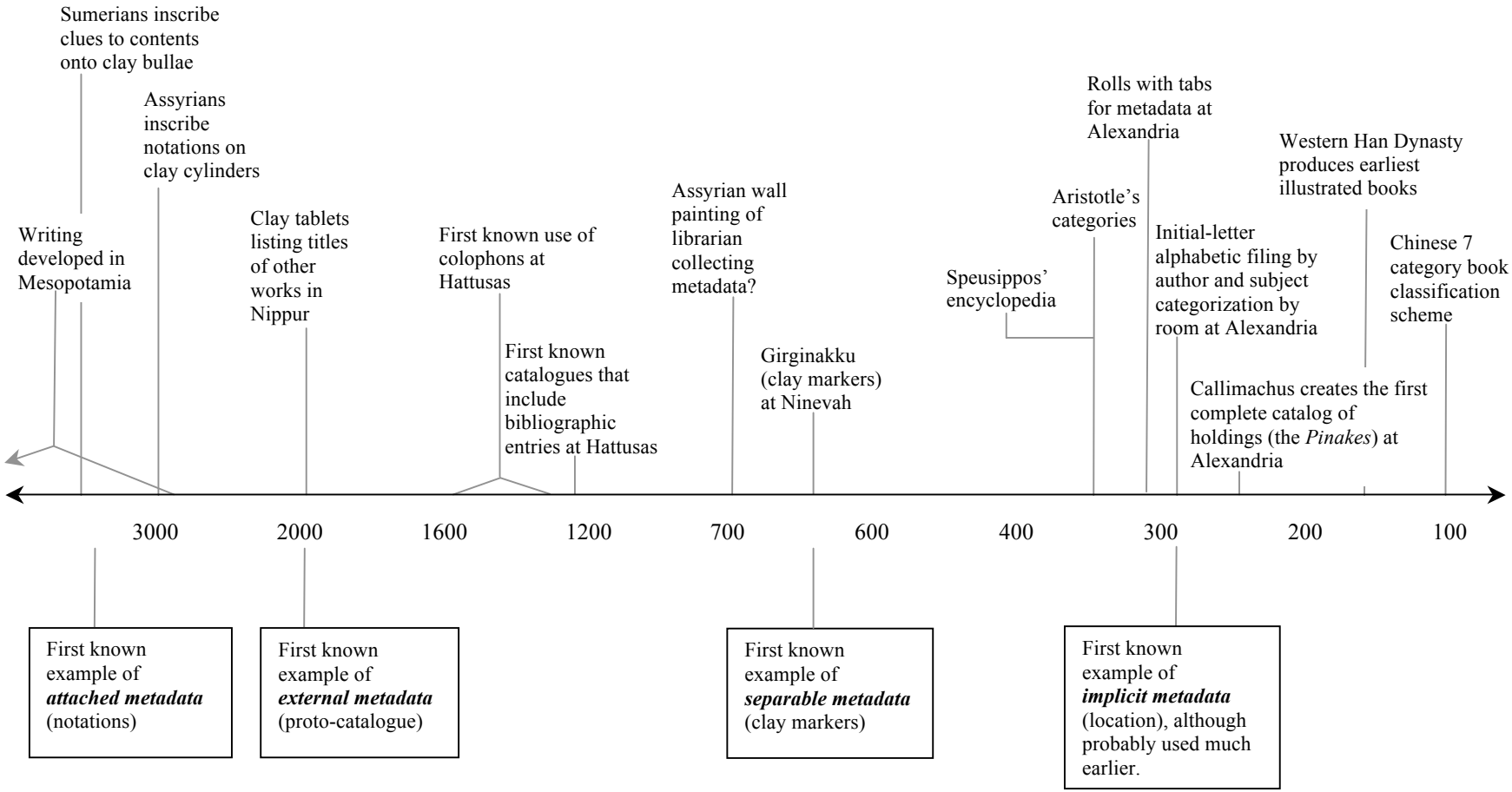


Figure 3-2. Timeline of Early Metadata Development (Years BCE, dates approximate)

3.3 Medieval, Renaissance and pre-Modern Metadata

Western thinking about the organization of knowledge continued along the path set by the Greeks, in that attempts were routinely made to gather the "most important" knowledge together [Stockwell 2001, pp. 97]. Only minor advances were made in metadata concepts though until the end of the nineteenth century. Librarians focused on classification and indexing during these periods. Metadata was viewed as an implementation mechanism for those activities.

A minor medieval advance in the development of metadata was made by Antonio Zara, bishop of Petina, in his *Anatomia Igeniorum et Scientiarum* ("Anatomy of Talents and Sciences") in 1614. This encyclopedia contained the first encyclopedic index. Johann Jacob Hoffman created an index for his encyclopedia *Lexicon Universale* in 1677. These early indexes were of a rough grade, including relatively few terms and inadequately dealing with polymorphism. The first "acceptable" indexes for encyclopedias were not developed until the 1830s [Stockwell 2001, pp. 101].

Indexes, as useful as they are, address Callimachus' Quandary in a very simplistic and limited way. The further development of categorization schemes was also undertaken to ease the burden of finding information scattered within and across texts. The Renaissance brought a rediscovery of ancient Greek texts and ideas to the West and a new way of thinking about old problems. Significant scientific advances including the works of Galileo and Newton were predicated on a type of analysis, "the practice of isolating a physical system from the outside universe and focusing attention on the phenomenon of interest." [Davies 1992, pp. 37-38] That very non-Aristotlean idea precipitated an explosion of new discoveries in everything from physics to biology.

Encyclopedists further experimented with three types of classification: the systematic method (similar to the bottom-up approach taken by Carolus Linnaeus in the eighteenth century CE for biological classification), alphabetical by broad subjects and alphabetical by specific subject. Over time, the alphabetical system by specific subject won out because it was both more approachable for new users and required less movement from section to section. The other methods are still in occasional evidence but are no longer used as the primary indexing scheme.

Francis Bacon's three-category classification scheme developed in 1605 and mentioned earlier led to the development of several other subject-based classifications for books: The Dewey Decimal System (1876), The Library of Congress Classification System (1897, based on Dewey but many more major and minor categories), Cutter Expansive Classification (1904), Bibliographic Classification and S.R. Ranganthan's Colon Classification (used mostly in India) [Gates 1968 pp.

145-147, Stockwell 2001 pp. 95-98, Cutter 1904]. The Dewey Decimal System was created by American librarian Melvil Dewey, and has been owned by the non-profit Online Computer Library Consortium (OCLC) since 1988. It is updated periodically for both new content categories and to refine linkages between categories. All of these systems retain a strong bias toward hierarchy traceable directly to Aristotle's original hierarchical categorization scheme, although cross-referencing has become commonplace in all of them to partially address Callimachus' Quandary.

Universal Decimal Classification, developed between 1904 and 1907 by Belgians Paul Otlet and Henri La Fontaine, is a significant iteration to the above classification schemes, although it was itself based on Dewey. Faceted classification, the ability to classify a single entity in multiple ways, was pioneered by Ranganthan but fully implemented in UDC. UDC was "the first – and one of the only – full implementations of a faceted classification scheme" [Wright 2007, pp. 187].

The twentieth century brought new concerns, including totalitarian regimes on a grand scale. Responses within Western libraries, especially in America, provided the first indication of forms of metadata deemed "unacceptable". Labeling of content in totalitarian countries as contrary to prevailing political thought resulted in somewhat of a rebellion among American librarians. In 1951 the American Library Association produced its Statement on Labeling denouncing the practice of labeling library holding for political purposes, saying, "Labeling is an attempt to prejudice the reader, and as such, it is the censor's tool." Anti-communist hysteria of the time prompted them to defend their position against both perceived domestic as well as foreign threats: "We are, then, anticommunist, but we are also opposed to any other group which aims at closing any path to knowledge." [Gates 1968, pp. 366-7] Metadata had found its first limit. It would not be the last.

Chapter IV builds on the description of metadata developed in this chapter and describes metadata as it applies to virtual resources. A special focus is given to the World Wide Web and to the application of metadata to the descriptions of software systems.

IV. Metadata Applied to Software Systems

"I have yet to see any problem, however complicated, which,
when you looked at it in the right way, did not become still more complicated."
-- Poul Anderson (1926 - 2001)

4.1 Virtual Metadata Before the World Wide Web

Paul Otlet presaged some of the key concepts of the World Wide Web in 1934 [Wright 2007, pp. 184-192]. Otlet articulated a vision for a "universal book" and is credited with coining the now-familiar terms "link" and "web" (or "network" from the French *réseau*) to describe the navigation of his "book". He was able to get substantial funding from the Belgian government for the realization of his work (the *Mundaneum*), which operated between 1910 and 1934 in the form of a massive collection of index cards and responded to queries by postal mail. A museum version of the Mundaneum was opened in 1996 with the surviving elements of the original. Otlet's web was a web of metadata; his index cards provided multi-subject faceted classification across and between resources. Although implemented in physical form, the faceted classification system pioneered by Otlet is the intellectual precursor to modern systems using metadata-based information organization, including the World Wide Web.

Vannevar Bush followed Otlet, although there is no indication that the two knew of each other's work. Bush's well known treatise As We May Think set the tone for the application of then-modern technologies to both library science and non-library holdings [Bush 1945]. He noted the problems with finding and organizing knowledge resources and the lack of recent progress in the field:

> The difficulty seems to be, not so much that we publish unduly in view of the extent and variety of present day interests, but rather that publication has been extended far beyond our present ability to make real use of the record. The summation of human experience is being expanded at a prodigious rate, and the means we use for threading

> through the consequent maze to the momentarily important item is the same as was used in the days of square-rigged ships.

Bush's suggestions were not just of recording knowledge resources, but of organizing them better for search and retrieval. He suggested that the individual human memory be augmented with an external device that he famously called a "memex". The concept of the memex is considered by many to be the intellectual precursor to today's hypertext systems [e.g. Nelson 1972 and Alesso 2006] However, the memex was specifically intended to be a device used by an individual – in this Bush revealed his own heritage of Greek individualism. The librarians had larger problems connecting knowledge resources, but it took decades for digital computers to become mature enough for librarians to contemplate them as meaningful alternatives to card-based indexing systems.

Bush did not explain precisely how the memex was to be implemented, nor did he mention the term "metadata". He did, however, intimate that a form of metadata would be collected, stored and used for later retrieval: "Wholly new forms of encyclopedias will appear, ready-made with a mesh of associative trails running through them, ready to be dropped into the memex and there amplified." [Bush 1945] There may be a good reason to use an associative model to describe information: Human memories have been claimed to be associative in nature [Collins 1975], and recent functional magnetic resonance imaging studies lends credence to that view [Mitchell 2008]. Viewed from the modern perspective, it is difficult not to suggest that Bush's "associative trails" are a collection of metadata.

By 1965 the librarian Jesse Shera was ready to formulate some "axiomatic principles for the classification of bibliographic materials". As a librarian, he thought that "*every attempt*" (emphasis in original) to analyze bibliographic materials must be made upon the basis of classification. This was a traditional and non-controversial thought in his field and dated back directly to Aristotle. In other ways, though, he was years ahead of his time. He noted the problems caused by hierarchical thinking and suggested a solution:

> The Aristotelian concept of the hierarchy must be rejected as a basic principle of classification if classification is to adapt itself to constantly changing needs, uses, and points of view. This is so because a hierarchy, of whatever structural design, implies a specific philosophical orientation, and the crystallization of a hierarchical structure into a *particular* orientation precludes, or

> does violence to, all others. [Shera 1965, pp. 104] (emphasis in original)

Shera had established the need for a general classification scheme that was non-hierarchical (like Otlet's) and, in the terminology of logic, "Open World". He made a call, twenty-four centuries in the making, for a general solution to Callimachus' Quandary (see Chapter III). The call could not be heeded until the digital computer made adequate inroads into both libraries and the general public. The "See Also" references in encyclopedias and categorization systems may implement a poor man's hypertext, but a general solution required a more flexible medium than paper.

Shera also noted that bibliographic classification be "completely *independent* of physical objects (books), for no arrangement of such objects can reveal the complex of relationships that may exist among the many separate thought-units that make up the subject content of these books." His solution was for librarians to give up not only hierarchical thinking but also any idea that their job was the protection of books. He wanted to redirect their dedication to "the *ideas* the books contain." [ibid. pp. 105] Shera, alone of library theorists, presaged the changes that would come over the discipline of library science with the advent of digital computers.

The idea that ideas matter more than form was compelling, but the means to reorganize library holdings around such a concept could not be implemented without inexpensive digital computers. Knowledge resources, regardless of their physicality, would have to be stored and maintained in a manner consistent with their content. Their form matters less. The preservationist Paul Conway put it this way, in 1996 (nearly a half century after Shera's call to arms):

> The digital world transforms traditional preservation concepts from protecting the physical integrity of the object to specifying the creation and maintenance of the object whose intellectual integrity is its primary characteristic. [Gilliland-Swetland 1998]

The canonical source for the above quote is no longer available on the World Wide Web, highlighting the new classes of difficulty in digital reference and preservation.

Librarians adopting computers initially carried over the types of metadata used during the last two millennia (*attached* metadata like indexes and *external* metadata like catalogues), without reference to the concepts of *separable* or *implicit* metadata used in antiquity. Physical card catalogues became virtual card catalogues. The U.S. Library of Congress established a system to make the translation from physical to virtual card catalogues in the 1960s. The Machine-Readable

Cataloging (MARC) system became a de facto industry standard. The current version of the format, called MARC 21, is still maintained by the Library of Congress and used worldwide. Unfortunately, the MARC format is arcane at best due its early history. Memory and storage were expensive in the 1960s and software developers responded by using the smallest possible identifiers for each item stored. A MARC record containing a "signpost" of "100 1# $a" can be read only by a highly trained user. A casual observer would have a difficult time translating that string into "Main entry, personal name with a single surname". [Furrie 2003] Contrast that emphasis on small storage space with today's XML. XML chose verbosity to ease human readability of content primarily intended for machine use. The switch from succinct to verbose tagging was only possible once the costs of computation and digital storage dropped dramatically in the 1980s and 1990s.

Librarians, once the keepers of metadata formats and standards, became somewhat sidelined as computing technology developed. Libraries became users of the new technologies, but not developers of it. The Online Community Library Consortium (OCLC), for example, was founded in 1967 to use computer networking to facilitate sharing of resources between U.S. libraries. Today, the inter-library search product WorldCat is used by more than 57,000 libraries in 112 countries. OCLC is a non-profit corporation supported by its member libraries. The World Wide Web, widely seen at that time as a threat to the very existence of libraries, spurred OCLC researchers to facilitate the adoption of metadata for the "finding, sharing and management of information". The Dublin Core Metadata Initiative was founded by OCLC to do that in 1995. The Dublin Core element set, consisting of tags to describe both physical and virtual knowledge resources, was later adopted by the International Organization for Standardization (ISO) as ISO standard 15836. Dublin Core metadata is now used as the basis for both Open Source and proprietary card cataloguing systems (e.g. the Vital and Valet products from VTLS, Inc and the Fedora digital content repository). Dublin Core metadata is verbose enough for human readers, unlike MARC, and has become widely used for other purposes by Semantic Web applications, as described in the next section.

The computer science community fielded many application-specific metadata solutions from the 1970s to 1990s. These ranged from file system metadata at the operating system level (a form of *implicit* metadata) to metadata describing the schemas of relational databases and data warehouses (typically *attached* metadata). Information retrieval systems built upon private data collections (e.g. legal documents collected by Lexis Nexis or West Group) augmented full text searches by attaching metadata to documents prior to indexing (*attached* metadata). Metadata also appeared in mass-market document file formats, such as Microsoft Office properties and

Exchangeable Image File Format (EXIF) metadata for digital camera image formats [JEITA 2002] during the same period of time.

Embedded forms of *attached* metadata paved the way for the forms of *attached/separable* metadata to come in later document format standards, such as XML and HTML. Metadata that is attached to virtual resources may be separated for other uses. Unlike the *Girginakku* discussed in Chapter III, metadata attached to virtual resources may be easily and non-destructively copied without removing it from its container. Such metadata may be classified as *attached/separable* to distinguish it from the *separable* metadata defined in ancient times. *Attached/separable* metadata only applies to virtual resources. The ability to separate metadata from virtual resources has ramifications regarding the use of metadata associated with virtual resources: It leads to a change in the best practice of metadata application.

Hypertext systems use metadata (hyperlinks) to associate different information resources. Hypertext systems predated the World Wide Web by many years. Ted Nelson coined the term hypertext (in [Nelson 1965]) and envisioned a complex hypertext system called Xanadu [Nelson 1987]. Xanadu was a complicated system that attempted to transcend properties of physical books, such as the notion of pages, and was to include automated management of bi-directional hyperlinks. Xanadu's concepts have not been fully implemented, although periodic software releases are accessible via, ironically, the World Wide Web.

Nelson's celebrated invention of the hyperlink should be seen in context, as Otlet's links implemented in software. Nelson's Xanadu should also be viewed in context, as iteration on Bush's Memex that allowed multiple users to share a centralized data store.

Important, early implementations of the hypertext concept included Doug Englebart's oN Line System (NLS) in the late 1960s [Berners-Lee 2000, pp. 6], Randy Trigg's Notecards [Halasz 1987], Akscyn's KMS [Akscyn 1987] and Apple Computer's HyperCard system started in 1985 and released in 1987 [Wright 2007, pp. 223]. Lesser-known, but important in the development of the World Wide Web was Enquire, Tim-Berners-Lee's first attempt at building a hypertext system in 1980 [Berners-Lee 2000, pp. 4]. Englebart, at least, was directly inspired by Bush [Nyce 1991, pp. 128].

Early hypertext systems were centralized applications that stored information in a single repository. Hyperlinks, as in Nelson's vision, were bi-directional and maintained; both linking and target entities were updated when one of them changed. Berners-Lee's later iteration of the ideas from Otlet, Bush, Nelson and Englebart led to his changing a single presumption that would lead to

the World Wide Web; distribution of hypertext information across multiple locations, which in turn necessitated the Uniform Resource Locator (URL) to address those locations [Berners-Lee 1996].

Berners-Lee's breakthrough simplification, hypertext through distribution, solved a long-standing limit to scalability of hypertext systems. That simplification was also responsible for a so-called "link maintenance problem"; hyperlinks became unidirectional in practice. Although some Web servers attempt to track references from hyperlinks ("back-links") within the confines of their own data sets (notably wikis), the link maintenance problem led hypertext pioneer Ted Nelson to sneer, "The reaction of the hypertext research community to the World Web is like finding out that you have a fully grown child. And it's a delinquent." [Gill 1998]

Hyperlinks within a hypertext document describing the content of the document are *attached/separable* metadata. They may be separated from virtual resources, for example by search applications to yield a map of inter-document relationships.

Hyperlinks within a hypertext document describing the content of another resource are acting as *external* metadata, while retaining the capability of separation for reuse. *External* metadata in the form of hyperlinks are relatively difficult to find, collect and identify on the World Wide Web due to the link maintenance problem and one reason Web search engines have gained widespread popularity in recent years.

Berners-Lee chose to use an application of the Structured Generalized Markup Language (SGML), the Hypertext Markup Language (HTML), as a presentation container [Berners-Lee 1995]. HTML contains both a resource's content and metadata, including hyperlinks. The tag structure of HTML and related markup languages may itself be considered to be *attached/separable* metadata.

Metadata, initially considered as a way to find information, thus became a critical component in the way information was structured, described, prepared for reuse and presented to human readers.

4.2 Metadata on the World Wide Web

The World Wide Web is an information retrieval system built upon a foundation of metadata. Metadata was a first class component in the earliest proposal for the Web, although it was not then called "metadata" [Berners-Lee 1989]. The first version of the Hypertext Transfer Protocol (HTTP) considered by the Internet Engineering Task Force (IETF) used *attached/separable*

metadata in message headers (adapted from the Internet's Simple Mail Transfer Protocol) to identify a requested resource, provide information about client capabilities and describe content that a message contains [Berners-Lee 1996]. Similarly, the first standardized version of Hypertext Markup Language (HTML), called version 2.0 to avoid confusion with earlier drafts, used *attached/separable* metadata to identify hyperlinks and arbitrary statements that an author may wish to make about content (the META tag) [Berners-Lee 1995].

The implicit relationship of metadata to content in the World Wide Web is essential to its operations. The ability to reuse *attached/separable* metadata allows Web browsers to follow hyperlinks, and also allows Web search engines to find (and better index) resources, and meta-applications that transcend search. Web-based social networks rely on reusing *external* metadata (Uniform Resource Locators or URLs) and *attached/separable* metadata (hyperlinks) and adding new, related *attached/separable* metadata (microformat tags).

Berners-Lee intended that metadata take on a wider role as the Web matured [Berners-Lee 1997]. He introduced the concept of machine-understandable metadata, which he defined as "information which software agents can use in order to make life easier for us, ensure we obey our principles, the law, check that we can trust what we are doing, and make everything work more smoothly and rapidly. Metadata has well defined semantics and structure." The concept of machine-understandable metadata led to a "road map" for its implementation [Berners-Lee 1998] and eventually to standardization efforts at the W3C.

Otlet noted the social value of metadata in his *réseau*, although mostly the World Wide Web's implicit relationships remain untracked and unexploited:

> In the Web's current incarnation, individual "authors" (including both individuals and institutions) exert direct control over fixed documents. It takes a meta-application like Google or Yahoo! to discover the broader relationships between documents (usually through some combination of syntax, semantics and reputation). But those relationships, however sophisticated the algorithm, remain largely unexposed to the end user, never becoming an explicit part of the document's history. In Otlet's vision those pathways constituted vital information, making up a third dimension of social context that made his system so potentially revolutionary. [Wright 2007, pp. 191]

Exploitation of relationships between resources on the Web remains incomplete for an excellent reason: The very design decision that allowed the Web to scale (distribution of resource servers) removed the ability to monitor links bidirectionally (the link maintenance problem). Some services exist to track and retain inter-resource metadata, such as Technorati, a service for tracking the relationships between information resources (especially Web logs ("blogs") and social content tagged with microformats). Technorati requires a publisher to register with them so that inter-resource monitoring may be accomplished. The majority of Web resources are not so registered and thus may not be monitored; Google claimed to have indexed roughly 6 billion documents in 2004 [Olsen 2004] whereas Technorati monitors roughly 110 million blogs (each consisting of multiple documents) [Technorati 2008].

Use of metadata on the Web has not been a linear process. Research and implementations have used metadata in varying ways. Key to the differing approaches is who controls the metadata and which metadata is required. Some metadata is required (such as necessary HTTP headers and the use of URLs) and is necessary to ensure consistent interactions between Web clients and servers. Required metadata is also known as authoritative metadata [Fielding 2006]. Other metadata is optional (such as the HTTP "Accept" header or various ways to describe a document's author, provenance or meaning). In the early Web, authors were also publishers, a situation that is not as generally true today, and metadata was seen as the province of authors.

The HTML META tag placed the power to define metadata about an information resource in the hands of the resource author. Users of the resource were thus required to trust, or decide not to trust, authors. Unfortunately, most of information used by people in social settings to determine trust "is not available (e.g. we do not know the history between people, the user's own background and how likely they are to trust in general, the familial/business/friend relationship between users, etc.). Thus, we must understand trust from only available information." [Golbeck 2006]. Users of information resources on the Web may choose to trust metadata from sources where trust is established outside of the Web relationship (e.g. by brand, employment relationship, or subscription to curated content). Contrarily, trust is generally not established between Web users and authors who may have something to gain by lying in their metadata.

Early abuses of the HTML META tag included the placement of words related to pornography in non-pornographic content for the purpose of impacting search engine result ordering [E.J. Miller (personal communication, January 26, 2008)]. The prevalence of pornographic content on the Web and the desire by many to protect children from (intentional or inadvertent) access to it led to

several proposals for legislative censorship in the United States (especially [Communications Decency Act of 1996]). The World Wide Web Consortium attempted to develop a metadata standard for the Web known as the Platform for Internet Content Selection (PICS) to forestall legislation [E.J. Miller (personal communication, January 26, 2008)]. PICS was standardized by the W3C in 1996 and included a new feature for metadata management; the ability for end users to subscribe to third party "rating services" [Miller 1996]. Rating services were to provide metadata about content that users could trust (a "brand"). PICS attempted to add third-party (*external*) metadata to the Web.

PICS failed to gain widespread acceptance, in spite of being implemented by many pornographic content providers and search engines. The primary reason was that URL-based censorship schemes were rendered less effective by changing URLs for given resources [E.J. Miller (personal communication, January 26, 2008)]. Browser makers ignored arguments by the PICS community that PICS' third-party rating services could provide trust without censorship [Resnick 1996]. Lack of client support doomed widespread use of the PICS standard. The W3C is trying again to create community consensus around a PICS-like standard for Web content control with the Protocol for Web Description Resources (POWDER) Working Group [Archer 2007]. POWDER's future standardization and use remains uncertain as of this writing.

Ramanathan Guha, while working on the Cyc knowledge base [Lenat 1990], developed an iteration of the knowledge base formulation most commonly used in symbolic artificial intelligence systems [Guha 1995]. Guha introduced the notion of context into AI knowledge bases and formalized a description for context information. He later used those ideas to develop metadata descriptions of Web site changes, first the Meta Content Framework (MCF) [Guha 1997] and later RDF Site Summaries (RSS) [Guha 1999].

Lessons learned from the implementations of Guha's work on contexts, PICS and Dublin Core metadata [Weibel 1998] led to a standard for the description of metadata on the World Wide Web, the Resource Description Framework (RDF) [E.J. Miller (personal communication, January 26, 2008)]. RDF [Manola 2004] is a standardized mechanism for deploying metadata on the World Wide Web.

Lessons from the deployment of the HTML META tag led supporters of RDF to conclude that some issues of trust are best solved socially [E.J. Miller (personal communication, January 26, 2008)]. In RDF, as in PICS or RSS, the author or publisher of information may choose to lie and the consumer of the information may choose whether to believe. Trust relationships, a social

consideration, are necessary to determine how to use the information contained in metadata. Trust relationships must exist for metadata to be useful (e.g., trust of a curation source such as an employer or service provider, trust that a user wouldn't lie about their own contact information in a FOAF document, or trusting the aggregate judgment of a many people in a social network) [Golbeck 2005]. The RDF standard attempts to facilitate trust relationships by allowing metadata to be *attached/separated* and/or *external*, so that authors, publishers and users may each have some ability to describe Web-based resources. Figure 4-1 illustrates the relationship between authors, publishers and users on the World Wide Web. Authors, publishers and users may all add RDF metadata to Web resources. Metadata may be cached at Web clients and servers.

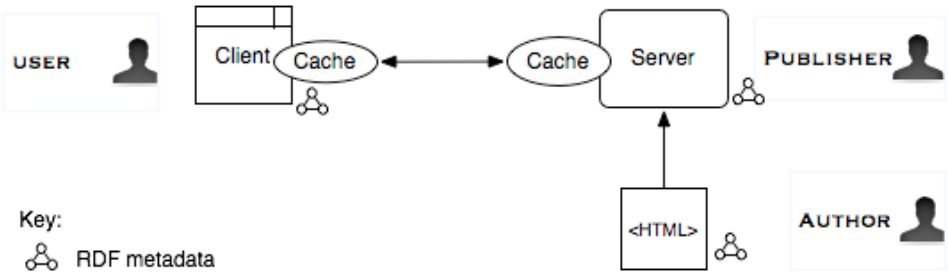


Figure 4-1. Metadata may be applied to Web resources by authors, publishers and/or users.

RDF provides a number of critical enhancements to the art of metadata. Callimachus' Quandary is (at last) addressed with the appearance of an international standard based on a graph data structure. The removal of Aristotle's hierarchical thinking from the library and computer science communities may well be decades in coming, but coming it is. Hierarchies may be seen as they should be; as degenerate conditions of the more general graph structure.

RDF metadata, like any other data structure, requires some form of vocabulary description. Three standards-track vocabulary description languages have been defined for RDF: RDF Schema (RDFS) [Brickley 2004], the Web Ontology Language (OWL) [McGuinness 2004] and the Simple Knowledge Organization System (SKOS) [Miles 2008]. RDFS provides a standard, but quite limited, means for describing information in RDF. OWL extends RDFS to facilitate logical processing of that information. SKOS, the only one of the three not yet a W3C Recommendation as of this writing, defines a common framework for representing (typically, but not necessarily, hierarchical and previously existing) knowledge organization systems in RDF.

The creation of the RDF standard, a graph-based solution to Callimachus' Quandary, is only theoretically sufficient to challenge more widely implemented hierarchical data structures such as

relational databases. RDF-based technologies must also be fielded. *External* RDF metadata may be created, stored and analyzed in databases capable of holding RDF metadata. Although it is possible to store RDF statements in conventional relational databases, and this is sometimes done, it is far more efficient to use a database with explicit RDF support. Dedicated RDF databases include Mulgara and its predecessor Kowari [Wood 2005] and RDF libraries backed by a choice of databases include Redland [Beckett 2006] and RDFLib [Krech 2005]. Oracle Corporation's flagship database also ships with RDF support as of version 10g.

The use of RDF has some advantages and disadvantages as compared to the more commonly fielded relational database model. RDF, while itself a "relational" model, is a more open data model than the extended hierarchy supported by relational databases. The disadvantage of RDF is obvious: Relational databases are widely deployed, available and supported. RDF has at least two significant advantages beyond providing a generalized resolution to Callimachus' Quandary: It is built upon a Web foundation, allowing immediate potential for distribution and it allows for flexibility of schema.

Relational databases have proven to be a profitable technology and the presence of profit has naturally led to market competition. Competition, in turn, has assured that no vendor is able to develop a distributed query language acceptable to other vendors. The Structured Query Language (SQL) is and is likely to continue to be a query language for centralized systems. RDF's query language SPARQL is in its infancy, but already supports standardized distributed queries across Web resources.

RDF schemas are optional and may be represented in a variety of vocabulary description languages, such as RDFS, SKOS and the varieties of OWL. RDF schemas are therefore explicitly defined in RDF data, such data possibly being external to the RDF data it is describing. Relational databases, by comparison, use implicit schemas that must generally be known *a priori* to form a query. RDF queries are thus more expressive in that vocabulary description (schema) information is explicit, not implicit as in SQL.

Although there is little disagreement about the use of metadata in protocols or machine-readable formats such as HTML layout tags (as opposed to descriptive tags or attributes), there is substantial disagreement on the use of user-defined metadata to describe the meaning of resources or suggest potential uses. Bulterman has suggested large-scale comparisons of binary data (such as a mathematical hash function on a resource's bytestream, or geometric modeling of image content) for the purposes of replacing the use of textual metadata on the Web [Bulterman 2004]. It is

interesting to note that Bulterman would like to do away with all metadata used for describing virtual image resources "other than perhaps required citation information and a URI"; even as a harsh critic he did not deny the need for some descriptive metadata.

Doctorow presented seven criticisms of the use of metadata to describe virtual resources on the Web [Doctorow 2001]. Doctorow's criticisms are summarized in Table 4-1 alongside the positions suggested by the RDF community in relation to them.

Each of Doctorow's criticisms is valid for the case of metadata added by an author (*attached/separated* metadata). The history of the HTML META tag illustrates every one of the failures suggested by Doctorow. However, metadata is known to avoid each criticism in some social environments both on and off the Web. The use of *external* metadata in PICS (via rating services) was intended to broaden the social discourse to address those criticisms. RDF, by addressing resources by URL (unidirectional hyperlinks), also encourages *external* metadata. External metadata may include vocabulary descriptions (schemas), content descriptions, and metrics. External metadata may also come from a source more trusted than the source of the content to which it refers.

Table 4-1. Metadata Criticisms and Responses

Doctorow's Criticisms	RDF Approaches
"People lie"	Allow users to choose a social trust model
"People are lazy"	Automate where possible and encourage authoring where needed
"People are stupid"	Automate where possible, check where possible
"Mission: Impossible – know thyself"	Allow multiple sources of metadata
"Schemas aren't neutral"	Allow multiple schemas
"Metrics influence results"	Allow multiple metrics
"There's more than one way to describe something"	Allow multiple descriptions

One of Doctorow's most stinging criticisms was that users must know themselves in order to create metadata worth having. This criticism makes an assumption about the relationship between metadata and the content to which it refers; the assumption is that a resource's author also creates the metadata.

There are at least two mechanisms that can produce high-quality metadata without reliance upon the motivations of an individual user. Professional ontologists within corporations, such as at Yahoo!, Google and many corporate information technology departments, routinely provide trusted, quality metadata in the same way that librarians have done for millennia. A more recent approach is the so-called "hive mind" of social networks. Many individual categorizations are combined and weighted, with the most popular tags being given precedence. Web-based systems that use the latter include the Web bookmarking community del.icio.us and the book review system at Amazon.com. Doctorow claimed that knowing ourselves was impossible, but it appears that we need not all know ourselves, after all.

The reasons that individuals donate their time and attention contributing to online social networks was studied by Smith and Kolloch [Smith 1992 and Kolloch 1999]. They determined that individuals contributed for both economic reasons (anticipated reciprocity and increased recognition) and psychological reasons (a sense of efficacy and a sense of community). These reasons, also present in traditional social networks, work to build trust in the aggregate [Golbeck 2005].

Both Doctorow and Shera complained about the lack of neutrality in schemas and noted that different people naturally describe resources differently. RDF addresses these concerns directly by (a) making schemas optional, (b) allowing multiple (possibly overlapping) schemas and (c) allowing the arbitrary application of schemas.

Doctorow's criticisms, although accurate, are shown to be bounded. People do lie, but the Open World assumption coupled with the ability to combine RDF graphs has provided technical infrastructure for distributed trust. RDF metadata may be applied to resources by others, not just the author of the resource. Social networks, within or outside of an organization, may be used to assign trust levels to a resource or to override metadata provided by an author.

Mechanisms exist to include *attached/separable* metadata in XML documents, including XHTML. Gleaning Resource Descriptions from Dialects of Languages (GRDDL) [Connolly 2007] and RDFa [Adida 2007] are competing, partially complementary proposals at the World Wide Web Consortium. GRDDL provides a URL pointing to one or more XSLT transformations that are capable of extracting metadata from the source document. Thus, any format for the embedded metadata may be used as long as a transformation can be defined to extract it. RDFa takes the opposite approach, embedding formatted metadata descriptions into the XML and XHTML standards so that only a single transformation is needed. GRDDL is a W3C Recommendation and

RDFa is on the standards track as of this writing. GRDDL and RDFa are discussed again in Chapter VII as useful mechanisms for linking traditional software project documentation with metadata describing software projects.

Implicit metadata on the Web includes the interpretation of URL components to suggest content that will be returned. Such a practice is dangerous, at best. For example, the URL http://example.ca/toronto/images/cntower.jpg might suggest that the resource to be returned is an image of the Canada National Tower in Toronto. Unfortunately, that might not be the case. The idealized architectural style of the World Wide Web is known as Representational State Transfer (REST) [Fielding 2000]. A key concept in REST is the late binding of addresses (URLs) to resources. Late binding, at the time the resource must be served, allows a Web server to return *any type of content for a URL*. Our example URL may return an HTML page of today's news or an audio recording of Beethoven's Ninth Symphony. Apparently worse, a URL may return many different types of content *at different times*. It may be considered impolite, but such actions are not excluded by Web architecture. The dissociation of a URL from the content that it returns is known as the Principle of URI Opacity [Jacobs 2004, Section 2.4.1]. The application of URI opacity has been a topic of much discussion for the last decade and is widely violated. It does suggest, however, that one may not rely on *implicit* metadata on the Web.

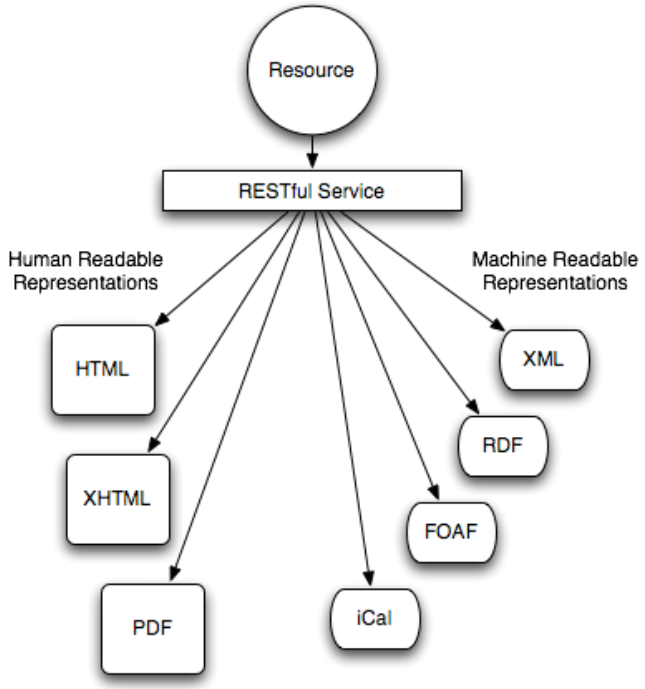


Figure 4-2. Attached/Separable Metadata Facilitates Multiple Representations

Content negotiation on the Web is enabled by *attached/separable* metadata embedded in HTTP headers and allowed by the late binding of a resource to its address [Fielding 1999]. A Web client can provide a list of MIME content types that it will accept. A Web server may use the list of MIME content types to determine which representation to return. Figure 4-2 suggests some resource representations that might be returned from a single URL.

Web client implementations limit the usefulness of HTTP content negotiation, as they did with PICS. Some browsers choose to display XML content preferentially over HTML, for example, as a legacy of a past belief that HTML would eventually become deprecated. Content providers must therefore sometimes choose to override a Web client's content request, violating content negotiation principles. A REST developer recently lamented, "Choosing a different MediaType for your XML, that the browser doesn't ask for, is the usual solution. Another workable solution I have found - if you are using XML and will get criticized for making up MIME types -- is to expose the browser-friendly HTML variant by itself on a distinct URI (e.g. person.html). That's sloppy too, just in a different way." [Heittman 2008] It is worth noting in any Web implementation that architectural theory and implementation practice are not always identical.

4.3 Metadata Applied to Software

Metadata is present in a wide variety of software systems. Metadata is commonly used to facilitate operations of computing systems (e.g. inodes on traditional Unix-like file systems [Rochkind 2004], file location information on network file systems such as Google's [Ghemawat 2003], manifest information in Java Archive (JAR) files [Sun Microsystems 1999]) and relational database schemas [Date 2004, pp. 47]. Secondly, metadata is used to facilitate later search and retrieval, as in document and image formats (e.g. [JEITA 2002]) and geospatial information systems [FDGC 1998]. A third use for metadata is to model existing or proposed software systems (as in the Object Management Group's Knowledge Discovery Metamodel [OMG 2008]). This thesis applies metadata to software for all three purposes; facilitating operations of tools, search and retrieval of content and the modeling of systems.

The SGML, HTML and XML markup languages make heavy use of metadata by their very nature as markup languages. The Extensible Markup Language (XML) community, assisted by the W3C, has defined a particularly large number of metadata mechanisms, including Document Type

Definitions (DTDs), XML Schema, XML Linking Language (XLink), XML Pointer Language (XPointer), XML Inclusions (XInclude), XML Topic Maps (XTM), XML Metadata Interchange (XMI), Extensible Metadata Platform (XMP), Meaning Definition Language (MDL) and many others that are more ancillary. Apparently metadata is important to the XML community. Additionally, the Semantic Web standards (RDF, RDFS and OWL) are serializable in XML and are used by some as XML metadata mechanisms.

Complexity of modern software systems has long led researchers to suggest the use of automated methods to record, track and leverage information about them [Tjortjis 2001]. Natural language processing has been used to evaluate the use of terms in requirements documentation for the purpose of finding inconsistencies [Kof 2005]. Relational navigation of software systems based on semantic representations goes back to the early 1990s and the LaSSIE knowledge base [Devanbu 1991]. Many of the ideas pioneered in LaSSIE may now be improved upon by implementing them with standard semantic representations and distributing descriptive components. Wongthangtham et al. related information regarding software projects to source code concepts via an ontology, but the researchers noted difficulties in creating and maintaining informational linkages [Wongthangtham 2006]. This thesis partially addresses the automated creation and maintenance of those linkages (see Chapters VI and VII). Semantic Web techniques have been applied to many aspects of the problem, including a recording of design patterns [Dietrich 2005] and organizational maturity [Soydan 2006]. Happel et al. prototyped an RDF/OWL-based system to describe software system components and domain knowledge related to them for the purposes of locating components for reuse [Happel 2006].

Formal ontologies have been used to model software systems [Meng 2006]. Ontologies have also been extracted from software systems to create a formal description [Yang 1999]. In a proposal significantly overlapping ours, Ankolekar discussed navigating both human and technical relationships within a software project to facilitate maintenance, but no evidence of implementations of her ideas has been found [Ankolekar 2004].

4.4 A Best-Practice Approach for Describing Software

We may draw some conclusions from the way metadata has been handled historically for physical and virtual resources. Physical resources hold content and also have their own physical form. Tightly coupling the ideas contained in the resource with the physical form was a necessity

until the advent of digital computers, but also caused archivists to view metadata through a warped lens. The inability to duplicate physical resources without error applied to both resource content and the metadata associated with it. The tight binding of form and content caused physical resource metadata to migrate away from *separable* locations and toward a combination of *external*, *attached* and *implicit*.

Virtual resources have many advantages over physical resources. As Shera and Conway wanted, their form is separable from their content, at least in those systems that attempt to do so (e.g. the SGML, XML, HTML family of formats) [Raggett 1999, Section 2.4.1].

Importantly, we see that most metadata regarding physical resources in libraries has been either *attached* (to facilitate internal searching of a resource by an end user) or *external* (to facilitate finding resources). By comparison, metadata for virtual resources on the World Wide Web (especially knowledge resources) has been either *external* (such as search engine indexes) or *attached/separable* to facilitate the using, processing or combining of resources. Importantly, *external* and *attached/separable* metadata may be readily combined for virtual resources. *Implicit* metadata, still widely used in libraries for physical resources, has become *informatio non grata* on the World Wide Web [Jacobs 2004, Section 2.5].

Table 4-2. A Comparison of Physical and Virtual Resources

Physical Resources	Virtual Resources
Form inseparable from content	Form separable from content
Error-prone copying	Error detection and correction upon copying
Transformations limited to copying	Arbitrary transformations possible
Lifetimes limited by: • Unbroken chain of faithful copies • Lifetime of physical media	Lifetimes limited by: • Unbroken chain of faithful copies • Availability of readers • Availability of media
Metadata may be: • External • Attached • Separable • Implicit	Metadata may be: • External • Attached/Separable
Metadata determined by: • Authors/Publishers (Attached) • Curators (External, separable, implicit)	Metadata determined by: • Authors/Publishers (Attached/Separable) • Curators (External) • Readers (External)

Table 4-2 summarizes the primary differences between physical and virtual resources. Virtual resources are shown to be different in their presentation and description, not simply their form. The differences in descriptive metadata suggest that software, always virtual for the purposes of this thesis, may best be described by a combination of *external* and *attached/separable* metadata and that the metadata be created and maintained by a combination of authors, publishers, curators and even readers.

Software, unlike any other form of information, has undergone a transition from existing physically (punch cards, paper tape) to existing purely in a virtual form. Software in the modern world exists purely as virtual (knowledge) resources. Software, as a type of resource, has made the jump from *purely* physical to *purely* virtual during the twentieth century. Library books may be either physical or virtual or both. Most Web resources are solely virtual. Many other forms of resources are solely virtual because they did not exist in any other form, indeed could not exist, until the creation of the computer.

The transition of software from physical to virtual has led to a mistake in the manner in which software metadata is treated. Like physical resources in a library, the application of metadata to software began as external. Metadata describing early software could not be held internally to the software itself, at least until the invention of the comment in programming languages. Physical software resources were tagged physically, with labels on containers naming authors, titles, and dates. Even today, when many facilities exist to store metadata within the source code of a software program, much, perhaps most, of a program's description is held externally in the form of user documentation, developer documentation, directions for installation, directions for removal, directions for upgrading, directions for bug reporting, user forums and sales pitches. In short, metadata for software today is more akin to metadata for (mixed physical and virtual) library resources than it is to the purely virtual knowledge resources of the World Wide Web. Put another way, the treatment of software metadata has not made the transition to best practices used for purely virtual knowledge resources.

Treating software descriptions as metadata for virtual resources on the World Wide Web encourages us to think of software resources universally addressed and accessible via transformation to a representation. The Perl programming language's Plain Old Documentation (POD) system, in which documentation is embedded with source code, parsed as needed and presented in various ways, provides an early example of representational transformations of

software descriptions [Wall 1987]. Each component of a software system, such as a class, function, method or package, may be described as fully as necessary for a task. Reasonable tasks using those descriptions naturally include relational navigation of the software collection for the purposes of making development or maintenance decisions.

Transformations of canonical content ("resources") into particular views for particular purposes ("representations") are features of REST. So are other architectural principles that we desire in modern software systems, such as simplicity, generality, cachability, scalability, reliability, modifiability and extensibility. This suggests that REST may be an architectural style appropriate for use with software, particularly distributed software.

Descriptions of software as a collection of virtual resources require us to think about describing one more type of resource: conceptual ones. Conceptual resources are those resources that do not exist in the physical realm, as a book does, nor in virtual form, as a Web page or electronic mail message does. When we refer to an object-oriented method in a software project or a unit test, we may think of them as virtual resources, able to be addressed and retrieved. But what about the notion of a test or a requirement? Those concepts, too, must be identified so that our metadata has meaning. We may assign URIs to those concepts, making them conceptual resources. The URIs addressing conceptual resources may not resolve to Web resources (that is, the URIs may not be URLs). The naming of conceptual resources by URI is not new; it is the basis for naming predicates (relationships) and other terms in RDF.

A combination of Semantic Web technologies and the REST architectural style provides some benefits for describing metadata applied to software systems. The most important single problem in software maintenance is the loss of coupling between source code and system metadata. Both RDF and REST apply metadata and supply mechanisms for metadata deployment. RDF was designed specifically for the purpose of handling metadata in a Web environment. As such, any RDF resource may link to any other resource via URI. We may use RDF to describe system metadata and link to source code as required. REST includes the concept of metadata accompanying a resource representation, which provides a standard means to pass metadata to a client. Due to Web client implementation limitations, however, this possibility has not yet been explored.

Both source code and system metadata are mutable. That is, they change (sometime rapidly) with time. RDF's open content structure allows for version information to be defined and maintained easily. REST's late binding allows URIs to be assigned to aspects of resources, as well as concrete resources. The combination of versioning metadata and URI-addressable aspects would

seem to ease the implementation of a versioning system for software that is capable of retaining and maintaining system metadata. Similarly, RESTful content negotiation or URI-addressability of aspects can provide sufficient flexibility of presentation that existing development tools and user applications (such as a Web browser) can be used.

Modern software systems are large and will require the generation and maintenance of a large volume of system metadata. This can have ramifications beyond simple scalability. The distributed nature of much development, both in a corporate world enamored of outsourced development and in Open Source projects, suggests that centralized repositories are more harmful than helpful. REST's replication and caching structure, as well as URI addressability, will be helpful in defining a truly distributed software maintenance environment.

Code reuse has been a hot topic for nearly a generation. How, exactly, should a developer find and reuse existing code? A distributed software development and maintenance environment could address this question by providing a Web-accessible interface to code modules, complete with accurate metadata. RDF graphs combine well, in a way that, say, XML's hierarchical structures do not. Graph combinatorics would allow metadata from disparate sources to combine cleanly in a new system.

Table 4-3. Software Maintenance Problems and Approaches

Problem	Semantic Web Approach	REST Approach
Loss of code/metadata coupling	Metadata related to code via URI(s)	Metadata accompanies a Representation
Code and metadata change with time	Code and metadata may be versioned	Dereference URIs as appropriate
We don't want/need a separate client application!	GRDDL, RDF/A	Content negotiation and/or URI-referenced aspects
Need support for remote operations for distributed teams	URI Addressability	
Require significant scalability		
Desire to foster code reuse	Metadata graphs combine well	Supports decentralised information, scales well
Wish to foster IDE/environment independence	Defined in international standards (W3C)	Defined in international standards (W3C, IETF)

Lastly, any attempt to create a software maintenance environment should be based upon international standards in an open manner. The choices of Semantic Web technologies and a RESTful architecture were made partly due to their statuses as international standards.

Table 4-3 summarizes problems of metadata application and suggests the features of Semantic Web technologies and the REST architectural style that may be applied.

The remainder of this thesis will describe software as a collection of virtual resources, addressable by URIs (sometimes URLs), related to other virtual and conceptual resources and representable on the World Wide Web. Chapter V will present a system for curating *external* metadata in order to reduce problems with author- or publisher-centric descriptions discussed in this chapter.

V. Management of Distributed Metadata

"All problems in computer science can be solved by another layer of indirection,
but that will usually create another problem."
-- David John Wheeler (1927 - 2004)

5.1 Persistent URLs

The problems of trust related to author- and publisher-centric metadata discussed in Chapter IV may be partially addressed in at least two ways: by allowing resource users and/or third parties to manage the process of URL resolution or the provision of metadata independently of Web resource author and publishers. An implementation of third party URL resolution management is the Persistent Uniform Resource Locator (PURL) scheme [Shafer 1996]. This chapter describes how the public PURL service has been extended to allow third party control over both URL resolution and resource metadata provision. The new PURL service was designed by the author as part of this thesis, in conjunction with others as noted in the statement of originality.

A URL is simply an address of a resource on the World Wide Web. A Persistent URL is an address on the World Wide Web that causes a redirection to another Web resource. If a Web resource changes location (and hence URL), a PURL pointing to it can be updated. A user of a PURL always uses the same Web address, even though the resource in question may have moved. PURLs may be used by publishers to manage their own information space or by Web users to manage theirs; a PURL service is independent of the publisher of information. PURL services thus allow the management of hyperlink integrity. Hyperlink integrity was noted in Chapter IV as a design trade-off of the World Wide Web, but may be partially restored by allowing resource users or third parties to influence where and how a URL resolves.

A simple PURL works by responds to an HTTP GET request by returning a response of type 302 ("Found"). The response contains an HTTP "Location" header, the value of which is a URL that the client should subsequently retrieve via a new HTTP GET request. Figure 5-1 summarizes the operations of a simple PURL redirection service.

A public PURL service has been operated by the Online Computer Library Center at

http://purl.org since 1995. The source code was released under an Open Source Software license and now forms the basis for several other PURL services, both public and private.

PURLs implement one form of persistent identifier for virtual resources. Other persistent identifier schemes include Digital Object Identifiers (DOIs) [Paskin 2006], Life Sciences Identifiers (LSIDs) [OMG 2004] and INFO URIs [Van de Sompel 2003]. All persistent identification schemes provide unique identifiers for (possibly changing) virtual resources, but not all schemes provide curation opportunities. Curation of virtual resources has been defined as, "the active involvement of information professionals in the management, including the preservation, of digital data for future use." [Yakel 2007]

For a persistent identification scheme to provide a curation opportunity for a virtual resource, it must allow real-time resolution of that resource and also allow real-time administration of the identifier. PURLs provide both criteria, as do DOIs. DOIs resolve to one of many possible repositories that provide administration capabilities, but have been criticized for their commercial nature [E.J. Miller (personal communication, January 26, 2008)]. LSIDs may be mapped to a URL scheme and an administration service [Clark 2004], in which case they would be functionally similar to PURLs. INFO URIs provide neither real-time resolution, nor real-time administration.

PURLs have been criticized for their need to resolve a URL, thus tying a PURL to a network location. Network locations have several vulnerabilities, such as Domain Name System registrations and host dependencies. A failure to resolve a PURL could lead to an ambiguous state: It would not be clear whether the PURL failed to resolve because a network failure prevented it or because it did not exist [Martin 2006].

PURLs are themselves valid URLs, so their components must map to the URL specification. Figure 5-2 shows the parts of a PURL. The scheme part tells a computer program, such as a Web browser, which protocol to use when resolving the address. The scheme used for PURLs is generally HTTP. The host part tells which PURL server to connect to. The next part, the PURL domain, is analogous to a resource path in a URL. The domain is a hierarchical information space that separates PURLs and allows for PURLs to have different maintainers. One or more designated maintainers may administer each PURL domain. Finally, the PURL name is the name of the PURL itself. The domain and name together constitute the PURL's "id".

PURLs require a Domain to hold them. Domains have to be created before PURLs may be placed in them. Some PURL servers automatically create domains when a PURL is created and others require domains to be created separately. A domain maintainer is responsible for

administering a PURL domain record. Domain “writers” are allowed to add or remove PURLs from a domain.

The original PURL service thus provided a general mechanism for third party URL curation using established elements of the HTTP standard. It was, however, simplistic in its scope. A more general solution to common issues of metadata trust may be found by extending the scope of PURL services.

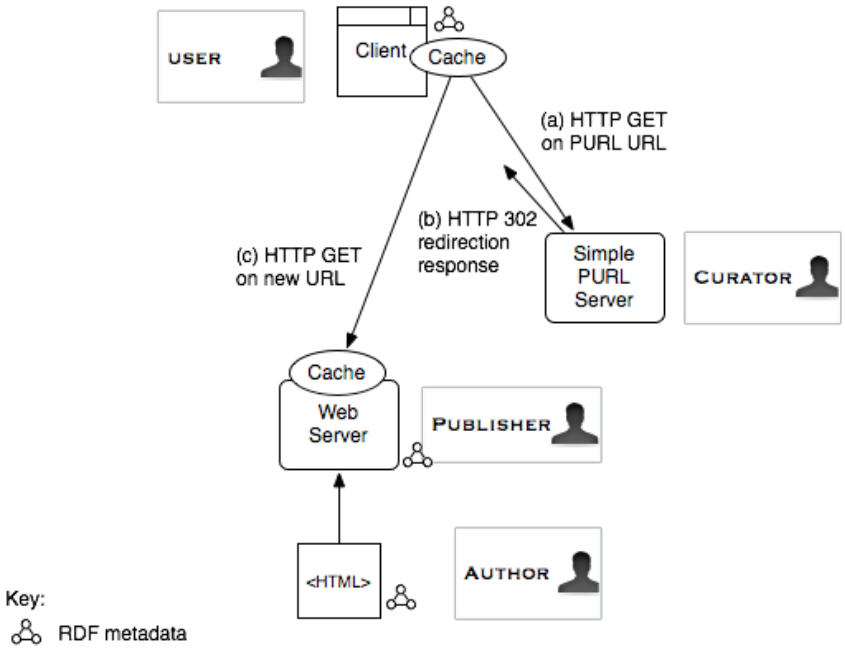


Figure 5-1. Simple PURL Redirection Service

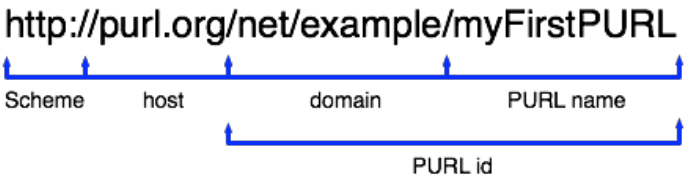


Figure 5-2. Parts of a PURL

5.2 Extending Persistent URLs for Web Resource Curation

The most obvious extension to PURL services is the creation of a greater number of PURL types. PURLs are categorized into different types depending on how they respond to a request. Simple PURLs that redirect to another URL (the "target" URL) via an HTTP 302 (Found) response are known as type 302. The new PURL service includes nine different types of PURLs that return

six different HTTP response codes. Figure 5-3 illustrates the behavior of the new Typed PURL service and Table 5-1 summarizes the different types of PURLs and their response codes.

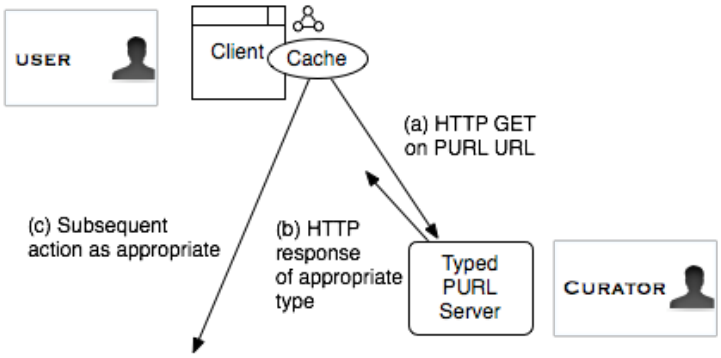


Figure 5-3. Typed PURL Service

The most common types of PURLs are named to coincide with the HTTP response code that they return. Not all HTTP response codes have equivalent PURL types. Some HTTP response codes (e.g. 401, Unauthorized) have clear meanings in the context of an HTTP conversation but do not apply to the process of HTTP redirection. Three additional types of PURLs ("chain", "partial" and "clone") are given mnemonic names related to their functions.

Table 5-1. PURL Types

Type	PURL Meaning	HTTP Meaning
301	Moved permanently to a target URL	Moved permanently
302	Simple redirection to a target URL	Found
Chain	Redirect to another PURL within the same server	Found
Partial	Redirect to a target URL with trailing path information appended	Found
303	See other URL	See Other
307	Temporary redirect to a target URL	Temporary Redirect
404	Temporarily gone	Not Found
410	Permanently gone	Gone
Clone	Copy the attributes of an existing PURL	N/A

Some have argued that a PURL server should return an HTTP 301 (Moved Permanently) response instead of a 302 (Found) response [E.J. Miller (personal communication, January 26, 2008)]. OCLC chose to use the HTTP 302 response code in 1995 in an attempt to encourage the further use of the PURL; a 301 response would suggest that a client should use the target URL in future requests. The controversy between 301 and 302 response codes continues as of this writing. The new PURL service allows sites or individual PURL maintainers to choose between 301 and 302 response codes. Major Web browsers currently handle HTTP 301 and 302 responses identically; an implicit redirection to the target URL results regardless.

A PURL of type “chain” allows a PURL to redirect to another PURL in a manner identical to a 301 or 302 redirection, with the difference that a PURL server will handle the redirection internally for greater efficiency. This efficiency is useful when many redirections are possible; since some Web browsers will stop following redirections once a set limit is encountered (in an attempt to avoid loops).

The introduction of type 303 PURLs has particular significance for Semantic Web techniques. The World Wide Web Consortium's (W3C) Technical Architecture Group (TAG) attempted to settle a long-standing debate about the things that could be named by HTTP URLs. The debate is formally known as the range of the HTTP dereference function (and called “http-range-14” for reasons of TAG issue numbering). After three years of debate from March 2002 to June 2005, the TAG ruled as follows [W3C TAG 2005]:

> The TAG provides advice to the community that they may mint "http" URIs for any resource provided that they follow this simple rule for the sake of removing ambiguity:
>
> - If an "http" resource responds to a GET request with a 2xx response, then the resource identified by that URI is an information resource;
> - If an "http" resource responds to a GET request with a 303 (See Other) response, then the resource identified by that URI could be any resource;
> - If an "http" resource responds to a GET request with a 4xx (error) response, then the nature of the resource is unknown.

The principal design goal for the support of typed PURLs was to allow PURLs to be definable

within the scope of the TAG guidance.

The TAG made a very subtle point. The idea was to cleanly separate those resources that are referred to by an HTTP URL and those that cannot be referred to directly, but might have an HTTP URL assigned to them anyway. The latter include physical and conceptual resources in the real world. Importantly, many of the objects assigned URIs in Semantic Web descriptions are given HTTP URLs but cannot be directly referred to on the Web. The TAG's guidance, having nearly the weight of a standard until such time as a standard may override it, insists that physical and conceptual resources addressed by HTTP URL return HTTP 303 responses. They may not return a 301 (Moved permanently), 302 (Found) or 200 (OK) response since they are not information resources.

Programmatic resolution of an HTTP URL may refer to an object in the real world (that is, a physical or conceptual resource) or a virtual resource (such as an HTML page or an image or a movie). In that case, the HTTP response code would be 303 (See Other) instead of 200 (OK). A 303 is an indication that the thing referred to may not be an information resource, it may be either an information resource or a "real" object. The body of the 303 (and the Location header) can provide information about the resource without encouraging one to think that what was returned really was a representation of the resource (as one would with a 200 response).

The W3C characterization of an information resource is that the entire content of the referred object may be "conveyed in a message". But what about resources that cannot be conveyed in a message? My dog is a resource, as is my car, or myself. These things cannot be conveyed in a message, they can only be referred to. That is where HTTP 303 response codes come in.

RFC 2616 [Fielding 1999] defines HTTP version 1.1 and its response codes. Section 10 defines a 303 thusly:

> The response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource. The new URI is not a substitute reference for the originally requested resource. The 303 response MUST NOT be cached, but the response to the second (redirected) request might be cacheable.
>
> The different URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the

response SHOULD contain a short hypertext note with a hyperlink
to the new URI(s).

RFC 2616 clearly predated the TAG guidance on http-range-14. The TAG (mildly) extended the use of an HTTP 303 without changing the manner in which it is generally currently applied. Further, the TAG guidance does not preclude the complete lack of a Location header in a 303 response referring to a physical or conceptual resource. RFC 2616 allows a Location header to be missing (by the use of "SHOULD" and not "MAY", terms used in accordance with RFC 2119 [Bradner 1997]). Typed PURLs referring to physical or conceptual resources may thus be distinguished from typed PURLs referring to virtual (information, knowledge) resources by the presence or absence of a resolvable URL in a Location header.

A PURL of type 307 informs a user that the resource temporarily resides at a different URL from the norm. PURLs of types 404 and 410 note that the requested resource could not be found and suggests some information for why that was so. Support for the HTTP 307 (Temporary Redirect), 404 (Not Found) and 410 (Gone) response codes are provided for completeness.

There are some differing interpretations of the HTTP specification and hence some decisions were made in the implementation of PURLs. For example, an HTTP server may respond with a 404 response code if a resource is not found, if it is temporarily not present or if it simply does not want to provide it to a requester. The new PURL service treats a 404 as representing a temporarily gone status and uses a 410 for those resources that are permanently not resolvable. Similarly, Eric Miller and I have noted the need for a way to ground non-information resources into the World Wide Web and supported that concept with PURLs by suggesting that any resource addressed by a 303 PURL and returning a "See also URL" be explicitly considered not to be an information resource if the response does not include a resolvable URL in the Location header. This decision allows physical resources (such as your car) or conceptual resources (such as the idea of a car) to be given a PURL and referred to in a sharable manner (as when using Semantic Web techniques). Where a particular interpretation of the HTTP status code definitions differs from the way an HTTP response code is used by a PURL server, the intent of the PURL should be interpreted via the Meaning column in Table 5-1.

It should be noted that two other HTTP response codes indicate that a resource may be found elsewhere without providing a URL to such a location. They are 304 (Not Modified) and 305 (Use Proxy). Those response codes were not implemented as PURL types because their meanings within the context of URL curation is unclear. Overloading the meaning of the presence or absence of the

Location header in a type 303 PURL addressed the range of HTTP responses without otherwise applying meaning to the 304 and 305 response codes.

The new PURL service includes administrative functionality, such as the ability to create and manage user accounts, assign users to groups for convenience and create and manage PURL domains. A new PURL may be created with the same values as an existing PURL (via the PURL type "clone"). PURLs may also be created or modified in batches by the submission of an XML document complying with a published XML schema. The new PURL service [Hyland-Wood 2008e] has been released under an Apache License version 2.0 (an Open Source license) and is available for download at the PURL community site http://purlz.org.

To allow Web clients to programmatically determine when they have resolved a PURL, the new PURL service adds a non-standard header to HTTP responses called X-Purl. A PURL header reports the base URL for the PURL server and the version number of the PURL server release. A PURL header for a PURL version 2.0 service running on localhost at port 8080 looks like "X-Purl: 2.0; http://localhost:8080".

The addition of a typing system to PURLs provides greater descriptive and managerial capability to URL curators. The specific addition of type 303 PURLs provides a way to cleanly and programmatically separate addresses for virtual resources from addresses for physical and conceptual resources, thus overcoming the objections made by Ogbuji in regard to the overuse of the HTTP URI scheme for the addressing of non-resolvable resources [Ogbuji 2007].

5.3 Redirection of URL Fragments

The original PURL service included a concept known as partial redirection. If a request does not match a PURL exactly, the requested URL is checked to determine if some contiguous front portion matches a registered PURL. If so, a redirection occurs with the remainder of the requested URL appended to the target URL. For example, consider a PURL with a URL of http//purl.org/some/path/ with a target URL of http://example.com/another/path/. An attempt to perform an HTTP GET operation on the URL http//purl.org/some/path/and/some/more/data would result in a partial redirection to http://example.com/another/path/and/some/more/data. The concept of partial redirection allows hierarchies of Web-based resources to be addressed via PURLs without each resource requiring its own PURL. One PURL is sufficient to serve as a top-level node for a hierarchy on a single target server. The new PURL service uses the type "partial" to denote a

PURL that performs partial redirection.

Partial redirections at the level of a URL path do not violate common interpretations of the HTTP 1.1 specification. However, the handling of URL fragments across redirections has not been standardized and a consensus has not yet emerged. Fragment identifiers indicate a pointer to more specific information within a resource and are designated as following a # separator in URIs [Berners-Lee 2005].

Partial redirection in the presence of a fragment identifier is problematic because two conflicting interpretations are possible [Bos 1999]. If a fragment is attached to a PURL of type "partial", should a PURL service assume that the fragment has meaning on the target URL or should it discard it in the presumption that a resource with a changed location may have also changed content, thus invalidating fragments defined earlier? Bos suggested that fragments should be retained and passed through to target URLs during HTTP redirections resulting in 300 (Multiple Choice), 301 (Moved Permanently), 302 (Found) or 303 (See Other) responses unless a designated target URL already includes a fragment identifier. If a fragment identifier is already present in a target URL, any fragment in the original URL should be abandoned. Unfortunately, Bos' suggestion failed to navigate the IETF standards track and expired without further work. Dubost et al. resurrected Bos' suggestions in a W3C Note (not a standard, but guidance in the absence of a standard) [Dubost 2001]. Makers of Web clients such as browsers have "generally" [ibid.] failed to follow Bos' guidance.

The new typed PURL service implements partial redirections inclusive of fragment identifiers by writing fragments onto target URLs in an attempt to comply with [Dubost 2001] and avoid problematic and inconsistent behavior by browser vendors.

5.4 Using Persistent URLs and Retrieved Metadata

The typed PURL service may be used in a manner similar to, or slightly more flexible than, the original PURL service. That is, the service allows simple redirection of URLs to content provided elsewhere and the ability to manage (curate) the URL resolution process. Other opportunities for management of Web-based services exist, however, by making use of the combination of curation services and published metadata.

Primary goals of URL curation are the ability for a user to choose a curator of choice and for a curator to choose a metadata provider of choice. These goals were first articulated for PICS, as

discussed in Chapter IV. When PURLs are used to implement a curation service that meets these goals, they are called "Rich" PURLs to denote the widening of metadata choices. Rich PURLs were first called by that moniker by Uche Ogbuji [U. Ogbuji (personal communication, October 14, 2007)] and first fielded as part of a production metadata service for a Web magazine, Semantic Report (http://purl.semanticreport.com) in 2007.

Rich PURLs are an application of typed PURL services that particularly redirect to structured metadata on the Web. No code changes were made to the typed PURL code to implement Rich PURLs.

Figure 5-4 illustrates a typical Rich PURL application data flow. A user has a URL to a Web resource that resolves to a PURL service, shown in the figure as label a. The PURL service resolves the URL and returns the HTTP response code appropriate for the URL (typically 301 (Moved Permanently), 302 (Found), 303 (See Other) or 307 (Temporary Redirect) (label b). The user's Web client is redirected to a target URL that resolves to a metadata description of a Web service, typically published in RDF (label c). The Web client is responsible for understanding how to parse and take action on the RDF metadata (label d). If the client is so capable, it will have a rich description of the content held at the final resource, the Web service at label e.

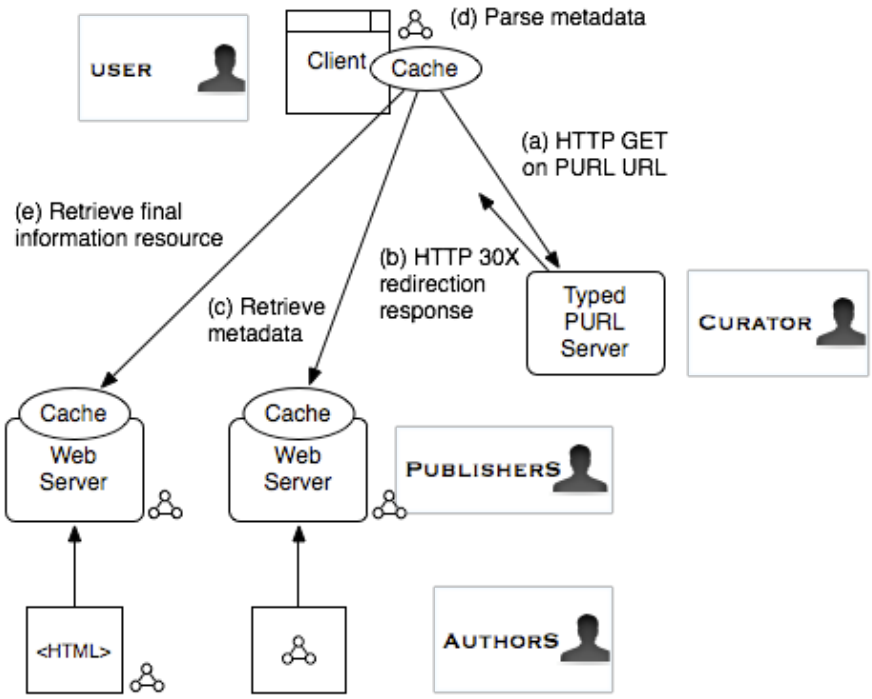


Figure 5-4. Rich PURL Service Data Flow

The term “Web service” is used here to include any Web-based service providing answers to queries and is intended to include, but not be limited to, Web Services (typically capitalized) based on the Web Service Description Language (WSDL) and the Simple Object Access Protocol (SOAP).

Rich PURLs of the form described above have the obvious disadvantage that most Web clients do not natively support the parsing or even meaningful display of RDF metadata as of this writing. Even some Semantic Web applications require non-generic formatting or use of particular vocabulary descriptions to operate fully. Rich PURLs are therefore limited to building the data set available for semantic processing on the Web and do not by themselves implement a user-directed experience.

It is possible to extend Web clients to handle Rich PURLs via their existing extension mechanisms, such as JavaScript. A Rich PURL client could, for example, display metadata about an information resource and link to or retrieve the resource itself as well. Since the metadata may include additional resolvable URLs, a display of the metadata may include hyperlinks based on those URLs. This is the approach taken by Semantic Web applications such as Exhibit, a JavaScript in-browser presentation application for semantically structured information [Huynh 2007].

Naturally, native Semantic Web applications already implement support for resolving URLs, following HTTP redirections, parsing and taking action on RDF metadata and following hyperlinks to referred content. Examples include the Haystack [Karger 2005] and Longwell Semantic Web browsers [Butler 2006]. Rich PURLs may be used directly by tools implementing those features.

5.5 Active PURLs

It is possible to conceive of more interesting opportunities for metadata management beyond simple URL curation if the PURL service itself is a more active participant in the provision of metadata. Such services may be called Active PURLs. An Active PURL is a PURL that redirects to a target URL addressing dynamic content, such as a Web service, and itself provides more information about the target URL than simply its location. That is, an Active PURL server provides metadata regarding the resources to which it refers, instead of hyperlinking to an external metadata location (as in Rich PURLs). Brian Sletten coined the term Active PURL [B. Sletten (personal communication, November, 2007)].

Consider, for example, a simple Active PURL service that periodically checks the state of its target service and confirms that the service is available. An HTTP HEAD request on the target service and noting whether or not the HTTP response code from that service is 200 (OK) would implement an availability check. Maintainers of the PURL could be notified (e.g. by electronic mail) if any other response code is returned. Figure 5.5 illustrates such a simplistic Active PURL. Active PURLs of this type were implemented in the Open Source PURL code and fielded in early 2008.

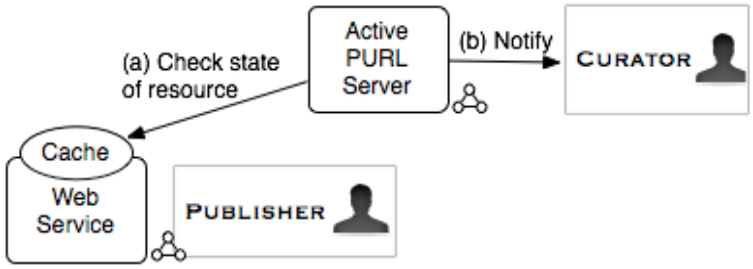


Figure 5-5. Hyperlink Integrity Management with an Active PURL Service

A hyperlink integrity service is trivial to implement by extending the capabilities of a PURL server and effectively eliminates the onus on a human curator for manually checking the status of a target URL. An Active PURL for availability checking enhances hyperlink integrity management over the original PURL service.

A more complicated Active PURL service can add to the functionality facilitated by Rich PURLs. An Active PURL service provides additional opportunities for computation because it is not restricted to simply pointing to metadata about a Web service. It may also take a role in creating metadata based on variables such as the results of a request to the target service. For this reason, an Active PURL service may not redirect to a target Web service at all. It may act as a proxy instead. Figure 5-6 illustrates the data flow of a fully qualified Active PURL service. Figure 5-6 shows the Active PURL service requesting (or supplementing) metadata from a separate service, although it may store, produce and serve the metadata directly. It may also redirect a user to its target or issue a target URL that resolves to itself, thus allowing it to act as a proxy for the final URL resolution if the ability to act as a proxy is advantageous. An example of an Active PURL service with a proxy is described later in this chapter in Section 5.6 and put to use in Chapter VIII to demonstrate the application of Active PURLs to software maintenance activities.

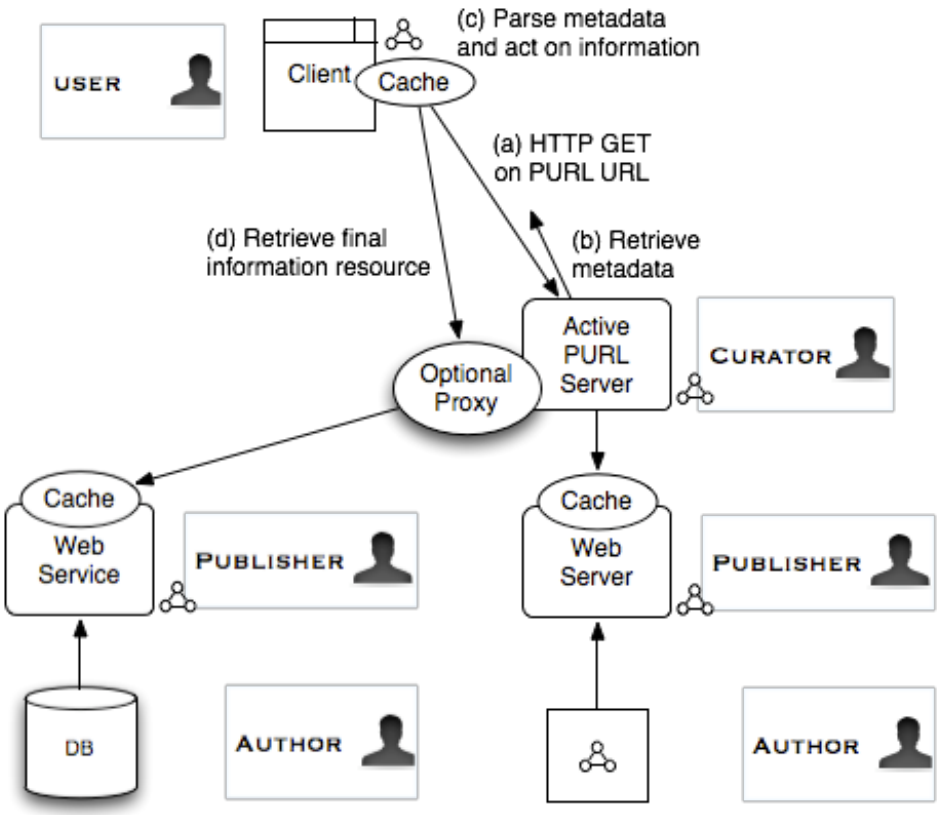


Figure 5-6. Active PURL Service Data Flow

Active PURLs allow greater control by a curator than is available in simpler PURL service definitions by allowing a PURL service to participate in the creation or modification of (possibly dynamic) metadata. An active PURL may redirect a Web client to the final Web service or act as a proxy. Acting as a proxy to a target Web service allows an Active PURL service to make decisions based on the state of the metadata and the target service at URL resolution time. The Web's feature of late binding of address to resolved resource is used to advantage in order to shift the balance of descriptive power from a resource publisher or author to a curation service. Given that a user may choose which curation service to use, and possibly participate in one, the potential for the types of metadata-related spam noted by Doctorow and experienced by search engines in relation to the HTML META tag has been significantly reduced. The Active PURL therefore appears to provide significant advantages for Web usage scenarios where metadata spam by authors or publishers is a concern.

An Active PURL, like a Rich PURL, is most effective when a Web client is able to parse and understand structured metadata. A Semantic Web client application, in-browser application or browser extension allowing a Web client to parse and understand RDF is therefore required for

most uses.

5.6 Querying Web Services with Active PURLs

Active PURLs may be used to facilitate access to Web services providing dynamic content by allowing users to access descriptive metadata of arbitrary complexity about a service and by using that metadata to assist users in the refinement of queries against the service. The results of Web service queries may naturally be chained or combined to create derivative results. Such combination of results is known as composition of Web services [Srivastava 2003].

Available Web services and metadata describing their use may be published using Rich or Active PURLs. Web publication of lists of Web services facilitates their discovery by potential users.

The combination of published metadata and Active PURLs provides a RESTful architecture for the discovery and composition of Web services. The REST architectural style has been widely recognized as a competitor to Web services based on Service Oriented Architectures (SOA) and the Universal Description, Discovery, and Integration (UDDI) registry services standard (e.g. [Szepielak 2007] and [Schroth 2007]), although the application of PURL services to the description of Web services is new.

A Web service requiring no query parameters always returns a representation of its resource based on criteria other than the query. In other words, such a service may be addressed in exactly one way. An Active PURL would not assist a user to determine how to use a service that requires no query parameters. URL curation services could be provided using a simpler typed or Rich PURL.

The PURL header for an Active PURL is modified to assist Web clients to determine when an Active PURL is being used. The header is for an Active PURL running on localhost on port 8080 is “X-Purl: 2.0 (Active); http://localhost:8080”.

A Web service requiring query parameters and exposed via an Active PURL could assist a user in the formulation of queries (via use of the service’s descriptive metadata) and in determining at what point query results should be returned to a user.

An application of Active PURLs to implement query formulation and results counting was developed as a JavaScript library. The library queries a URL to ensure that the URL resolves to an

Active PURL, retrieves metadata describing a Web service sufficient to construct a user interface for a query against the target service and sends a completed query to the target service.

Query results may be reviewed and analyzed in an Exhibit user interface if the service's metadata allows structuring of results in a manner acceptable to Exhibit and defines Exhibit-specific parameters in its metadata. This use case highlights an advantage of URL curation; a Web service need not be aware that its query results are to be used in an Exhibit since the metadata may be defined at or by the Active PURL service.

An Active PURL implements a count service for metadata results so that a Web client, in this case our JavaScript library, may make a display decision based on the size of the results. Exhibit performs poorly on result sets larger than roughly one thousand records. Thus, the Active PURL acts as a throttling mechanism until a query is refined to the point where less than one thousand records (or a similar settable number) will be returned to a client. The combination of the JavaScript library and Active PURL, acting as a throttling mechanism for an Exhibit, is called "Inhibit" for obvious reasons. The Inhibit JavaScript library is provided as Appendix B and is available online at [Hyland-Wood 2008d]. The use of Inhibit to query a Web service representing a software project's descriptive metadata is demonstrated in Chapter VIII.

A Web client needs to perform a minimum of four steps to query a Web service using Inhibit. Table 5-2 summarizes the necessary steps. First, an HTTP HEAD request is made to determine if the URL it has been given for a Web service is an Active PURL. If the URL is not an Active PURL, the client is immediately redirected to the service and the results sent to the underlying Web client for display as best it can. If an Active PURL is found, the URL is again queried to get information with which to construct a query interface for the Web service, this time by making an HTTP GET request to the URL.

The HTTP specification requires that an HTTP HEAD request return the same information that a GET would return ("The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request." [Fielding 1999, Section 9.4), with the exception that the message body is not sent. An HTTP GET is required to return an information resource upon success. Our use of HEAD and GET is consistent with the HTTP specification.

Ideally, HTTP content negotiation [Fielding 1999, Section 12] would be used to determine the content type of the results. XML or HTML could be returned, as appropriate, with XML as the preferred alternative. An XSLT style sheet would be provided to translate a platform-independent

XML description into an HTML fragment constituting a query user interface form. The XML could be returned if the Web client provides an Accept header inclusive of text/xml. Unfortunately, support for both content negotiation and XSLT processing in JavaScript from within Web browsers is inconsistent [e.g. Apple 2007]. The sample implementation of an Active PURL thus provides a query user interface as an HTML fragment.

The Web client provides the user some means of providing the required input. In the case of an HTML fragment, the client draws a query form. The user completes the query form to their satisfaction and submits the form.

The Inhibit JavaScript library issues another HTTP HEAD request, this time with the query parameters appended to the URL using a URL query string. The presence of the query string is the indication to the Active PURL that it should respond with a count of the results. The Active PURL queries the target Web service to get either a count of the results (if the Web service supports that operation) or counts the results itself. Having the Active PURL perform a count allows for less capable Web clients (such as mobile devices) to more safely participate in the querying of large data sets.

A user may refine their query based on the result count, using the generated query interface. When a Web client is satisfied that the count of results is small enough for it to handle, a final query is issued using an HTTP GET request and inclusive of the current value of the query string. The Active PURL either redirects the query to the target Web service or proxies the query, as it is configured.

Table 5-2. Example HTTP Request Types for an Active PURL Application

Step	Client Action	HTTP Method
1	Determine whether the URL is an Active PURL	HTTP HEAD on <PURL>
2	Determine necessary query parameters for a Web service	HTTP GET on <PURL>
3	Count results from a query	HTTP HEAD on <PURL> with query string input
4	Redirect to or proxy a final query to the target	HTTP GET on <PURL> with query string input

Consider a Web service that provides a SPARQL query language endpoint. SPARQL queries

may be issued to the Web service and results returned in RDF. An Active PURL may be defined that accepts parameters for use in defining a SPARQL query before issuing a query to the Web service. In this way, an Active PURL may act in a manner similar to that of a stored procedure in a relational database. Note that the Active PURL in this case is abstracting (and limiting) the interface provided by the Web service. A user of the Active PURL need not know that the Web service is a SPARQL endpoint, nor have knowledge of the SPARQL query language. To the user, the Web service might as well be implemented in SQL or another query language. The Active PURL and the late-binding aspect of URL addressability have hidden implementation details.

An example SPARQL query modified to define replaceable query parameters is shown in Listing 5-1. The query, made against the Web service's default graph, returns a list of names and telephone numbers of people with a surname beginning with a given set of characters. The term '`**SUBT**`' will be replaced with a query parameter from the client.

Listing 5-1. Parameterized SPARQL Query for a Hypothetical Active PURL

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?phone
FROM <http://zepheira.com/team/dave/dave.rdf>
WHERE { ?x foaf:family_name ?family_name ;
           foaf:name ?name ;
           foaf:phone ?phone
       FILTER regex(str(?family_name), "^**SUBT**") }
```

A client would start by determining that a URL addressing the Active PURL resolves to an Active PURL by the header check already described. The client would then issue an HTTP GET request to retrieve either an XML description of the query parameters accepted by the Active PURL or an HTML fragment, depending on the outcome of the content negotiation. An example XML response is shown in Listing 5-2.

Listing 5-2. Example XML Describing Query Parameters for a Web Service

```
<queryparams>
  <title>Find people based on their surname</title>
  <param type="simple">
    <label>Initial letters of surname</label>
    <varname>SUBT</varname>
  </param>
</queryparams>
```

The contents of the <title> tag act as a human-readable hint to a user regarding the purpose of the Web service. Each query parameter has a type, a label and a variable name. The type is used to determine which user interface element to use to present information to a user. A type of 'simple' suggests that a labeled text box is sufficient. A type of 'bulk' is used for text requiring a textarea, such as might be used for a raw SQL or SPARQL query. Other types are possible, such as radio buttons or checkboxes, but have yet to be implemented. A label is a human-readable hint to inform a user what a particular parameter means. A varname is the variable name to be used by a client when passing information back to the Active PURL server.

The XML response is transformed into an HTML fragment via XSLT. The resulting HTML fragment is shown in Listing 5-3. Cascading stylesheet (CSS) hints are provided so that Inhibit users can create their own presentation styles for query form fragments.

The client presents the HTML fragment to the user to gather query parameters. At any time, a client may submit a HEAD request to retrieve a count of the expected results. Counts are provided by an Active PURL server via an HTTP header (e.g. 'ResultCount: 450').

Listing 5-3. HTML Fragment Resulting from Processing Listing 5-2

```
<p class="querytitle">
        Find people based on their surname
</p>
<form action="#" id="queryform">
<table class="querytable">
  <tr class="queryrow">
    <td class="querycol">
        Initial letters of surname
    </td>
    <td class="querycol">
      <input id="q_SUBT" type="text" />
    </td>
  </tr>
  <tr class="queryrow">
    <td class="querycount">
      <input value="Count" type="submit" />
    </td>
    <td class="querysubmit">
      <input value="Submit" type="submit" />
    </td>
  </tr>
</table>
<form>
```

Finally, a client requests that a refined query be executed against the Web service, as previously described. Query results may be absorbed in the client as it prefers or as a proxy Active PURL provides, depending on how the Active PURL has been configured.

Active PURLs are intended to comply with the restrictions of the HTTP specification and the REST architectural style. An Active PURL server keeps no state about a client; each conversation between client and server is self contained and fully defined in the request. Each request has been carefully constructed to provide the necessary information in the ways intended by the relevant standards.

This chapter has described how the public PURL service has been extended to allow third party control over both URL resolution and resource metadata provision. PURLs have been extended to have types, initially defined to parallel relevant HTTP response codes. A use case for typed PURLs was given (Rich PURLs) that is currently being used in a production commercial service. A further extension to the PURL concept was then defined that allows PURL services to actively participate in the creation and modification of metadata (Active PURLs).

Finally, the Inhibit service was described, that is an example application of Active PURLs, in terms of the standards-based communication between Web clients, an Active PURL service and target Web services. An implementation of the Inhibit application is used in Chapter VIII to illustrate the use of URL curation services in a software maintenance context.

VI. Formal Description of Software Systems

"Design and programming are human activities; forget that and all is lost."
-- Bjarne Stroustrup (1950 -)

"There is nothing permanent except change."
-- Heraclitus (ca. 535–475 BCE)

6.1 Formal Descriptions

One must have a means of describing software systems in order to manage their life cycle. This chapter presents a formal ontology sufficient to represent the features of object-oriented software systems for the purpose of documenting their state in relation to maintenance activities. The ontological representation of software systems presented in this chapter builds on the theory of metadata application developed in Chapters III, IV and V and provides sufficient descriptive capacity to support the life cycle methodology presented in Chapter VII.

An ontology of software engineering concepts was created to represent the state of a software system as it applies to maintenance activities. The ontology, hereafter referred to as the SEC ontology, describes the relationships between software components and maintenance-oriented metadata.

We chose to create a formal ontology in a structured ontology language 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. Both reasons have been identified as common reasons to use an ontological approach by Noy and McGuinness [Noy 2001].

The SEC ontology was developed in the Web Ontology Language (OWL) [McGuinness 2004]. The levels (also sometimes called "species") of OWL are defined in the OWL standards; OWL Full, OWL DL and OWL Lite. OWL Full has the unfortunate property of being logically

undecidable in the general case. OWL Lite is a limited vocabulary description language that fails to provide for certain restrictions on data required for the purposes of SEC (use of the disjoint classes axiom and the union operator). OWL DL was therefore chosen. The "DL" in OWL DL stands for "description logic", a subset of First Order Logic. A more detailed analysis of why OWL DL was chosen is provided in Section 6.3.

Although it would have been possible to describe software systems using a collection of identified terms, such as a relational schema, the degree of logical expressiveness gained by using an ontology description language, coupled with existing tools capable of parsing and checking that language, provide the ability to confirm the logical consistency of the ontological description by use of an OWL reasoner. Indeed, mistakes were found in early versions of the SEC ontology by using that method.

Complexity of modern software systems has long led researchers to suggest the use of automated methods to record, track and leverage information about them [Tjortjis 2001]. Natural language processing has been used to evaluate the use of terms in requirements documentation for the purpose of finding inconsistencies [Kof 2005]. Relational navigation of software systems based on semantic representations goes back to the early 1990s and the LaSSIE knowledge base [Devanbu 1991]. Many of the ideas pioneered in LaSSIE may now be improved upon by implementing them with standard semantic representations and distributing descriptive components. Wongthangtham et al. related information regarding software projects to source code concepts via an ontology, but the researchers noted difficulties in creating and maintaining informational linkages [Wongthongtham 2006]. This paper partially addresses the automated creation and maintenance of those linkages. Semantic Web techniques have been applied to many aspects of the problem, including a recording of design patterns [Dietrich 2005] and organizational maturity [Soydan 2006]. Happel et al. prototyped an RDF/OWL-based system to describe software system components and domain knowledge related to them for the purposes of locating components for reuse [Happel 2006].

Formal ontologies have been used to model software systems [Meng 2006]. Ontologies have also been extracted from software systems to create a formal description [Yang 1999]. In a proposal significantly overlapping ours, Ankolekar discussed navigating both human and technical relationships within a software project to facilitate maintenance, but no evidence of implementations of her ideas has been found [Ankolekar 2004].

The Core Ontology of Software Components (COSC) [Oberle 2006] is a rich ontological

description of software components. COSC is, however, intended for a different purpose than the SEC ontology. Where COSC is intended to formalize "the most fundamental concepts which are required to model both software components and Web services" (quoted from cos.ontoware.org), the SEC ontology is illustrative of the application of formatted metadata to the process of software documentation. There is naturally a great degree of overlap, such as in the description of Classes and Interfaces. However, the differences between our two ways of describing software highlights the difficulties in getting two parties to agree to any single description of a complicated system. COSC attempts to define a "software component" as a conception different from an instantiation in an object-oriented Class or Method. For the purposes of documentation, however, the desired outcome is to record, search, validate and track what has been done as much as what was intended. Classes and Methods are, in fact, the atomic elements of the SEC ontology.

The facets by which people view a system change their perception of it. There are only two approaches to align different views; either by limiting all users to a single, imposed descriptive mechanism (the dreaded and impractical "upper-level ontology") or by providing a mechanism to interrelate common components and equivalent meanings between an arbitrary number of descriptions. RDF and OWL formalize the latter approach and that decision aligns well with the goals of this thesis. Thus, there is no claim that the SEC ontology is in any way superior to COSC, but there is a claim that the SEC ontology is well suited to the purpose of capturing the state of extant software.

The SEC ontology 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. Evidence for this is presented in Section 6.4.

It is hoped that an ontology encoding such common characteristics of the software engineering domain will be reused. Bontas et al studied the costs and benefits of ontology reuse and found gaps in current approaches to reusing ontological elements beyond simply reusing individual vocabulary terms [Bontas 2005]. Although research regarding ontology reuse is ongoing, publication of the ontology on the Internet in a standard (OWL-DL) format will facilitate its potential reuse by making its vocabulary terms and structure accessible to others.

6.2 Paradigmatic Software Documentation

Lehman proposed a classification scheme for software systems based on their environment for

the purpose of explaining their tendencies to evolve. The scheme is known as SPE, for its categorization of software systems as being based upon specifications (S), real-world problems (P) or embedded (E) into the part of the world they model. Importantly, he based that work on a realization that, "any program is a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse." [Lehman 1983] (emphasis removed). Software documentation, then, may be described as a **description** of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse. Note that the description does not necessarily describe the program, but the model that the program tries in its turn to model. In other words, documentation and code are two attempts to record an understanding of a problem at hand. They are parallel activities and do not completely correspond. Our work may be seen as an attempt to more closely relate the description and the encoded model in order to minimize divergence between them.

Cook, et al, recently proposed a refinement of the SPE classification scheme that takes into account the way programs are allowed to evolve [Cook 2006]. The default case, E-type systems, are renamed "evolving". The other two are artificially restricted in their evolution by either a specification (S) or a paradigm (P). The term paradigm is used by Cook and his co-authors in accordance with Masterman's categorization [Masterman 1970] of the "sociological" aspect of Kuhn's original usage [Kuhn 1996], whereby a group of people agree to sustain a consensus in a field. Such consensus is often facilitated by artificially constraining the evolution of a system; in this case a software system.

Noting the assertion by Cook, et al, that, "The characteristics of the P category give it a strong association with software reusability", we suggest that a P-type system is also desirable for software documentation.

Our methodology is based upon several existing and widespread paradigms, ensuring that the methodology is itself bound by social constraint as a P-type system, even while it is (using Masterman's term) an "emerging" paradigm due to its lack of widespread social consensus. The methodology becomes an extension of paradigms, a term we carefully use to separate it from the polymorphic term "extended paradigm". The methodology builds upon the standards and structures of the World Wide Web and existing software documentation paradigms. Software documentation paradigms include the capturing of a system's intended usage in end-user documentation, the recording of a system's design in developer documentation and the measurement of a system's state at a point in time via unit, integration and/or acceptance tests. These actions become "sociological"

paradigms (in Masterman's sense) as they become enshrined in software development methodologies. We consider all of these software documentation paradigms to be equally important, but suggest that their usage needs to become more structured in order to facilitate understanding and maintenance of large-scale software systems.

The Web itself is a combination S- and P-type software system, being constrained in its evolution both by specifications, such as the Hypertext Transfer Protocol (HTTP), and an idealized notion of architectural style, Representation State Transfer (REST). The Web's evolution is further hermeneutically guided by decisions of the community. A recent example is a decision by the World Wide Web Consortium's Technical Architecture Group (TAG) to interpret the 303 ('See Also') HTTP response code in a new and meaningful manner [W3C TAG 2005].

Our goal therefore becomes to systematically constrain the process of software documentation so that it is sufficiently specified to be viewed as a paradigm. We show that this is possible by defining documentation as a set of simple relationships between intent and code, the strong coupling of addressable code structures to addressable documentation elements and the basing of the technical infrastructure upon existing paradigmatic systems (the Web and software documentation practices) that serve to bind our methodology as an extension of those paradigms.

The definition of metadata developed in Chapters III and IV apply to the way the term "metadata" is used in this chapter. As discussed in Chapter III, the wide availability of computers has only recently shifted the archival focus away from physical objects to the information they contain. In contrast, best practices for managing virtual metadata rely on the ability to easily separate metadata from a resource without loss or error, such as with the SGML/XML/HTML and Postscript/PDF families of file formats or the newer means of embedding RDF (e.g. RDFa, GRDDL). These approaches resemble older attempts to create software documentation styles that were physically attached, such as in self-documenting PL/1 code [Brooks 1995, pp. 172-175] and the Perl programming language's Plain Old Documentation (POD) mechanism [Wall 1987], but add the critically important property of a standardized machine-readable structure.

A benefit to recording software documentation in RDF is that RDF was designed for use on the World Wide Web. RDF graphs representing distributed data (that is, data developed and published by separate individuals working, possibly remotely, on a larger project) may be readily combined to form composite graphs. Traditional hierarchical or tabular structures found in XML documents or relational database tables do not combine readily, except at the root of the hierarchy. Metadata may be universally addressed and accessible via transformation to a representation. Those capabilities

cleanly separate the concerns of storage from access and enable software system metadata to be stored in either an attached, separate or separable manner in relation to the related source code. Transformations of canonical content ("resources") into particular views for particular purposes ("representations") are features of the REST architectural style. So are other architectural principles that we desire in modern software and software documentation systems: simplicity, generality, cachability, scalability, reliability, modifiability and extensibility. These goals suggest that REST may be an architectural style appropriate for use with software documentation, particularly when managers, developers, maintainers and customers are geographically distributed. REST is itself a paradigm and the slow evolution of its core components helps REST-based systems to serve as "stable intermediate forms" as identified by Cook et al. [Cook 2006].

6.3 An Ontology Of Software Engineering Concepts

The SEC ontology consists of an OWL DL representation of 14 RDFS classes, 32 object properties, 1 datatype property and 3 annotation properties. The logical level of the properties used ("DL expressivity") is SIF(D): Noting that S is an abbreviation for attribute logic (AL), the use of complements (C) and role transitivity (R+), the SEC ontology also uses role inverses (I), functional properties (F) and datatypes (D). SIF(D) logic may be represented in OWL DL. Domain and range restrictions are used to ensure that logical consistency checking of example data is facilitated. Inverse relationships, often not explicitly present in data sets, are used to provide flexibility for queries and data representation.

The 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 example, 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 en-coded 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`.

A key to creating a graph useful for software engineering queries 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 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.

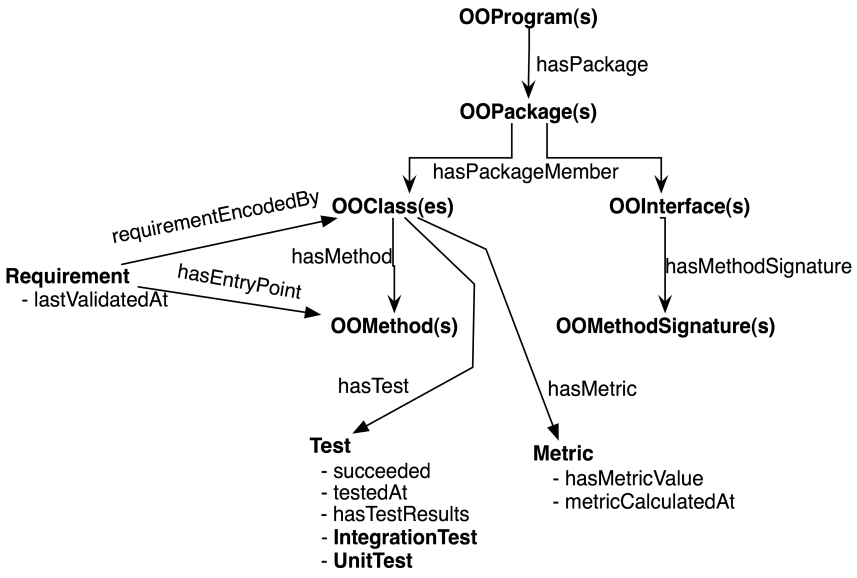


Figure 6-1. A Simplified View of SEC Ontology Concepts

A portion of the ontology is presented here in order to explain general design decisions. The portion presented describes software requirements. An overview is depicted in Figure 6-1. Some

ancillary properties (e.g. `seeAlso` and `hasDeveloper`) have been omitted for clarity. The `seeAlso` property allows a component to be tagged with an arbitrary URI so that more information may be retrieved regarding the component. `seeAlso` URIs may, for example, link an object oriented class description to a revision control system interface for the retrieval of the class itself. The `hasDeveloper` property refers to URIs representing developers of components. An example use of the `hasDeveloper` property is shown later in this chapter, in Listing 6-2.

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(
  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 manner as the properties just described. 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 provided as Appendix A and is available online at [Hyland-Wood 2008c]. It is an initial version and was developed to highlight the use cases described later in this chapter, in Section 6.5.

We acknowledge that 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 ability to refer to such implementation details is not necessary for a design language (such as UML), but is needed to record implementation decisions in a complete documentation system. The SEC ontology further separates the concept of interfaces from abstract classes, a distinction that 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 or constructors. Classes may contain methods and constructors, which may use other methods and constructors (and hence those methods' parent classes). The direct relationships between the classes could be inferred, but are not explicitly stated.

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 of the ontology.

The SEC ontology is designed to be reliant solely upon one or more RDF data stores and a SPARQL query engine. That is, a run time OWL-DL reasoner is not required in the general case. 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 of common reasoners lead us to avoid runtime dependence, although one was used during the development of the ontology. However, methodological theory should not be constrained by the current state of engineering development. The scalability of reasoners is still being studied. Our SEC ontology was therefore designed (e.g. by liberal use of inverse properties) to allow many useful queries without runtime reliance on a reasoner.

Early versions of the SEC ontology were published as a conference poster [Hyland-Wood 2006a], a workshop paper [Hyland-Wood 2006b] and a journal article [Hyland-Wood 2008a]. At each stage reviewers criticized the suggestion that runtime reliance on reasoners should be avoided. The author would bring attention to remarks made by Jim Hendler, an AI researcher and chair of the W3C Web Ontology Working Group that produced the OWL standard and co-author with Tim Berners-Lee and Ora Lassila of the seminal article that first described the vision for the Semantic Web [Berners-Lee 2001]. Famous for the saying, "a little semantics goes a long way", Hendler has commented, "…the REST approach to the world is a wonderful way to use RDF and it is empowered by the emerging standards of SPARQL, GRDDL, RDF/A and the like. In short, it is the Semantic Web vision of Tim's, before Ora and I polluted it with all this ontology stuff, coming real!" [Hendler 2006].

The author remains convinced that strict reliance on logical consistency checking at runtime over-constrains information architectures and leads to difficulties with scalability. Referring to the state of enterprise information architectures, a colleague recently observed, "enterprise data management suffers not from lack of technology, but from outdated modes of data ownership" [B. Hyland-Wood, personal communication March 19, 2008]. The outdated modes of data ownership include the over-constraining of software interfaces used to access information; technologies such as SOAP, WSDL and CORBA "leak" system implementation details to their users, thus limiting the ability to replace a service without changing its interface. REST, by contrast, encourages logical addressing of information without passing implementation details. Arguments between supporters of REST and SOAP/WSDL Service Oriented Architecture are reminiscent of conversations in the library community regarding the curation of physical forms versus information contained therein (as discussed in Chapter III).

Designing and testing of vocabulary description schemas in strong OWL species such as DL with a reasoner is both prudent and helpful. Runtime reliance on a reasoner, however, unnecessarily constrains scalability and ensures that "dirty" data, an all-too common phenomenon, will result in unnecessarily broken applications. Like the World Wide Web's 404 (Not Found)

status code, semantic applications are better served by routing around dirty data instead of insisting on perfection. However, the ability to ignore bad ontological descriptions presumes that an end user can still make use of the information that is available. Much like a user of a Web browser might look for a different resource to replace one that proved to be inaccessible, a user of a broken ontology may choose to use a different vocabulary description or to construct a query based upon implicit knowledge of available data. The architectural property of flexibility is gained by a refusal to tightly bind vocabulary descriptions to a data set.

6.4 Creation of Metadata Elements

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 (that would require manual input) would require a large or small amount of screen space in a user interface.

Table 6-1. 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

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 6-1.

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 that 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 6-2 shows the remainder of the ontological properties in the SEC ontology. They define relationships between ontological elements that 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.

It is possible to automatically generate metrics and tests using plug-ins for common integrated development environments, although such plug-ins are generally commercially licensed. Automatically-generated metrics and tests would be able to be automatically associated with their software components, by either actions of the plug-ins or reliance on naming conventions.

Tests may be associated by automatic creation (as in some plug-ins to integrated development environments) or simply by naming conventions. It has become common practice to name a unit test by appending a term to the name for a software component that it tests. For example, a class called DefaultSparqlConnection might have a unit test called DefaultSparqlConnectionUnitTest or the package Builder might have an integration test called BuilderIntegrationTest. Using naming conventions provides a simple way of associating tests with software components.

Table 6-2. SEC ontology properties that may require human input.

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

Table 6-3 lists the minimal set of ontological properties requiring human input. All of the properties requiring human input relate to requirements, as is expected. Requirements, after all, map human intentions and desires to code.

Table 6-3. Minimal SEC ontology properties that always require human input.

Concept	Requires Human Assignment	Requires Human Generation
Requirements	hasEntryPoint requirementEncodedBy	lastValidatedAt

Note that all information relating directly to software components may be generated based on the inputs shown in Table 6-3. As few as three properties require human input, suggesting that this approach may be implemented in an IDE or project management software system without burdening the end user. Even if metrics and tests are manually assigned, augmentation of a user interface for an integrated development environment would be minimal.

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 also 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.

6.5 Usage

Example data based on the SEC ontology was developed and is available online at [Hyland-Wood 2008b]. 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 published via a Web server and SPARQL queries run against them. 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, for example:

1) Requirements that were currently validated against associated software components;

2) Requirements that required revalidation following a change to an associated software component or to the requirement;

3) Tests that have failed;

4) Requirements that relate to failed tests; and

5) Object-oriented classes that have associated tests.

Additionally, queries were developed that included external data sources. Example queries are provided later in this chapter.

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 shows 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. The creation of large amounts of metadata provides 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. The time required to execute queries against large RDF graphs has been reported to be, at best, proportional to Olog(n), where n is the number of nodes in the graph [Wood 2005].

An initial analysis of the thirty-six relationships in the SEC ontology suggested that as few 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 may be treated as mandatory or optional, variant or invariant, as the needs of a particular project dictate.

Preferably 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 [Eclipse 2007].

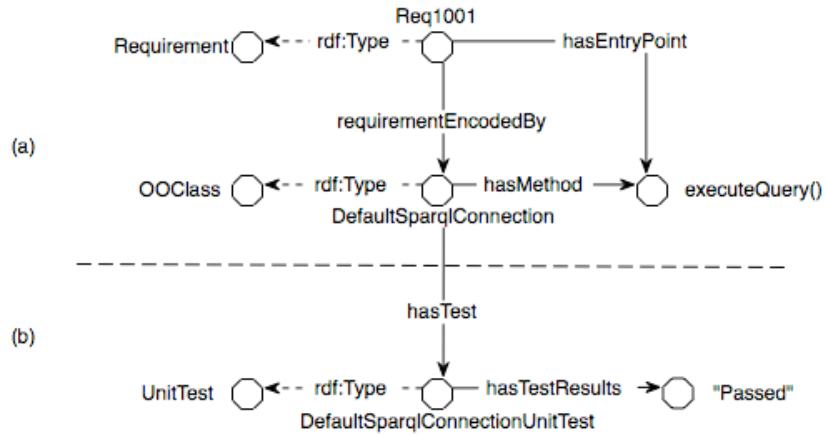


Figure 6-2. A portion of the sample data, shown in a composite graph derived from two data sources, (a) from sec-example.owl and (b) from sec-testresults.owl

A portion of the example data is shown as a graph in Figure 6-2. 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.

A simplistic SPARQL query is shown in Listing 6-1. This query returns all object-oriented classes in the example data that have tests associated with them that, in turn, have test results. Note especially that the software structure and the test results come from two different data sources. The RDF graphs are combined dynamically and queried to achieve the result. The result, in Table 6-4, contains three columns (defined in the SELECT clause), one for the classes, one for associated tests and the last for the status of the tests.

The trivial SPARQL example attempts to demonstrate the relative ease of integrating data from distributed developers when using the combination of Web addressing and Semantic Web techniques. The providers of the software structure description and test results could have been different people, performing their work in different locations, yet their combined data may be queried without additional infrastructure such as a centralized database. Only commodity Web

services and a reformatting of the data are required.

```
prefix sec: <http://www.itee.uq.edu.au/~dwood/ontologies/sec0.2.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?class ?test ?testresults
FROM <http://www.itee.uq.edu.au/~dwood/sampledata/sec-example2.owl>
FROM <http://www.itee.uq.edu.au/~dwood/sampledata/sec-testresults2.owl>
WHERE {
  ?class rdf:type sec:OOClass .
  ?test sec:isTestOf ?class .
  ?test sec:hasTestResults ?testresults
}
```

Listing 6-1. Example SPARQL Query using the SEC Ontology

A list of only those tests that failed could be returned by modifying the last term of the query from `?testresults` to the literal string `"Failed"` (inclusive of the quotation marks).

Table 6-4. Query results for Listing 6-1

Class	Test	Test Status
sec-example:SparqlQueryBuilder	sec-example:SparqlQueryBuilderUnitTest	Passed
sec-example:DefaultSparqlConnection	sec-example:DefaultSparqlConnectionUnitTest	Failed

The examples are meant to demonstrate the use and advantages to software documentation of distributed data sources, not the complexity of the underlying data. An attempt was made to implement an example that represented, but did not surpass, the state of the art in SPARQL client implementation in order to ensure the example query may be executed by readers. The SPARQL query is complete and refers to actual online data. It will run on any SPARQL-compliant query engine that supports SPARQL datasets (the FROM clauses in the query).

The desire for less-than-trivial SPARQL queries was helpful to push the state of SPARQL implementations. Several implementers responded with better dataset support when asked [Hyland-Wood, 2007]. The SPARQL queries given in this chapter were checked for correct operation using multiple SPARQL implementations, including the SPARQLer online query parser (http://www.sparql.org/sparql.html), OpenLinks (http://demo.openlinksw.com/sparql/) and the Redland RDF application framework [Beckett 2006].

A more complicated query is shown in Listing 6-2. It returns a list of all methods that are members of classes that are associated with both failing tests and requirements that require revalidation. Metric values are shown for any classes that have metrics where the metric value exceeds a maximum value provided in a separate RDF document. The names of developers associated with the methods are displayed if they are available by introspecting an RDF document listing them.

Note that test results, developer information and metric scope information were acquired from RDF documents not associated with the ontology; the data was developed and published separately. The distributed nature of the data is enabled by URI addressing support within SPARQL and enables information describing software projects to be distributed and yet queried with relative ease. We make use of this property to create a methodology for distributed software maintenance, as described in Chapter VII.

The query in Listing 6-2 refers to actual data and is functional. The query may be validated by running it in any SPARQL-compliant parser.

Results of the query shown in Listing 6-2 are shown in Table 6-5. Abbreviated URIs ("QNames") are used in the results for brevity and clarity. For example, the lengthy URI http://www.itee.uq.edu.au/~dwood/ sampledata/sec-example2.owl# SparqlQueryBuilder.buildQuery is abbreviated simply as sec-example2:#SparqlQueryBuilder.buildQuery.

```
prefix sec: <http://www.itee.uq.edu.au/~dwood/ontologies/sec0.2.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?method ?name ?metric ?metricval
FROM <http://www.itee.uq.edu.au/~dwood/sampledata/sec-example2.owl>
FROM <http://www.itee.uq.edu.au/~dwood/sampledata/sec-testresults2.owl>
FROM <http://www.itee.uq.edu.au/~dwood/sampledata/devteam.rdf>
FROM <http://www.itee.uq.edu.au/~dwood/sampledata/metricranges.rdf>
WHERE {
  ?method rdf:type sec:OOMethod .
  ?method sec:methodOf ?class .
  ?method sec:lastModifiedAt ?moddate .
  ?class rdf:type sec:OOClass .
  ?test sec:isTestOf ?class .
  ?test sec:hasTestResults "Failed" .
```

```
  ?requirement sec:requirementEncodedBy ?class .
  ?requirement sec:lastValidatedAt ?valdate .
    FILTER(?moddate > ?valdate) .
  OPTIONAL { ?method sec:hasDeveloper ?developer .
             ?developer foaf:name ?name} .
  OPTIONAL { ?metric sec:isMetricOf ?class .
             ?metric sec:hasMetricValue ?metricval .
             ?metric sec:hasMaxValue ?maxval .
               FILTER (?metricval > ?maxval) }
}
```

Listing 6-2. Extended SPARQL Query using the SEC Ontology

Building a more complicated SPARQL query is analogous to building a more complicated SQL query; it is well within the capabilities of many people, once the data is present. With SPARQL, though, the data does not have to be centralized, nor curated. That is both good and bad. If we want curation, we have to allow for it explicitly. In the case of the Web we can curate the URI resolution process with PURLs and show in Chapter VII that we can use that feature to curate access to useful data.

Table 6-5. Query results for Listing 6-2

Method	Name	Metric	Metricval
sec-example2: #SparqlQueryBuilder.buildQuery	"Tom Adams"	sec-example2: #SparqlQueryBuilder.SLOC	"153"

This chapter has presented an ontology of software engineering concepts suitable for formally describing object-oriented software systems and their requirements, tests, metrics and developer relationships. The SEC ontology defines a set of simple relationships between intent and code to allow us to treat software documentation as a paradigm.

The structuring of software documentation information in RDF allows for software elements to be addressable on the World Wide Web, which in turn allows for the development of a methodology that allows for modern distributed development practices, presented in Chapter VII. Data may be developed by various actors and published in locations most appropriate for those actors. The SEC ontology serves as a catalyst to relate possibly disparate data relating to a software project. The definition of metadata developed in Chapters III and IV was used to ensure that metadata descriptions were used in the manner most appropriate for virtual resources.

VII. A Methodology for Distributed Software Maintenance

"The reasonable man adapts himself to the world;
the unreasonable one persists in trying to adapt the world to himself.
Therefore all progress depends on the unreasonable man."
-- George Bernard Shaw (1856 -1950)

7.1 SWAMM: A Software Maintenance Methodology

Having defined metadata, discovered the best ways to apply it to virtual resources such as software descriptions and formalized the description of object-oriented software systems using metadata, the time has come to use those tools to facilitate software maintenance. This chapter describes the Software Agent Maintenance Methodology (SWAMM), a software maintenance methodology.

SWAMM is a methodology to collect metadata during software development and use that metadata to facilitate later software maintenance tasks. SWAMM makes use of the ontology of software engineering concepts (SEC) described in Chapter VI to provide metadata in RDF and OWL describing software projects.

The URL curation capabilities of persistent URLs (PURLs), as described in Chapter V, are used to manage version releases in distributed environments. Other Web-based mechanisms for serving metadata, such as the Web Document Authoring and Versioning (WebDAV) property mechanism [Whitehead 2003], could also be used, but the PURL approach has some advantages. PURLs and WebDAV differ in that PURLs allow for the separation of metadata from data whereas WebDAV attaches metadata to content. Although this thesis suggests that *attached/separable* metadata is the preferred approach, PURLs' flexibility in this matter may be useful in some circumstances, such as the serving of historical metadata after the removal of associated content. PURLs also use unmodified HTTP, a weak advantage over WebDAV in that more clients may be used without modification.

PURLs and WebDAV share the RESTful property that their implementation is separated from

their URI-based public interface. PURLs, especially Active PURLs, may be backed by various databases or other query interfaces without regard to protocol implementation (other than providing HTTP responses). DASL (DAV Searching and Locating [Reschke 2008]) is a protocol for searching WebDAV repositories, but a query mapping for RDF would need to be developed to make it a candidate for the purposes of this thesis. Perhaps the greatest single difference between the WebDAV and PURL approaches is that WebDAV properties "tend to encourage breaking up metadata into smaller chunks, rather than large RDF graphs." [Whitehead, E.J. (personal communication, November 5, 2008)] Breaking up an RDF graph makes it more difficult to query and thus more difficult to put to a particular purpose. PURLs, at least in theory, offer the opportunity to serve RDF graphs both in chunks and as a single graph as the occasion demands.

Developers using SWAMM provide structured metadata describing software components and their state during the development phase of a software project. That metadata is then used to suggest or direct maintenance activities during the maintenance phase. RDF/OWL representations of the metadata are published using a RESTful architecture (using a World Wide Web server) so that the files are network-accessible and URL-addressable for SPARQL queries. Other people involved in a software project also contribute metadata, such as customers, testers and project managers (called "monitors", for their role in monitoring, reviewing and approving states of development).

SWAMM is a full, prescriptive methodology for the collection and use of software maintenance information sufficient to allow relational navigation over a software project during, and in support of, maintenance activities. It is, however, not a software development methodology.

SWAMM may be used in conjunction with many existing development methodologies and is intended to both augment and be compatible with them. If SWAMM is used with a development methodology, the only demand made on the software development cycle is the collection of metadata.

SWAMM was designed to allow post-facto and/or partial application, as in reverse engineering activities for recovery of software system metadata to avoid maintenance failure. When used solely during the maintenance phase of a software lifecycle, metadata is collected iteratively as reverse engineering proceeds. Naturally, SWAMM becomes more useful as the basis of maintenance decisions as more metadata is collected.

SWAMM is described in terms of actors (customers, developers, testers and monitors) and their relationships to both the code being created or maintained and the metadata describing the code. A

workflow is then developed for both the development and maintenance phases of a software project. Care is taken to ensure that SWAMM may be applied to existing projects that may not have used a structured methodology during development that captured relevant metadata. Indeed, SWAMM may be applied iteratively as metadata is collected during maintenance.

Many software projects considering the use of SWAMM for maintenance will develop traditional documentation. Thus, mechanisms for relating traditional documentation to the SWAMM system metadata are discussed and examples provided.

Consideration is given to distributed development teams. The use of SWAMM is discussed when development teams are physically separate from each other. The World Wide Web, RESTful information architecture and URL curation are combined and applied to the SWAMM description.

SWAMM interactions have been modeled to determine properties of the methodology and validate presumptions. The SWAMM model is available online at http://www.itee.uq.edu.au/~dwood/models/SWAMMApplet/SWAMM.html. A discussion of the SWAMM model and its capabilities is provided in Section 7.7. The model's user interface is discussed so that readers may use the model to evaluate SWAMM for particular project parameters

The scalability of the SWAMM methodology was analyzed to determine the systems requirements necessary to implement SWAMM. The analysis shows that SWAMM scales linearly with the number of requirements for a software project and suggests that currently available RDF databases are sufficient to store system metadata for most software projects. The detailed scalability analysis is provided in Section 7.8.

SWAMM was designed specifically to be compatible with Agile development methodologies, including Extreme Programming (XP) and Test Driven Development (TDD). A detailed comparison of SWAMM to these methodologies is provided at the end of this chapter.

7.2 Actors and Relationships

Figure 7-1 depicts the relationships between actors, code and documentation in the SWAMM methodology. The relationships are not hierarchical and no attempt is made to confine the workflow to be linear in time. The workflow suggests repeated iteration, before, during and after both development and maintenance operations. Indeed, many software projects do not invest heavily in documentation until value is shown in developing it; development of documentation may

not occur until after an initial delivery. Given the tendencies of development phase documentation to be both incomplete and inaccurate as discussed in Chapter 2, it is important to allow and even encourage the later development of documentation for the purposes of system maintenance. SWAMM metadata is simply a form of documentation and one that is particularly designed to foster maintenance decisions.

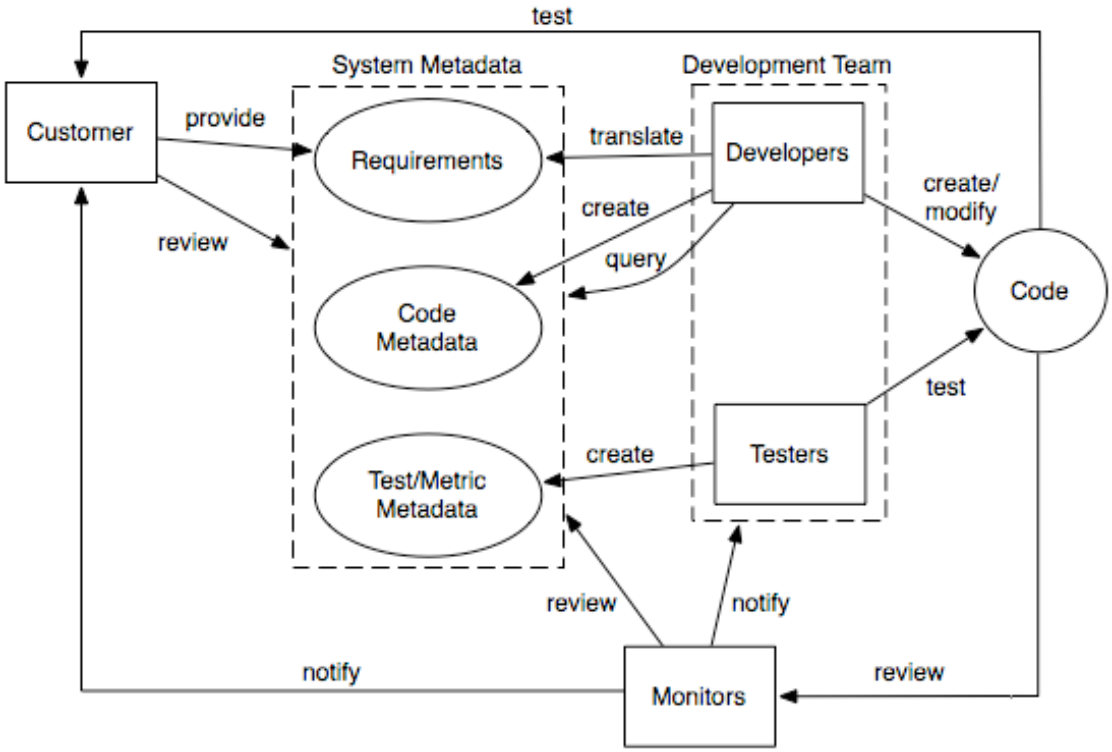


Figure 7-1. Actors and Relationships

Four types of actors are defined; Customers, Developers, Testers and Monitors. Customers are responsible (directly or indirectly) for providing requirements. They may also review system metadata descriptions at any time. Customers are provided access to versions of code ("releases"), typically upon completion of the encoding of a requirement, the completion of integration, and other releases made for the purposes of testing and acquiring feedback on suitability. Monitors notify customers when releases are available for review.

The view of a customer as a source of requirements and an approval authority for suitability follows the approach taken by Agile methodologies (e.g. [Beck 2000]).

Developers translate requirements for the purpose of implementing those requirements in software. Both the software (code) and metadata describing the software are created when a

developer uses requirements to produce an implementation. They are responsible for the creation of metadata describing code, although the actual creation of such metadata is likely to be accomplished via tools supporting the developer. Developers are expected to work closely with testers, to inform testers when a code component exists and what it is intended to accomplish. In some cases, developers and testers may be the same people.

Testers validate code operation against requirements and are responsible for the creation and maintenance of metadata describing their work. Testers may use tools to create routine tests, but should also understand and interpret requirements to ensure that the tests they create provide code coverage and exercise code components to test for the implementation of requirements.

Monitors review the progress of a system via reviews of both code and metadata and notify customers, developers and testers of necessary actions. Monitors, or project managers, may also act as developers, testers or even customers. Monitors may even be scripted processes in some cases.

The defined roles of customers, developers, testers and monitors are not intended to constrain the actions of a software development team during development. The descriptions of roles are provided only to provide context; the descriptions are consistent with common practice and the titles and terms are expected to be familiar to readers. The only new mandatory tasks for actors in the SWAMM methodology during the development phase of a software project are those relating to the creation, review and refinement of metadata.

In the SWAMM methodology, developers and testers comprise a development team. Monitors support the development team by evaluating system metadata against source code. The presence of software system metadata is an indication of how completely this methodology is being used. Lack of metadata, or a certain type of metadata, would indicate a condition that should be brought to the team's attention. Monitors notify the development team when software components are missing descriptive (structural, metric) metadata or prescriptive (test/requirement) metadata.

Descriptive metadata describes the state of a software project. It may be readily created by automatic processes, as described in Chapter VI. Descriptive metadata includes descriptions of the software itself (describing code components and their relationships to each other) and metrics that report the state of a code component. For example, a count of source lines of code, or a cyclomatic complexity calculation relate to the state of a code component at a particular time. Those metrics describe the code.

Tests may be considered to describe the state of code components in a sense, but are better thought of as prescriptive. A failing test demands action to repair its outcome and the metadata

related to a test reports the state of the test. Similarly, requirements and the metadata that describe them refer to actions required of the development team. The job of the team is to implement the requirements. Therefore, metadata relating to tests and requirements is known as prescriptive.

All metadata taken together (requirements and metadata describing code, tests and metrics) is called system metadata. The totality of system metadata is available for review at any time by monitors, developers and testers and upon releases by customers. The purpose of restricting review of system metadata by customers to defined releases is to ensure that the system metadata describes a stable, buildable and runnable system.

SWAMM actors are physical resources, as defined in Chapter III. They may therefore be assigned URIs, but those URIs may only act as identifiers. System metadata and code are knowledge (or virtual) resources; they may be assigned URIs and representations of them may be returned upon the resolution of those URIs. SWAMM actors, metadata and code are assigned URIs to allow their descriptions to be encoded in RDF.

7.3 Development Workflow

The SWAMM development workflow is simple and consistent with existing methodologies in order to be compatible with them. The only substantial addition is the provision of structured metadata by actors during their normal development activities.

A typical SWAMM development workflow consists of the following activities:

1. **Requirement Injection**: One or more customers provide requirements; the requirement descriptions are encoded in RDF and published to a Web server.
2. **Requirement Interpretation**: One or more developers interpret or translate a requirement and implement it in code. Metadata describing any newly created code components is created and published to a Web server. Modification of existing code components may (or may not) result in changes to the metadata describing them.
3. **Testing**: Testers test code components (and collections of code components) upon completion and publish the results of their tests as metadata to a Web server. Successful testing of all completed code components relating to a given requirement ends a development iteration for that requirement.
4. **Internal Validation**: Monitors review the state of both code and metadata and have

the option to accept or reject the iteration. Reviews are accomplished by querying metadata. A rejection occurs when a monitor interprets a requirement differently than a developer, desires more tests or is unhappy with the state of either the code or metadata. Additional iterations follow until accepted by the monitors.

5. **Release**: Monitors determine readiness for a release. When the code exists in a runnable state and test results are acceptable for a release, monitors make a release available by the identification of code component versions that make up the release.
6. **External Validation**: Customers are notified of a release and asked to review and test it. Customers may initiate additional iterations or requirements at any time.

Metadata creation and/or modification occurs at each of the six steps. Steps 2 and 3 (development and testing) may be reversed and iterated if desired (e.g. when a Test Driven Development methodology is being extended with SWAMM). Similarly, other steps necessary for particular methodologies may be inserted.

The SWAMM development workflow is based upon a classic Agile methodology, with each iteration starting and stopping with customer interaction, but with the addition of the recording of descriptive and prescriptive metadata. Figure 7-2 illustrates the development workflow and highlights metadata-related actions.

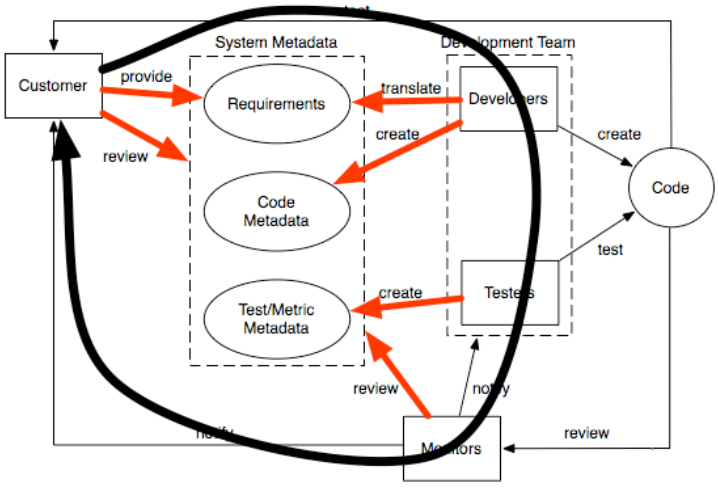


Figure 7-2. SWAMM Development Workflow

A common difficulty during development occurs when one programmer makes a change that negatively impacts another programmer's work. Increasing the visibility of system changes is

therefore desirable. Thus, in SWAMM each metadata element is tagged with not only a date-time of the last change (the `lastModifiedAt` ObjectProperty), but also a version number. Version numbers may be assigned in one of several ways; the addition of an ObjectProperty to the SEC ontology, the assignment of version-specified URIs for each version or use of a Persistent URL (PURL). Revision control systems, in common use in modern software development practices, already assign version numbers to code component revisions and several of the most popular (e.g. Subversion or the older Concurrent Versioning System, CVS) optionally allow Web interfaces to be placed over revision control repositories: A representation of each revision is therefore accessible from the Web and assigned a URL. The existence of a URL for each revision suggests the use of a simple PURL to point to the preferred version of a code component and for Active PURLs to resolve collections of metadata associated with each resource, including code components, requirements, tests and metrics. Higher-order collections of material (e.g. a software package consisting of a number of code components and inclusive of associated requirements, tests and metrics, may also be tracked and referred to by Active PURLs. A more detailed explanation of the use of simple and Active PURLs for tracking software releases in conjunction with the SWAMM methodology is provided later in this chapter in Section 7.6.

The majority of metadata elements may be automatically generated; there is no intent to suggest that a developer be required to enter metadata elements by hand. Many tools for automatic metadata extraction exist commercially, in academe and as Open Source Software projects.

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. The prototypes developed for this thesis made use of the Doxygen document generator (http://www.stack.nl/~dimitri/doxygen/) and the Java RDFizer (http://simile.mit.edu/wiki/Java_RDFizer) to extract the code structure from source code.

The choice of metrics to apply to a software project may be a choice of a developer, a project manager or performed automatically, such as when using a continuous build system. A metric's relationship to software components, value and time of calculation can in some cases be automatically established.

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

success/failure status may be automatically generated.

All ontology classes should have a last-modified property indicating when objects of the class were last modified. The last modified property could be updated automatically if the modification takes place via a system capable of knowing about it, such as an IDE or a continuous build system. Last modified properties are key to determining the state of a software project, as illustrated in Chapter VIII and discussed in Section 7.6.

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 an IDE or project management system when the requirement has been fulfilled in software because that is an inherently subjective judgment. The decision to call a requirement complete is made by monitors and confirmed by customers.

Monitors use SPARQL queries over dynamically assembled graphs of metadata to determine completeness of information and maturity of a project. The "review" action taken by monitors during development is thus in reality a query procedure, and is similar to the queries performed by developers during the maintenance phase (see Section 7.4 for details on the maintenance phase). Local policies controlling compliance to a methodology and establishing points of control could be set by choosing the types of SPARQL queries to execute. Examples include:

1. Change to a software component (a class or method) would require a revalidation of any associated requirements. The names of unvalidated requirements are available via a SPARQL query, the results of which should be published on a Web page. The list of requirements needing revalidation cues developers to revalidate them.
2. Failing tests relate to requirements, as well as to software components, via the metadata. Requirements impacted by failing tests should be displayed alongside test results.
3. Metrics calculated to be out of desired scope should be treated as equivalent to failed tests. The names of software components and requirements relating to such metrics should be displayed until the metrics are calculated to be within desired scope.

The example SPARQL queries should be viewed as representative suggestions only. 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. For example, metrics calculating the rarely implemented Maintainability Index [Van Doren 1997] could be used to

suggest when a particular software component should be considered for refactoring and which requirements those actions would potentially impact.

Table 7-1. Actor's Responsibilities for Metadata–Related Actions

Actor	Action	Metadata Elements
Customer	provides	Requirements
Developer	creates	Code metadata
Tester	creates	Test metadata Metric metadata
Monitor	reviews curates	System metadata & code Version URLs

Table 7-1 summarizes SWAMM actors and the metadata-related actions for which they are responsible.

7.3.1 Application to Existing Software Projects

SWAMM was designed to be applicable to projects post development. The metadata required by SWAMM may be collected throughout the development phase of a software project, later during development, at the start of the maintenance phase or at any later time. The metadata upon which SWAMM relies does not need to be "complete" in any sense of the word for SWAMM to provide valuable insights into a software project. Although more metadata allows more useful queries, simply generating structural metadata provides some value, as referenced in Chapter II.

Thus, SWAMM may be used as a purely maintenance-oriented methodology at any point in a project's lifecycle. Metadata may be generated, used and iterated as a development team proves the value of the methodology to itself.

Chapter VIII provides two real-world examples of SWAMM implementations; a simplistic proof-of-concept application and the application of SWAMM to a real world software project. SWAMM was applied to the latter following the 1.0 release of the project, proving the claim of application subsequent to development.

7.4 Maintenance Workflow

The SWAMM maintenance workflow is necessarily a modification of the development workflow for the simple reason that much of maintenance consists of new development (as described in Chapter II). Approximately 60% of maintenance activities on average involve the implementation of code to support new requirements. The remaining 40% consist of non-requirement modifications, such as defect repairs (bug fixes) and migration of the code to different operating environments. Treatment of new requirements and non-requirement modifications differs in SWAMM.

Development of code to support new requirements is just that – development. Development follows the development workflow described earlier in this chapter. SWAMM developers have a significant advantage once metadata has been collected about a software project, however: They may make use of that metadata to assist in new development. Leveraging the relational navigation of both descriptive and prescriptive metadata associated with a project may reduce the significant portion of maintenance activities that relate to understanding the existing code.

Developing an understanding of the code base and the requirements and code components impacted by any change also facilitates handling of non-requirement modifications. A developer repairing a bug can immediately determine which requirements need to be revalidated and which code components may be affected. Access to that information is rarely available under current software development methodologies.

As much as 45% of maintenance activities (including both new requirement implementation and non-requirement modifications) are estimated to relate to understanding an existing encoding and redesigning code structures based on that knowledge [Glass 2003, pp. 120-1]. Those activities are at least partially related to "findability" within existing code and documentation. Structured metadata, as a form of documentation, serves to reduce the relative cost of those activities if it is kept up to date. Glass calls the understanding of an existing software product "the dominant maintenance activity". The purpose of SWAMM is to allow for continued understanding of a software project throughout its lifecycle so that maintenance failure may be delayed or even avoided entirely.

A typical SWAMM maintenance workflow is identical to the SWAMM development workflow with the following modification:

2a. **State Determination**: Developers perform a query of system metadata to determine

relevant locations, properties and/or state prior to creating or making modifications to code components. State Determination occurs prior to Requirement Interpretation.

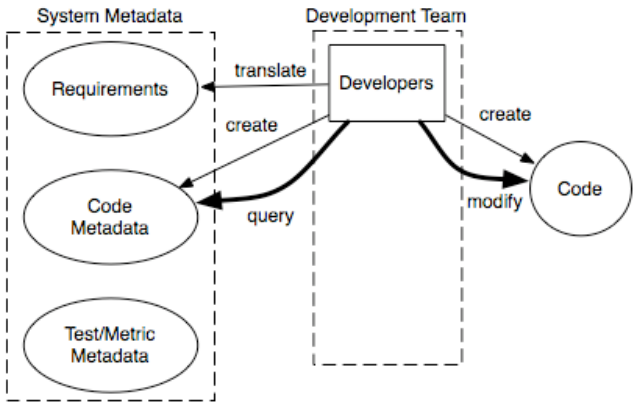


Figure 7-3. Additional Developer Actions During SWAMM Maintenance Workflow

Figure 7-3 illustrates the additional steps present in the SWAMM maintenance phase.

7.5 Relating Traditional Documentation

Most software development methodologies include the documentation of requirements and implementation results. The use of SWAMM is not intended to replace such traditional documentation, but to augment it. It is possible, however, to do more than augment traditional documentation. By placing traditional documentation into Web-ready formats and using standard metadata markup techniques, it is possible for traditional documentation to provide structured metadata about a software project and to use that metadata to augment SWAMM's system metadata. In other words, a careful choice of documentation format and encoding techniques can formally link traditional documentation with system metadata with code.

The symptoms of maintenance failure discussed in Chapter II, especially the inability to understand a software project, may be avoided or significantly delayed if traditional documentation may be related to system metadata created during SWAMM. Hyperlinking the elements of traditional documentation and SWAMM metadata facilitate understanding and provide a means to find (and possibly subsequently update) relevant traditional documentation.

Two recent developments in the encoding of metadata within documentation formats were briefly mentioned in Chapter IV: GRDDL and RDFa. GRDDL allows for metadata to be defined

and later extracted as an RDF serialization from XML and XHTML documents. RDFa allows for metadata to be defined in XHTML for later extraction, again in an RDF serialization.

Consider a simple document fragment describing part of a software project as shown in Listings 7-1 and 7-2. The document fragment is an excerpt from actual software project documentation. It describes a RESTful API for a Web-based service. RDFa markup has been added to the fragment to define structured, extractable metadata. The RDFa markup is shown in bold typeface.

Listing 7-1 shows the document header. XHTML documents using RDFa require a new DOCTYPE declaration. XML namespaces used for RDFa are listed as attributes of the HTML tag. The Friend-of-a-Friend (FOAF) and Dublin Core (DC) vocabulary descriptions have been used. The BASE tag is used to provide a default URL to an RDFa parser so that otherwise unanchored metadata relationships may form complete RDF statements.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN"
    "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:dc="http://purl.org/dc/elements/1.1/" xml:lang="en">
    <head>
            <title>Foo</title>
            <base
href="http://purlz.org/project/purl/documentation/requirements/URLs.html" />
    </head>
```

Listing 7-1. Example XHTML Header for RDFa Content

Listing 7-2 shows the document fragment itself. The RDFa markup results in three metadata statements regarding the document itself (its title, description and source document) and three metadata statements regarding its author (name, URL at his workplace and URL of employer). The extracted RDF syntax is provided in Listing 7-3.

Metadata extracted from XML or XHTML documentation has no requirement to be defined in terms of the SEC ontology. The examples above made use of the FOAF and DC vocabularies to make a point. Individual organizations may choose to extend metadata descriptions as they see fit and use knowledge of the vocabularies used in their queries. Queries are not necessarily limited to a pre-defined and relatively fixed schema, as with relational databases. That design decision allows

RDF to provide flexible, robust data services at the expense of pre-determination of query limits.

```
<h1 rel="dc:title">PURL 2 Public URL Design</h1>
<p rel="dc:description">This document describes Uniform Resource Locators
(URLs) and HTTP verbs for each public-facing requirement.  See the <a
href="index.html" rel="dc:source">Requirements Page</a> for a full
requirements list.</p>

<h2>Account Creation and Maintenance</h2>
<table border="1" cellpadding="5px">
  <tr>
    <th>Requirement</th>
    <th>Public URL</th>
    <th>HTTP Verb</th>
    <th>Parameters</th>
    <th>Auth required</th>
  </tr>
  <tr>
    <td><a href="TLregisterUser.html">Register a new user</a></td>
    <td>$PURL_SERVER$/admin/user</td>
    <td>POST</td>
    <td>Full name (name)<br />
        Affiliation (affiliation)<br />
        E-mail address (email)<br />
        User ID (id)<br />
        Password (passwd)<br />
        Hint (hint)<br />
        Justification (justification)</td>
     <td>no</td>
  </tr>
…
</table>
<p>Author:
    <span about="http://zepheira.com/team/dave">
    <a href="http://zepheira.com/team/dave/index.html"
       rel="foaf:workplaceInfoHomepage">
    <span property="foaf:name">David Wood</span></a>,
    <a href="http://zepheira.com/"
       rel="foaf: foaf:workplaceHomepage">
       Zepheira</a>
```

```
    </span>
</p>
```

Listing 7-2. Example Documentation Fragment

RDFa or GRDDL markup may naturally make use of terms in the SEC ontology, but the maintenance of those uses may be difficult. The idea presented here is to tie traditional documentation to automatically extracted and human-generated metadata by means of a RESTful architecture and hyperlinks. It is not the intent to so over-define terms in documentation that they become out of date rapidly. Creating an incentive to ignore software project documentation because it is too costly or time-consuming to maintain would work against the very core of this thesis. It seems better, then, to mark up traditional documentation in ways that minimize authorial involvement and look for ways to automatically generate and maintain linkages encoded within it.

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
  xmlns:foaf='http://xmlns.com/foaf/0.1/'
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:dc='http://purl.org/dc/elements/1.1/'>
<rdf:Description rdf:about="http://zepheira.com/team/dave">
<foaf:name rdf:datatype="http://www.w3.org/1999/02/22-rdf-syntax-
   ns#XMLLiteral">David Wood</foaf:name>
    <foaf:workplaceHomepage rdf:resource="http://zepheira.com/"/>
   <foaf:workplaceInfoHomepage
      rdf:resource="http://zepheira.com/team/dave/"/>
  </rdf:Description>
  <rdf:Description rdf:about="
http://purlz.org/project/purl/documentation/requirements/URLs.html">
    <dc:title rdf:datatype="http://www.w3.org/1999/02/22-rdf-syntax-
ns#XMLLiteral">PURL 2 Public URL Design</dc:title>
    <dc:description rdf:datatype="http://www.w3.org/1999/02/22-rdf-syntax-
ns#XMLLiteral">This document describes Uniform Resource Locators (URLs)
and HTTP verbs for each public-facing requirement.  See the Requirements
Page for a full requirements list.</dc:description>
    <dc:source rdf:resource="
http://purlz.org/project/purl/documentation/requirements/index.html"/>
  </rdf:Description>
</rdf:RDF>
```

Listing 7-3. Metadata Created from Listing 7-2.

7.6 Distributed Maintenance

The SWAMM methodology relies on metadata published using Web servers and URL addressable resources. That metadata is then used to direct or suggest maintenance activities. Although Web techniques are often used within software development organizations, they may naturally also be used on the public Internet. Distributed development teams can use the same maintenance methodology as teams that are geographically co-located, with minor modifications to allow for the distributed nature of endeavor. In this section, we explore opportunities for the SWAMM methodology operating across distributed development teams.

Free/Libre/Open Source Software (FLOSS) projects are prime examples of distributed software development and maintenance efforts. A distributed maintenance methodology would be particularly advantageous to FLOSS projects because it would assist non-developers to contribute meaningfully to software maintenance activities by reducing the barriers to entry.

One way for FLOSS projects to keep documentation current is by leveraging larger social networks. Non-developers from a FLOSS project's community may be willing to serve as monitors and/or maintainers of system metadata. Contributions to social networks such as FLOSS projects generally occur for either economic or social reasons [Kollock 1999]. That is, contributors anticipate receiving a delayed economic advantage (e.g. via building skills or raising their profiles) or they enjoy participating in the community and perhaps receiving positive acknowledgment. A recent study of Wikipedia contributors has concluded that anonymous, infrequent contributors yield material that is as reliable as major contributors [Johnson 2007]. If that finding holds for FLOSS development projects as well, it suggests that harnessing the power of a wider community could be particularly advantageous.

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

1. Enabling wider contributions to software documentation;
2. 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;
3. 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;

4. Advising users of code modules of version changes and potential impacts of those changes;
5. Advising code maintainers of the popularity (or lack thereof) of code modules.

URLs are used to uniquely identify each published version of a software component. PURLs provide an HTTP redirection to an online resource and may be modified to point to different resources over time. PURLs may further be used to resolve to current versions of software components, collections of software components and even to define releases.

PURLs are used to redirect to the current stable version, the latest version in development and any other variations that a development team may determine to be useful. Stable collections of given version numbers (in our case, addressed via PURLs) would constitute a representation of a project. Such a collection may be readily constructed from versioned software components by means of a SPARQL CONSTRUCT query. A CONSTRUCT operation produces an RDF graph from query results. In SWAMM methodology, a SPARQL CONSTRUCT produces a composite RDF graph from multiple data sources representing each of the software components, known as the project graph. The project graph is comprised of all system metadata for a particular project version. It is evaluated, again via SPARQL queries, to determine its state, such as the number and location of failing tests and requirements to which they relate.

Active PURLs are a key component of SWAMM when the methodology is applied in a distributed environment. Monitors manage redirection services to ensure that PURLs redirect to appropriate code components and return metadata describing those components. Active PURLs were described in Chapter V.

A developer, tester or monitor uses Active PURLs to narrow metadata search results to a level where the search results may be absorbed in Semantic Web user interfaces such as the Simile Project's Exhibit or Welkin [Huynh 2007], Salzburg Research's RDF Gravity [Goyal 2004] or other tool for viewing RDF metadata. None of those tools allow direct navigation of metadata repositories on the scale required by SWAMM; the use of Active PURLs to narrow search results in a user-definable manner prior to analyzing search results is therefore required. An example use of Active PURLs to narrow search results is provided in Chapter VIII.

A customer or other end user of a project using SWAMM acquires a release from a URL. The URL resolving to the release is an Active PURL and the Active PURL concept is then used to

return metadata appropriate to that release. Figure 7-4 illustrates the use of Active PURLs to determine a software release. Compare Figure 7-4 with Figure 5-6.

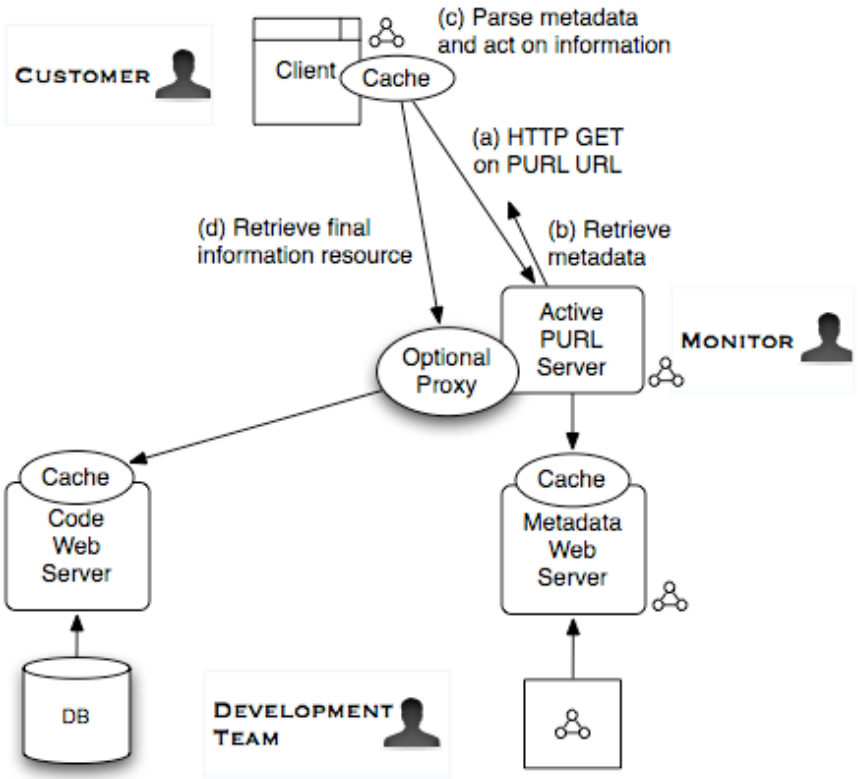


Figure 7-4. Monitors Use Active PURLs to Curate Software Project Information

Note that a monitor assembling a project chooses components by selecting the appropriate URL for each component. They are not constrained to choose a particular version, and may choose to override hints provided by metadata or the desires of a component's developer. Thus, this methodology retains the same level of flexibility currently offered by revision control systems.

Metadata could be used to manage software dependencies 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 [Krech 2005] and an extension to a REST-based microkernel such as NetKernel [1060 Research 2004] would be straightforward.

SWAMM using PURLs to identify and collect code components into a coordinated release replaces the need for centralized revision control systems. The Web becomes the revision control system. Individual authors maintain control over the act of publishing their components. Existing Web security systems may 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) URLs

eases searchability.

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.

The ability for a developer to determine impact in advance of upgrades or modifications 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. Since each URL-referenceable code module has associated version information, documentation on the versions of code modules that constitute a project release could be automatically generated. Project version numbers could also be automatically assigned upon decision to publish.

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.

It is not strictly necessary to replace revision control systems and there are strong arguments against doing so. The most important argument may be that centralizing code components is necessary for compilation and packaging anyway. Another is that centralized systems are easier to back up. As noted earlier in this chapter, several widely used revision control systems optionally assign URLs to code components and make metadata about each revision available via the Web. Such functionality is not to be lightly dismissed. Early implementations of SWAMM preferred to leverage the functionality of revision control systems instead of attempting to replace them.

7.7 Modeling SWAMM Interactions

The Software Agent Maintenance Methodology (SWAMM) was modeled to determine properties of the methodology. The model simulates *in silico* the development and maintenance phases of a software project and applies the SWAMM development and maintenance workflows described earlier in this chapter.

The model allows a software project to be simulated by determining the number of customers, developers, testers and monitors to be used, followed by injections of requirements (during the development phase) and modifications (consisting of both new requirements and non-requirement modifications) during the maintenance phase. Multiple and non-sequential development and maintenance phases are supported.

An estimated amount of metadata and tests created and review requests generated are graphed at the bottom of the user interface versus the number of requirements and modifications injected.

The NetLogo modeling environment, a functional Logo-like language and development system, was used to model the methodology [Wilensky, 1999].

A version of the model exported as a Java applet is available via http://www.itee.uq.edu.au/~dwood/models/SWAMMApplet/SWAMM.html and requires a Web browser capable of running Java applets. Java 1.4.1 or higher is needed. The NetLogo applet implementation is known not to run on Windows 95 or versions of Mac OS prior to OS X.

The applet version of the model has some limitations compared to the same code running in the native NetLogo environment. The applet version does not have a command-line interface to the NetLogo interpreter, nor may the code be viewed or edited via the user interface. The source code is available from http://www.itee.uq.edu.au/~dwood/models/SWAMMApplet/SWAMM.nlogo, a file that may be loaded into the full NetLogo environment by those wishing to access those features.

The main area of the model's user interface, in the center, represents actors and relationships between them. Actors are shown as iconographic nodes in a network and relationships are shown as links between the nodes. Relationships represent actions such as creating metadata or notifying another actor. The main area of the user interface is initially black until initialization parameters are chosen and the model run.

Taking the following steps will run either version of the model:

(1) Start in the upper left of the user interface. Select the number of customers that will provide

requirements to a software project, the number of developers working on the project, the number of testers and the number of monitors (i.e. project managers or others with responsibility of reviewing work product).

(2) Select the "Setup" button. That will create nodes ("turtles" or "agents" in NetLogo terminology) representing the actors you requested as well as nodes representing code and metadata about the software project.

(3) Development Phase: Choose a number of requirements to inject into the software project during the development phase. Make sure to use the tab key or mouse to remove focus from the "requirements_to_inject" text input field so your number will be accepted. Select the "Inject Requirements" button when ready. This action will simulate customers requesting requirements to be coded and will result in development, testing and metadata collection occurring. Links will be created between the actors as development proceeds. Selecting a link will allow introspection of its type ("breed") and weight (number of times it has occurred) when running in the full NetLogo environment. The NetLogo applet does not support variable introspection. More requirements may be injected at any time.

(4) Maintenance Phase: Choose a number of modification requests to inject into the software project during the maintenance phase. Select the "Inject Modifications" button when ready. Note that a number of modifications was set for you when you injected requirements in (3) above. That number is suggested based on the 60/60 Rule of software maintenance and is representative of the number of modifications expected in a typical software system with the number of requirements you specified earlier. The suggested number of modifications to inject may be overridden as desired.

The state of the network at the end of a project's maintenance phase may be reviewed to determine the amount of metadata created. The amount of metadata and tests created and the maximum number of possible review opportunities are graphed versus the number of requirements/modifications requested. Monitors at the bottom of the interface show the number of requirements and modifications injected and the total number of development iterations.

A single requirement or modification request may result in a number of iterations. The number of iterations for any particular request is estimated via a Poisson distribution. Poisson distributions are also used to calculate the number of code components to create for a given request. The numbers of code-, test- and metric-related metadata elements are estimated as functions of the number of code components created. The exact formulas for estimating those values are discussed

in Section 7.8.

Note that the amount of metadata, tests and reviews generated are substantial, compared to the number of requirements and software modifications, but that the complexity is linear.

There are six different types (“breeds”, in NetLogo terminology) of links: provide, review, create, notify, test and translate. Visibility of each type may be turned on or off to assist the analysis of relationships between nodes. The visibility of each link type may be turned on or off with the switches on the upper right side of the user interface. Link visibility will update when the "Update link display" button is selected.

Widths of displayed links grow as the links' weight increases. The relative widths of the links may be changed to make the display more readable by adjusting the "link_size" slider and selecting the "Redraw links" button on the middle right side of the user interface.

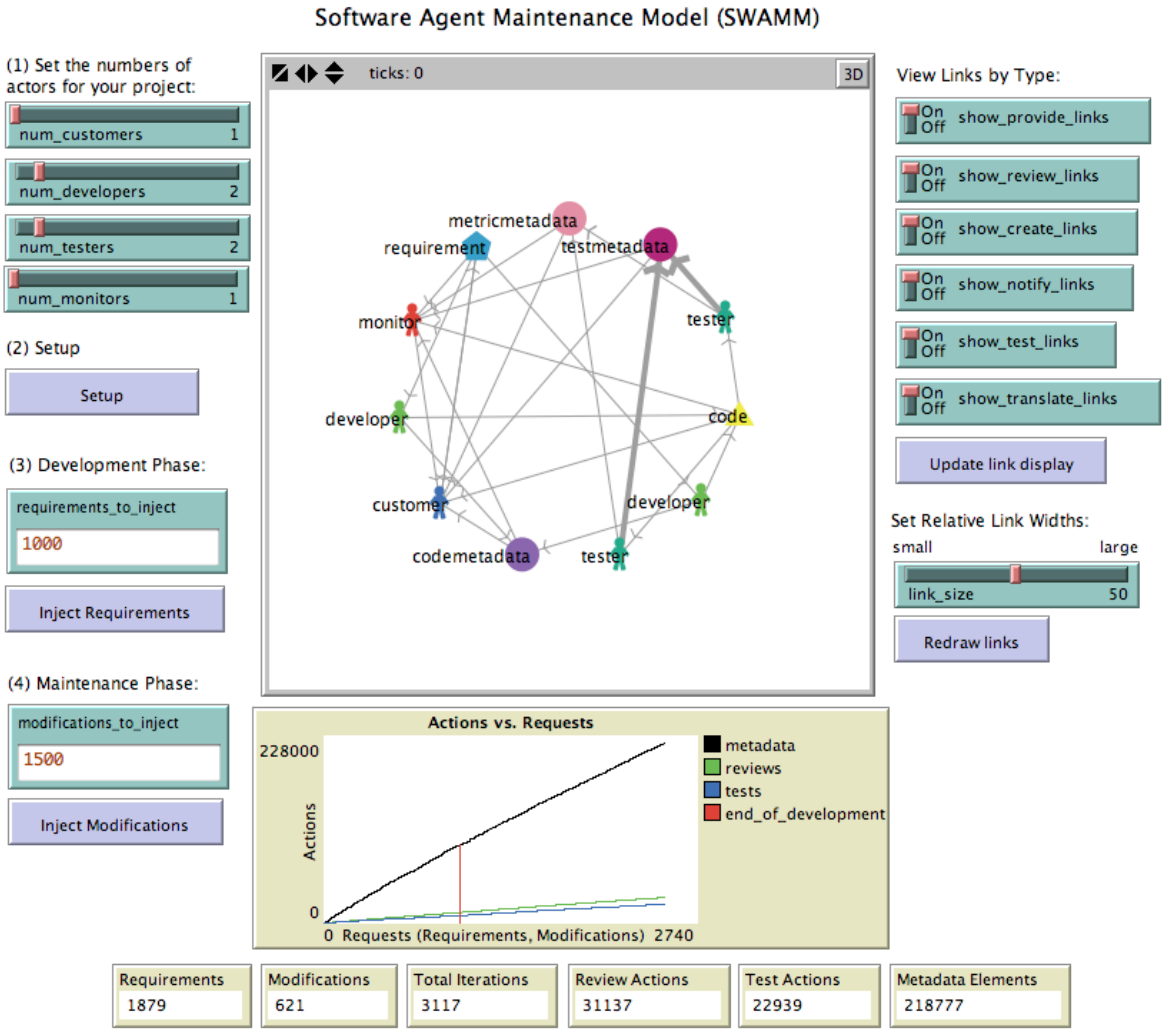


Figure 7-5. User Interface of SWAMM Interactions Model.

Figure 7-5 shows a screenshot of a typical model run. One customer, two developers, two testers and one monitor were used. One thousand requirements were injected during the single development phase and one thousand five hundred modifications were injected during the maintenance phase. Approximately 220 metadata elements were created per development phase requirement. Section 7.8 analyzes scalability expectations in more detail.

7.8 SWAMM Scalability

Usability of the SWAMM methodology in practice is limited by the ability of an information system to provide storage and query facilities for the metadata produced. This section provides an analysis of SWAMM scalability to determine practical limits on its usability.

The total number of requests made by customers of a software project (R_{total}) may be divided into requests made during development (R_d) and requests made during maintenance (R_m), as was considered in Chapter II. All requests made during development are expected to relate to requirements. A different division may also be made by considering requirements that are expected to add functionality to the project (R_{req}) and modifications that do not change the intended functionality (such as bug fixes and migrations to different environments, R_{mod}). Although it is recognized that the separation between R_{req} and R_{mod} may not be as cleanly separable in practice (e.g. when a migration requires a feature addition), the relationships shown in Equation 8-1 generally hold.

$$R_{total} = R_d + R_m \approx R_{req} + R_{mod} \qquad \textbf{Equation 8-1}$$

The 60/60 Rule, described in Chapter II, suggests that (on average) 60% of the overall cost of a software project is incurred during maintenance activities. Equation 8-2 is readily derived, presuming that salaries and other "people costs" and not capital expenditures dominate overall costs and thus the cost bases during the development and maintenance phases are similar. Equation 8-2 describes maintenance requests (costs) in terms of the requests made by customers during the development phase of a software project and is a useful thumb rule.

$$R_m \approx 1.5R_d \qquad \textbf{Equation 8-2}$$

Further, 60% of maintenance activities are new requirements, again on average. Using the 60/60 Rule we can suggest that the number of requirements (R_{req}) bears a relationship to the number of modification requests (R_{mod}). The total number of requirements consists of all requests made during the development phase (by definition) and roughly 60% of the requests made during the maintenance phase ($R_d + 60\%(1.5\ R_d)$), yielding Equation 8-3.

$$R_{req} \approx 1.9R_d \qquad \textbf{Equation 8-3}$$

The number of non-requirement modifications consists of the remainder, namely approximately 40% of requests made during the maintenance phase, as shown in Equation 8-4.

$$R_{mod} \approx 0.6R_d \qquad \textbf{Equation 8-4}$$

The total number of metadata elements created during a software project may be estimated and is based upon analysis of the SEC ontology and the SWAMM model. The total number of metadata elements created during the lifespan of a project (M_{total}) is the total of the metadata associated with each requirement (M_{req}) plus the metadata relating to non-requirement modifications (M_{mod}), as shown in Equation 8-5.

$$M_{total} = M_{req} + M_{mod} \qquad \textbf{Equation 8-5}$$

Metadata associated with requirements includes information describing the requirements themselves (approximately 6 elements per requirement), and information describing code components, tests and metrics. Six or seven metadata elements are required to describe a code component, but it is important to note that the number of code components varies per requirement. A Poisson distribution is used to model the variable number of code components that may be created to implement any given requirement. An analysis of several Open Source software projects suggested a mean for the Poisson distribution of 3.2 code components per requirement. The notation P_x is used to denote a Poisson distribution with a mean of x.

A trend in Agile and Test Driven methodologies is to create an increasingly large number of

tests per code component. Another Poisson distribution was used to model the number of tests, with a mean of 4.2 tests per code component. Six metadata elements are necessary to describe a test.

Given the relatively limited use of metrics noted in Chapter VI, a single metric (e.g. source lines of code) was modeled per code component. Three metadata elements are used to describe each metric.

The metadata associated with requirements may therefore be estimated using Equation 8-6.

$$M_{req} \approx 6R_{req} + M_c + M_t + M_m \qquad \textbf{Equation 8-6}$$

where:

Number of iterations, $i = \text{integer}(R_{req}P_{1.3})$
Number of code components, $N_c = iP_{3.2}$
Code metadata elements, $M_c = 6.5N_c$
Test metadata elements, $M_t = 6N_cP_{4.2}$
Metric metadata elements, $M_m \approx 3N_c$

Metadata created during non-requirement modifications is limited to the creation of new tests (although existing metadata is expected to be modified). No metadata is added describing new requirements or code components since there aren't any. A reasonable simplifying presumption is made that new metrics will not be created during non-requirement modifications. Six metadata elements are used to describe a test.

$$M_{mod} \approx 6\ R_{mod} \qquad \textbf{Equation 8-7}$$

The NetLogo model implements the relationships encoded in Equations 8-6 and 8-7.

Equations 8-6 and 8-7 may be substituted into Equation 8-5 and transposed into terms of the number of requirements injected during development (R_d) using Equations 8-3 and 8-4. The resulting estimation of the total number of metadata elements created during a project is shown in Equation 8-8.

$$M_{total} \approx 11.4R_d + M_{cd} + M_{td} + M_{md} + 3.6R_d \qquad \textbf{Equation 8-8}$$

where:

Number of iterations, id = 1.9RdP1.3
Number of code components, Ncd = idP3.2
Code metadata elements, Mcd = 6.5id
Test metadata elements, Mtd = 6NcdP4.2
Metric metadata elements, Mmd ≈ 3Ncd

Equation 8-8 may be expanded and the terms collected to yield Equation 8-9.

$$M_{total} \approx R_d\,(15 + P_{1.3}\,(12.4 + P_{3.2}\,(11.4P_{4.2} + 5.7)\,) \approx KR_d \qquad \textbf{Equation 8-9}$$

where:
$K = 15 + 1.3*(12.4 + 3.2*(11.4*4.2 + 5.7))$
$= 254$, for this example

Equation 8-9 demonstrates that the total amount of metadata created in the SWAMM methodology scales linearly with the number of requirements injected during development (R_d). The values of the Poisson distributions will naturally collapse to their means on average. We can write the scalability finding more formally with the use of "Big O" (or Landau) Notation by stating that the amount of metadata created is on the order of the number of requirements injected during development, as shown in Equation 8-10.

$$g(M_{total}) \in O(f(R_d)) \qquad \textbf{Equation 8-10}$$

Linear scalability on the order of the number of development requirements ensures that the amount of metadata created is both estimable and tractable. The SWAMM methodology thus requires one or more queriable metadata stores on the order of the development requirements. The most scalable RDF metadata store currently available as Open Source Software is the Mulgara Semantic Store, which is capable of storing some hundreds of millions of metadata statements using commodity hardware and is being extended as of this writing to store many billions. One billion metadata statements would allow for the description of a software project with nearly four million Development Phase requirements (a very large software project indeed).

Note that metadata is produced at a higher rate during development than during maintenance. This is because roughly 40% of maintenance activities do not result in the creation of new software components (by the 60/60 Rule: 60% of software cost is incurred during maintenance and 60% of

maintenance activities relate to enhancements). Thus, the slope of the line representing the total number of metadata elements created compared to the number of requirements and modification requests is greater during development and lessens during the maintenance phase. This is shown in the widgets on the model's user interface labeled "Development slope" and "Maintenance slope". The model shows that behavior, as shown in Figure 7-6.

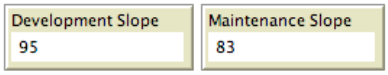


Figure 7-6. Reduction in Metadata-Request Slope During Maintenance (following Figure 7-5).

Scalability in the number of review requests may also be considered. The model implements the methodology illustrated in Figure 7-1 literally and therefore creates a review request for every metadatum and code component upon completion of each customer request. Automated tests are expected to catch regressions, as in much modern practice, and therefore Customers, Monitors and even Testers are expected to review only newly created material, existing material likely to be affected and overall state upon review requests. The total number of review requests generated is much smaller in practice than illustrated in the NetLogo model. Scalability in review requests is also linear with the number of requirements.

7.9 Relationships to Other Methodologies

7.9.1 Waterfall

Perhaps the most obvious argument for SWAMM use is for those software projects developed with the Waterfall methodology. The Waterfall methodology poorly defines maintenance phase activities and SWAMM is focused on maintenance phase activities. Waterfall and SWAMM are nearly orthogonal.

Software projects that are developed using a Waterfall methodology may transition to SWAMM during their maintenance phases. SWAMM metadata may be collected during development (as an adjunct to the Waterfall methodology) or afterward.

If maintenance activities return to a Waterfall or overlapping Waterfall ("sashimi")

methodology during requirement iterations, the existence of SWAMM metadata would help to drastically reduce the costs associated with understanding and navigating the code base.

7.9.2 Extreme Programming

The SWAMM methodology fits well with Agile methodologies. Extreme Programming (XP), as the rather elderly grandfather of Agile methodologies, has some long-identified drawbacks. Controversies surrounding XP include the lack of captured requirements and documentation of requirements and the management of user conflicts [Copeland 2001]. The SWAMM methodology provides partial solutions to these problems.

Capture of requirements as metadata may occur with the Extreme Programming development cycle at the time that user stories are captured on Class-Responsibility-Collaboration (CRC) cards. Developers and testers are generally the same actors in XP and the creation of the metadata associated with them may occur within the same development cycle.

The capture of structured metadata during an XP development cycle is not a particularly burdensome requirement, but it does address the capturing of requirements and the documentation of requirements. Further, analysis of requirements metadata by monitors (who, in XP, may be the developers themselves) provides a tool to discover and address user conflicts. The discovery of user conflicts early in development prevents the cycling of changes to source code that is perhaps XP's most important failure.

7.9.3 Test Driven Development

Test Driven Development (TDD) is an Agile methodology that has grown out of Extreme Programming. A TDD developer writes tests first, and then develops code until the tests pass. TDD is generally not recommended for software projects where code components may not be easy to isolate, such as the development of graphical user interfaces.

SWAMM fits well into the TDD test-development-test cycle (noting that the order of metadata creation would be modified since the order of component creation is modified during TDD), but does nothing to address TDD's limitations. TDD and SWAMM do not conflict, but do not provide any particular benefits during the development phase. SWAMM's benefits occur in the maintenance phase, where TDD is less commonly applied. SWAMM therefore provides a

maintenance methodology suitable to projects that used TDD during initial development.

7.10 Conclusions

This chapter defined the Software Agent Maintenance Methodology (SWAMM), a software maintenance methodology. SWAMM collects metadata regarding requirements, code component relationships, tests and metrics to facilitate the understanding of a software project during the maintenance phase. Chapter VIII reports on early implementation experiences with SWAMM for both a simple toy example and a real-world software project.

VIII. Experience and Evaluation

"Experience converts us to ourselves when books fail us."
-- Amos Bronson Alcott (1799 - 1888)

The SWAMM methodology described in Chapter VII has been applied twice to ensure that the procedures were understood and to expose any practical problems with its use. Manual creation of metadata alone does not prove that such metadata is readily obtainable from existing sources and using (mostly) existing tools, nor do the SPARQL queries given in Chapter VI exercise the entire workflow of the SWAMM methodology. This chapter attempts to fill those gaps by acting out the aspects of SWAMM that transcend the mere theory of metadata creation and direct applications of the SEC ontology.

In the first application of SWAMM, metadata representative of a portion of an existing small software project (JRDF, an RDF API written in Java) was created manually as described Section 6.5. The SWAMM methodology was applied to the pre-defined metadata to determine reasonable means of using each step.

In the second application, existing tools were used to extract information about a larger existing software project (the PURL project discussed in Chapter V) and the information manipulated by scripts to fill gaps and fit the metadata to the SEC ontology. Once the project was described by SEC metadata, the metadata was viewed in several Semantic Web user interfaces. Results were analyzed to ensure that relational navigation of the metadata could be accomplished and that user interactions required by the SWAMM methodology could be successfully implemented. The SWAMM methodology requires users to gather information (e.g. to determine software components that may be impacted by a maintenance action) and to make decisions (e.g. when to release a software version). User interfaces appropriate for SWAMM must create flexible analysis environments for those actions. The user interfaces were judged in relationship to their usefulness for SWAMM.

Representative statistics regarding the two applications of SWAMM are shown in Table 8-1. The portion of the JRDF project represented was trivial with the purpose being to demonstrate the viability of basic techniques in SWAMM. By comparison, the PURL project was represented

nearly in its entirety, thereby demonstrating that the techniques proposed in Chapters VI and VII are applicable to a real-life software project of some reasonable level of complexity.

The remainder of this chapter provides details of these applications of SWAMM. Procedures used to develop the metadata representations, types of queries developed to show efficacy to software maintenance and tools used for analysis are discussed.

Table 8-1. SWAMM Application Summary

Project	JRDF	PURL
Non-commented Lines of Code	47	125179
Packages	1	8
Classes	2	79
Methods	4	427
Requirements	1	87
Tests	2	107
Metrics	1	87
Metadata creation	Manual	Automated, with manual mapping of requirements and tests to code components

Careful readers will note the multiple occurrences of the number 87 in the PURL column of Table 8-7. One metric (source lines of code) was calculated for each package and class in the PURL project. The number of packages and classes total 87 and therefore 87 metrics were calculated. The number of requirements defined separately for the project was coincidently also 87. The number of requirements has no direct relation to the number of code components.

8.1 RDF User Interfaces for Relational Navigation

The SWAMM State Determination activity is where a developer queries system metadata prior to changing one or more code components in a system during the maintenance phase. The State Determination activity is the most important step in SWAMM, in that most assistance provided to a developer using SWAMM is presented during the activity. The State Determination activity

requires some form of user interface by which developers can interact with the collected metadata. Providing an end user, even a developer, with a SPARQL API is insufficient. Fortunately, a number of generic Semantic Web user interfaces have been developed. The use of Semantic Web standards allows for the use of these user interfaces without modification and should be seen as a benefit to our approach.

Tim Berners-Lee has classified Semantic Web user interfaces (often called "browsers", not to be confused with traditional Web browsers) into two categories: Those designed to present an entire dataset for perusal and those designed to follow linked data from link to link [Berners-Lee 2007]. Examples of the former include RDF Gravity [Goyal 2004], mSpace [Schraefel 2003], Exhibit [Huynh 2007], Longwell [Butler 2006], Haystack [Karger 2005] and ontology editors/viewers such as Protégé [Knublauch 2004] and SWOOP [Kalyanpur 2004]. Examples of the latter include Tabulator [Berners-Lee 2007], Disco [Bizer 2007], Oink [Lassila 2006] and the OpenLink RDF Browser [Open Link Software 2008].

Semantic Web browsers designed to explore a single data set may be further classified into those that support faceted navigation on generic data, those that directly visualize an arbitrary RDF graph and those that visualize specific forms of data in RDF for particular purposes. Exhibit and Longwell are faceted navigation systems; RDF Gravity is a graph viewer; Ontology editors, mSpace and Haystack view specific forms of data.

SWAMM metadata is generic RDF, described by one or more generic ontologies. It is most appropriate to explore SWAMM metadata with a user interface supporting arbitrary information. The simulated JRDF metadata was developed in OWL using the Protégé and SWOOP ontology editors, as was the SEC ontology itself. Although the metadata is viewable in such editors, they are inappropriate to use as search and retrieval tools for end users due to their primary purpose being editors of logical relationships. Similarly, mSpace and Haystack could not be used without appending the SWAMM metadata with information specific to the data formats recognized by those applications.

Graph viewers do not scale particularly well because RDF is a very verbose data structure. That is one price RDF pays for its flexibility; the atomic unit of RDF (the RDF statement) contains relatively little information, equivalent to a cell in a relational table. One result of RDF's verboseness is that graph viewers cannot display much information at one time. RDF Gravity was used to investigate the JRDF metadata set, but larger metadata sets could not be realistically explored using JRDF Gravity.

Semantic Web browsers designed to follow linked data are not considered optimal for SWAMM metadata for two reasons; SWAMM metadata is quite large and tends to be mostly centralized, even for distributed development teams. Linked data browsers have (to date) been shown to successfully navigate small amounts of linked data (e.g. from FOAF page to a personal Web page to a linked document), but do not fare as well when faced with multi-thousand statement RDF stores. It may be possible to publish SWAMM metadata on a multiplicity of sites, as briefly suggested in Chapter VII, and to allow developers to "follow their nose" from one small document to another. This possibility has yet to be fully explored.

By process of elimination, faceted navigation user interfaces are most appropriate for the exploration of SWAMM metadata. Maintenance of the relationships between software components and system metadata has been recognized to require some form of relational navigation [Jarrott 2003], and faceted navigation is relational in nature.

Exhibit and Longwell are both Web-based RDF interfaces. Exhibit is a client-side faceted browser implemented in Javascript and HTML. Longwell is a server-side Web application with a user interface served as HTML with Javascript. The difference between the two approaches may seem subtle to an end user, but the former is limited severely in terms of scalability and the latter is much less so.

The JRDF and PURL projects' metadata was displayed via RDF Gravity, Exhibit and Longwell. As expected, the faceted navigation capabilities of the faceted navigation interfaces allowed rapid and useful review of the JRDF and PURL projects via their metadata. Each interface provided slightly different facets and user interface paradigms, but the results were similar. The representative maintenance actions identified in Chapters VI and VII were recreated in each interface. Detailed experiences with these three RDF user interfaces are presented in Sections 8.2 and 8.3.

8.2 Manual Application of SWAMM: JRDF Project

Example metadata was manually developed to represent a small portion of the org.jrdf.sparql package of the JRDF project, version 0.3.4 (http://jrdf.sourceforge.net). The example metadata was described via the SEC ontology and presented in Chapter VI. The metadata is available online at [Hyland-Wood 2008b].

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. One class has a requirement associated with it.

The JRDF projects' metadata was viewed using RDF Gravity, Exhibit and Longwell. Exhibit, like RDF Gravity, can handle roughly one thousand nodes of RDF metadata in a single instance. The JRDF test data consisted of 160 RDF statements plus an additional 451 RDF statements in the SEC ontology, for a total of 611 RDF statements. Thus, even the toy example of the JRDF data set neared the limits for RDF Gravity and Exhibit. The PURL project metadata was much larger; it consisted of more than 4,000 RDF statements. Display and analysis of the PURL project metadata is discussed in Section 8.3.

The JRDF metadata was loaded into a dynamically generated HTML page using the Exhibit Javascript library and facets defined for navigation. The entire JRDF data set was readily viewable in Exhibit.

The development phase of JRDF is complete and the project has been in maintenance for some years. The SWAMM methodology was therefore applied to the maintenance phase of the project.

The maintenance phase of SWAMM consists of the following activities: Requirement Injection, State Determination, Requirement Interpretation, Testing, Internal Validation, Release, and External Validation. Those activities were described in detail in Chapter VII. Three of the activities, Requirement Injection, Requirement Interpretation and Testing, were simulated by the manual creation of metadata. A requirement was created and manually mapped to an existing code component. Two tests were manually created and manually mapped to existing code components. One of the test results was arbitrarily set to "Passed" and the other to "Failed". The creation of this metadata was accomplished by manually adding twenty-five RDF statements to the automatically generated metadata. Searching for the strings 'sec:Requirement' and 'UnitTest' in [Hyland-Wood 2008b] will show those statements.

The State Determination activity requires some form of user interface, as noted in Section 8.1. RDF Gravity was selected for the simple JRDF metadata set. The JRDF metadata set was loaded into RDF Gravity and viewed. Figure 8-1 shows the failing test in the JRDF data set being inspected. RDF Gravity is a graph viewing application; that is, it shows each RDF node as a graph node in the display and each RDF predicate as a line connecting two nodes. The nodes and lines are styled according to their RDF types. The amount of information shown in the graph window may be impacted by applying filters (in the right-hand windows) or by zooming (via the control in

the lower right of the screenshot). Figure 8-1 shows information relating to the class SparqlQueryBuilderUnitTest, which is a unit test for the JRDF SparqlQueryBuilder class. The `hasTestResults` property is shown with a literal value of "Failed", indicating a failed test. A passing test may be seen in the upper left of the screenshot.

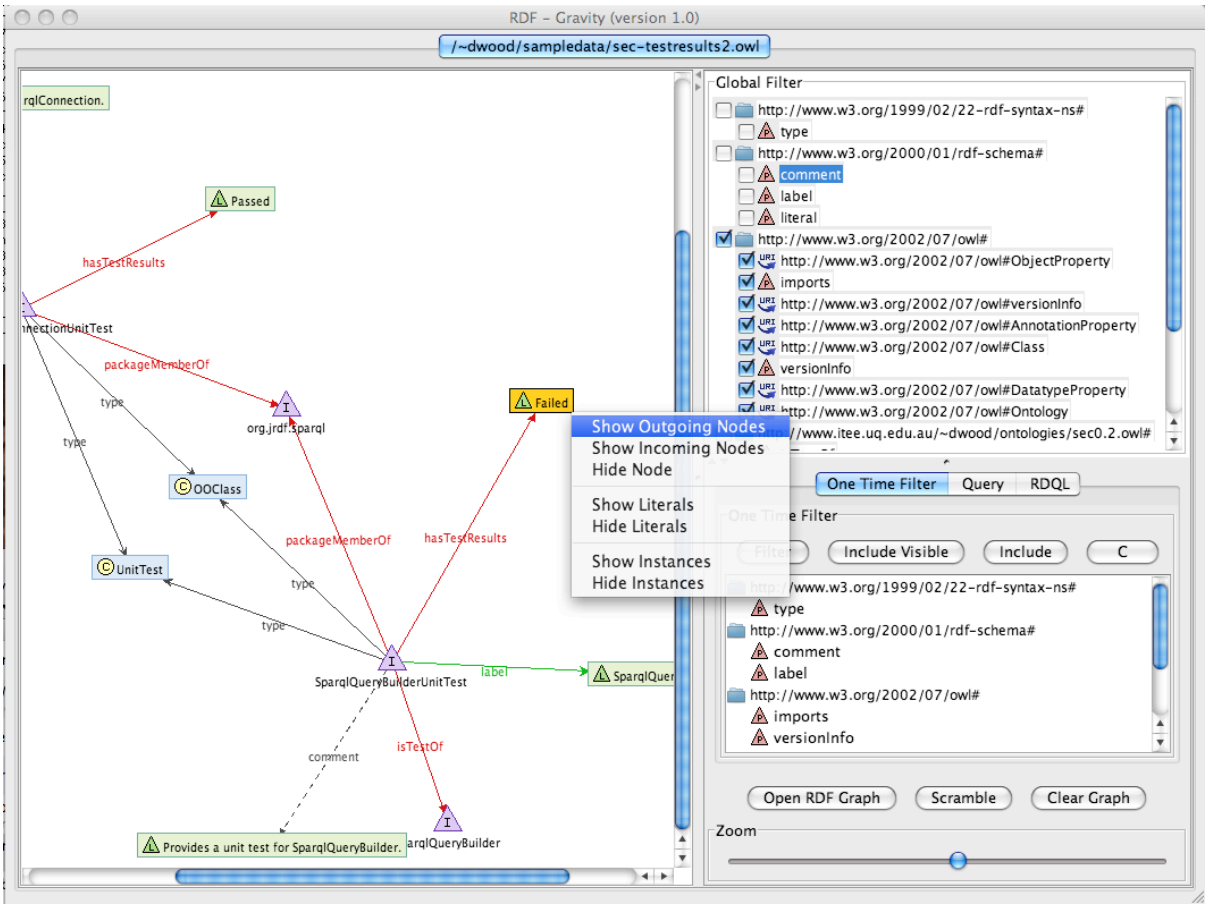


Figure 8-1. Inspecting a Failed Test in RDF Gravity

RDF Gravity, like Protégé, SWOOP and others, allows RDF metadata to be loaded from a URL or a local file. Since the JRDF metadata is small (roughly 54 KB, including the SEC ontology), the entire metadata set could be loaded into RDF Gravity. Given that RDF Gravity can only visualize on the order of one thousand RDF nodes, this approach is not scalable since it could it adequately represent an entire real-world software project. The next section, regarding the SWAMM application to the PURL project, provides a solution to the scaling problem by using an Active PURL.

The Internal Validation activity (conducted by monitors) is similar in scope and actions to the State Determination activity (conducted by developers). In both cases, an actor reviews the states

of both code and metadata by first querying the metadata and possibly updating the metadata (or causing it to be updated via a tool). Thus, the approach taken for State Determination applies to the Internal Validation activity as well.

The Release activity consists of identifying versions of code components by URL and collecting them into a release state. Each code component, requirement, test and metric was given its own URL, but the URLs were not versioned. Therefore the JRDF case study did not address the full actions inherent in the Release activity because temporal changes in code components were not taken into account. The Release activity was more fully addressed in the PURL project's case study in Section 8.3; that section discusses how a unique URL may be easily assigned to each code revision.

The External Validation activity consists of actions by customers to validate that the code that was created matches their understanding of the requirements presented. Customers may review metadata in the same manner as developers during the State Determination activity and monitors during the Internal Validation activity, but they are not constrained in any way to do so. Customers may, and often do, simply run the code presented to them following a release. There was thus no prototyping of the External Validation activity since it devolves either to a null action (in the case where no external feedback is offered) or to equivalence with the State Determination and Internal Validation activities.

8.3 Semi-Automated Application of SWAMM: PURL Project

Example metadata from the PURL project (http://purlz.org) was automatically extracted and subsequently manually augmented with test results and requirements mappings to represent the entire Java portion of the software project. Substituting SEC ontology terms for terms used by the various metadata extraction tools normalized automatically extracted metadata. This section describes the metadata extraction process in detail.

The example data consists of seventy-nine object-oriented Java classes that implement eighty-seven documented requirements. The PURL project was chosen partially because its requirements are both documented and publicly available on the Web.

Metadata from Java class files was extracted using the MIT Simile Project's Java RDFizer [Mazzocchi 2007] and the resulting RDF changed by Bourne Again Shell (BASH) scripts to use the SEC ontology. Detailed information was not collected at the method level of the code base since

the Java RDFizer produced descriptions only of packages and classes. Descriptions at the package and class levels were deemed sufficient for the purposes of the PURL project's case study.

The PURL project's requirements are available on the Web as a series of XML documents. Those XML files were transformed via an XSLT script, developed for the purpose, into an RDF description using the SEC ontological relationships.

The JavaNCSS utility [Lee 2006] was used to create a simple source-lines-of-code metric for each package, class and method. Package and class information was retained and mapped via script to the RDF collected from the Java RDFizer.

The PURL project's code components are available from a Subversion revision control system that includes a Web interface. The URL pattern used by the Subversion Web interface was used to create seeAlso references for each code component. The seeAlso references were used during the modeling of the SWAMM Release activity, as discussed later in this chapter.

A manual mapping of requirements to classes and tests to classes was required because such mappings were not available in the available data. A portion of the project's requirements and tests were mapped to classes in the org.purl.accessor package in order to demonstrate relational navigation in the user interfaces described later in this section: All unit tests were captured, but only two relating to the class java:org.purl.accessor.PURLSAccessor were manually provided with test results. All requirements were captured but only three requirements were manually mapped to the java:org.purl.accessor.PURLSAccessor class. The purpose of providing test results and requirements mappings relating to a particular class was to demonstrate relational navigation in SWAMM without having to manually provide test results and requirements mappings for each class in the project.

The development phase of the PURL project is complete and the project has entered maintenance during this writing. The PURL project represents a real-world project of moderate complexity that recently completed its initial development as of this writing. Only the Java portion of the PURL project code was used in the use case due to the ready availability of automated extraction tools for software component relationships. The vast majority of the project was implemented in Java.

The Requirement Injection activity was captured in its entirety by the availability of written requirements. All requirements were converted into SEC requirement descriptions regardless of their implementation details; that is, requirements were represented that may not have been implemented in the portion of the source code represented. This was simple to arrange because the

author of this thesis was the lead architect and analyst for the PURL project.

The State Determination and Internal Validation activities were acted out by analyzing collected metadata in a series of Semantic Web user interfaces; Longwell, Exhibit and RDF Gravity.

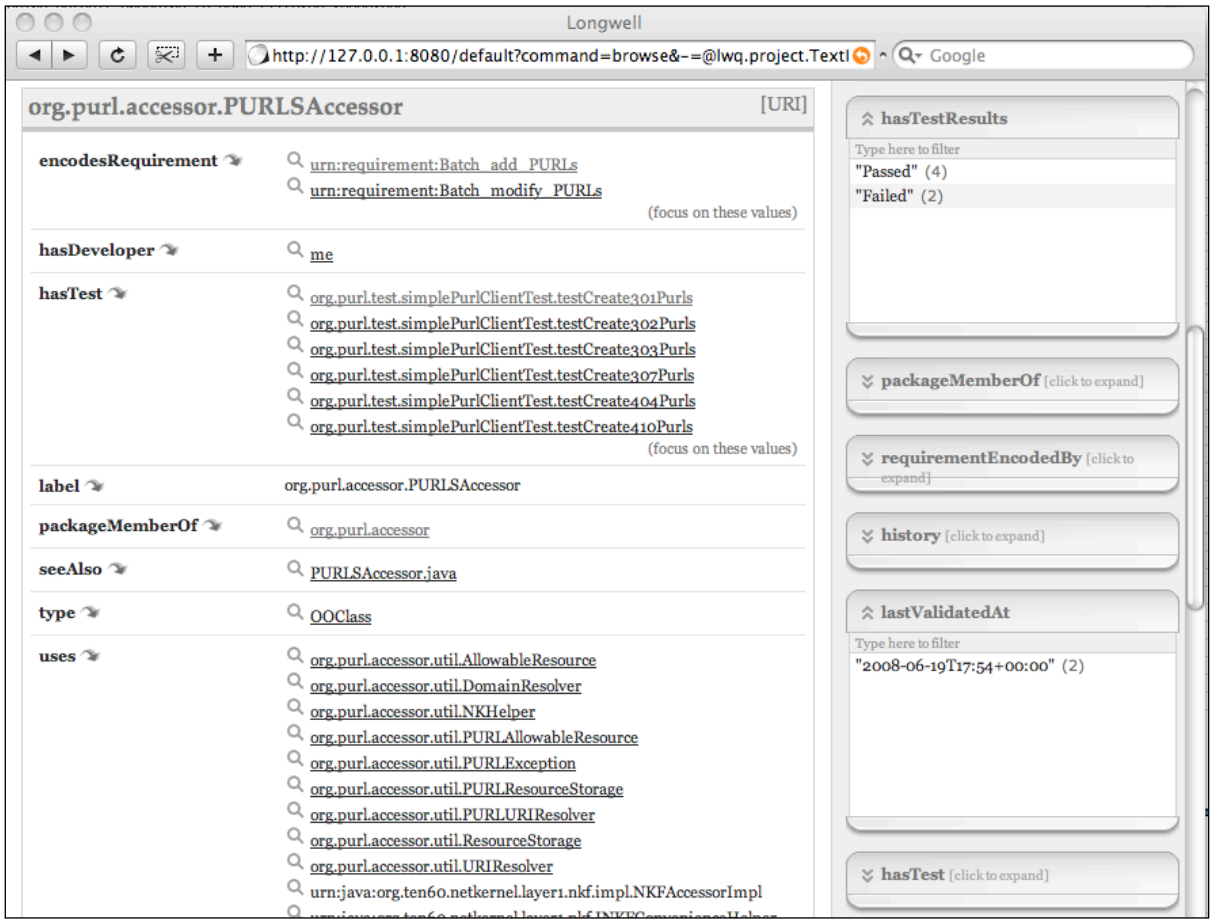


Figure 8-2. Faceted Navigation of PURL Project Metadata Using Longwell

The entire metadata set loaded into Longwell. Longwell uses the Jena Semantic Web Framework [McBride 2002] and thus scales reasonably well. Some tens of millions of RDF statements may be stored in Jena depending on the backing store chosen. Thirty million RDF statements, by the estimate developed in Chapter VII, would be sufficient to represent a software project with roughly 120,000 Development Phase requirements by use of Equation 7-9 in Chapter VII. Current efforts are being made to use the Mulgara Semantic Store [Gearon 2006] as a storage layer for Longwell, which will extend Longwell's scalability. Longwell is thus considered to be acceptable in terms of scalability for most SWAMM activities because it can provide relational navigation over more SWAMM metadata than are expected for most software projects.

Figure 8-2 shows Longwell displaying some RDF metadata for the PURL project. The class org.purl.accessor.PURLsAccessor has been selected. Careful inspection of the screenshot shows that the class has six tests associated with it and two of them are currently failing. Longwell's generic faceted navigation capabilities allowed investigations to start with failing tests and work toward classes associated with them and the reverse. Critical SWAMM metadata, such as last validated date-times and navigational paths to requirements are readily available. Longwell's facets, as well as its look and feel, are configurable. The case study described here used Longwell's default configuration.

There are at least two reasons to consider user interfaces for SWAMM other than Longwell: A centralized repository of metadata may not suit a software project team or the amount of metadata may exceed the comfortable limits of Longwell's storage layer. For scaling beyond the scope of Longwell, a way is needed to query a subset of the metadata available.

Mapping a query to a unique URL allows several existing user interfaces for RDF data to visualize query results. That is because some existing RDF user interfaces allow the addressing of data sets by URL. One way to map a query to a URL is with an Active PURL, as discussed in Chapter V. An Active PURL was prototyped to provide a URL for a SPARQL endpoint (that is, a Web service listening for SPARQL protocol connections). The use of the prototype Active PURL is described later in this section for Exhibit and RDF Gravity as user interfaces for the State Determination and Internal Validation activities.

The metadata collected contained 4,446 RDF statements in 795 records. Each record contained metadata statements regarding a given resource (actor, code component, metric, test, etc). Although a single instance of Exhibit was capable of viewing the entire metadata set, some performance degradation was noticed. Experimentation with Exhibit suggested that it ceases to perform well with data sets larger than roughly 1,000 records. Larger software projects would certainly be represented by a larger amount of metadata though, so this case study demonstrated that Exhibit would not scale in the general case.

One approach to ensuring that Exhibit acts on data sets of acceptable size is the "Inhibit" Javascript library discussed in Chapter V. Inhibit was developed by the author to provide an interactive means of refining a SPARQL query against the entire metadata set until a user-definable threshold number of records was returned. Inhibit "negotiates" with an Active PURL until a result set may be safely handled by Exhibit. Once an acceptable number of records is returned from a query, the negotiated data set must be made available to Exhibit. This may be done in one of two

ways: Either a unique URL is assigned to the request by an Active PURL and returned to the client in a Location header (an HTTP redirect) or the Active PURL may proxy the query results and return them directly. Active PURL implementation alternatives are discussed in detail in Chapter V. The prototype Active PURL used for the PURL project use case proxied query results to avoid cross-browser scripting limitations. The Inhibit library is provided in Appendix B, inclusive of inline documentation on its usage. Figure 8-3 shows a screenshot of the Inhibit user interface being used. In the figure, a SPARQL query is shown that returned an unacceptably high number of results. A message to the user below the query shows the current number of results and the acceptable threshold. The query is refined iteratively until acceptable and then a unique URL assigned to it. The URL representing an acceptable query includes the query in the Query String fields of the URL and is suitable for resolution via an Active PURL's proxy or HTTP redirection or by cutting and pasting directly into an Exhibit HTML page description.

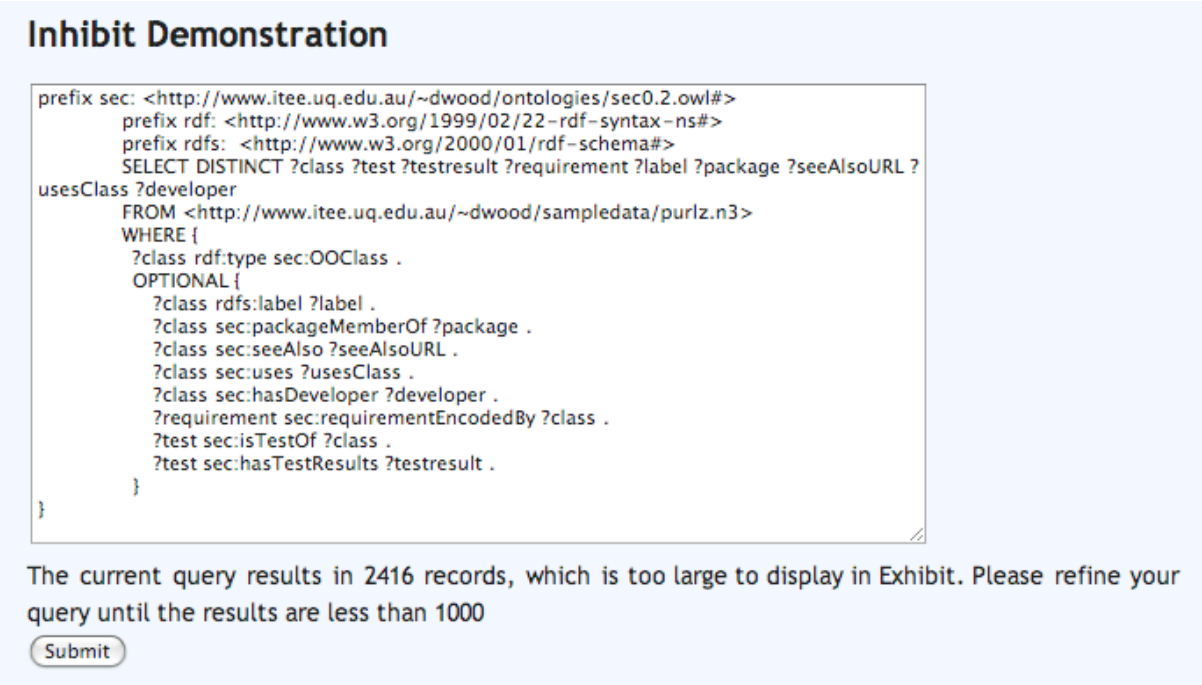


Figure 8-3. Resolving an Active PURL to Refine a SPARQL Query in Inhibit

Figure 8-4 shows a dynamically generated HTML user interface created using Exhibit. The RDF data set displayed in the figure is the result of a SPARQL query using Inhibit. The query results are a subset of the RDF metadata for the PURL project. The class org.purl.accessor.PURLsAccessor has been selected and is visible in the resulting dialog box. Navigational facets are shown on the left of the screenshot. Facets were defined for SEC type,

packageMemberOf, History and Developer. The SEC type includes OOPackage, OOClass, OOMethod, Requirement, Test, Metric and represents a means of narrowing a display set based on the type of component one is looking for. The packageMemberOf relationship is sufficient to narrow a display to the members of a particular software package. The Developer facet names the developers who worked on the project and allows the displayed set to be narrowed based on authorship.

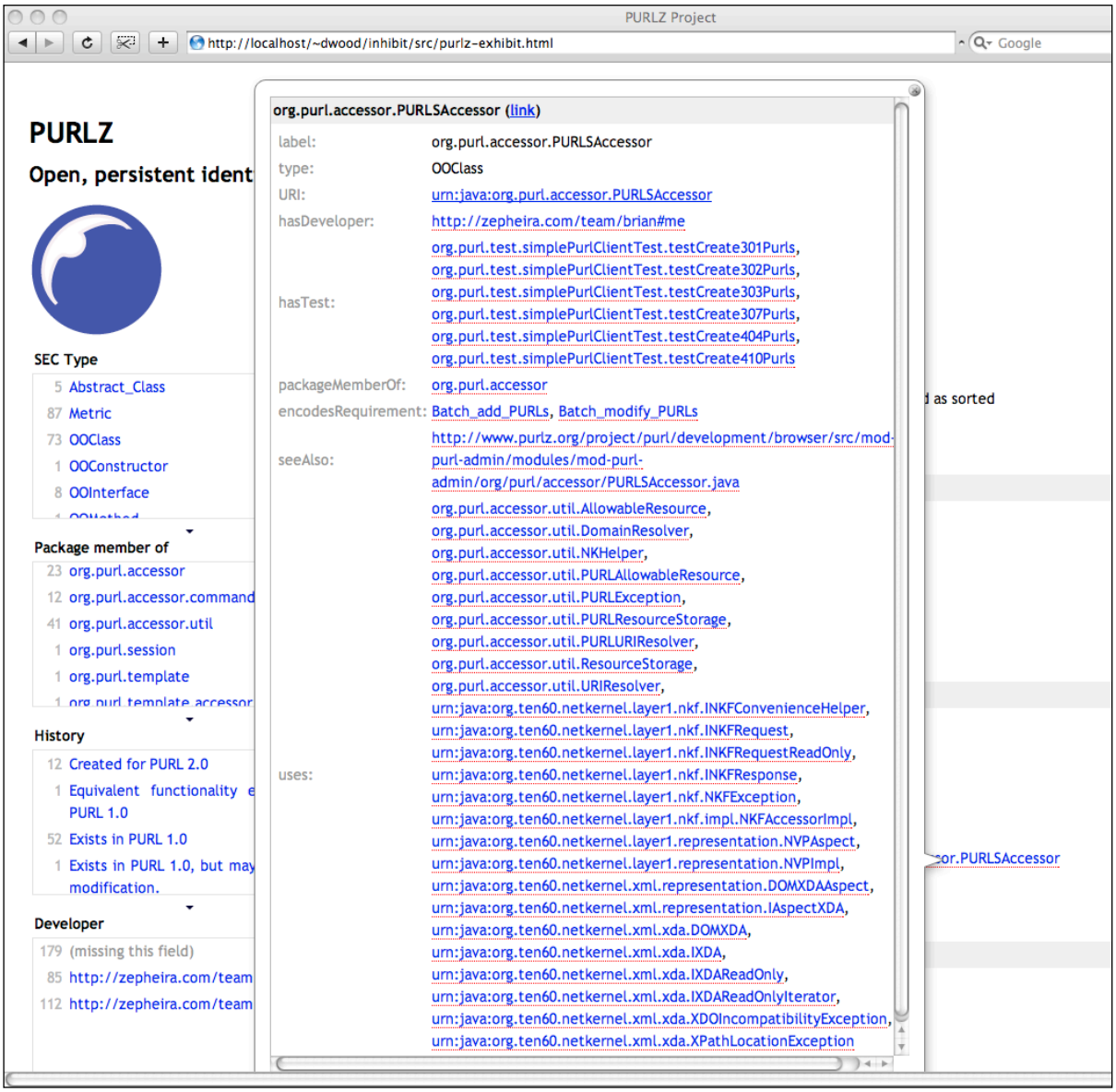


Figure 8-4. Faceted Navigation of PURL Project Metadata Using Exhibit/Inhibit

The History facet made use of a metadata element captured in the PURL requirements documentation; a notation was made to show whether a particular requirement was new to the

current version of the PURL project or had existed in the original PURL code base from OCLC. Use of the History facet clearly demonstrates the usefulness of a Semantic Web approach. There is no need to restrict one's view of a software project to a pre-defined schema (in this case, the SEC ontology). Other metadata elements may be defined internally or externally and made available for future analysis via RDF's general extension mechanisms.

Experience with Exhibit suggests that it is a very flexible (and therefore powerful) faceted navigation environment, especially since one may define one's own facets. Other facets may be defined than the ones used for the case studies. Facets are defined as markup language in an HTML page that calls the Exhibit JavaScript libraries. Exhibit's only significant problem is its lack of scalability. The layering of Inhibit over Exhibit served to reduce the impact of Exhibit's scalability limitations at the expense of forcing a user to know the SPARQL query language. Different Active PURL implementations (upon the same Active PURL infrastructure) could have been used to isolate the user from a query language at the expense of limiting flexibility of user interaction.

The metadata collected was too large to be viewed in RDF Gravity in its entirety. However, the prototype Active PURL was used to provide an interactive means of refining a SPARQL query until the records returned were sufficiently small to be directly viewable in RDF Gravity. A unique URL was assigned by the Active PURL negotiation described in Chapter V at the completion of the SPARQL query refinement, and that URL was then cut-and-pasted into RDF Gravity's user interface so the metadata could be loaded. The use of an Active PURL to provide a URL for use by a different application such as a "fat client" user interface such as RDF Gravity makes use of the first three of the four steps described for an Active PURL in Chapter V.

The Release activity was modeled by the inclusion of seeAlso references for each code component. seeAlso URLs provide hyperlinks to additional information regarding code components. In this case, the seeAlso URLs were links to each code component's entry in the PURL project's Subversion revision control system. They were calculated from a pattern used by the project's Subversion revision control system Web interface. The default, or top-level, URL was used for each code component. That is, no version numbers were appended. Version numbers may be readily appended during a Release activity in the case where a Subversion Web interface is used by adding a revision number to a URL Query String (e.g. ?rev=446). The feasibility of a monitor creating a release by assigning URLs to each code component in a release was thus shown.

The Requirement Interpretation, Testing and External Validation activities were adequately addressed in the JRDF prototype and were not revisited during the evaluation of the PURL

metadata.

8.4 Survey of Software Maintenance Information

A small-scale survey was conducted to determine whether the type of metadata collected during SWAMM is useful for practicing software maintainers. Sample metadata regarding a Java class in the PURL Project source code was collected and presented for review. Questions regarding the usefulness of the sample metadata were asked of survey respondents. The survey was cleared in accordance with the ethical review processes of the University of Queensland (clearance number ITEE 2008/02), and was conducted from 6-15 June, 2008, using a World Wide Web survey system (SurveyMonkey.com).

The hypotheses for the survey were:

1) The sample metadata would be useful for Java programmers performing maintenance activities.
2) Each type of metadata presented would be useful for the purposes of software maintenance.
3) The sample metadata presented would be more useful than Javadoc or similar documentation systems for the purposes of software maintenance.

The survey targeted programmers who perform maintenance activities on Java source code. Each respondent was asked to confirm that they programmed in the Java language and that they maintained Java source code.

Six individual maintainers of Java software known to the author were sent direct electronic mail messages inviting them to participate in the survey. An additional eleven Java team leaders were asked to invite participants from their teams. Further invitations were sent to two Java-related mailing lists (The XML Guild and the Mulgara developers lists). A total of twenty-seven people began the survey, but only twenty-three people completed it. One respondent stated that he did not maintain Java source code and his responses were removed from consideration. Twenty-two complete responses from Java maintainers were thus collected.

Sample metadata were gathered regarding a single Java class from the PURL Project (org.purl.accessor.PURLSAccessor) and presented for consideration to each respondent. Respondents were asked to study the sample metadata and answer questions regarding it. The survey questions are provided in Appendix C. The sample metadata is identical to that shown for the class in Figure 8-4, although the format was changed for the purposes of the survey. The

Exhibit-based interface shown in Figure 8-4 is rather verbose and suffers from cross-browser display differences. The information presented in the survey was formatted into a simple HTML table to ensure cross-browser consistency and to reduce the amount of explanation necessary for survey participants to understand the survey questions.

Respondents were asked to identify how useful a particular type of metadata was to them in the context of software maintenance. A scale of responses was used (e.g. Decline to answer, Unable to tell, Not valuable at all, Somewhat valuable, Very valuable, Essential). Five types of metadata were presented regarding the sample Java class:

- hasRequirements: Information regarding requirements encoded by the class.
- seeAlso: A hyperlink to the source code for the class and its history in a revision control system.
- uses: Other classes used by the subject class.
- hasTest: Information regarding unit tests that exercise the class.
- hasDeveloper: Information regarding the original author of the class.

For each type of metadata, a strong majority of respondents (86% or more) found the metadata to be useful (either Somewhat valuable, Very valuable, or Essential). The mean, median and mode of the responses relating to the hasRequirements and seeAlso metadata were higher (centered around "Very valuable") than those for uses, hasTest and hasDeveloper (centered around "Somewhat valuable").

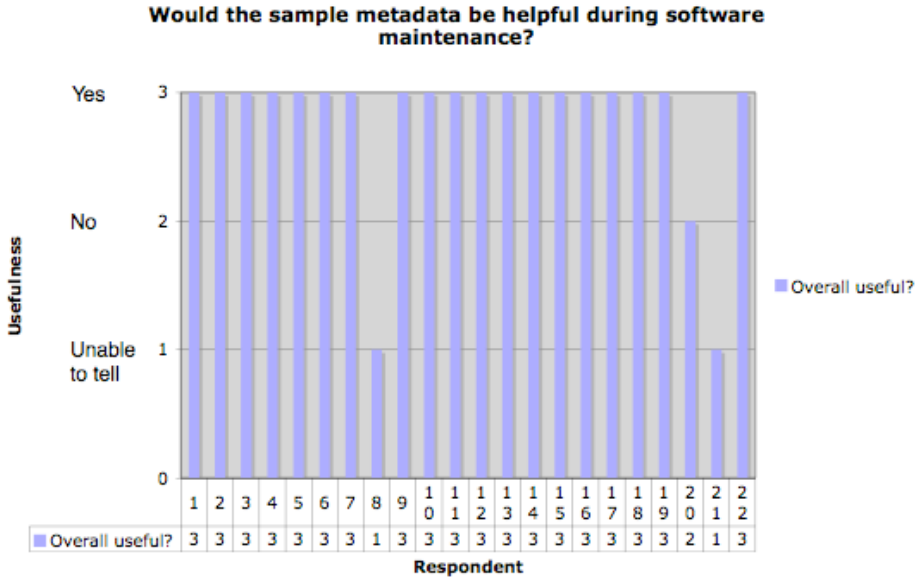
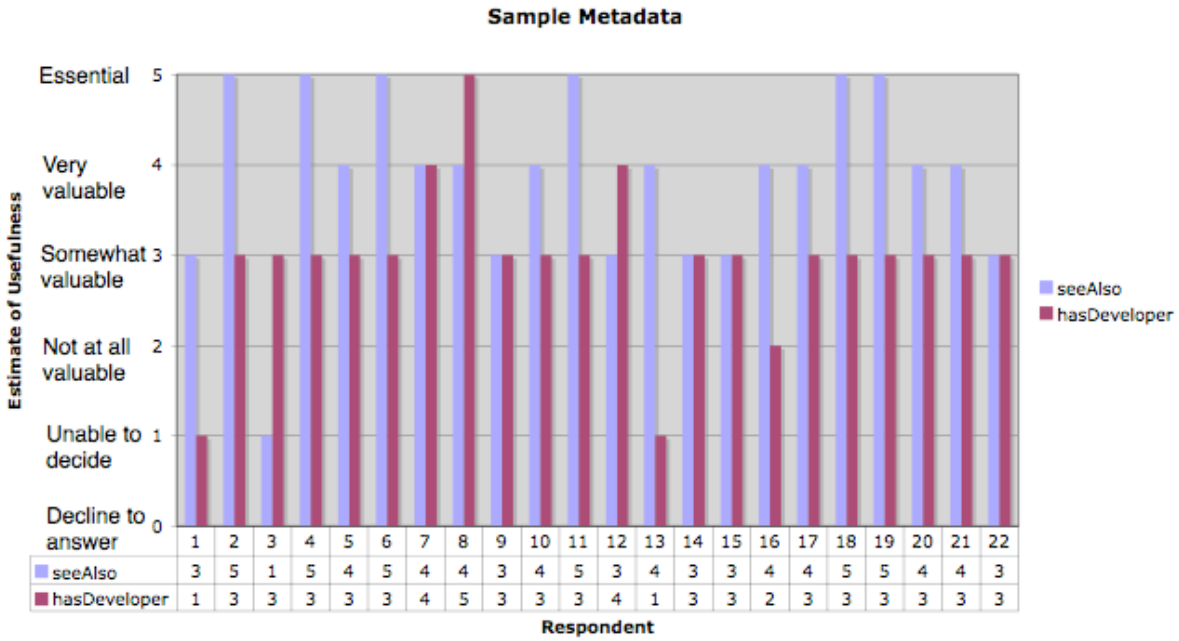


Figure 8-5. Overall Usefulness of the Sample Metadata

Respondents were asked two subjective questions relating to the overall utility of the sample metadata. They were asked whether the information presented was better, worse or about the same than information available in Javadoc or similar documentation systems (50% said better) and whether the information presented would be useful when performing maintenance on the example class (86% said yes). Figure 8-5 summarizes the responses to the latter question.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
seeAlso	3	5	1	5	4	5	4	4	3	4	5	3	4	3	3	4	4	5	5	4	4	3
hasDeveloper	1	3	3	3	3	3	4	5	3	3	3	4	1	3	3	2	3	3	3	3	3	3

Figure 8-6. Sample Metadata (Statistically Significant Portion)

Statistical tests were in agreement with hypothesis (1); A t-test (using α=0.05) shows that Java maintainers consider the sample metadata to be useful for the purposes of software maintenance with a probability $P \approx 0.05$. The small size of the study, however, was not sufficient to produce statistically significant results for the validation of hypotheses (2) or (3). Statistically significant data was collected for seeAlso and hasDeveloper metadata ($P \approx 0.01$ and 0.02, respectively), which tends to support hypothesis (2). Figure 8-6 summarizes the statistically significant results for the seeAlso and hasDeveloper metadata.

The population of interest includes some three and a half million Java maintainers, presuming the accuracy of Sun Microsystems' estimate released at JavaOne 2008 of seven million Java programmers, roughly half of whom may be expected to perform maintenance [Jones 2007]. The difference between the population of interest and the operationalized population was the difference

between those Java maintainers who are accessible by electronic mail and have Web access and those who don't. That difference may be reasonably expected to be small. The sampling error between the operationalized population and the identified sample was caused by any differences between the people invited and the operationalized population. This difference was minimized by inviting maintainers from several countries (Australia, Canada, France, The Netherlands, Russia, the United Kingdom, and the United States) and several types of organizations (Fortune 500 companies, non-profit corporations, privately held corporations, government and individual consultants). Finally, the achieved sample was analyzed on the basis of email address domains and country of residence in an attempt to identify demographic clues to bias. No such biases were identified.

The SWAMM methodology was exercised twice, once with a trivial amount of metadata representing a portion of a small software project and again with a moderate amount of metadata representing all the Java classes in a medium-sized software project. Both software projects modeled were real world, although the PURL project was modeled in substantially more detail. Several types of Semantic Web user interfaces were considered. Two faceted navigation interfaces (Exhibit and Longwell) and one graph viewer (RDF Gravity) were used. Exhibit and Longwell were shown to provide faceted navigation over software project metadata when provided RDF metadata extracted from existing software projects. The SWAMM methodology was discussed in the context of the prototypes and was shown to be feasible to apply to real-world data. Finally, metadata produced was evaluated for usefulness by a small-scale survey of practicing Java programmers. The survey suggested that metadata produced during SWAMM is useful to practitioners.

IX. Conclusions and Further Work

"An acquaintance wishes to know if I intend to go on writing in chunks for the rest of my life.
That is a matter I have not yet decided. I may, and I may not.
How on earth does one know whether or not one is going to go on writing in chunks?"
-- Will Cuppy (1884-1949)

9.1 Conclusions

Software maintenance is, often by far, the largest portion of the software lifecycle in terms of both cost and time. Yet, in spite of thirty years of study that successfully uncovered the mechanisms and attributes of maintenance activities, there exist a number of significant open problems in the field: Software still becomes unmaintainable with time.

Software maintenance failures result in significant economic costs because unmaintainable systems generally require wholesale replacement. Maintenance failures occur primarily because software systems and information about them diverge quickly in time. This divergence is typically a consequence of the loss of coupling between software components and system metadata and eventually results in an inability to understand or safely modify a system. Accurate documentation of software systems, long the proposed solution for maintenance problems, is rarely achieved and even more rarely maintained. Inaccurate documentation may exacerbate maintenance costs when it misdirects the understanding of maintainers.

At least three computing pioneers have attempted to define and collect "laws" of software: Brooks, Lehman and Glass. It is instructive to review them briefly to determine whether the claims of this thesis violate any of those laws.

Brooks defined four "inherent properties" of software: complexity, conformity, changeability and invisibility [Brooks 1987]. Software is complex because programmers continue to define and use new levels of abstraction. This is the reverse procedure from the physical sciences, where practitioners have spent generations by doing the exact opposite. Software becomes more complex as it is forced to conform to other systems, some of which are arbitrary (and arbitrarily complex)

such as human social systems. Software evolves, as we have seen, and so changes throughout its life cycle. Finally, the geometric abstractions of software are mostly difficult or impossible to visualize. They are, to use Brooks' term, invisible.

Lehman's laws are well known [Lehman 1997]. There are eight laws, but arguably the most important echo Brooks: The Law of continuing change notes that a software project evolves continuously (Brooks' changeability) and the Law of Increasing Complexity suggests that a software project becomes less structured, or more complex, with time (Brooks' complexity and conformity). One may also consider Lehman's Law of Declining Quality, which states that a project's quality decreases during its lifecycle, although that would appear to be a consequence of increasing complexity.

Glass collected fifty-five "facts" of software engineering that he summarized into four themes: complexity, poor estimation coupled with schedule pressure, a disconnect between software managers and technologists, and the delusion of hype. Glass goes on to say, "I would suggest that practitioners considering some tool, technique, method or methodology that is at odds with one or more of these facts should beware of serious pitfalls in what they are about to embark on." [Glass 2003, pp 187]

I contend that the central ideas of this thesis are in conformance with Brooks' inherent properties, Lehman's Laws and Glass's Facts. The new version of Persistent Uniform Resource Locators (PURLs), the SEC ontology of software engineering concepts and the Software Agent Maintenance Methodology (SWAMM) do not remove complexity; they are merely tools for better managing the complexities inherent in software projects. They are not a "silver bullet" (as Brooks put it) to "solve" the woes of software developers, nor a means to remove schedule pressure or bad management. Indeed, the curation of URL resolution and the application of SWAMM may actually increase schedule pressures due to their slight overhead. However, PURLs, the SEC ontology and the SWAMM methodology are tools that provide information and guidance. Good teams can make use of that information and guidance to ease social problems (Glass's disconnect) and hopefully soften the impact of Lehman's Law of Declining Quality.

This thesis described an approach for increasing the maintainability of software systems via the application and maintenance of structured metadata at the source code and project description levels. The application of specific metadata approaches to software maintenance issues was suggested for the reduction of economic costs, the facilitation of defect reduction and the assistance of developers in finding code-level relationships. The vast majority of system metadata (such as

that describing code structure, encoded relationships, metrics and tests) required for maintainability was shown to be capable of automatic generation from existing sources, thus reducing needs for human input and likelihood of inaccuracy. Suggestions were made for dealing with metadata relating to human intention, such as requirements, in a structured and consistent manner.

The history of metadata was traced to illustrate the way metadata has been applied to physical and virtual resources, including software. Best practice approaches for applying metadata to virtual resources were discussed and compared to the ways in which metadata has historically been applied to software. This historical analysis was used to explain and justify a specific methodological approach and place it in context. Specifically, the inability of Callimachus of Cyrenae to categorize his own works, called Callimachus' Quandary, was shown to be a direct result of Aristotle's insistence on hierarchies for the management of information. The generalized resolution of Callimachus' Quandary required both the expressivity of a graph data structure and the affordable computational power to implement and query it. The Resource Description Framework (RDF) and associated Semantic Web techniques were discussed as international standards for the resolution of Callimachus' Quandary.

Three historical fallacies were suggested: The fallacy of perfect knowledge, the fallacy of the big round ball and the fallacy of perfect execution (in Chapter II). The admission of all three fallacies is encouraged in order to adopt appropriate tools for the management of software's inherent complexities. PURLs, the SEC ontology and the SWAMM methodology are tools for addressing software systems that continuously evolve, whose requirements are not static and which were imperfectly executed during their initial development.

The SWAMM methodology was proposed for capturing, describing and using system metadata, coupling it with information regarding software components, relating it to the SEC ontology of software engineering concepts and maintaining it over time. Unlike some previous attempts to address the loss of coupling between software systems and their metadata, the described methodology is based on standard data representations and may be applied to existing software systems.

The collection of metadata-based descriptions of software projects has been tried before. Barriers to adoption of such techniques include, importantly, schedule pressure and unwillingness to perform work that leads to benefits only at some later time. Such benefits are often unspecified or irrelevant to the provider. The approach taken in this thesis differs in that the SWAMM methodology may be applied piecemeal and at any time during software development or

maintenance that the benefits are clearly required.

The methodology is supportive of distributed development teams and the use of third-party components by its grounding of terms and mechanisms in the World Wide Web. Scaling the methodology to the size of the Web required mechanisms to allow the Web's URL resolution process to be curated by metadata maintainers. Extensions to the Persistent URL concept were made to allow for curation of URL resolutions as part of this thesis.

Relational navigation of software systems via technical and sociological metadata was used to facilitate understanding. Understanding, in turn, is known to forestall maintenance failure. The efficacy of this approach was demonstrated via a modeling of the methodology and two case studies. A brief survey of Semantic Web user interfaces was provided that demonstrated user interface properties suitable for projects of the size and complexity of SWAMM, with extensions for scalability.

The approaches taken in the SWAMM methodology are in conformance with the eight "good practice" recommendations of the latest draft finding of the World Wide Web Consortium's Technical Architecture Group (TAG) [Mendelsohn 2008].

The following contributions to the field of Software Engineering have been made as part of this thesis:

- A general classification for applications of metadata to resources, inclusive of trust relationships, on the World Wide Web and a theory of the application of metadata to software source code supported by that classification.
- Extensions to the Persistent URL concept to facilitate third party curation of metadata describing Web resources. These extensions included means to computationally disambiguate virtual, physical and conceptual resources identified by HTTP URLs.
- Definition and analysis of an ontology of software engineering concepts suitable for formally describing object-oriented software systems and their properties that relate to software maintenance activities.
- Description of a methodology for distributed software maintenance (the SWAMM methodology) using a RESTful architectural style and structured Semantic Web-based metadata descriptions of software, system and requirements components.
- Validation of the SWAMM methodology's viability via two enactments of the methodology, a model of the methodology *in silico* and a survey of the use of the

methodology by software maintenance practitioners.

9.2 Further Work

Traditional software engineering analyses, tools and models presume a closed system; all software developed during a project, and the hardware upon which the software relies, is treated as existing within the control of the project. The closed system presumption was often valid when Brooks developed the OS/360 operating system, but is generally less true during multi-year maintenance of modern software. Use of third party libraries, some of which may be Free/Libre/Open Source software developed by distributed communities and the increasing use of distributed software developers and maintainers suggest that the modern software engineering environment is no longer a closed system. The development of tools and techniques to conduct software engineering in open and distributed environments is needed.

Mechanisms for representing and using metadata on the World Wide Web are rapidly changing. Any new mechanism has the potential to have an impact on software maintenance activities, especially if it becomes widely fielded. The recognition and indexing of metadata by Web search engines, for example, has the potential to change the way software developers find code components that require modification. Further work to utilize metadata search capabilities of Web search engines may yield important results.

Web resources are controlled by their (distributed) publishers. Distributed publishing is one potential problem with a Web-based software development environment. A code module might disappear from the Web without warning, especially if such a module was published by a third party, and was not under the control of a software project that relied upon it. 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. PURLs provide a technical mechanism of control for such problems by allowing a curator to modify the location of a target resource (e.g. to point to a local copy of a code module that was removed from the Web). However, further work is needed to determine the sociological controls necessary to facilitate software development and maintenance in a Web environment.

The applications of SWAMM presented in Chapter VIII of this thesis presume that SWAMM metadata is located in a centralized repository. Further work is needed to determine the impact upon SWAMM specifically and software maintenance in general of widely distributing code components and metadata on the World Wide Web. Truly distributed software projects, where each code component and associated metadata is published by its most recent modifier, have not

been tried, much less studied. If distributed code repositories are feasible, new social structures for the management of such projects would need to be developed.

Presuming that widely distributed publishing of code and system metadata is managerially feasible, new mechanisms would need to be developed to discover code components on the Web. Either external metadata, as in SWAMM, could be used or distributed developers might prefer to insert metadata directly into their code (internal metadata). For example, each code component could be represented as a separately addressable Web resource and semantic microformats and extraction techniques used to acquire the metadata (e.g. GRDDL or RDFa extractions from embedded XML or HTML blocks). Yahoo! Inc.'s forthcoming SearchMonkey Web search platform may provide a general solution to Web-based search of resources with descriptive metadata [Fox 2008]. Unfortunately, SearchMonkey developer tools have not been released to the public as of this writing.

Distributed publishing of system metadata would change the assumptions made in Chapter VII regarding the choice of user interface for SWAMM's State Determination activity. Smaller, discrete amounts of distributed metadata would suggest the use of a user interface capable of "following your nose" from link to link, such as Disco or Tabulator.

Distributed publishing of system metadata would also present challenges when tracking software components that are deleted or otherwise made obsolete. The choice of *external* metadata would lead to consistency issues whereby the metadata would need to be deleted (or marked as referring to an inactive component) at the time a component is deleted. The use of *attached/separable* metadata might ease consistency concerns, but consistency would remain a problem in the case where metadata relating to deleted components is retained. If retention of metadata is desired, as in the case where metadata might be used to track software component versions, additional metadata elements would be required to be added to the SEC ontology. One might add, for example, metadata elements for a version number, a date-time of deletion and an active status flag. Keeping multiple active versions of a component (e.g. for differentiating multiple versions of a complete software system) would require yet more metadata elements to relate component versions to versions of a software system.

Additional work to implement and extend the SEC ontology and the SWAMM methodology are necessary regardless of the publication style of system metadata. Tools are needed to ease difficulties in the creation, collection and storage of metadata. SEC and SWAMM need to support a wider range of languages and environments before they may be applied to most real-world

projects and SWAMM needs to be validated in production environments.

Chapter VI suggested that SWAMM metadata regarding requirements and unit tests could be collected via a plug-in to the popular Eclipse integrated development environment. Support for other development environments may also be desirable. Tools are also required for the continuous and automated extraction of software collaboration graph information from source code. Additional tools are needed to extract information on the state of tests and the values of metrics at any given time. All of these tools are required before SWAMM can be used in a production software maintenance effort.

The SEC ontology needs to be extended to support additional software engineering concerns (such as aspects, computing environments and extended definitions of requirements) and non-Java language constructs. The SEC ontology could be extended to incorporate the work done by Buckley et al. [2005] and Kitchenham et al. [1999].

The SWAMM methodology and the SEC ontology together describe software maintenance efforts using standard formats and vocabulary description techniques. Some simple extensions to link external data were shown, such as information describing development team members. Further work to integrate SWAMM/SEC information describing software projects to other linked metadata on the World Wide is suggested.

SWAMM should be extended as necessary to support any new tools that may provide useful metadata or metadata collection capabilities. Training materials for the use of SWAMM and the application of the SEC ontology will be necessary for those techniques to be applied to production environments.

SWAMM activities were enacted by query of (potentially distributed) metadata stores addressed by URLs; a "pull" information architecture. "Push" architectures, such as proactive notification of failing tests, invalidated requirements or out-of-scope metrics should also be explored. Existing Web mechanisms such as peer-to-peer systems and/or RSS/Atom news feeds may be used.

When tool support, necessary extensions and development of training materials are complete, application of the SEC ontology and the SWAMM methodology should be applied to real-world production software maintenance activities.

The Inhibit JavaScript library and the Active PURL concept also deserve attention past the scope of this thesis. The use of Inhibit and an Active PURL to narrow the size of a dataset capable of being displayed in Exhibit works around a scalability problem in Exhibit. Inhibit may eventually

be replaced by a project to allow Exhibit to spool large data sets on a server (code-named Backstage) [Huynh 2008]. It is not yet known whether Backstage will obviate the current need for Inhibit. If Inhibit remains useful, it should be extended to more generically interoperate with Active PURL implementations. Possibilities for extensions include recognition of browser environments to optimize the amount of calculation performed within a Web client, validation of returned content from an Active PURL, better error handling and extended support for user interfaces, including cascading style sheets and data types.

Active PURLs may, in concept, provide metadata for any type of Web service. The use of Active PURLs may far surpass their use with Inhibit. Development of Active PURLs that support generic queries to relational databases (via the Structured Query Language, SQL) and RDF data sources (via SPARQL) are underway as of this writing.

Extensions to Active PURLs could include the definition and development of a XML vocabulary to define query parameters that is independent of the user interface (to replace the HTML fragments that are currently returned). Such a vocabulary should include simple component definitions that may be mapped directly to common user interface widgets (e.g. text areas, radio buttons, pull-down menus) as well as compound component definitions that map to multiple widgets (such as dates, date-times or hyperlinks). Active PURLs could then optionally return XML for processing by a client or HTML fragments for those clients incapable of client-side processing of XML. Choice of content type to be returned could be determined by HTTP content negotiation or by explicit user interface selection based upon the HTTP User-Agent header. A non-HTML vocabulary for query parameters would allow Active PURLs to support a broader range of clients, including those that may not be Web browsers.

The Active PURL concept should be evaluated to determine whether it provides significant advantages over other mechanisms for the provision of metadata describing Web services, such as Universal Description, Discovery, and Integration (UDDI) [Clement 2004] and the Web Ontology Language for Services (OWL-S) [Martin 2004].

References

1060 Research Limited (2004). NetKernel Open Source Community. Retrieved July 29, 2007 from http://www.1060.org/.

Adida, B., Birbeck, M., McCarron, S. and Pemberton, S. (2007, October 18). RDFa in XHTML: Syntax and Processing, W3C Working Draft, Retrieved February 17, 2008 from http://www.w3.org/TR/rdfa-syntax/.

Akscyn, R., McCracken, D. and Yoder, E. (November 1987). KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations. Proc. HYPERTEXT '87, pp. 1-20.

Alesso, H. P. and Smith, C.F. (2006). Thinking on the Web: Berners-Lee, Gödel, and Turing, Wiley, Hoboken, New Jersey.

Ankolekar, A. (2004). Towards a Semantic Web of Community, Content and Interactions, Doctoral Consortium, Computer-Supported Coop. Work (CSCW), Chicago.

Antoniol, G., Canfora, G., Casazza, G., de Lucia, A., Merlo, E. (2002). Recovering Traceability Links Between Code and Documentation. Proc. IEEE Transactions on Software Engineering, 28 (10), pp. 970-983.

Apple, Inc. (2007, September 13). Safari FAQ, Apple Developer Connection, Question 21. Does Safari support XSLT? Retrieved May 26, 2008 from http://developer.apple.com/internet/safari/faq.html#anchor21

Archer, P. (ed.) (2007, October 31). POWDER: Use Cases and Requirements, W3C Working Group Note. Retrieved March 13, 2008 from http://www.w3.org/TR/2007/NOTE-powder-use-cases-20071031/.

Basili, V., and Mills, H. (1982). Understanding and Documenting Programs. IEEE Transactions on Software Engineering SE-8, 3, pp. 270-283.

Beck, K. (2000). Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading, Massachusetts.

Beck, K., Andres, C. (2005). Extreme Programming Explained: Embrace Change, 2nd ed. Addison-Wesley Professional, New York.

Beckett, D. (2006). Redland RDF Libraries, Retrieved July 29, 2007 from http://librdf.org/.

Berners-Lee, T. (1989). Information Management: A Proposal, Retrieved February 17, 2008 from

http://www.w3.org/History/1989/proposal.html.

Berners-Lee, T. and Connolly, D. (1995, November). Hypertext Markup Language - 2.0, Internet Engineering Task Force, Network Working Group, RFC 1866. Retrieved February 8, 2008 from http://www.ietf.org/rfc/rfc1866.txt.

Berners-Lee, T., Fielding, R. and Frystyk, H. (1996, May). Hypertext Transfer Protocol -- HTTP/1.0, Internet Engineering Task Force, Network Working Group, RFC 1945. Retrieved February 8, 2008 from http://www.w3.org/Protocols/rfc1945/rfc1945.

Berners-Lee, T. (1997, January 6). Metadata Architecture. Retrieved February 20, 2008 from http://www.w3.org/DesignIssues/Metadata.html.

Berners-Lee, T. (1998, September). Semantic Web Road Map. Retrieved February 20, 2008 from http://www.w3.org/DesignIssues/Semantic.html.

Berners-Lee, T. (2000). Weaving the Web. The Original Design and Ultimate Destiny of the World Wide Web. Harper Business, New York, pp. 4.

Berners-Lee, T., Hendler, J., Lassila, O. (2001, Dec 31). The Semantic Web. Scientific American. 2001 (5):28–37. Retrieved March 24, 2008 from http://www.sciam.com/2001/0501issue/0501berners-lee.html.

Berners-Lee, T., Fielding, R. and Masinter, L. (2005, January). Uniform Resource Identifier (URI): Generic Syntax, Internet Engineering Task Force, Network Working Group, RFC 3986 (STD 66), Section 1.2.3. Retrieved March 1, 2008 from http://tools.ietf.org/html/rfc3986#section-1.2.3.

Berners-Lee, T., Hollenbach, J., Lu, K., Presbrey, J., Pru d'ommeaux, E. and Schraefel, M.C. (2007, November 2). Tabulator Redux: Writing Into the Semantic Web, University of Southampton Technical Report 14773. Retrieved May 31, 2008 from http://eprints.ecs.soton.ac.uk/14773/.

Bizer, C. and Gauß, T. (2007). Disco - Hyperdata Browser. Retrieved May 31, 2008 from http://www4.wiwiss.fu-berlin.de/bizer/ng4j/disco/.

Boehm, B. (1973). Software and its Impact: A Quantitative Assessment. Datamation, 19, pp. 48-59.

Boehm, B. (1981). Software Engineering Economics. Prentice-Hall, Englewood, NJ.

Bontas, E., Mochol, M. and Tolksdorf, R. (2005). Case Studies on Ontology Reuse, Proc. 5th International Conference on Knowledge Management (I-Know 2005).

Booth, D. and Liu C.K. (eds.) (2007, June 26). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer, Section 5.3, W3C Recommendation. Retrieved May 12, 2008 from http://www.w3.org/TR/wsdl20-primer/#adv-service-references.

Bos, B. (1999, June 30). Handling of Fragment Identifiers in Redirected URLs, Internet Engineering Task Force, Expired Internet Draft. Retrieved March 1, 2008 from http://www.w3.org/Protocols/HTTP/Fragment/draft-bos-http-redirect-00.txt.

Bradner, S. (1997, March). Key Words for Use in RFCs to Indicate Requirement Levels, Internet Engineering Task Force Best Current Practice (BCP) 14, RFC 2119. Retrieved March 1, 2008 from http://www.ietf.org/rfc/rfc2119.txt.

Brand, A. and Daly, F. (2003). Metadata Demystified: A Guide for Publishers, Sheridan Press and NISO Press. Retrieved June 10, 2007 from http://www.niso.org/standards/resources/Metadata_Demystified.pdf.

Brickley, D. (2001). Metadata at the W3C. Retrieved June 6, 2007, from http://www.w3.org/Metadata/.

Brickley, D. and Guha, R.V. (eds.) (2004). RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation. Retrieved February 19, 2008 from http://www.w3.org/TR/rdf-schema/.

Brooks, F.P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering, Computer, IEEE Computer Society Press, 20 (4), pp. 10-19.

Brooks, F.P. (1995). The Mythical Man Month, Anniversary edition. Addison Wesley.

Brown, P.J. (1980). Why Does Software Die? in Parikh, G. and Zveginitzov, N., eds. (1983). Tutorial on Software Maintenance, IEEE Computer Society, Silver Spring, MD, pp. 279-286.

Buckley, J., Mens, T., Zenger, M., Rashid, A. and Kniesel, G. (2005). Towards a Taxonomy of Software Change, Journal of Software Maintenance: Research and Practice, Wiley, 17, (5), pp. 309-332.

Bulterman, D.C.A. (2004). "Is It Time for a Moratorium on Metadata?," IEEE MultiMedia, 11(4), pp. 10-17.

Bush, V. (1945, July). As We May Think, *The Atlantic Monthly*, pp.101-108.

Butler, M., Huynh, D., Hyde, B., Lee, R., Mazzocchi, S. (2006) Longwell Project Page. Retrieved March 3, 2008 from http://simile.mit.edu/wiki/Longwell.

Campbell, J. (1991a). The Masks of God : Primitive Mythology, Arkana edition, Penguin Books, New York.

Campbell, J. (1991b). The Masks of God : Oriental Mythology, Arkana edition, Penguin Books, New York.

Caplan, P. (2003). Metadata Fundamentals for All Librarians, American Library Association, Chicago, pp. 3-5.

Casson, L. (2001). Libraries in the Ancient World, Yale University Press, New Haven & London.

Clark, T., Martin, S. and Liefeld, T. (2004, March). Globally Distributed Object Identification for Biological Knowledgebases, Brief Bioinform, 5 (1), pp. 59-70.

Clement, L., Hately, A., von Riegen, C. and Rogers, T. (eds.) (2004, October 19). UDDI Version 3.0.2, UDDI Spec Technical Committee Draft, OASIS. Retrieved June 2, 2008 from http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

Collins, A.M. and Loftus, E.F. (1975, November). A Spreading-Activation Theory of Semantic Processing. Psychological Review, 82, pp. 407-428.

Communications Decency Act of 1996 (1996) Section 5 of Telecommunications Act of 1996, 104th Congress of the United States of America, Retrieved February 17, 2008 from http://www.fcc.gov/Reports/tcom1996.txt.

Connolly, D. (ed.) (2007, September 11). Gleaning Resource Descriptions from Dialects of Languages (GRDDL), W3C Recommendation, Retrieved February 17, 2008 from http://www.w3.org/TR/grddl/.

Cook, S., Harrison, R., Lehman, M.M. and Wernick, P. (2006). Evolution in Software Systems: Foundations of the SPE Classification Scheme, Software Maintenance and Evolution Research and Practice, 18 (1), pp. 1-35.

Copeland, L. (2001, December 3). Extreme Programming. Computerworld. Retrieved April 18, 2008 from http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html

Coyle, A. (2002). 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.

Cutter, C.A. and Cutter, W.P., ed. (1904). Rules for a Dictionary Catalog. 4th Edition, Government Printing Office, Washington, D.C. Retrieved November 7, 2008 from http://books.google.com/books?hl=en&id=WPMRAAAAYAAJ&dq=Rules+for+a+Dictionary+Catalog&printsec=frontcover&source=web&ots=VUDWG_RMe4&sig=r2DzMQNbGel2PQWCROaypKmF-kg&sa=X&oi=book_result&resnum=1&ct=result#PPA1,M1

Date, C.J. (2004). An Introduction to Database Systems, 8th Edition, Pearson, Boston.

Davies, P. and Gribbon, J. (1992). The Matter Myth, Penguin, New York.

Dawkins, R. (1989). The Selfish Gene, 1989 edition, Oxford University Press, Oxford, pp. 192.

Day, M., ed. (2007). Metadata, UKOLN. Retrieved June 10, 2007 from http://www.ukoln.ac.uk/metadata/.

Devanbu, P., Brachman, R.J., Selfridge, P.G. and Ballard, B.W. (1991). "LaSSIE: a Knowledge-based Software Information System", Comm. of the ACM, 34(5), pp. 36–49.

Diamond, J. (1998). Guns, Germs and Steel, A Short History of Everybody for the Last 13,000 Years, Random House, Sydney.

Dickerson, R., Nehaniv, C. and Wernick, P. (2006). Plus ça Change… But What Changes and What Stays the Same?, Proc. 2nd Int'l IEEE Workshop on Software Evolvability, Philadelphia, pp. 11-15.

Dietrich, J., and Elgar, C. (Nov 2005). Towards a Web of Patterns, Proc. 4th International Semantic Web Conference, Galway, Ireland.

Doctorow, C. (2001, August 26). Metacrap: Putting the torch to seven straw-men of the meta-utopia. Retrieved June 12, 2007 from http://www.well.com/~doctorow/metacrap.htm.

Dubost, K., Haas, H. and Jacobs, I. (2001, February 6). Common User Agent Problems, W3C Note, Section 4. Retrieved March 1, 2008 from http://www.w3.org/TR/2001/NOTE-cuap-20010206#uri.

Dunbar, B. (2007, April 13). Report Reveals Likely Causes of Mars Spacecraft Loss. NASA Press Release 2007-040. Retrieved July 16, 2007 from http://www.nasa.gov/mission_pages/mgs/mgs-20070413.html

Dunbar, R.I.M. (1993). Coevolution of Neocortical Size, Group Size and Language in Humans. Behavioral and Brain Sciences 16 (4), pp. 681-735.

Durrell, W.R. (1985). Data Administration: A Practical Guide to Data Administration, McGraw-Hill.

Eclipse Foundation (2007). Eclipse – An Open Development Platform, Retrieved July 29, 2007 from http://www.eclipse.org/.

Farb, P. (1969). Man's Rise to Civilization as Shown by the Indians of North America from Primeval Times to the Coming of the Industrial State, Avon, New York, pp. 259-264.

Federal Geographic Data Committee (FDGC) (1998). The Value of Metadata; National Spatial Data Infrastructure, U.S. Geological Survey.

Fielding, R.T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1, Internet Engineering Task Force, Network Working Group, RFC 2068. Retrieved February 19, 2008 from http://www.w3.org/Protocols/rfc2616/rfc2616.html.

Fielding, R.T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.

Fielding, R.T. and Jacobs, I. (2006, April 12). Authoritative Metadata, W3C TAG Finding. Retrieved March 6, 2008 from http://www.w3.org/2001/tag/doc/mime-respect-20060412.

Fox, V. (2008, April 24). Yahoo! Launches SearchMonkey Developer Tool in Limited Preview, Search Engine Land article. Retrieved April 26, 2008 from http://searchengineland.com/080424-113600.php.

Furrie, B. and The Follett Software Company (2003). Understanding MARC Bibliographic: Machine Readable Cataloging, Seventh Edition, Cataloging Distribution Service, Library of Congress. Retrieved June 12, 2007 from http://www.loc.gov/marc/umb/.

Gamble, C. (1999). The Paleolithic Societies of Europe, Cambridge University Press, Cambridge.

Gantz, J.F., Chute, C., Manfrediz, A., Minton, S., Reinsel, D., Schlichting, W. and Toncheva, A. (March 2008). The Diverse and Exploding Digital Universe, IDC, Framingham, MA. Retrieved November 7, 2008 from http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf.

Gates, J.K. (1968). Introduction to Librarianship, McGraw-Hill.

Gearon, P. and Muys, A. (2006). Mulgara Semantic Store, http://mulgara.org/.

Ghemawat, S., Gobioff, H., and Leung, S. (2003, Dec). The Google File System, SIGOPS Oper. Syst. Rev. 37, 5, pp. 29-43.

Gill, T. (1998). Metadata and the World Wide Web in Introduction to Baca, M. (ed). Metadata; Pathways to Digital Information, Getty Information Institute, pp. 11.

Gilliland-Swetland, A.J. (1998). Defining Metadata, in Baca, M., ed., Introduction to Metadata, Pathways to Digital Information, Getty Information Institute.

Glass, R. L. (2003). Facts and Fallacies of Software Engineering. Pearson Education, Boston, MA.

Glass, R.L. (2006). Software Maintenance is a Solution, Not a Problem. IT Metrics and Productivity Institute, Retrieved July 17, 2007 from

http://www.compaid.com/caiInternet/ezine/maintenance-as-solution.pdf.

Godfrey, M. and Qiang Tu (2000). Evolution in open source software: a case study, Proc. Intl Conf on Software Maintenance, pp. 131-142.

Golbeck, J. (2005). Computing and Applying Trust in Web-based Social Networks, Ph.D. dissertation, Department of Computer Science, University of Maryland College Park, pp. 30-34. Retrieved February 17, 2008 from http://trust.mindswap.org/papers/GolbeckDissertation.pdf.

Golbeck, J. (2006). Trust and Nuanced Profile Similarity in Online Social Networks, MINDSWAP Technical Report TR-MS1284, pp. 1. Retrieved February 2, 2008 from http://trust.mindswap.org/papers/trustStudy.pdf.

Goth, G. (2007, July-Aug). Will the Semantic Web Quietly Revolutionalize Software Engineering?, IEEE Software, 24, (4), pp. 100-105.

Goyal, S. and Westenthaler, R. (2004). RDF Gravity (RDF Graph Visualization Tool). Salzburg, Austria. Retrieved April 13, 2008 from http://semweb.salzburgresearch.at/apps/rdf-gravity/index.html.

Guha, R.V (1995, February 10). Contexts: A Formalization and Some Applications, Ph.D. thesis, Department of Computer Science, Stanford Univesity.

Guha, R.V. and Bray, T. (1997, June). Meta Content Framework Using XML, W3C Technical Note, Retrieved February 17, 2008 from http://www.w3.org/TR/NOTE-MCF-XML/.

Guha, R.V. (1999, March 15). RSS 0.90 Specification, Netscape Communications Corporation, Retrieved February 17, 2008 from http://www.rssboard.org/rss-0-9-0.

Guthrie, R.D. (2005). The Nature of Paleolithic Art, The University of Chicago Press, Chicago & London.

Halasz, F.G., Moran, T.P. and Trigg, R.H. (May 1987). Notecards in a nutshell, ACM SIGCHI Bull., 17, SI, pp. 45-52.

Han, J. (1994). Software documents, their relationships and properties, Proc. Asia-Pacific Software Eng. Conf., Tokyo, Japan, Dec. 1994, pp. 102-111.

Happel, H.-J., Korthaus, A., Seedorf, S., Tomczyk, P. (July 2006). KOntoR: An Ontology-enabled Approach to Software Reuse, Proc. 18th Int. Conf. on Software Eng. and Knowledge Eng.(SEKE), San Francisco, CA.

Harris, M.H. (1995). History of Libraries in the Western World, 4th Edition, The Scarecrow Press, Metuchen, NJ & London.

Hay, D.C. (2006). Data Model Patterns: A Metadata Map, Morgan Kaufman.

Heittman, R. (2008, February 18). Re: HTML with REST, electronic mail message to mailto:discuss@restlet.tigris.org. Retrieved February 20, 2008 from http://comments.gmane.org/gmane.comp.java.restlet/4203.

Hendler, J. (2006, Dec 14). The Dark Side of the Semantic Web, Mindswap Weblog. Retrieved March 19, 2008 from http://www.mindswap.org/blog/2006/12/13/the-dark-side-of-the-semantic-web/.

Holt, P.O. (1993). 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, pp. 1/1-1/3.

Holzmann, G.J. (2003, September). Trends in Software Verification. Proc. Formal Methods Europe Conference, Pisa, Italy. Retrieved July 21, 2007 from http://spinroot.com/gerard/pdf/fme03.pdf.

Huynh, D.F. (2007, August). User Interfaces Supporting Casual Data-Centric Interactions on the Web, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, MIT.

Huynh, D.F. (2008, February 7). Scaling Up Exhibit – An Early Experiment. Electronic mail message to the Simile Project General Mailing List. Retrieved June 1, 2008 from http://www.mail-archive.com/general@simile.mit.edu/msg03119.html.

Hyland-Wood, D., Carrington, D. and Kaplan, S. (2006a). Enhancing Software Maintenance by using Semantic Web Techniques, International Semantic Web Conference (ISWC), poster.

Hyland-Wood, D., Carrington, D. and Kaplan, S. (2006b) Toward a Software Maintenance Methodology using Semantic Web Techniques, Proc. of 2nd Intl IEEE Workshop on Software Evolvability, pp. 23-30.

Hyland-Wood, D. (2007, November 11). The Poor State of SPARQL Implementations. Retrieved November 11, 2007 from http://prototypo.blogspot.com/2007/11/poor-state-of-sparql-implementations.html.

Hyland-Wood, D., Carrington, D. and Kaplan, S. (2008a). Toward a Software Maintenance Methodology using Semantic Web Techniques and Paradigmatic Documentation Modeling in Harrison, R. (ed). IET Software Special Issue on Software Evolability. To appear.

Hyland-Wood, D. (2008b, March 20). Examples of Use for an Ontology of Software Engineering Concepts, version 0.2. Retrieved April 20, 2008 from http://www.itee.uq.edu.au/~dwood/sampledata/sec-example2.owl and http://www.itee.uq.edu.au/~dwood/sampledata/sec-testresults2.owl.

Hyland-Wood, D. (2008c, March 23). An OWL-DL Ontology of Software Engineering Concepts,

version 0.2. Retrieved May 29, 2008 from http://www.itee.uq.edu.au/~dwood/ontologies/sec0.2.owl and http://www.itee.uq.edu.au/~dwood/ontologies/reqs.owl.

Hyland-Wood, D. (2008d). Inhibit: A Javascript Library to Negotiate Exhibit Data Sets, http://www.itee.uq.edu.au/~dwood/publicweb/js/inhibit.js.

Hyland-Wood, D., Sletten, B. and Miller, E. (2008e). PURLZ: Open, persistent identifiers for managing Web resources. Retrieved March 25, 2008 from http://purlz.org/.

IEEE Standard Glossary of Software Engineering Terminology (1990). Institute of Electrical and Electronic Engineers, New York, NY.

Jackson, M. (1983). System Development, Prentice-Hall.

Jackson, S.L. (1974). Libraries and Librarianship in the West: A Brief History, McGraw Hill.

Jacobs, I. and Walsh, N. (eds.) (2004). Architecture of the World Wide Web, Volume One, W3C Recommendation, Retrieved June 7, 2007, from http://www.w3.org/TR/2004/REC-webarch-20041215/.

Japan Electronics And Information Technology Industries Association (JEITA) (2002, April). JEITA CP-3451: Exchangeable Image File Format for Digital Still Cameras: Exif, Version 2.2, retrieved February 16, 2008 from http://www.digicamsoft.com/exif22/exif22/html/exif22_1.htm

Jarrott, D. and MacDonald, A. (2003). Developing relational navigation to effectively understand software, Proc. 10th Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand, Dec. 2003, pp. 144-153.

Johnson, B.K. (2007). Wikipedia As Collective Action: Personal Incentives And Enabling Structures, Masters Thesis, Michigan State University. Retrieved Nov 9, 2007 from http://www.msu.edu/~john2429/Wikipedia%20as%20Collective%20Action.pdf.

Jones, C. (1995). Patterns of Software System Failure and Success. International Thomson Computer Press, Boston, MA.

Jones, C. (2006, February 14). The Economics of Software Maintenance in the Twenty First Century, Version 3. The IT Metrics and Productivity Institute. Retrieved July 18, 2007 from http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf.

Jones, C. (2007, March 19). Geriatric Issues of Aging Software. The IT Metrics and Productivity Institute. Retrieved July 18, 2007 from http://www.compaid.com/caiinternet/ezine/capers-maintenance.pdf.

Kalyanpur, A., Sirin, E., Parsia, B. and Hendler, J. (2004). Hypermedia Inspired Ontology Engineering Environment: SWOOP, Third International Semantic Web Conference (ISWC), Hiroshima, Japan.

Karger, D., Bakshi, K., Huynh, D., Quan, D. and Sinha, V. (2005). Haystack: A General Purpose Information Management Tool for End Users of Semistructured Data. Proc. 2nd Biennial CIDR, Asilomar, CA, pp. 13-27.

Kitchenham, B.A., Trarassos, G.H., Mayrhauser, A.V., Niessink, F., Schneidewind, N.F., Singer, J., Takada, S., Vehvilainen, R. and Yang, H. (1999). Toward an Ontology of Software Maintenance, Journal of Software Maintenance: Research and Practice, Wiley, 11, (6), pp. 356-389.

Knublauch, H., Fergerson, R.W., Noy, N.F. and Musen, M.A. (2004). The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. Third International Semantic Web Conference (ISWC), Hiroshima, Japan.

Knublauch, H., Oberle, D., Tetlow, P and Wallace, E. (2006, March 9). A Semantic Web Primer for Object-Oriented Software Developers, W3C Editors' Draft, http://www.w3.org/2001/sw/BestPractices/SE/ODSD/.

Knuth, D.E. (1992). Literate Programming, Center for the Study of Language and Information, Stanford, California.

Kof, L. (2005). "Natural Language Processing: Mature Enough for Requirements Documents Analysis?" in Gelbukh, A., Martínez-Barco, P., Muñoz, R., Sugumaran, V., Llopis, F., Gómez Hidalgo, J.M. (eds.): Natural Language Processing and Information Systems, Springer Lecture Notes in Computer Science, Berlin, pp. 91-102

Kollock, P. (1999). "The Economies of Online Cooperation: Gifts and Public Goods in Cyberspace." In Communities in Cyberspace, edited by Marc Smith and Peter Kollock. London, Routledge.

Krech, D. (2005). Redfoot Hypercoding System, Retrieved July 29, 2007 from http://redfoot.net/.

Kuhn, T.S. (1996). The Structure of Scientific Revolutions, The University of Chicago Press, 3rd ed.

Lassila, O. (2006, November 10-11). Generating Rewrite Rules by Browsing RDF Data, Proc. 2nd Intl Rule ML Conference. Retrieved June 1, 2008 from http://2006.ruleml.org/online-

proceedings/submission_36.pdf.

Lee, C.C. (2006, October 6). JavaNCSS - A Source Measurement Suite for Java, version 28.49. Retrieved March 24, 2008 from http://www.kclee.de/clemens/java/javancss/.

Lehman, M.M. (1969, Sept). The Programming Process, IBM Res. Rep. RC 2722, IBM Res. Centre, Yorktown Heights, NY.

Lehman, M.M. (1983). Programs, Life Cycles, and Laws of Software Evolution in Parikh, G. and Zvegintzov, N. (eds.) (1983). Tutorial on Software Maintenance, IEEE Computer Society Press, pp. 199-215

Lehman, M.M. and Stenning V. (1996, March). FEAST/1: Case for Support, ICSTM, DoC, EPSRC Proposal. Retrieved May 28, 2008 from http://www.cs.mdx.ac.uk/staffpages/mml/feast2/case96-2.html

Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E and Turski, W.M. (1997, November 5-7). Metrics and Laws of Software Evolution-The Nineties View, Proc. 4th Annual Software Metrics Symposium, pp. 20-32.

Lehman, M.M. (1998, July). FEAST/2: Case for Support, ICSTM, DoC, EPSRC Proposal. Retrieved May 28, 2008 from http://www.cs.mdx.ac.uk/staffpages/mml/feast2/case98-2.html.

Lehman, M.M. and Ramil, J. (2001a). Rules and Tools for Software Evolution Planning and Management, Annals of Software Eng., Special Issue on Softw. Manag., 11, Autumn 2001.

Lehman, M.M. (2001b, December 20). A Brief Introduction to the FEAST Hypothesis and Projects, http://www.doc.ic.ac.uk/~mml/feast/.

Lenat, D., Guha, R.V., Pittman, K., Pratt, D. and Shepherd, M. (1990, August). Cyc: Towards Programs with Common Sense, Communications of the ACM, 33/8.

Lerner, F. (1998). The Story of Libraries, From the Invention of Writing to the Computer Age, Continuum, New York.

Lovell, J. (2006). The Great Wall, China Against the World 1000 BC – AD 2000, Grove, New York.

Manola, F. and Miller, E. (eds.) (2004). RDF Primer, W3C Recommendation, Retrieved July 29, 2007 from http://www.w3.org/TR/rdf-primer/.

Martin, J. and McClure, C. (1983). Software Maintenance, The Problem and Its Solutions. Prentice Hall, Englewood Cliffs, NJ.

Martin, J. and McClure, C. (1985). Structured Techniques for Computing. Prentice Hall, Englewood Cliffs, NJ.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N. and Sycara, K. (eds.) (2004, November 22). OWL-S: Semantic Markup for Web Services, W3C Member Submission. Retrieved June 2, 2008 from http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/.

Martin, S. (2006, June 30). LSID URN/URI Notes, W3C ESW Wiki. Retrieved February 24, 2008 from http://esw.w3.org/topic/HCLSIG_BioRDF_Subgroup/LSID_URN_URI.

Masterman, M. (1970). The Nature of a Paradigm, in Lakatos, I and Musgrave, A (eds.). Criticism and the Growth of Knowledge: Proceedings of the International Colloquium in the Philosophy of Science, Cambridge University Press, 4, pp. 59-89.

Mazzocchi, S. and Elmer, F-J. (2007). Java RDFizer, M.I.T. Simile Project. Retrieved April 21, 2008 from http://simile.mit.edu/wiki/Java_RDFizer.

McBride, B. (2002). Jena: A Semantic Web Toolkit, Internet Computing, 6 (6), pp. 55-59.

McCoyd, E., ed. (2000). AAP Metadata Standards for Ebooks, Version 1.0, Association of American Publishers. Retrieved June 10, 2007 from http://www.publishers.org/digital/metadata.pdf.

McGuinness, D. and van Harmelen, F. (2004). Web Ontology Language (OWL) overview, W3C Recommendation. Retrieved July 29, 2007 from http://www.w3.org/TR/owl-features/.

Mendelsohn, N (ed.) (2008, February 8). The Self-Describing Web, W3C Draft TAG Finding. Retrieved April 27, 2008 from http://www.w3.org/2001/tag/doc/selfDescribingDocuments.

Meng, W. J., Rilling, J., Zhang, Y., Witte R. and Charland, P. (2006, October). An Ontological Software Comprehension Process Model, 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM), Genoa, Italy, pp. 52-66.

Miles, A. and Bechhofer, S. (eds.) (2008). SKOS Simple Knowledge Organization System Reference, W3C Working Draft 25 January 2008, Retrieved February 19, 2008 from http://www.w3.org/TR/skos-reference/.

Miller, J., Resnick, P. and Singer, D. (1996, October 31). Rating Services and Rating Systems (and Their Machine Readable Descriptions) Version 1.1, W3C Recommendation, Retrieved February 17, 2008 from http://www.w3.org/TR/REC-PICS-services.

Mitchell, T.M., Shinkareva, S.V., Carlson, A., Chang,K.-M., Malave, V.L., Mason, R.A. and Just, M.A. (2008, May 30). Predicting Human Brain Activity Associated with the Meanings of Nouns, Science, 320 (5880), pp. 1191-1195.

Nehaniv, C. (1997). Algebraic Models for Understanding Coordinate Systems and Cognitive Empowerment, 2nd Int'l Conf on Cognitive Technology: Humanizing the Information Age, IEEE Computer Society Press, pp. 147-162.

Nehaniv, C., Hewitt, J., Christianson, B. and Wernick, P. (2006). What Software Evolution and Biological Evolution Don't Have in Common, Proc. 2nd Int'l IEEE Workshop on Software Evolvability, Philadelphia, pp. 58-65.

Nelson, T.H. (1965). Complex Information Processing: A File Structure for the Complex, the Changing and the Indeterminate. Proc. 1965 20th National Conference of the ACM, ACM Press, New York, pp. 84-100.

Nelson, T.H. (1972). As We Will Think, in Nyce, J.M. and Kahn, P. (1991). From Memex to Hypertext. Academic Press, Boston, pp. 245-260.

Nelson, T.H. (1987). Computer Lib; Dream Machines. Redmond: Tempus Books of Microsoft Press.

Noy, N. and McGuinness, D. (2001). Ontology Development 101: A Guide to Creating Your First Ontology. Stanford Knowledge Systems Laboratory, Retrieved December 12, 2006 from ftp://ftp.ksl.stanford.edu/pub/KSL_Reports/KSL-01-05.pdf.gz.

Nyce, J.M. and Kahn, P. (1991). From Memex to hypertext : Vannevar Bush and the Mind's Machine, Academic Press, 1991.

Oberle, D. (2006). Semantic Management of Middleware. Springer, pp. 127-135.

Object Management Group (OMG) (2004, April). Life Sciences Identifiers, OMG Adopted Specification dtc/04-05-01. Retrieved February 24, 2008 from http://www.omg.org/docs/dtc/04-05-01.pdf.

Object Management Group (OMG) (2008, 1 Jan). Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), v1.0, Retrieved February 16, 2008 from http://www.omg.org/docs/formal/08-01-01.pdf.

Ogbuji, C. (2007, May 26). Linked Data and Overselling the HTTP URI Scheme. Retrieved March 1, 2008 from http://copia.ogbuji.net/blog/2007-05-26/linked-data-is-overseling-http.

Olsen, S. (2004, February 17). Google, Yahoo duel for documents, C|Net News. Retrieved February 10, 2008 from http://www.news.com/Google,-Yahoo-duel-for-documents/2100-1038_3-5160480.html.

OpenLink Software (2008). OpenLink RDF Browser, version 2. Retrieved Juna 1, 2008 from http://demo.openlinksw.com/DAV/JS/rdfbrowser2/index.html.

Parikh, G. and Zveginitzov, N., eds. (1983). Tutorial on Software Maintenance, IEEE Computer Society, Silver Spring, MD.

Paskin, N. (ed.) (2006, October 6). The DOI® Handbook, Edition 4.4.1, International DOI Foundation, Inc., Oxford, UK. Retrieved February 24, 2008 from http://www.doi.org/handbook_2000/DOIHandbook-v4-4.pdf.

Petroski, H. (2006). Success Through Failure: The Paradox of Design, Princeton University Press, Princeton, NJ, pp. 57-65.

Pressman, R. (1988). Making Software Engineering Happen; A Guide for Instituting the Technology. Prentice Hall, Englewood Cliffs, NJ.

Raggett, D., Le Hors, A. and Jacobs, I. (1999, December 24). HTML 4.01 Specification, W3C Recommendation. Retrieved February 19, 2008 from http://www.w3.org/TR/html401/.

Reitz, M. (2006). Software Evolvability by Component-Orientation: A Loosely Coupled Component Model Inspired by Biology, Proc. 2nd Int'l IEEE Workshop on Software Evolvability, Philadelphia, pp. 66-73.

Reschke, J.F. (ed.), Reddy, S, Davis, J. and Babich, A. (2008, August). Web Distributed Authoring and Versioning (WebDAV) SEARCH, draft-reschke-webdav-search-18, IETF Internet Draft. Retrieved November 14, 2008 from http://greenbytes.de/tech/webdav/draft-reschke-webdav-search-18.html.

Resnick, P. and Miller, J. (1996). PICS: Internet Access Controls Without Censorship, Communications of the ACM, 1996, 39 (10), pp. 87-93.

Rochkind, M.J. (2004). Advanced Unix Programming, 2nd edition, Addison-Wesley Professional, New York, pp. 105.

Rogers, H. (2005). Writing Systems, A Linguistic Approach, Blackwell Publishing.

Rugaber, S. (2000). The use of domain knowledge in program understanding', Annals of Software Eng., 9, pp. 143–192.

Sandq, S.F. (2000). The Software Development Guide Book. Sandq Telesis, Atlanta, GA.

Schraefel, M.C., Karam, M. and Zhao, S. (2003, August 26) mSpace: Interaction Design for User-Determined, Adaptable Domain Exploration in Hypermedia. Proc. Workshop on Adaptive Hypermedia and Adaptive Web Based Systems, Nottingham, UK. Retrieved June 1, 2008

from http://eprints.ecs.soton.ac.uk/8803/1/schraefel_mspaceAH03.pdf.

Schroth, C. (2007). Web 2.0 versus SOA: Converging Concepts Enabling Seamless Cross-Organizational Collaboration, Proc. CEC-EEE 2007, pp. 47-54. Retrieved March 4, 2008 from http://doi.ieeecomputersociety.org/10.1109/CEC-EEE.2007.105.

Selby, R.W., ed. (2007). Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research. Wiley-IEEE Computer Society.

Shafer, K., Weibel, S., Jul, E. and Fausey, J. (1996). Introduction to Persistent Uniform Resource Locators, OCLC Online Computer Library Center, Inc. Retrieved February 23, 2008 from http://purl.oclc.org/docs/inet96.html.

Shera, J.H. (1965). Libraries and the Organization of Knowledge, Archon Books, Hamden, Connecticut.

Shere, K.D. (1988). Software Engineering and Management. Prentice Hall, Englewood Cliffs, NJ.

Shooman, M.L. (1983). Software Engineering; Design, Reliability and Management. McGraw-Hill.

Smith, M. (1992). Voices from the WELL: The Logic of the Virtual Commons, Master's thesis, Department of Sociology, University of California at Los Angeles.

Soydan, G. H. and Kokar, M.M. (Nov 2006). An OWL Ontology for Representing the CMMI-SW Model, Workshop on Semantic Web Enabled Software Engineering (SWESE), Athens, GA.

Srivastava, B. and Koehler, J. (2003). Web Service Composition - Current Solutions and Open Problems, Workshop on Planning for Web Services, ICAPS 2003, Trento, Italy. Retrieved March 4, 2008 from http://icaps03.itc.it/satellite_events/documents/WS/WS5/05/srivastava-icaps2003-p4ws.pdf.

Stockwell, F. (2001). A History of Information Storage and Retrieval, McFarland & Company, Jefferson, NC.

Sun Microsystems (1994), Javadoc Tool. Retrieved November 7, 2008 from http://java.sun.com/j2se/javadoc/.

Sun Microsystems (1999). JAR File Specification. Retrieved February 19, 2008 from http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html.

Szepielak, D. (2007). REST-Based Service Oriented Architecture for Dynamically Integrated Information Systems, Proc. IBM PhD Student Symposium, ISCOC 2007. Retrieved March 4,

2008 from http://infolab.uvt.nl/phd-icsoc07/files/phdicsoc06szepielak.pdf.

Technorati, Inc (2008). Technorati: About Us. Retrieved February 10, 2008 from http://technorati.com/about/.

Tetlow, P., Pan, J., Oberle, D., Wallace, E., Uschold, M. and Kendell, E. (2006, February 11). Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering, W3C Editors' Draft, http://www.w3.org/2001/sw/BestPractices/SE/ODA/060211/.

Thornton, J.L. (1941). The Chronology of Librarianship, An Introduction to the History of Libraries and Book Collecting, Grafton, London.

Tjortjis, C. and Layzell, P. (2001) Expert Maintainers' Strategies and Needs when Understanding Software: A Case Study Approach, Proc. Eighth Asia-Pacific Software Engineering Conference (APSEC'01), Macau, pp. 281.

Tudge, C. (1998). Neaderthals, Bandits and Farmers, How Agriculture Really Began, Yale University Press, New Haven & London.

Van de Sompel, H., Hammond, T., Neylon, E., Weibel, S. (2003, September). The "info" URI Scheme for Information Assets with Identifiers in Public Namespaces, Internet Engineering Task Force, Internet Draft (expired). Retrieved February 23, 2008 from http://info-uri.info/registry/docs/drafts/draft-vandesompel-info-uri-00.txt.

Van Doren, E. (1997). Maintenance of operational systems - an overview. Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/str/descriptions/mos_body.html, accessed July 29, 2007.

Van Heesh, D. (1997). Doxygen Source Code Documentation Generator Tool. Retrieved November 7, 2008 from http://www.stack.nl/~dimitri/doxygen/.

Van Lamsweerde, A., Delcourt, B., Delor, E., Schayes, M.-C. and Champagne, R. (1998). Generic lifecycle support in the ALMA environment, Proc. IEEE Trans. on Software Eng., 14, (6), pp. 720-727.

Vernant, J-P (1982). The Origins of Greek Thought, Cornell University Press, Ithaca, NY.

Wall, L. and Burke, S.M. (1987, updated 2006). Perl POD Manual Page. Retrieved July 24, 2007 from http://perldoc.perl.org/perlpod.html.

Weibel, S., Kunze, J.,Lagoze, C. and Wolf, M. (1998, September). Dublin Core Metadata for Resource Discovery, Internet Engineering Task Force RFC 2413, Retrieved February 17, 2008 from http://www.ietf.org/rfc/rfc2413.txt.

Weick, K. (1976). Educational organizations as loosely coupled systems, Administrative Science Quarterly, 21 (1976), pp. 1-9 (part).

Weigand, W.A. and Davis, D.G. Jr., eds., (1994). Encyclopedia of Library History, Garland Reference Library of Social Sciences, 53, Garland Publishing, New York & London.

Welsh, J. (1994a). Software is history!, in Roscoe, W. (Ed.): A classical mind: essays in honour of C.A.R. Hoare, Prentice Hall, pp. 419–430.

Welsh, J. and Han, J. (1994b). Software documents: concepts and tools, Software Concepts and Tools, 15, pp. 12–25.

Wernick, P., Hall, T. and Nehaniv, C. (2006). Software Evolutionary Dynamics Modelled as the Activity of an Actor Network, Proc. 2nd Int'l IEEE Workshop on Software Evolvability, Philadelphia, pp. 74-81.

Whitehead, E.J. and Goland, Y.Y. (2003). The WebDAV Property Design, Software: Practice and Experience, 34, 2, pp. 135-161. Retrieved on November 12, 2008 from http://www.cs.ucsc.edu/~ejw/papers/spe-whitehead.pdf.

Wiegers, K.E. (2001, May). Requirements When the Field Isn't Green. Software Test and Quality Engineering. Retrieved 21 July 2007 from http://www.processimpact.com/articles/reqs_not_green.pdf

Wilde, N. (1990). Understanding Program Dependencies. Technical Report SEI-CM-26, Carnegie Mellon University.

Wilensky, U. (1999). NetLogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL. Retrieved April 9, 2008 from http://ccl.northwestern.edu/netlogo/.

Wongthongtham, P., Chang, E. and Dillon, T. (2006). Enhancing Software Engineering Project Information through Software Engineering Ontology Instantiations. Proc. 2006 IEEE/WIC/ACM Intl Conf on Web Intelligence, pp. 33-37.

Wood, D. (2005). Scaling the Kowari Metastore, in Dean, M. et al. (eds.): WISE 2005 Workshops, LNCS 3807, pp. 193-198.

World Wide Web Consortium (W3C) Technical Architecture Group (TAG) (2005, June 15). httpRange-14: What is the range of the HTTP dereference function?, TAG Issues List. Retrieved February 27, 2008 from http://www.w3.org/2001/tag/issues.html#httpRange-14.

Wright, A. (2007). Glut: Mastering Information Through the Ages. Joseph Henry Press, Washington, D.C.

Yakel, E. (2007). Digital Curation, OCLC Systems & Services, 23, (4), pp. 335-340. Retrieved May 26, 2008 from http://www.ingentaconnect.com/content/mcb/164/2007/00000023/00000004/art00003

Yang, H. J., Cui, Z. and O'Brien, P. (1999). Extracting Ontologies from Legacy Systems for Understanding and Re-Engineering, 23rd Intl Computer Software and Applications Conference. Washington, DC, pp. 21-26

Yanosh, S. (2007, June 7), The Metadata Company, personal communication, June 7, 2007

Yourdon, E. (1988). Managing the System Life Cycle. Yourdon Press, Upper Saddle River, NJ

Appendix A. SEC Ontology

An ontology of software engineering concepts (the SEC ontology) is presented in this appendix in RDF/XML format. The ontology is expressed in the Web Ontology Language in the DL (for "description logic") variant (OWL-DL). The ontology is described in Chapter VI and used to represent metadata in the SWAMM methodology described in Chapter VII. Examples of use of the SEC ontology are given in Chapters VI and VIII.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY owl "http://www.w3.org/2002/07/owl#">
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
  <!ENTITY sec0.2.owl "http://www.itee.uq.edu.au/~dwood/ontologies/sec0.2.owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
]>
<rdf:RDF xml:base="&sec0.2.owl;"
        xmlns:owl="&owl;"
        xmlns:rdf="&rdf;"
        xmlns:rdfs="&rdfs;">

<!-- Ontology Information -->
  <owl:Ontology rdf:about=""
              rdfs:label="Software Engineering Concepts"
              owl:versionInfo="0.2">
    <rdfs:comment>An ontology to represent concepts in software engineering, such as object-
oriented components and their relationships to each other and external
documentation.</rdfs:comment>
  </owl:Ontology>

<!-- Classes -->
  <owl:Class rdf:about="#IntegrationTest"
           rdfs:label="Integration Test">
    <rdfs:comment>An integration test of a software subsystem.</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#Test"/>
  </owl:Class>

  <owl:Class rdf:about="#Metric"
           rdfs:label="Metric">
    <rdfs:comment>An automatically generated metric about a software
component.</rdfs:comment>
```

```
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOSoftwareComponent"/>
      <owl:onProperty rdf:resource="#isMetricOf"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#OOSoftwareComponent"/>
  <owl:disjointWith rdf:resource="#Requirement"/>
  <owl:disjointWith rdf:resource="#Test"/>
</owl:Class>

<owl:Class rdf:about="#OOAbstractClass"
         rdfs:comment="An Object-Oriented Abstract Class.">
  <rdfs:label xml:lang="en">OO Abstract Class</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OOClass"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOMethodSignature"/>
      <owl:onProperty rdf:resource="#hasMethodSignature"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#OOMethodSignature"/>
</owl:Class>

<owl:Class rdf:about="#OOClass"
         rdfs:comment="An Object-Oriented Class">
  <rdfs:label xml:lang="en">OO Class</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OOSoftwareComponent"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#UnitTest"/>
      <owl:onProperty rdf:resource="#hasTest"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#IntegrationTest"/>
      <owl:onProperty rdf:resource="#hasTest"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOClass"/>
      <owl:onProperty rdf:resource="#extendedBy"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
```

```
        <owl:Restriction>
          <owl:allValuesFrom rdf:resource="#Requirement"/>
          <owl:onProperty rdf:resource="#encodesRequirement"/>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:allValuesFrom rdf:resource="#OOInterface"/>
          <owl:onProperty rdf:resource="#implementsInterface"/>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:allValuesFrom rdf:resource="#OOClass"/>
          <owl:onProperty rdf:resource="#extends"/>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:allValuesFrom rdf:resource="#OOPackage"/>
          <owl:onProperty rdf:resource="#packageMemberOf"/>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:allValuesFrom rdf:resource="#OOMethod"/>
          <owl:onProperty rdf:resource="#hasMethod"/>
        </owl:Restriction>
      </rdfs:subClassOf>
      <owl:disjointWith rdf:resource="#OOConstructor"/>
      <owl:disjointWith rdf:resource="#OOInterface"/>
      <owl:disjointWith rdf:resource="#OOMethod"/>
      <owl:disjointWith rdf:resource="#OOMethodSignature"/>
      <owl:disjointWith rdf:resource="#OOPackage"/>
      <owl:disjointWith rdf:resource="#OOProgram"/>
    </owl:Class>

    <owl:Class rdf:about="#OOConstructor"
            rdfs:label="OO Constructor">
      <rdfs:comment>Representation of a object-oriented constructor within an OO
class.</rdfs:comment>
      <rdfs:subClassOf rdf:resource="#OOSoftwareComponent"/>
      <owl:disjointWith rdf:resource="#OOClass"/>
      <owl:disjointWith rdf:resource="#OOInterface"/>
      <owl:disjointWith rdf:resource="#OOMethod"/>
      <owl:disjointWith rdf:resource="#OOMethodSignature"/>
      <owl:disjointWith rdf:resource="#OOPackage"/>
```

```
  <owl:disjointWith rdf:resource="#OOProgram"/>
</owl:Class>

<owl:Class rdf:about="#OOInterface"
         rdfs:comment="An Object-Oriented Interface.">
  <rdfs:label xml:lang="en">OO Interface</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OOSoftwareComponent"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOPackage"/>
      <owl:onProperty rdf:resource="#packageMemberOf"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOInterface"/>
      <owl:onProperty rdf:resource="#extendedBy"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOInterface"/>
      <owl:onProperty rdf:resource="#extends"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOMethodSignature"/>
      <owl:onProperty rdf:resource="#hasMethodSignature"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#OOClass"/>
  <owl:disjointWith rdf:resource="#OOConstructor"/>
  <owl:disjointWith rdf:resource="#OOMethod"/>
  <owl:disjointWith rdf:resource="#OOMethodSignature"/>
  <owl:disjointWith rdf:resource="#OOPackage"/>
  <owl:disjointWith rdf:resource="#OOProgram"/>
</owl:Class>

<owl:Class rdf:about="#OOMethod"
         rdfs:comment="An Object-Oriented Method.">
  <rdfs:label xml:lang="en">OO Method</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OOSoftwareComponent"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOMethod"/>
      <owl:onProperty rdf:resource="#methodUsedBy"/>
```

```
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#Requirement"/>
      <owl:onProperty rdf:resource="#entryPointFor"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#UnitTest"/>
      <owl:onProperty rdf:resource="#hasTest"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOMethod"/>
      <owl:onProperty rdf:resource="#usesMethod"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#OOClass"/>
  <owl:disjointWith rdf:resource="#OOConstructor"/>
  <owl:disjointWith rdf:resource="#OOInterface"/>
  <owl:disjointWith rdf:resource="#OOMethodSignature"/>
  <owl:disjointWith rdf:resource="#OOPackage"/>
  <owl:disjointWith rdf:resource="#OOProgram"/>
</owl:Class>

<owl:Class rdf:about="#OOMethodSignature"
          rdfs:comment="An Object-Oriented Method Signature."
          rdfs:label="OO Method Signature">
  <rdfs:subClassOf rdf:resource="#OOSoftwareComponent"/>
  <owl:disjointWith rdf:resource="#OOAbstractClass"/>
  <owl:disjointWith rdf:resource="#OOClass"/>
  <owl:disjointWith rdf:resource="#OOConstructor"/>
  <owl:disjointWith rdf:resource="#OOInterface"/>
  <owl:disjointWith rdf:resource="#OOMethod"/>
  <owl:disjointWith rdf:resource="#OOPackage"/>
  <owl:disjointWith rdf:resource="#OOProgram"/>
</owl:Class>

<owl:Class rdf:about="#OOPackage"
          rdfs:comment="An Object-Oriented package of classes."
          rdfs:label="OO Package">
  <rdfs:subClassOf rdf:resource="#OOSoftwareComponent"/>
  <rdfs:subClassOf>
    <owl:Restriction>
```

```
      <owl:allValuesFrom>
       <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
         <rdf:Description rdf:about="#OOClass"/>
         <rdf:Description rdf:about="#OOInterface"/>
        </owl:unionOf>
       </owl:Class>
      </owl:allValuesFrom>
      <owl:onProperty rdf:resource="#hasPackageMember"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#IntegrationTest"/>
      <owl:onProperty rdf:resource="#hasTest"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOProgram"/>
      <owl:onProperty rdf:resource="#packageOf"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#OOClass"/>
  <owl:disjointWith rdf:resource="#OOConstructor"/>
  <owl:disjointWith rdf:resource="#OOInterface"/>
  <owl:disjointWith rdf:resource="#OOMethod"/>
  <owl:disjointWith rdf:resource="#OOMethodSignature"/>
  <owl:disjointWith rdf:resource="#OOProgram"/>
</owl:Class>

<owl:Class rdf:about="#OOProgram"
          rdfs:comment="An Object-Oriented software program."
          rdfs:label="OOProgram">
  <rdfs:subClassOf rdf:resource="#OOSoftwareComponent"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOPackage"/>
      <owl:onProperty rdf:resource="#hasPackage"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#OOClass"/>
  <owl:disjointWith rdf:resource="#OOConstructor"/>
  <owl:disjointWith rdf:resource="#OOInterface"/>
  <owl:disjointWith rdf:resource="#OOMethod"/>
  <owl:disjointWith rdf:resource="#OOMethodSignature"/>
  <owl:disjointWith rdf:resource="#OOPackage"/>
```

```
</owl:Class>

<owl:Class rdf:about="#OOSoftwareComponent"
        rdfs:comment="A Object-Oriented software component."
        rdfs:label="OO Software Component">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#Test"/>
      <owl:onProperty rdf:resource="#hasTest"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#Metric"/>
      <owl:onProperty rdf:resource="#hasMetric"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Metric"/>
  <owl:disjointWith rdf:resource="#Requirement"/>
</owl:Class>

<owl:Class rdf:about="#Requirement"
        rdfs:comment="A requirement for a software system.">
  <rdfs:label xml:lang="en">Requirement</rdfs:label>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOClass"/>
      <owl:onProperty rdf:resource="#requirementEncodedBy"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOMethod"/>
      <owl:onProperty rdf:resource="#hasEntryPoint"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Metric"/>
  <owl:disjointWith rdf:resource="#OOSoftwareComponent"/>
  <owl:disjointWith rdf:resource="#Test"/>
</owl:Class>

<owl:Class rdf:about="#Test"
        rdfs:label="Test">
  <rdfs:comment>An automated test of a software component.</rdfs:comment>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#OOSoftwareComponent"/>
```

```
        <owl:onProperty rdf:resource="#isTestOf"/>
       </owl:Restriction>
      </rdfs:subClassOf>
      <owl:disjointWith rdf:resource="#Metric"/>
      <owl:disjointWith rdf:resource="#Requirement"/>
    </owl:Class>

    <owl:Class rdf:about="#UnitTest"
            rdfs:comment="A unit test of a software component."
            rdfs:label="Unit Test">
      <rdfs:subClassOf rdf:resource="#Test"/>
    </owl:Class>

  <!-- Annotation Properties -->
    <owl:AnnotationProperty rdf:about="&rdfs;comment"/>
    <owl:AnnotationProperty rdf:about="&rdfs;label"/>
    <owl:AnnotationProperty rdf:about="&owl;versionInfo"/>

  <!-- Datatype Properties -->
    <owl:DatatypeProperty rdf:about="#hasTestResults"
                    rdfs:label="has test results">
      <rdfs:comment>Defines the relationship between a Test and the its results.</rdfs:comment>
      <rdfs:domain rdf:resource="#Test"/>
    </owl:DatatypeProperty>

  <!-- Object Properties -->
    <owl:ObjectProperty rdf:about="#constructorOf"
                    rdfs:label="Constructor of">
      <rdfs:comment>Defines the relationship between an object-oriented Constructor and an OO
Class that contains it.</rdfs:comment>
      <rdfs:domain rdf:resource="#OOConstructor"/>
      <rdfs:range rdf:resource="#OOClass"/>
      <owl:inverseOf rdf:resource="#hasConstructor"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#encodesRequirement">
      <rdfs:comment>Defines the relationship between an OO Requirement and an OO Class which
encodes it.</rdfs:comment>
      <rdfs:domain rdf:resource="#OOClass"/>
      <rdfs:label xml:lang="en">encodes requirement</rdfs:label>
      <rdfs:range rdf:resource="#Requirement"/>
      <owl:inverseOf rdf:resource="#requirementEncodedBy"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#entryPointFor"
                    rdfs:label="entry point for">
      <rdfs:comment>Defines the relationship between a Method and a Requirement, where the
```

Requirement's implemention is best understood by starting with the Method.</rdfs:comment>
<rdfs:domain rdf:resource="#OOConstructor"/>
<rdfs:domain rdf:resource="#OOMethod"/>
<rdfs:domain rdf:resource="#OOMethodSignature"/>
<rdfs:range rdf:resource="#Requirement"/>
<owl:inverseOf rdf:resource="#hasEntryPoint"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#extendedBy"
rdfs:label="extended by">
<rdfs:comment>Defines the relationship between an OO super class and another class which extends it.</rdfs:comment>
<rdfs:domain rdf:resource="#OOSoftwareComponent"/>
<rdfs:range rdf:resource="#OOSoftwareComponent"/>
<owl:inverseOf rdf:resource="#extends"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#extends"
rdfs:label="extends">
<rdf:type rdf:resource="&owl;TransitiveProperty"/>
<rdfs:comment xml:lang="en">Defines the relationship between an OO Class which extends another and the OO CLass which it extends.</rdfs:comment>
<rdfs:domain rdf:resource="#OOSoftwareComponent"/>
<rdfs:range rdf:resource="#OOSoftwareComponent"/>
<owl:inverseOf rdf:resource="#extendedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasConstructor"
rdfs:label="has Constructor">
<rdfs:comment>Defines the relationship between an Object-Oriented Class and the constructors that it contains.</rdfs:comment>
<rdfs:domain rdf:resource="#OOClass"/>
<rdfs:range rdf:resource="#OOConstructor"/>
<owl:inverseOf rdf:resource="#constructorOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasDeveloper"
rdfs:label="has Developer">
<rdfs:comment xml:lang="en">Defines the relationship between a software construct (a software component, metric, test or requirement) and a developer responsible for creating or maintaining it..</rdfs:comment>
<rdfs:domain rdf:resource="#IntegrationTest"/>
<rdfs:domain rdf:resource="#Metric"/>
<rdfs:domain rdf:resource="#OOAbstractClass"/>
<rdfs:domain rdf:resource="#OOClass"/>
<rdfs:domain rdf:resource="#OOConstructor"/>
<rdfs:domain rdf:resource="#OOInterface"/>

```
    <rdfs:domain rdf:resource="#OOMethod"/>
    <rdfs:domain rdf:resource="#OOMethodSignature"/>
    <rdfs:domain rdf:resource="#OOPackage"/>
    <rdfs:domain rdf:resource="#OOProgram"/>
    <rdfs:domain rdf:resource="#OOSoftwareComponent"/>
    <rdfs:domain rdf:resource="#Requirement"/>
    <rdfs:domain rdf:resource="#Test"/>
    <rdfs:domain rdf:resource="#UnitTest"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#hasEntryPoint"
                      rdfs:label="has entry point">
    <rdfs:comment>Defines the relationship between a Requirement and a Method which is the
best place to start understanding the Requirement's implementation.</rdfs:comment>
    <rdfs:domain rdf:resource="#Requirement"/>
    <rdfs:range rdf:resource="#OOConstructor"/>
    <rdfs:range rdf:resource="#OOMethod"/>
    <rdfs:range rdf:resource="#OOMethodSignature"/>
    <owl:inverseOf rdf:resource="#entryPointFor"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#hasMethod">
    <rdfs:comment xml:lang="en">Defines the relationship between an Object-Oriented Class and
the methods that it contains.</rdfs:comment>
    <rdfs:domain rdf:resource="#OOClass"/>
    <rdfs:label xml:lang="en">has method</rdfs:label>
    <rdfs:range rdf:resource="#OOMethod"/>
    <owl:inverseOf rdf:resource="#methodOf"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#hasMethodSignature"
                      rdfs:label="has method signature">
    <rdfs:comment>Defines the relationship between an OO Interface or Abstract Class and an OO
Method Signature.</rdfs:comment>
    <rdfs:domain rdf:resource="#OOAbstractClass"/>
    <rdfs:domain rdf:resource="#OOInterface"/>
    <rdfs:range rdf:resource="#OOMethodSignature"/>
    <owl:inverseOf rdf:resource="#methodSignatureOf"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#hasMetric"
                      rdfs:label="has metric">
    <rdfs:comment>Defines the relationship between an OO Software Component and a Metric
which has been calculated for it.</rdfs:comment>
    <rdfs:domain rdf:resource="#OOSoftwareComponent"/>
    <rdfs:range rdf:resource="#Metric"/>
    <owl:inverseOf rdf:resource="#isMetricOf"/>
```

```
    </owl:ObjectProperty>

    <owl:FunctionalProperty rdf:about="#hasMetricValue"
                            rdfs:label="has metric value">
      <rdf:type rdf:resource="&owl;ObjectProperty"/>
      <rdfs:comment>Defines the relationship between a Metric and its calculated
value.</rdfs:comment>
      <rdfs:domain rdf:resource="#Metric"/>
    </owl:FunctionalProperty>

    <owl:ObjectProperty rdf:about="#hasPackage"
                        rdfs:label="has package">
      <rdfs:comment>Defines the relationship between an OO Program and a Package which it
contains.</rdfs:comment>
      <rdfs:domain rdf:resource="#OOProgram"/>
      <rdfs:range rdf:resource="#OOPackage"/>
      <owl:inverseOf rdf:resource="#packageOf"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#hasPackageMember"
                        rdfs:label="has Class">
      <rdfs:comment xml:lang="en">Defines the relationship between an OO Package and the OO
Classes/AbstractClasses/Interfaces which belong to it.</rdfs:comment>
      <rdfs:domain rdf:resource="#OOPackage"/>
      <rdfs:range rdf:resource="#OOSoftwareComponent"/>
      <owl:inverseOf rdf:resource="#packageMemberOf"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#hasTest"
                        rdfs:label="has test">
      <rdfs:comment>Defines the relationship between an OO Software Component and an
associated Test.</rdfs:comment>
      <rdfs:domain rdf:resource="#OOSoftwareComponent"/>
      <rdfs:range rdf:resource="#Test"/>
      <owl:inverseOf rdf:resource="#isTestOf"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#implementsInterface"
                        rdfs:label="implements">
      <rdfs:comment xml:lang="en">Defines the relationship between an OO Class and an OO
Interface which it extends.</rdfs:comment>
      <rdfs:domain rdf:resource="#OOClass"/>
      <rdfs:range rdf:resource="#OOInterface"/>
      <owl:inverseOf rdf:resource="#interfaceImplementedBy"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#interfaceImplementedBy"
```

```
                rdfs:label="implemented by">
    <rdfs:comment>Defines the relationship between an OO Interface and an OO Class which
implements it.</rdfs:comment>
    <rdfs:domain rdf:resource="#OOInterface"/>
    <rdfs:range rdf:resource="#OOClass"/>
    <owl:inverseOf rdf:resource="#implementsInterface"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#isMetricOf"
                rdfs:label="is metric of">
    <rdfs:comment>Defines the relationship between a Metric and an OO Software Component for
which it was calculated.</rdfs:comment>
    <rdfs:domain rdf:resource="#Metric"/>
    <rdfs:range rdf:resource="#OOSoftwareComponent"/>
    <owl:inverseOf rdf:resource="#hasMetric"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#isTestOf"
                rdfs:label="is test of">
    <rdfs:comment>Defines the relationship between a Test and an OO Software
Component.</rdfs:comment>
    <rdfs:domain rdf:resource="#Test"/>
    <rdfs:range rdf:resource="#OOSoftwareComponent"/>
    <owl:inverseOf rdf:resource="#hasTest"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#lastModifiedAt"
                rdfs:label="last modified at">
    <rdfs:comment xml:lang="en">Defines at which date and time a given element was
modified.</rdfs:comment>
    <rdfs:domain rdf:resource="#IntegrationTest"/>
    <rdfs:domain rdf:resource="#Metric"/>
    <rdfs:domain rdf:resource="#OOAbstractClass"/>
    <rdfs:domain rdf:resource="#OOClass"/>
    <rdfs:domain rdf:resource="#OOConstructor"/>
    <rdfs:domain rdf:resource="#OOInterface"/>
    <rdfs:domain rdf:resource="#OOMethod"/>
    <rdfs:domain rdf:resource="#OOMethodSignature"/>
    <rdfs:domain rdf:resource="#OOPackage"/>
    <rdfs:domain rdf:resource="#OOProgram"/>
    <rdfs:domain rdf:resource="#OOSoftwareComponent"/>
    <rdfs:domain rdf:resource="#Requirement"/>
    <rdfs:domain rdf:resource="#Test"/>
    <rdfs:domain rdf:resource="#UnitTest"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#lastValidatedAt"
```

```
                rdfs:label="last validated at">
    <rdfs:comment>Defines at which date and time a given Requirement was validated by a
person.</rdfs:comment>
    <rdfs:domain rdf:resource="#Requirement"/>
  </owl:ObjectProperty>

  <owl:FunctionalProperty rdf:about="#methodOf"
                rdfs:label="method of">
    <rdf:type rdf:resource="&owl;ObjectProperty"/>
    <rdfs:comment>Defines the relationship between a method and the Object-Oriented Class
which contains it.</rdfs:comment>
    <rdfs:domain rdf:resource="#OOMethod"/>
    <rdfs:range rdf:resource="#OOAbstractClass"/>
    <rdfs:range rdf:resource="#OOClass"/>
    <owl:inverseOf rdf:resource="#hasMethod"/>
  </owl:FunctionalProperty>

  <owl:ObjectProperty rdf:about="#methodSignatureOf"
                rdfs:label="method signature of">
    <rdfs:comment>Defines the relationship between an OO Method Signature and a containing
OO Interface or Abstract Class.</rdfs:comment>
    <rdfs:domain rdf:resource="#OOMethodSignature"/>
    <rdfs:range rdf:resource="#OOAbstractClass"/>
    <rdfs:range rdf:resource="#OOInterface"/>
    <owl:inverseOf rdf:resource="#hasMethodSignature"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#methodUsedBy"
                rdfs:label="method used by">
    <rdfs:comment>Defines the relationship between two OO Methods, from the used Method to
the using Method.</rdfs:comment>
    <rdfs:domain rdf:resource="#OOMethod"/>
    <rdfs:range rdf:resource="#OOMethod"/>
    <owl:inverseOf rdf:resource="#usesMethod"/>
  </owl:ObjectProperty>

  <owl:FunctionalProperty rdf:about="#metricCalculatedAt"
                rdfs:label="metric calculated at">
    <rdf:type rdf:resource="&owl;ObjectProperty"/>
    <rdfs:comment>Defines the relationship between a Metric and the date-time at which it was
calculated.</rdfs:comment>
    <rdfs:domain rdf:resource="#Metric"/>
  </owl:FunctionalProperty>

  <owl:FunctionalProperty rdf:about="#packageMemberOf">
    <rdf:type rdf:resource="&owl;ObjectProperty"/>
    <rdfs:comment xml:lang="en">Defines the relationship between an OO
```

```
Class/AbstractClass/Interface and the OO Package to which it belongs.</rdfs:comment>
      <rdfs:domain rdf:resource="#OOSoftwareComponent"/>
      <rdfs:label xml:lang="en">class of</rdfs:label>
      <rdfs:range rdf:resource="#OOPackage"/>
      <owl:inverseOf rdf:resource="#hasPackageMember"/>
    </owl:FunctionalProperty>

    <owl:FunctionalProperty rdf:about="#packageOf"
                            rdfs:label="package of">
      <rdf:type rdf:resource="&owl;ObjectProperty"/>
      <rdfs:comment>Defines the relationship between an OO Package and the OO Program which
contains it.</rdfs:comment>
      <rdfs:domain rdf:resource="#OOPackage"/>
      <rdfs:range rdf:resource="#OOProgram"/>
      <owl:inverseOf rdf:resource="#hasPackage"/>
    </owl:FunctionalProperty>

    <owl:ObjectProperty rdf:about="#requirementEncodedBy">
      <rdfs:comment xml:lang="en">Defines the relationship between an OO Requirement and an
OO Class which encodes it.</rdfs:comment>
      <rdfs:domain rdf:resource="#Requirement"/>
      <rdfs:label xml:lang="en">requirement encoded by</rdfs:label>
      <rdfs:range rdf:resource="#OOClass"/>
      <owl:inverseOf rdf:resource="#encodesRequirement"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#seeAlso"
                        rdfs:label="seeAlso">
      <rdfs:comment>Uniform Resource Locator describing or providing further information about a
class.</rdfs:comment>
      <rdfs:domain rdf:resource="#Metric"/>
      <rdfs:domain rdf:resource="#OOSoftwareComponent"/>
      <rdfs:domain rdf:resource="#Requirement"/>
      <rdfs:domain rdf:resource="#Test"/>
    </owl:ObjectProperty>

    <owl:FunctionalProperty rdf:about="#succeeded"
                            rdfs:label="succeeded">
      <rdf:type rdf:resource="&owl;ObjectProperty"/>
      <rdfs:comment>Defines the relationship between a Test and a Boolen indicating whether the
Test passed or failed.</rdfs:comment>
      <rdfs:domain rdf:resource="#Test"/>
    </owl:FunctionalProperty>

    <owl:FunctionalProperty rdf:about="#testedAt"
                            rdfs:label="tested at">
      <rdf:type rdf:resource="&owl;ObjectProperty"/>
```

```
      <rdfs:comment>Defines the relationship between a Test and the date-time at which the test
was concluded.</rdfs:comment>
      <rdfs:domain rdf:resource="#Test"/>
    </owl:FunctionalProperty>

    <owl:ObjectProperty rdf:about="#usesMethod"
                        rdfs:label="uses method">
      <rdfs:comment>Defines the relationship between two OO Methods, from the using Method to
the used Method.</rdfs:comment>
      <rdfs:domain rdf:resource="#OOMethod"/>
      <rdfs:range rdf:resource="#OOMethod"/>
      <owl:inverseOf rdf:resource="#methodUsedBy"/>
    </owl:ObjectProperty>
  </rdf:RDF>
```

Appendix B. Inhibit

This appendix provides source code for the Inhibit Javascript library described in Chapter V, Section 5.6. The Inhibit library is used with an Active PURL (also described in Chapter V) to negotiate an intermediate data set from a Web service that is acceptable for display in the Exhibit faceted navigation system.

Section B.1 shows a sample HTML document that uses the Inhibit Javascript library. The sample HTML may be compared to the HTML that directly calls an Exhibit library. The differences are intentionally minimized.

Section B.2 shows the source code for the Inhibit Javascript library itself.

B.1 Sample HTML Document Calling inhbit.js

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
        <head>
                <title>Inhibit Demo</title>
                <link id="inhibitdata" type="application/json" href="/cgi-
bin/activepurl.pl" />
                <!--[if lt IE 7]>
  <script defer type="text/javascript" src="javascripts/pngfix.js"></script>
  <![endif]-->
        <script type="text/javascript" src="javascripts/jquery.js"></script>
        <script type="text/javascript" src="javascripts/utilajax.js"></script>
        <script type="text/javascript" src="javascripts/inhibit.js"></script>
        <script type="text/javascript" src="javascripts/ajaxCaller.js"></script>
        <script type="text/javascript" src="javascripts/xmlsax.js"></script>
        <script type="text/javascript" src="javascripts/SaxEventHandler.js"></script>
        <script type="text/javascript" src="http://invariant.zepheira.com/exhibit/api-
2.0/exhibit-api.js?autoCreate=false"></script>
        </head>
        <body>
            <h2>Inhibit Demonstration</h2>
            <div id="inhibit-query" style="display: none;"></div>
                <table width="100%">
                    <tr valign="top">
                        <td ex:role="viewPanel">
                            <div ex:role="view"></div>
                        </td>
```

```
                        <td width="25%">
<div ex:role="facet" ex:expression=".discipline" ex:facetLabel="Discipline"></div>
<div ex:role="facet" ex:expression=".relationship" ex:facetLabel="Relationship"></div>      <div
ex:role="facet" ex:expression=".shared" ex:facetLabel="Shared?"></div>       <div ex:role="facet"
ex:expression=".deceased" ex:facetLabel="Deceased?"></div>
                        </td>
                    </tr>
                </table>
            </body>
    </html>
```

B.2 inhibit.js JavaScript Library

```
// ===================================================================
//
// inhibit.js - Provide functions to extend MIT Simile's Exhibit HTML to
//              throttle data sets by query so they don't overwhelm Exhibit.
//
// version 1.0, 03 June 2008
// David Hyland-Wood (dwood at http://itee.uq.edu.au/)
//
// ===================================================================
//
// Copyright (C) 2008 Zepheira, LLC (http://zepheira.com)
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//   http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
//
// ===================================================================
//
// Design Notes:
//
// This library works with an Active PURL to "negotiate" the number of query
// results from a Web service until the number of results is less than
// exhibitMaxCount, then an Exhibit is built to handle the results.
// There are four steps:
// 1) HTTP HEAD on <URL> to see if it is an Active PURL.
// 2) HTTP GET on <PURL> to get the query parameters
// (used to build an HTML query UI).
// NB: In this version, an HTML fragment is returned from the Active PURL.
// TODO: Handle XML from the Active PURL and perform XSLT in the client.  See:
// http://johannburkard.de/blog/programming/javascript/xslt-js-version-3-0-released-xml-xslt-
jquery-plugin.html
// 3) HTTP HEAD on <PURL> with query string input to get a count of the results.
//    When the count is less than exhibitMaxCount, proceed to Step 4.
// 4) HTTP GET on <PURL> to be redirected to the Web service for final query.
// ===================================================================
// Avoid Mozilla/Firefox "same origin" policy for some servers.
//document.domain = "zepheira.com";
```

```
// Document globals
var dataURL;  // The URL of the Web service providing data to load into Exhibit.
var countURL; // The dataURL with a query string of parameters as defined by the user.
var exhibitMaxCount = 1000;

// Executed on document load.
// NB: Can't use JQuery's $(document).ready() here because it is in an external script.
window.onload = function(){
// # (1a)

		// Get the data URL.
		dataURL = document.getElementById("inhibitdata").href;

		// Retrieve query metadata from the URL, build and display the query form.
		if ( dataURL != null ) {
			checkIfActivePURL();
		} else {
			alert ("No data URL found.  Nothing to do.");
		}
}

// Check the URL to see if is an Active PURL.
function checkIfActivePURL() {
// # (1b)

		// Check the HEAD response from the URL to see if
		// it contains a PURL header and, if so, whether it is an Active PURL.
		ajaxCaller.head(dataURL, null, onCheckIsActivePURLResponse, false,
"checkIsActivePURL");

}

// Callback for checking whether a URL is an Active PURL.
function onCheckIsActivePURLResponse (message, headers, callingContext) {
// # (1c)

		if ( headers["X-Purl"] && headers["X-Purl"].indexOf("(Active)") > -1 ) {

			// Build a query interface for the PURL's Web service.
			getQueryMetadata();
		} else {
			alert("The URL " + dataURL + " is not an Active PURL.  It cannot be
used with Inhibit.");
			reportHeaders();
		}
}

// Gets HTML representing a query interface for a PURL.
function getQueryMetadata() {
// # (2a)
```

```
		// GET its query metadata by retrieving <URL>.
		// The query metadata should be in XML.  Run it through
		// the inhibit.xsl transformation to build an HTML
		// fragment representing a query interface.
		// NB: Safari does not support in-browser XSLT calls
		// from Javascript :(
		// (http://developer.apple.com/internet/safari/faq.html#anchor21)
		// Either call a Web service to do the transformation or have
		// the Active PURL optionally return HTML fragments...
		signalLoadingResults();
		ajaxCaller.get(dataURL, null, onGetQueryMetadataResponse, false,
"getQueryMetadata");
}

// Callback for getting a metadata form from the Active PURL.
function onGetQueryMetadataResponse (message, headers, callingContext) {
// # (2b)

		// Make sure we only accept the expected HTML fragment.
		if ( message.length > 0 ) {

				document.getElementById("inhibit-query").innerHTML = message;

		} else {
				// Throw an error.
				alert("Error: Bad content received.  Aborting." + "  Content-Type: " +
headers["Content-Type"]);
				// DBG
				var report;
				for (key in headers) {
						report += "[" + key + "] -&gt; [" + headers[key] + "]<br/>";
				}
				document.getElementById("inhibit-query").innerHTML += report;
		}
		setVisibility("inhibit-query", "inline");
}

// Submits a query and gets a count of results.
function getCount() {
// # (3a)
		// Create the appropriate query string from the form elements.
		countURL = dataURL;

		jQuery('.active_purl').each(function(index, element) {
				countURL += (index == 0 ? "?" : "&");
				var id = jQuery(this).attr('id');
				var value = jQuery(this).val();
				countURL += escape(id) + "=" + escape(value);
		});
```

```
	ajaxCaller.head(countURL, null, onGetCountResponse, false, "getCount");
}

// Callback for getting a count of query results.
function onGetCountResponse (message, headers, callingContext) {
// # (3b)

	if ( headers["X-Purl-Count"] ) {
		if ( headers["X-Purl-Count"] <= exhibitMaxCount ) {
			// # (4)

			// Load Exhibit.  Thanks to David Huynh for this invocation.
			var fDone = function() {
		window.exhibit = Exhibit.create();
		window.exhibit.configureFromDOM();
			};
			window.database = Exhibit.Database.create();
			// Load the data URL into Exhibit.
			window.database._loadLinks(
		[ countURL ],
		fDone
			);

			// Hide the Inhibit divs.
			setVisibility("inhibit-count", "none");
			setVisibility("inhibit-query", "none");
		} else {
			// Show the count and leave the form so the query may be
modified.
			document.getElementById("inhibit-count").innerHTML =
"The current query results in " + headers["X-Purl-Count"] + " records, which is too large to
display in Exhibit.  Please refine your query until the results are less than " + exhibitMaxCount;
			setVisibility("inhibit-count", "inline");
			setVisibility("inhibit-query", "inline");
		}
	} else {
		// Throw an error.
		alert("Error: Bad content received.  Aborting.");
		// DBG
		reportHeaders();
	}
}

/*
 * Utility Methods
 */
// Sets the visibility of an HTML element.
function setVisibility(id, visibility) {
	document.getElementById(id).style.display = visibility;
}
```

```
function signalLoadingResults() {
	document.getElementById("inhibit-query").innerHTML = "<p>Loading ...<\/p>";
}

function reportHeaders() {
	var report;
	for (key in headers) {
		report += "[" + key + "] -&gt; [" + headers[key] + "]<br/>";
	}
	document.getElementById("inhibit-query").innerHTML += report;
	setVisibility("inhibit-query", "inline");
}
```

Appendix C. Survey Questions

A small-scale survey of software maintenance information is described in Chapter VIII. This appendix provides the questions asked of survey participants. Section C.1 provides questions relating to demographics of the participants and Section C.2 provides questions regarding metadata describing a sample Java class. The metadata for the sample Java class is presented at http://www.itee.uq.edu.au/~dwood/sampledata/PURLSAccessor.html.

C.1. Demographic information

Questions marked with an asterisk (*) are required.

* 1. Please enter the country in which you reside. This information is desired, but optional.

Please enter a (legitimate!) email address. You will not be contacted and your email address will NOT be sold. This information is for survey demographic purposes only.

Country:
Email Address:

* 2. Do you program in the Java programming language?

Multiple choice: No answer, Yes, No

* 3. Do you maintain Java source code?

Multiple choice: No answer, Yes, No

C.2. Questions regarding the class org.purl.accessor.PURLSAccessor

All of the questions on this page refer to the Java class org.purl.accessor.PURLSAccessor that may be found at http://www.itee.uq.edu.au/~dwood/sampledata/purlz-exhibit.html#org.purl.accessor.PURLSAccessor

* 1. Information about the org.purl.accessor.PURLSAccessor class includes links to requirements associated with the class (the "encodesRequirement" links).
How helpful would this information be to you if you were performing software maintenance tasks on this class?

Multiple choice: Decline to answer, Unable to tell, Not valuable at all, Somewhat valuable, Very valuable, Essential

* 2. Information about the org.purl.accessor.PURLSAccessor class includes links to the history of changes to the class in a revision control system (the "seeAlso" links).
How helpful would this information be to you if you were performing software maintenance tasks on this class?

Multiple choice: Decline to answer, Unable to tell, Not valuable at all, Somewhat valuable, Very valuable, Essential

* 3. Information about the org.purl.accessor.PURLSAccessor class includes links to other classes associated with the class (the "uses" links).
How helpful would this information be to you if you were performing software maintenance tasks on this class?

Multiple choice: Decline to answer, Unable to tell, Not valuable at all, Somewhat valuable, Very valuable, Essential

* 4. Information about the org.purl.accessor.PURLSAccessor class includes links to unit tests associated with the class (the "hasTest" links).
How helpful would this information be to you if you were performing software maintenance tasks on this class?

Multiple choice: Decline to answer, Unable to tell, Not valuable at all, Somewhat valuable, Very valuable, Essential

* 5. Information about the org.purl.accessor.PURLSAccessor class includes a link to the original developer of the class (the "hasDeveloper" link).
How helpful would this information be to you if you were performing software maintenance tasks on this class?

Multiple choice: Decline to answer, Unable to tell, Not valuable at all, Somewhat valuable, Very valuable, Essential

* 6. Is the information shown about the PURLSAccessor class better, worse or about the same FOR THE PURPOSES OF SOFTWARE MAINTENANCE as the information available in Javadoc or similar documentation systems?

Multiple choice: Decline to answer, Unable to tell, Not valuable at all, Somewhat valuable, Very valuable, Essential

* 7. If you were asked to perform software maintenance tasks on this project, would it help you to have the information presented about the PURLSAccessor class?

Multiple choice: Decline to answer, Unable to tell, Not valuable at all, Somewhat valuable, Very valuable, Essential