TAGGINGSENSE: Method Based On Sensemaking For Object-Oriented Source Code Comprehension

Daniel Schreiber
Post Graduate Program in Informatics (PPGIa) - Polytechnic School
Pontifícia Universidade Católica do Paraná – PUCPR
Curitiba, Brazil
xiraba@gmail.com

André Menolli
Computer Science Departament
Universidade Estadual do Norte do Parná - UENP
Bandeirantes, Brazil
menolli@uenp.edu.br

Sheila Reinehr, Andreia Malucelli
Post Graduate Program in Informatics (PPGIa) - Polytechnic School
Pontifícia Universidade Católica do Paraná – PUCPR
Curitiba, Brazil
sheila.reinehr@pucpr.br, malu@ppgia.pucpr.br

Abstract— **All software requires maintenance, either for error correction or for implementing updates. However, maintenance is often complex and expensive, and one of the main problems in the high cost of maintenance is the difficulty of understanding the source code of other authors. Thus, this research presents TaggingSense, a method based on sensemaking that aims to reduce object-oriented source code comprehension time on systems maintenance. Through experimentation, it was possible to observe knowledge extracted from the source code, processing, and sharing, to be positively assisted in the source code maintenance and comprehension process, thus bringing benefits such as reduction time spent, quality, and greater security in the changes made.**

Keywords-knowledge;sensemaking;source code maintenance; ontology.

I. INTRODUCTION

Software maintenance is one of the activities that consume substantial resources in software projects. In the mid-1980s, the total cost invested in maintenance and improvement accounted for over 60% of the total cost of software systems [1]. In contrast, in the 2000s, total maintenance cost exceeded more than 90% [2]. Maintenance is inevitable because we must ensure updated and efficient software, and this activity is performed for various reasons, such as changes in requirements, bug fixes, component modifications, software improvement, source code optimization, and efficiency improvement, among others [3].

Among several proposed techniques and processes to improve software maintenance, some studies explore cognitive aspects related to software comprehension. With source code being the main maintenance component, comprehension is the predominant factor for providing effective software maintenance, thus allowing the development of computerized systems [4].

Software comprehension corresponds to activities that people perform in order to understand, conceptualize, and reason about software [5]. It is estimated that developers dedicate an average of 40% to 90% of the maintenance effort to the software comprehension process [6] [7]. One of the possible reasons for difficulty in source code comprehension is the lack of knowledge by people without experience, as well as by programmers from other fields.

One method to build knowledge and make sense of things is through sensemaking. Sensemaking is the process of turning circumstances into situations that can be comprehended explicitly in words, and that serves as a catalyst for actions [8]. Weick [8] considers labeling (assigning explicit names) an essential step in sensemaking.

In maintenance activities, it is in the analysis and comprehension stage that those involved do work to extract knowledge and use it to continue with maintenance. During this activity, the acquired knowledge is conserved in people's memories, and such knowledge is divided into two classes: syntactic and semantic [9]. Both semantic and syntactic knowledge are directly and indirectly related to source code comprehension. Many studies and models of comprehension identified different types of knowledge, including knowledge of programming, knowledge of real-world situations addressed by software, and knowledge of the application domain [10].

After comprehension, the coding activity, a process through which developers declare their intentions for the computer, is performed. This activity implies high processing power and storage in the memory of people, because, in addition to the domain, developers need to visualize the organization of objects and routines, as well as the data flow [11]. These challenges, coupled with the effort applied to maintenance and the absence of an ideal solution to these problems, led to the development of this research. It is believed that a comprehension method applied to the source code related to the extraction and dissemination of knowledge can assist in the comprehension process, thus reducing uncertainties and the time dedicated to maintenance tasks.

Therefore, this study aims to develop a method based in sensemaking to reduce object-oriented source code comprehension time on system maintenance. More specifically, it is intended to answer the following question: Is it possible to reduce the time and effort of source code comprehension, and thus increase the quality and efficiency of software maintenance?

II. RELATED LITERATURE

Of all the activities involved in the process of maintenance, comprehension is the most important, as it is considered to be the essential basis for modifying a software product [12]. Studies

DOI reference number: 10.18293/SEKE2015-038

show that efforts applied on maintenance are mainly targeted to the comprehension part [11].

Several works were developed related to software maintenance and comprehension, not all of which are focused on serving the same purpose. However, these studies use similar techniques for working on the source code. For example, in research [10], the complexity of understanding a program at the time of maintenance was studied for the purpose of calculations and estimates of effort metrics. Work [13] identified two levels of comprehension: syntactic and semantic. The work proposed in [14], by means of cataloging source code, already seeks to discover programmers' knowledge on application domain. [12] explored a method for maintaining software engineering artifacts "connected" through semantic connections, starting from the source code, by means of ontologies. Work [15] proposes a union of the ontology of code knowledge with domain knowledge, and lastly, work [16] developed source code and documentation ontology to assist in the comprehension process through complex searches inferred on ontology populated from text mining applied in the source code.

The use of ontologies has been significantly explored in software maintenance activities for much of the works highlighted here. Among the techniques for applying ontology to the source code, this paper proposes a new approach: using ontology as a consequence of the knowledge extracted from the source code by using the sensemaking technique. Based on sensemaking, we propose the development and implementation of a method with the principle of formalizing and implementing a folksonomy within the source code, so that it is possible to extract knowledge and maintain it in a knowledge base, with the goal of extracting and disseminating both domain knowledge and the features contained in the source code.

III. TAGGINGSENSE METHOD

In this section, we present the "TaggingSense" method, which supports the steps and the intrinsic processes involved in source code comprehension during software maintenance. This method combines the tagging concepts of folksonomy, and the stages and processes identified by sensemaking, with the goal of accelerating and improving the comprehension process of unknown source codes.

A. *Method Structure*

From the eight stages of sensemaking (Organizes Flux, Noticing and Bracketing, Labeling, Retrospective, Presumption, Social and Systemic, Action, and Organizing through Communication) conceived by [8], four activities have been defined for the proposed method, and are described as follows:

- Observation: consists of the superficial analysis that a programmer performs when starting the maintenance activity. Owing to such observations, in this activity, the programmer formulates ideas and structures based on the experience of past projects.
- Extraction: activity related to the extraction and development of knowledge contained in the source code. This activity starts sensemaking. Knowledge is formalized and archived by the programmer with the source code.
- Organization: organizing and structuring the extracted knowledge. This activity consists of provided support and the support or rejection of raised ideas and hypotheses in order to improve knowledge structuring. It is in this activity that the programmer identifies phenomena and observed patterns, improves externalization, and catalogs the acquired knowledge.
- Collaboration: the main component of this activity is communication. In this activity, the sharing and development of knowledge with the group of people involved in the process occurs through the exchange of experience and the refinement of learning.

Based on the activities defined, details of the method and the steps involved in each activity are described in Table I.

In total, four activities were created with a subtotal of 15 interrelated steps. Each activity has a purpose that serves as input to generate a specific output. The outputs generated by the activities are: (i) formulation and structuring of ideas and hypotheses (observation activity): ideas are formulated and structured tacitly, where externalization occurs in the execution of the next stage; (ii) formalized knowledge (extraction activity): transformation of tacit knowledge into explicit; (iii) restructured and organized knowledge (organization activity): this activity organizes knowledge in a structured way, and enriches existing knowledge with more information; and (iv) knowledge base (collaboration activity): this is the location where all knowledge extracted from the source by one or more programmers is stored.

B. *Knowledge Representation*

The TaggingSense method proposes the use of a folksonomy-based ontology for organizing and managing tags. In [27], the authors defined folksonomy as the result of a personal free marking (tag) of information and objects for retrieval. The use of tags through folksonomy fits best to factor a demonstration of human thought, compared with those methods related to automatic extraction of text [17]. Through a manual process, the user develops source code sensemaking and identifies a topic/knowledge through tagging. In this process, folksonomy is the result of the sensemaking process designed by the user. One of the strengths of folksonomy is the free assignment of words to features. Annotating a feature with multiple keywords requires less cognitive effort than selecting a single category [18].

Folksonomy is represented through ontology, which serves as basis for supporting the processes. This helps to solve the main problems of folksonomy, such as synonyms, ambiguities, and searches. The main ontologies developed to support the tagging process were evaluated, such as Newman [19], SCOT [20], MOAT [21], Knerr [22], and NAO [23]. After analyzing the available ontologies, the ontology of Knerr [22] was chosen, due to its better compliance with the requirements of the problem, its availability, and easy access to documentation of their classes and properties.

TABLE I. TAGGINGSENSE METHOD ACTIVITIES AND STEPS

Activities	Steps	Description
Observation	**Structure analysis**	Preliminary study of the structuring of the source code.
	Technical knowledge and domain search	Improvement of technical knowledge in relation to the source code structure, such as programming language, paradigms, architecture, and standards, in addition to complementary studies related to the domain.
Extraction	**Knowledge extraction**	Development of domain concepts. Assimilation between domain issues and technical issues related to the source code.
	Tagging	Marking source code through tags. Use of folksonomy to assist, support, and organize tags created during knowledge extraction.
	Externalization	Knowledge articulation occurs, i.e., transformation of tacit knowledge into explicit or usable knowledge. This task represents the continued task of Tagging.
	Guides	Improvement of source code tagging. Tagging is structured in a way that helps programmers find such markings in the source code through waypoints.
	Enrichment tags	Development of new concepts related to those already developed and identified by means of tags.
Organization	**Knowledge refinement**	Refinement of points related to application domain. Revaluation and continuation of "Knowledge extraction" task of previous activity.
	Tag re-evaluation	Importance validation with project and domain. Redundant tags are eliminated; common tags are reused in the project.
	Cataloging standardization	Standardization between the terms already created.
	Release	All new created tags have visibility property set to private because they are developed at this stage and can change or be eliminated by the creator.
Collaboration	**Storage**	Throughout the process, extracted knowledge is stored in a database called knowledge base, through ontologies meaning.
	Sharing	Database must be shared with everyone specifically involved in the process of project maintenance.
	Refinement	Enhancement and improvements in tags created by other programmers.
	Reuse	Reuse of tags created by other programmers.

IV. TAGGINGSENSE ENVIRONMENT

To support the TaggingSense method, we implemented an environment to allow tagging the source code in order to assist in its comprehension. The tagging process consists of manually extracting source code knowledge, and adding it in the folksonomy ontology. This information corresponds to keywords for the tag, date, time, and creator, in addition to the class, method, variable, or related code snippet, that can be inserted to the same tag created for other individuals. This environment was built as a plug-in for the Eclipse development IDE (integrated development environment). In this environment, interaction starts from the programmer's comprehension of the source code from the bottom to the top ("bottom-up") of the source code lines that represent the domain knowledge, through the identification of relevant chunks. Chunks are code portions that programmers can recognize. Large chunks contain several smaller chunks [16].

After this step, it is necessary for the source code to be processed and synchronized with the source code ontology. In the environment, SCRO (Source-code Ontology) is used as the source code ontology because it was created to support the main tasks of software comprehension through the explicit representation of conceptual knowledge found in the source code [24]. This synchronization consists of the extraction of information from the project's source code, such as methods, input and output values of each method, attributes, and classes, and population of the source code ontology.

Once the source code ontology is populated, the next step is to interact with the folksonomy ontology. This allows new individuals created in this ontology through the creation of tags by the programmer to be associated according to the instances of individuals of the source code ontology. Lastly, the process results in a knowledge base that contains all created tags and their respective associations, derived from the domain knowledge received from the source code. The knowledge base consists of the very folksonomy ontology populated and inferred by inference mechanisms. The environment implementation is presented in the next subsection.

A. *Environment Implementation*

The environment was implemented according to the assumptions of sensemaking, folksonomy, and knowledge base. In addition, six implementation requirements were raised to support the source code comprehension process. They are:

- Requirement 1: query and record domain information in the folksonomy ontology. Information refers to the knowledge acquired during the comprehension process, and should be semantically linked to allow queries and inferences (reasoning).
- Requirement 2: populating source code ontology. The plug-in must provide a method for extracting semantic information from the source code and automatically populating the source code ontology.
- Requirement 3: populating folksonomy ontology. Populating the domain ontology, which corresponds to the tags created, must be performed manually. As a result of sensemaking, the source code comprehension process is best developed manually because it is at this moment that the user assimilates and understands the source code.
- Requirement 4: searches of instances in the ontology.
- Requirement 5: allows to create, connect, provide, identify, query, and share tags during the source code comprehension process.

- Requirement 6: integration with the working environment.

In order to automatically extract the source code and allow direct interaction with the user, the system was designed and developed based on Eclipse 3.6 and Java 6 platform.

The source code ontology is automatically populated by the plugin, through QDox library [25], whereas the tags manipulation is manual, according to user action. The source code is the only input software artifact, whereas the remaining entries in the system are through manual intervention. Queries by created and populated tags occur through SPARQL-DL with OWL-API support library because there was no native support for SPARQL queries during the development of this research.

Based on the requirements for extraction and manipulation of gathered knowledge, the TaggingSense plug-in was developed to manipulate ontologies and tagging in the source code, with the following functionalities: (i) Display tags related to the selected code: from a window, it is possible to analyze the relationship between the programming-related object and the associated domain concept (tag); (ii) Display tags in tree format: from the list of tags, it is possible to find the source code related to the selected tag; and (iv) Display use of all tags: list all public tags created by any person, in addition to private tags authored by the current user.

In addition to the features described, the plug-in allows the addition of new tags and makes the tags public, thus allowing other users to view the tags and use them collaboratively.

V. Experiment

To evaluate the feasibility of the method and the environment, an experiment was proposed with the goal of answering the initial question of this research: Is it possible to reduce the time and effort of source code comprehension, and thus increase the quality and efficiency of software maintenance?

To evaluate the experiment, three criteria were defined: (i) programmer behavior: evaluation based on observations from an expert who accompanied the experiment; (ii) development time: this was considered a metric to measure method efficiency; (iii) quality of maintenance performed: an assessment as to whether the requested improvements were implemented as expected.

To conduct the experiment, four IT professionals, who work in a midsize software company, were selected. The selected professionals belong to two distinct classifications: junior, professionals with less than five years of experience in OOP (Object-Oriented Programing), software architectures, design patterns, organization and best coding practices; and senior, programmers with equals or more than five years of experience in system development with knowledge of working on large, complex projects. The participants were requested to make two improvements to an existing system that was unknown to them. The system consisted of a salesforce automation project developed in Java language for mobile devices. Its initial release was designed to run on PALM OS, Windows Mobile, and Android devices. The experiment was divided into three parts, each part containing a specific purpose and applied to specific participants, as summarized in Table II. In addition, a maximum execution time for each maintenance task was stipulated.

TABLE II. Experiment description

	Participants	Objective	Procedure
Experiment 1	Junior A Senior A **Evaluation** Improvement time and location	Evaluate understanding difficulty and comprehending source code of other authors.	Same activity for participants with and without experience. Activity consists of making improvements to existing system. For this experiment, features for using tags were not available, only features offered by IDE.
	Participants	**Objective**	**Procedure**
Experiment 2	Junior B Senior B **Evaluation** Improvement time and location; name and number of new tags created during the process.	Evaluate comprehension of source code of other authors that performed tagging.	Same activity for both types of participants. Source code is not tagged, but developers are allowed to add, share, and use tags to assist maintenance process.
	Participants	**Objective**	**Procedure**
Experiment 3	Junior B Senior B **Evaluation** Maintenance time and quality; number of new tags created.	Evaluate comprehension of project already tagged by someone familiar with the project.	Repeat experiment 1 with project already tagged. Those involved should use tags as guides to reach system critical point, thus performing maintenance at correct location.

A. Results

Analysis of the results was performed mainly in a qualitatively manner. In this analysis, the purpose of the experiments was considered, and the experiments were designed so that a comparison could be made, as described in Table III.

In experiment 1, senior participant A showed difficulties when attempting to find the location (class/method) that caused the parameter to perform the validation requested for this experiment. However, he was able to perform the experiment successfully in 16 minutes, and executed the maintenance in the expected class and method. Junior participant A could not find the correct location of the maintenance in the stipulated time. Even after being shown the location where the maintenance should be performed, the participant failed to complete the task successfully within the stipulated time because, although the maintenance was performed correctly, the code was not implemented in the expected method.

In experiment 2, junior participant B did not use the plug-in as a support tool and could not find the correct method where the improvement should be implemented. Senior participant B achieved this improvement in 12 minutes, and did not need to receive any type of help or advice. However, neither senior participant B nor junior participant B implemented an improvement on the desired method and class.

TABLE III. EXPERIMENT COMPARISON

Relationship	Evaluation - Objective
Experiment 1 x Experiment 2	Check maintenance performance without the use of tags (experiment 1) and with the use of tags (experiment 2); evaluate performance of maintenance performed between junior programmers, among senior programmers, and between junior and senior programmers.
Experiment 1 x Experiment 3	Analyze performance of maintenance performed by senior programmer without the use of tags and by junior programmer with the use of tags.
Experiment 2 x Experiment 3	Evaluate impact on improvement maintenance when there are no tags; that is, comprehension is initiated without the aid of previously created domain concepts (experiment 2). Evaluate impact on improvement maintenance when tags are identified previously (experiment 3) and are available to assist in the comprehension process.

In experiment 3, participants had access to the tags. Junior participant B started the maintenance using the available tags. Through the tags, the class attribute that had the value that needed to be changed was easily deduced. After the locating task was performed all locations that called the attribute in question were searched by the programmer in the source code. Every item in each code snippet that was located was verified against the related tag. Junior participant B performed the activity in merely eight minutes, without any type of help or support. Compared with senior participant A who ran the same maintenance in experiment 1 without the aid of tags, junior participant B was faster because senior participant A performed the same maintenance in 16 minutes. In turn, senior participant B, who had access to the tags, implemented the proposed improvement in four minutes; half the time displayed by junior participant B. Table IV presents a summary of the maintenance time required by senior and junior programmers.

TABLE IV. COMPARISON BETWEEN TIME OF SAME MAINTENANCE WITH AND WITHOUT TAG

Participant	Without tags	With tags
Junior Group	30 min	8 min
Senior Group	16 min	4 min

VI. DISCUSSION

In experiments 1 and 2, tag features to be used in the comprehension process were not available to programmers. However, for experiment 3, the tags were made available to assist in the comprehension process. From the results, it can be concluded that sensemaking development is influenced heavily by the availability of features. The group of junior programmers who did not use tags required an average of 30 minutes to perform the proposed maintenance. However, through the tags, this time decreased to eight minutes, demonstrating a 74% productivity improvement in performance.

In the same sense, the senior group performed the same maintenance in 12 minutes, whereas by means of tags, this time decreased to four minutes, showing a gain of 75% for this class of developers

In experiment 2, wherein the tags were not available, but the possibility of creating and using them was offered, only the group of senior participants benefitted. However, the tags created were used as waypoints (identification of locations), and as memorization topics that were extracted from the source code. Thus, the created tags helped in source code navigation, assisting developers to locate code among the many classes and methods, avoiding them to get lost on source code navigation.

In contrast, in the experiment where the tags were already created and available, only the group of juniors added a new tag. The new tag served the same objective as for the other group, that is, as a waypoint.

We can conclude that in unfamiliar environments, extracting source code knowledge is easier for more experienced developers precisely because they have more experience. It was also observed that in environments where knowledge of the code was already present, senior programmers did not process new knowledge, whereas junior programmers were led by the existing tags, and even added a new related tag. The failure to process new knowledge puts in evidence the conclusion of the study by [26], which showed that there is no interest on the part of software engineers to study application domain knowledge when performing specific maintenance, where only knowledge related to software engineering (programming, development environment, and application implementation) are considered. The authors in [26] concluded that developers cultivate past knowledge, and searching for new knowledge is a costly process that is performed only when there is a clear need for the programmer and there is no easier alternative. According to [26], software engineers attempt to understand only what is necessary for a system to solve the current problem, and then tend to forget the details of what they learned.

Senior programmers in experiment 2 showed an average of 60% higher performance in the same experiment performed by the group of junior programmers. In this experiment, only the senior programmers used the feature for extracting knowledge from the source code. This justifies the fact that sensemaking is best developed when there already exist foundations and past experience [8].

However, as already discussed, in an environment where the knowledge contained in the source code is extracted previously by an expert with greater knowledge, and is made available via tags for those with less experience, a significant gain in performance is demonstrated.

Thus, we can conclude that the proposed method for extracting and sharing knowledge of the source code is sufficiently effective for improving overall performance of the development team.

VII. CONCLUSIONS

The software maintenance field is complex, mainly because it is dependent on a source code comprehension process, an activity that involves greater cognitive effort of the people involved. Several studies have been developed to facilitate code comprehension. However, this process can still be improved. Knowledge extracted directly from the source code through sensemaking is rich in important and valuable details that can be applied to source code comprehension. This knowledge can be best utilized when stored by means of ontologies and disseminated to more people using Semantic Web. With this process, the knowledge can not only be extracted, but also shared with those involved, thus benefitting the entire team. Through the results of our experiments, we demonstrated that the proposed TaggingSense method is viable because we were able to conclude that knowledge extraction, processing, and sharing assists positively in the process of source code maintenance and comprehension, thus obtaining benefits such as reduced time, increased quality, and greater security in the changes made. We also showed that our proposed method can guide programmers to the exact location of the improvement required, thus causing maintenance to not occur in wrong places that could affect the quality of the program or open the possibility for security breaches. Thus, the main issue of this research could be answered: it is possible to reduce the time and effort for source code comprehension during maintenance. However, we plan to extend the study to a larger number of participants. We also intend to evaluate the reaction of programmers with different educational backgrounds, as well as evaluate the question of the impact of personal and organizational culture and customs.

REFERENCES

[1] M. Zelkowitz, A. Shaw, and J. Gannon, Principles Of Software Engineering And Design. Prentice Hall Inc., , 1979, pp 157 – 178.

[2] L.Erlikh, "Leveraging Legacy System Dollars For E-Business," It Professional, Ieee, vol. 2, n. 3, 2000, pp. 17 – 23.

[3] B. B. Argawal, and S. P. Tayal, Software Engineering. Laxmi Publications: New Delhi, 2007.

[4] A. Mayrhauser, and A. Vans, "Program Comprehension During Software Maintenance And Evolution, " Ieee Computer vol. 28, 1995, pp. 44 – 55.

[5] W. Meng, J. Rilling, Y. Zhang, R. Witte, S. Mudur, and P. Charland, "A Context-Driven Software Comprehension Process Model," Ieee Software Evolvability Workshop, 2006.

[6] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding Function Behaviours Through Program Slicing, " In 4th Ieee Workshop On Program Comprehension, Ieee, 1996, pp. 9 – 18.

[7] A.Telea, and V. Lucian, "Visual Software Analytics For The Build Optimization Of Large-Scale Software Systems, " Computational Statistics, vol 26, n. 4, 2011, pp. 635 – 654.

[8] K. E. Weick, K. M. Sutcliffe, and D. Obstfeld, "Organizing And The Process Of Sensemaking, " Organization Science, 2005, pp. 409 – 421.

[9] B. Shneiderman, Designing The User Interface: Effective Strategies For Effective Human-Computer Interaction, 2rd ed. Addison Wesley, 1992.

[10] J. Yang, D. Hendrix, K. Chang, and D. Umphress, "An Empirical Validation Of Complexity Profile Graph, " In Proceedings Of The 43rd Annual Southeast Regional Conference, Acm, vol 1, 2005, pp. 143 – 149, 2005.

[11] C. L.Corritore, and S. Wiedenbeck, "Mental Representations Of Expert Procedural And Object-Oriented Programmers," In A Software Maintenance Task. In International Journal Of Human-Computer Studies, vol 50, n. 1, 1998, pp. 61 – 83.

[12] R. Witte, Y. Zhang, and J. Rilling, "Empowering Software Maintainers With Semantic Web Technologies," Springer-Verlag Berlin, n. 4519, 2007, pp. 37 – 52.

[13] C. Dasgupta, "That Is Not My Program Investigating The Relation Between Program Comprehension And Program Authorship," In Acm Se '10 Proceedings Of The 48th Annual Southeast Regional Conference, Acm, n. 103, 2010.

[14] R. Sindhgatta, "Identifying Domain Expertise Of Developers From Source Code, " In Proceeding Of The 14th Acm Sigkdd International Conference On Knowledge Discovery And Data Mining, Acm, 2008, pp 981-989.

[15] H. Zhou, F. Chen, and H. Yang, "Developing Application Specific Ontology For Program Comprehension By Combining Domain Ontology With Code Ontology, " In Quality Software, 2008. Qsic '08. The Eighth International Conference, Ieee, 2008, pp. 225 – 234.

[16] Y. Zhang, An Ontology-Based Program Comprehension Model. Doctoral Thesis, University Of Concordia, Canada, 2007.

[17] H. Al-Khalifa, and H. Davis, " Exploring The Value Of Folksonomies For Creating Semantic Metadata, " International Journal On Semantic Web & Information Systems, vol 1., n. 3, 2007, pp. 13 – 39.

[18] R. Sinha, "Tagging From Personal To Social: Observations And Design Principles," In Tagging Workshop, World Wide Web Int. Conf., 2006.

[19] R. Newman, Tag Ontology Design. Available On <Http://Www.Holygoat.Co.Uk/Projects/Tags/>. Accessed On 27 March 2012.

[20] H. L. Kim, A. Passant, J. G. Breslin, S. Scerri, and S. Decker, "Review And Alignment Of Tag Ontologies For Semantically-Linked Data In Collaborative Tagging Spaces, " In Proceeding Of The 2nd International Conference On Semantic Computing, Ieee, 2008. pp. 315 – 322.

[21] A. Passant, "Using Ontologies To Strengthen Folksonomies And Enrich Information Retrieval In Weblogs: Theoretical Background And Corporate Use-Case," In International Conference On Weblogs And Social Media, Boulder, United States, 2007.

[22] T. Knerr, "Tagging Ontology—Towards A Common Ontology For Folksonomies," Available On <Http://Tagont.Googlecode.Com/files/Tagontpaper.Pdf>. Accessed On 27 March 2012.

[23] S. Scerri, M. Sintek, and L. Van Elst, " Handshuch, S. Nepomuk Annotation Ontology (Nao)," Available On <Http://Www.Semanticdesktop.Org/Ontologies/Nao/>. Accessed On 28 March 2012.

[24] A. Alnusair, "Scro – Source-Code Ontology," Available On<Http://Www.Indiana.Edu/~Awny/Index.Php/Research/Projects-Tools/15-Research/Ontologies/10>. Accessed On 22 February 2012.

[25] Qdox: Qdox Java Parser Extractor. Available On <Http://Qdox.Codehaus.Org/>. Accessed On: 31 May 2012.

[26] M. R. Ramal, R. M Meneses, and N. A. Anquetil, "Disturbing Result On The Knowledge Used During Software Maintenance, " In 9th Working Conference On Reverse Engineering, Ieee, 2002, pp. 277 – 286.

[27] V. Wal, "Explaining And Showing Broad And Narrow Folksonomies, " Available On <Http://Www.Personalinfocloud.Com/2005/02/Explaining_And_.Html> . Accessed On 10 November 2011.