University of Alberta

EXTRACTING STRUCTURED KNOWLEDGE FROM TEXTUAL DATA IN SOFTWARE REPOSITORIES

by

Maryam Hasan

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Spring 2011
Edmonton, Alberta

Examining Committee

Eleni Stroulia, Computing Science

Denilson Barbosa, Computing Science

Ken Wong, Computing Science

Marek Reformat, Electrical and Computer Engineering

To my dear Mojtaba and Negar

Abstract

Software team members, as they communicate and coordinate their work with others throughout the life-cycle of their projects, generate different kinds of textual artifacts. Despite the variety of works for mining software artifacts, relatively little research has focused on communication artifacts. Software communication artifacts contain useful semantic information that is not fully explored by existing approaches.

This thesis, presents the development of a text analysis method and tool to extract and represent valuable information from a wide range of textual data sources from software projects. The developed system integrates Natural Language Processing techniques and statistical text analysis methods, with software domain knowledge. The extracted information is represented as RDF-style triples which constitute interesting relations between developers and software products. We applied our system to analyze five different textual sources, i.e., source code commits, bug reports, email messages, chat logs, and wiki pages. In the evaluation of the system, we found its precision to be 82%, its recall 58%, and its F-measure 68%.

Acknowledgements

First of all, I would like to thank my supervisor Dr Eleni Stroulia. Her guidance and experience have helped me to become a researcher. She gave me research freedom and support to explore and build up diverse ideas throughout my research as a graduate student.

I am also grateful to my co-supervisor, Dr Denilson Barbosa for his guidance and feedback on this research. I thank to his help and insightful comments during my graduate studies.

I would like to thank all the members of Software Engineering Research Lab (SERL) for all their help. In particular, many thanks are reserved to Marios Fokaefs, who kindly helped during the evaluation of the developed approach. As a domain expert, he selected a sample of sentences as our test set, and helped for the evaluation of our results.

I would like to thank the members of Stanford parser group for their great tool. Specially, I would like to thank Christopher Manning for his very helpful support.

Finally and foremost, my great appreciation goes to my parents and my husband for their endless love, support, and care that they have given me, and to my lovely daughter, for being a big part of my life.

Table of Contents

1 Introduction **1**
- 1.1 Motivation and Background . . . 1
- 1.2 The Research Problem . . . 3
- 1.3 Contributions . . . 3
- 1.4 Outline . . . 5

2 Related Work **7**
- 2.1 Ontology Learning for Software Domain . . . 7
  - 2.1.1 Summary . . . 9
- 2.2 Understanding the Software Project . . . 10
  - 2.2.1 Summary . . . 13
- 2.3 Understanding the Communications among Software Developers . . . 14
  - 2.3.1 Summary . . . 15

3 A Method for Analyzing Textual Artifacts of Software Projects **17**
- 3.1 Data Pre-processing . . . 18
  - 3.1.1 General Data Cleaning . . . 20
  - 3.1.2 Bug Reports . . . 21
  - 3.1.3 SVN comments . . . 21
  - 3.1.4 Wiki Pages . . . 22
  - 3.1.5 Email messages . . . 22
  - 3.1.6 IRC Chat Logs . . . 23
- 3.2 Term Extraction . . . 25
- 3.3 Syntactic Analysis . . . 29
- 3.4 Semantic Analysis . . . 32
  - 3.4.1 Semantic Annotation . . . 33
  - 3.4.2 Semantic Representation . . . 35
- 3.5 Pattern Extraction . . . 37

4 Evaluation **41**
- 4.1 Experiments . . . 41
  - 4.1.1 The Input Dataset . . . 41
  - 4.1.2 Experimental Results . . . 42
- 4.2 Evaluation . . . 45
  - 4.2.1 Empirical Evaluation Method . . . 46
  - 4.2.2 Evaluation Results . . . 49
- 4.3 Discussion . . . 51

5 Conclusions and Future Work **53**
- 5.1 Conclusions and Contributions . . . 53
- 5.2 Future Work . . . 56

Bibliography **58**

List of Tables

3.1 Data cleaning and acquisition 24
3.2 Created Domain Vocabulary, Including the Concepts and the Corresponding Sample Terms. 29

4.1 Syntactic/Semantic Analysis Results for the Wikidev2.0 data sources 44
4.2 Samples of Comparing Extracted Triples with Reference Triples . . 49
4.3 Comparison Results for the Extracted Triples 50
4.4 Evaluation Results of the Tool for each Data Source 50

List of Figures

3.1 Tool Architecture 19
3.2 Sample Result of Data Preprocessing on an Email message 25
3.3 Constituency tree for the sentence ”I used Java and Eclipse before.” 30
3.4 Dependency tree for the sentence “I used Java and Eclipse before.” 30
3.5 Augmented Dependency tree for the sentence “I used Java and Eclipse before.” 30
3.6 Stanford Parsing result for the sentence “I used Java and Eclipse before.” 32
3.7 The sentence “I used Java and Eclipse before.” after parsing and semantic annotation. 36
3.8 The semantic relations of the RDF triples produced by the pattern-extraction process. 38
3.9 Dependency paths in the XML tree to represent Rule 1, Rule 2, and Rule 3. 39

List of Plates

...

List of Symbols

...

Chapter 1

Introduction

1.1 Motivation and Background

During software-development activities (i.e. requirements analysis, system design, implementation and testing), software developers continuously communicate with each other to coordinate their work. Coordination becomes particularly important when developers have to work together on complex, long-term projects [3].

As software team members work and communicate with each other, their exchanges get stored in a collection of heterogeneous artifacts. Some of these artifacts are composed of more structured data (e.g., source code), yet others include unstructured information (e.g., documentation in natural language).

The software research community has long recognized the importance and potential usefulness of the information contained in these artifacts and has developed several approaches for analyzing them [41, 5, 33]. These approaches have been applied to many software engineering problems such as reverse engineering, traceability, program comprehension, software reuse, software maintenance and recovery [28]. The most interesting findings, and those most directly relevant to our own work, are reviewed in Chapter 2.

In spite of the predominance of natural-language text around software development activities, most research in the area of "mining software repositories" is focused on analyzing the more structured artifacts of the process, usually source code and bug reports. For example, the work presented in [42, 41] examines data from source code for project comprehension, but do not analyze other data sources.

However, the knowledge about a software project is usually spread in different data sources, namely the bug descriptions, discussion messages, documentation, and commit logs. There are quite a few research projects on analyzing multiple data sources associated with software development. However these approaches either ignore the semantic information that can be found in the text included in these data sources, or employ supervised techniques, assuming that someone will manually annotate (part of) the text in order to train a system to extract such semantic information. For example, the approaches presented in [3, 10] don't extract any semantic relations between domain terms; rather they rely on extraction and annotation of terms and concepts. To mention an example of supervised methods, [21, 22] manually classified at least 2000 commit messages in order to create the training set for the automatic classification of commit messages. As another example, [3] used a combination of hand-written parsing rules for ontology population.

Earlier research has demonstrated that the textual sources attached to the software artifacts contain a wealth of domain knowledge that can become a valuable resource of information for current and future members of the software community [10]. Specifically, the content of the developers' informal communications is a rich source of additional information that complements, and often elaborates on, the formal documents. For example, through exchanges over communication channels such as email and chat, developers discuss the requirements of their software system and the details of system design. They negotiate the distribution of tasks among them and make decisions about the internal structure and functionalities of code modules. Finally, they discuss ideas for eliminating bugs or the best way to implement new features [19].

In the rest of this chapter we state the research problem that we addressed in this thesis, we describe our contributions to the state of the art, and we provide an outline of the thesis.

1.2 The Research Problem

Although software development artifacts implicitly contain information about different aspects of the software development, they are weakly integrated, unstructured and hard to explore [10].

While previous research demonstrates the feasibility of the idea that important knowledge can be extracted from software artifacts, the nature of such artifacts should be taken into account. Indeed, software projects produce large, distributed, heterogeneous, and unstructured information sources. We believe that a logical next step is to provide methods that can explore the knowledge provided by the complete range of heterogeneous information sources associated with software projects. We need a way to explore the semantics of information on the variety of software artifacts, including the bug reports, discussion messages, documentation, and commit logs. A major challenge in achieving this goal is that they generate vast amounts of information as a result of their interactions, but that information is not well structured. The following requirements should be considered by the methods that explore software artifacts [10]:

1. Ability to deal with multiple document collections;

2. Coverage of heterogeneous data sources; and

3. Ability to deal with unstructured data sources.

Our work in this thesis develops an initial approach towards addressing this research problem and makes several contributions, as summarized in the next section.

1.3 Contributions

In this thesis, we developed a text analysis method and tool to extract and represent useful pieces of information from multiple data sources associated with the software project. Text analysis is commonly known as a knowledge discovery process that aims to extract information from unstructured text [44]. We proposed and validated the use of lexical, syntactic, and semantic analysis to extract semantic information

from software textual data and represent them using a structured and formal data model such that they can be queried by end users. Extracted semantic and structural information reduce the time and effort to obtain a comprehension level required for the software maintenance. The extracted information can also be combined with the source code information to better support various software maintenance tasks and activities [34, 35]. The major contributions of this thesis are as follows.

1. **Analyzing a wide range of textual data sources around software development**

   Despite the variety of works in the area of mining software artifacts, relatively little research has focused on the diverse set of communication artifacts, explored in this thesis. Software communication artifacts, in addition to source code artifacts, contain rich semantic information that is not fully explored by existing approaches. The proposed approach exploits a wider range of textual information sources associated with software projects, including, source code commits, bug reports, email messages, chat logs, and wiki pages. Section 3.1 describes the specific features of each data source and the challenges for analyzing it.

2. **Applying unsupervised methods**

   Our approach for analyzing textual sources around software artifacts relies on unsupervised methods that don't require the specification of rules or training sets. Our text-analysis system incorporates Natural Language Processing (NLP) techniques and statistical text-analysis methods, aware of software domain knowledge, and applies these methods in an integrated manner to five different sources of software artifacts, in order to understand the semantics conveyed by the artifacts. Within the text-analysis process, we make use of a number of standard NLP tools, including sentence splitter, Part-of-Speech (PoS) tagger, and dependency parser. We provide more details in Section 3.2.

3. **Applying a deep syntactic and semantic analysis**

In order to explore software project information, we apply a combined syntactic and semantic analysis on textual artifacts to extract valuable knowledge about various aspects of the software life-cycle. We propose a term extraction and semantic annotation method to obtain information about individual software terminologies mentioned in the text. The developed system also detects predicate-argument structures in order to extract the relations between the identified terms. Our approach attempts to obtain semantic information as RDF-style (Resource Description Framework) triples [1] from unstructured textual sources. The extracted triples constitute instances of a rich conceptual model of the domain that captures interesting relations between developers and software products. Sections 3.3 and 3.4 contain a detailed description of these analyses.

As a result, a deep analysis of such textual artifacts reveals valuable information about various aspects of the software life-cycle: the background and expertise of developers (e.g., "I have used JUnit before"), the roles and responsibilities assumed by the various team members (e.g., "I focused on testing for this package"), and their specific contributions to the various project artifacts (e.g., "I added method M to class C"). Furthermore, the informal nature of such communication channels can also offer ways of extracting more subtle information about the dynamics among a team's members (e.g., "who communicates the most/least and with whom", "who asks the most questions").

Our final goal is to explore a series of activities around the software code. These include the discovery that a bug exists or that a new functionality is needed, determining those active developers working on an issue, identifying a possible solution to the issue.

1.4 Outline

The rest of this thesis is organized as follows.

In Chapter 2, we review other researches related to analyzing software textual

[1] (http://www.w3.org/RDF/)

artifacts for different purposes including (a) ontology learning from information sources associated with software artifacts, (b) natural-language analysis for understanding the life-cycle of software projects, and (c) understanding the communications around software developers.

In Chapter 3, we provide the details of the proposed approach for analyzing textual information of software artifacts. We describe our framework and different tasks that should be accomplished including syntactic parsing, term extraction, semantic annotation and representation, and pattern extraction.

In Chapter 4, we present our evaluation of our framework. We describe our experiment on the Wikidev2.0 data sources. Moreover, we present an empirical evaluation method and describe the evaluation metrics that we employ. We use these metrics to assess the results of our experiment.

Chapter 5, provides a summary of the main points of our work and the results of the evaluation process. It also discusses some possible future works for this research.

Chapter 2

Related Work

There has been a significant amount of work in the context of analyzing software textual artifacts for different purposes including (a) ontology learning from information sources associated with software artifacts, (b) natural-language analysis for understanding the life-cycle of software projects, and (c) understanding the communications around software developers. In this chapter, we discuss the literature that is most closely related to our thesis in each of the above groups.

2.1 Ontology Learning for Software Domain

There are different works for ontology learning from information sources associated with software artifacts. In the following we review some of them which are more important and related to our work.

K. Bontcheva and M. Sabou [10], presented an approach for learning domain ontologies from multiple sources associated with the software projects, i.e., source code, user guides, and discussion forums. They used the natural language processing services provided by the GATE (General Architecture for Text Engineering) [13], which is one of the most widely used NLP tools. Their technique relies on unsupervised learning methods that are portable across application domains and consists of four stages. Firstly, relevant terms are extracted from source code. Then the domain terms identified in the source code are used as a starting point for exploring unstructured sources which include documentations and forums. Thirdly, the terms extracted from the source code and textual artifacts are submitted for

a concept identification process to identify terms referring to the same concept. Finally, the automatically learned ontology is modified manually by an ontology editor in order to remove the irrelevant concepts and add missed ones. Although their approach provide ontology learning from multiple data sources, they don't extract any semantic relations between domain terms; rather it relies on extraction and annotation of terms and concepts.

A similar approach was used by Y. Zhang, R. Witte and their colleagues [44, 37] to find the relations and dependencies among various software artifacts as traceability links, based on an ontology learning technique. To achieve this goal they utilized structural and semantic information in software artifacts, i.e., the documentation and source code, by means of text mining and source code parsing, and created a formal ontological representation for both software artifacts. In order to populate the source code ontology, they developed an ontology-based program comprehension tool, SOUND (Software Ontology for UNDerstanding), which identifies concept instances from source code. They implemented a text mining subsystem based on the GATE framework, to extract concept instances from software documents. After representing source code and documents in the form of an ontology, these ontological representations, along with semantic information conveyed by the artifacts, are then used to establish traceability links between software implementation and documentation at the semantic level. They tried to link instances from source code and documentation using ontology alignment techniques.

More recently, they extended their previous research to empower software maintenance with ontology [40]. They first defined an ontological representation model as RDF for the software artifacts (i.e., source code and documents), in order to map documentation to source code. The source code ontology contains concepts mapped to the source code entities and the document ontology contains concepts in programming domain, including programming languages, design architecture, data structures, and algorithms. Their ontological model, also include some semantic relations between concepts, such as "implements(class, interface)". Secondly, they deployed text mining and source code analysis to automatically populate the ontological model with concept instances, in a similar way as [44]. For the text

mining task, they deployed GATE, in order to perform sentence splitting, part-of-speech tagging, named entity recognition, coreference resolution, syntactic analysis and relation detection. For the relation detection subtask, they extracted predicate-argument structure which is similar to our approach. However they applied syntactic parsing and a number of predefined grammar rules, rather than dependency parsing. After the ontology populated from source code and documentation artifacts, it is explored, queried and reasoned by utilizing semantic web clients.

Another interesting project in this area is the work of Ankolekar [3]. They developed a prototype of semantic web model for the open source software (OSS) communities. They provided an ontology as a semantic model for bug reporting messages, source code, and developers. Their ontology contains several sub-ontologies including, code ontology, bugs ontology, developers ontology, and interactions ontology. In order to populate the ontology, they collected instances of concepts and roles from software artifacts such as source code files, bug reports, and CVS commit logs. Their ontology population is a semi-automated process and uses a combination of hand-written parsing rules and information extraction patterns. Their approach demonstrated the feasibility of capturing the semantics of different kinds of artifacts to help the task of bug resolution in OSS communities.

2.1.1 Summary

The research projects reviewed in this section is similar to our work in that they map information from software artifacts to a structured data model, such as ontology. Although they analyze information from a variety of repositories, such as source code repositories and bug reports, these projects ignore communication data sources, such as email messages, and chat logs, and do not extract any kind of semantic relations between concepts in the text, but focus solely on teh extraction of terms and concepts.

2.2 Understanding the Software Project

There is substantial research in the context of analyzing software artifacts for understanding the software project. In the following we review those, which are more important and related to our work.

T. Yoshikawa and his colleagues [42], proposed a technique for identifying the traceability links between a natural language sentence describing features of a software product and the source code of the product, using a domain ontology. The inputs of their system are the source code, domain ontology, and a description sentence in natural language. They first analyze the source code and extract a call-graph for the methods and their invocations. Then they extract the words exist in the sentence using stemming, unifying synonyms, and filtering stop words. Finally, they traverse the call-graph with the information of the words and the given ontology, to produce a subpart of call-graph as pieces of source code related to the input sentence. Although by using traceability links developers can potentially locate code fragments corresponding to documentation, this system requires the "domain ontology" as input which is a challenging task by itself.

Another similar work for analyzing the source code for program comprehension purposes presented by Würsch and his colleagues [41]. They developed an approach to answer questions about source code, such as "what are the subclasses of a class?", or "what fields are declared as this type?". They combined toolsets for source code analysis with ideas from the Semantic Web to enable developers to query software artifacts through natural language question answering. For this goal, they represented the information extracted from the source code with the Web Ontology Language (OWL). They first created an RDF graph data model of the key concepts of source code such as: packages, classes, methods, accesses, invocations. The created ontology then serves as input for a guided natural language interface in which the developers can query the data.

More recently, T. Fritz and G. Murphy [18] identified 78 interesting questions about software development through interviews with professional developers such as "What have developers been working on?", or "Who is working on what?". An-

swers to these questions require information from several data sources such as, source code, change sets, wiki pages, exception tracks, and test cases. They introduced an information fragment model and a prototype tool to automate the composition of different kind of data source related to software artifacts. They used the prototype tool to answer a set of eight questions, by composing information fragments, originating primarily from traditional repository-mining methods. The tool also examines textual data from Wikis but it does not do any substantial syntactic or semantic analysis; rather it relies on recognizing IDs and names.

Another project aiming at answering questions developers ask is Codebook [5], which is a framework for mining software repositories and connecting developers and their work artifacts. The Codebook creates a graph useful for answering end-user's questions about software development activities. First, a set of crawlers extract information from the project repositories (i.e., source code, work items, user directory) and store them in a database as a graph. Then, a set of paths created by domain experts is uploaded into database as regular expressions and used to discover the paths that exist in the graph. Finally, front-end applications use web services to query the database to answer end-user's questions. With respect to text, it simply recognizes names within textual objects and infers that the object in question "mentions" the named individual. One drawback of their framework is that it is based on a supervised technique and relies on the paths created by domain experts.

A. Hindle, and et al. [21, 22] proposed a method to automatically classify large commit messages into maintenance categories including, corrective, adaptive, perfective, feature addition and non-functional improvement. They applied several machine learners on features extracted from the commit data, such as frequency of the words in the commit messages, author of the commit, and modified modules. Their approach assumes that commit messages and the identity of authors provide learners with enough information to accurately and automatically classify large commit messages into maintenance categories. However one drawback of their approach is that they had to manually classify at least 2000 commits into the maintenance categories in order to create the training set for the automatic classification. They evaluated the results of the learners via 10-fold cross validation, and

achieved accuracies consistently above 50%. In the field of software evolution and process extraction, they published another similar work [23]. In order to characterize a projects behaviour, they classified revisions into four classes source code, testing, documentation, build. After extracting data for revisions and releases, they classified the revisions by aggregation and windowing. Their classification was based on the file types associated with software processes.

In other works, A. Hindle and his colleagues [20, 24], proposed a method to extract a set of independent topics from a corpus of commit-log comments. Their methodology to derive and label topics consists of several steps. First they gathered data from commit messages from source control repositories, by removing stop-words and using stemming. Then they extracted topics by applying Blei's LDA (Latent Drichlet Allocation) algorithm [8] on the word distribution of comment messages. In [20], they also applied unsupervised and supervised learning techniques to label extracted topics. For the unsupervised labelling they tried to associate each topic with a label from a list of keywords and related terms. They generated different sets of word-lists based on ISO9126 quality model [7], and Word-Net. For the supervised labelling they applied several machine learners on features extracted from the commit data, such as word counts per message. Their unsupervised method got the accuracy between 55% and 75%, while their supervised method achieved the accuracy between 60% and 80%. Their research tried to abstract software artifacts by their subject matter by these topics, however they only examined version control repositories and ignored other artifacts.

A similar work of topic labelling for software quality, proposed by Neil A. Ernst and John Mylopoulos [16]. They applied a software repository mining technique to investigate the software quality in software projects. Their assumption was that requirements for software quality can be conceived as a set of labels assigned to the conversations of project participants such as mailing list discussions, bug reports, and subversion commit logs. They labeled each message of conversations with appropriate software quality requirements. Their quality model was derived from ISO9126 [7] which is a standard for software quality and describes six quality requirements, reliability, usability, maintainability, functionality, portability, ef-

ficiency. For each quality label, they first construct a word list using Wordnet's synsets, hypernyms and stems. Then, they query the corpora of the software project with these word lists to create a table of events. Each event is any message which contains at least one term in the word lists. They evaluated their results based on precision and recall. They randomly selected 100 messages for each quality requirement, and achieved the precision as 79% and recall as 51%.

Finally, A. Hindle and his colleagues [25], extended their previous work [24, 21] and presented an approach for recovering software development processes from several software artifacts including, mailing list archives, commit log comments, and bug tracker reports. In order to extract the processes used by developers, they proposed a semi-automatic technique for analyzing software artifacts. Their methodology involved the following steps: acquisition and extraction, unsupervised analysis, manual annotation, supervised analysis, and results reporting. The acquisition and extraction step attempts to extract data from software artifacts including version control, mailing lists, and bug trackers. The unsupervised analysis creates word bags, using topic analysis and frequency analysis. Word bags are dictionaries of terms related to a concept such as efficiency or requirement. They are used for labelling of bug reports or commit comments with predefined concepts. The annotation step uses word bags to manually annotate bug reports or commit comments with predefined concepts. For the supervised analysis they used Bayesian classifier to label topics extracted during unsupervised analysis with non-functional requirements such as, efficiency, reliability, usability, maintainability, portability. topic labelling and maintenance classification.

2.2.1 Summary

The works that we reviewed in this section examine data from software artifacts for the purpose of project comprehension. However these works do not do any substantial syntactic or semantic analysis of the software artifacts to deeply explore the semantic information that can be found in textual units of these data sources; rather they rely on recognizing names and assigning labels. Another drawback of the reviewed approaches is that some of them are based on supervised techniques

which need some manual training data sets.

2.3 Understanding the Communications among Software Developers

There exists an interesting body of research in the context of analyzing software communication artifacts, to discover developer roles, contributions, and associations in the software development. In the following we introduce some of them which are more related.

Rigby and Hassan [36] used the LIWC, to examine the mailing list of Apache http server developers. LIWC, is a psychometrically-based analysis tool, similar to the TAPoR. They conduct three preliminary experiments to assess the appropriateness of this tool for information extraction from mailing lists. Their goal was to gain insight into the type of people who participate in and the discussions that occur on the mailing list. They also tried to determine the personality types of four top developers, to understand why they join and leave the project.

In a similar work, Yu and his colleagues [43] adopt a social-network model for representing the interactions of open-source software developers and mining their email archives using data clustering techniques. In fact, they applied data mining techniques to study the dynamics and the evolution of open-source social networks. They used communications in the mailing lists of two open-source projects, Linux and KDE to construct and analyze the social network. In order to group distributed developers according to their interaction, they applied hierarchical clustering. Although they are interested in the same informal communications as we are, they do not actually examine the text of the email messages but only explore the connectivity of the senders and recipients from email threads.

A similar paper in constructing social networks of email correspondents from the email archives published by Christian Bird et al. [6]. They extract a social network by analyzing the headers of the email archives of Apache HTTP server project, examining how the connections between developers on the mailing list relate to their development activity in the source code. They tried to find the relation-

ships between the activities of developers in the email archives and their software development activity, to see how the connections between developers on the mailing list relate to their development activity in the source code. From the social network measures, they concluded that the level of activity in the source code is a strong indicator of the social status of a developer among other developers. However, like Yu and his colleagues [43], they did not examine the body of the messages and only consider the name of the email participants.

Another approach to group developers and construct a social network from the change logs stored in the CVS repository was proposed by Huang and Liu [26]. They analyzed the log data to determine developers' contributions and construct a graph where a node represents a developer and an edge represents a contribution between developers. Then they analyzed the graph to find core and peripheral developers. The peripheral developers made minor contributions and the core members worked very closely with each other. Similarly Lopez-Fernandez and his colleagues [32] propose the idea of creating developer network from source code repositories. Their main goal was to categorize developers into different groups based on their collaboration. They also created a module network where two modules were connected if they were committed together.

Finally A. Meneely and his colleagues [33] proposed an approach for failure prediction based on developer network analysis. They constructed a developer network from version control commits in a similar way as Lopez-Fernandez and his colleagues [32]. In their developer network, each node represents a developer and two developers are connected if they have both made a change to at least one file during the same release. They found that the prediction model based on a network of developers performed significantly well in prioritizing files based on predicted failures.

2.3.1 Summary

Taking into consideration the lessons learnt from previous approaches, in this thesis we propose an unsupervised text-analysis approach to extract information from multiple sources associated with the software project, including, source code com-

mits, bug reports, mailing lists, chat logs, and wiki pages. In order to exploit such information, we integrate natural language processing techniques and statistical methods, with software domain semantic knowledge and apply them to five different sources of textual data. Comparing with the existing approaches, the strength of our technique is that (1) it explores a wider range of textual information sources associated with software projects; (2) it deals with the different types of data sources based on unsupervised techniques, i.e., natural language processing techniques, statistical methods, and domain specific knowledge; and (3) it applies a thorough lexical, syntactic and semantic analysis on textual artifacts to extract valuable semantic information about various aspects of the software life-cycle.

Chapter 3

A Method for Analyzing Textual Artifacts of Software Projects

In this chapter, we discuss our methodology for analyzing several sources of textual information in the software domain. Our main motivation is to gain insights about the information in the data sources, created by the individual team members over the various stages of the project. This information is collected through the WikiDev2.0 tool [4], as it is being used by a software team during a term-long course project.

The fundamental methodological assumption of our work is that, through their members' collaboration, software teams develop a "shared understanding" about what they discuss in each of their communication channels. In order to exploit this information, we developed a text analysis methodology, based on two complementary analyses, i.e., syntactic and semantic analysis. We applied our methodology to five different sources of the natural language textual data of a software development team. The five textual data sources were (a) wiki pages, (b) SVN comments, (c) bug descriptions, (d) email messages, and (e) IRC chats.

We developed our methodology for syntactic and semantic analysis, by integrating state-of-the art computational-linguistic techniques with domain-specific knowledge, in order to extract useful pieces of information about various aspects of the software life cycle.

Our overall methodology can be broken down into five main phases as the following:

1. Data Pre-processing, including data cleaning, sentence boundary detection,

and data acquisition.

2. Term extraction, including candidate extraction and term weighting.

3. Syntactic analysis, including syntactic tagging and dependency parsing.

4. Semantic analysis, including semantic annotation, and semantic representation.

5. Pattern extraction.

Generally, during the above five stages, the unstructured natural language data of software artifacts are analyzed in order to generate structured data annotated with domain semantics, which then can be queried to extract useful pieces of data.

Figure 3.1 presents the different phases of the developed system and their connections. As Figure 3.1 illustrates, after the raw data have been cleaned and preprocessed, the separate sentences first go through the syntactic analysis to produce syntax trees. The syntax trees of all the sentences, along with the domain vocabulary generated by the term extraction task, then serve as the inputs to a semantic analyzer which performs semantic annotation and produces semantic representations of the sentences. Finally, the annotated syntax trees represented in XML (eXtensible Markup Language) go through pattern extraction for further analysis to generate RDF-style (Resource Description Framework) [1] triples. More details of each process are presented in the following sections.

3.1 Data Pre-processing

The textual data created during software development are frequently very noisy. Noisy unstructured text data are typically found in informal texts such as online chat, text messages, emails, wikis and web pages. Unfortunately, simply applying text analysis tools, which are usually not designed for analyzing noisy data, may not bring good results. However, preserving such information makes traditional text processing methods (e.g., parsing, tagging, and stemming) unsuitable. Therefore,

[1] (http://www.w3.org/RDF/)

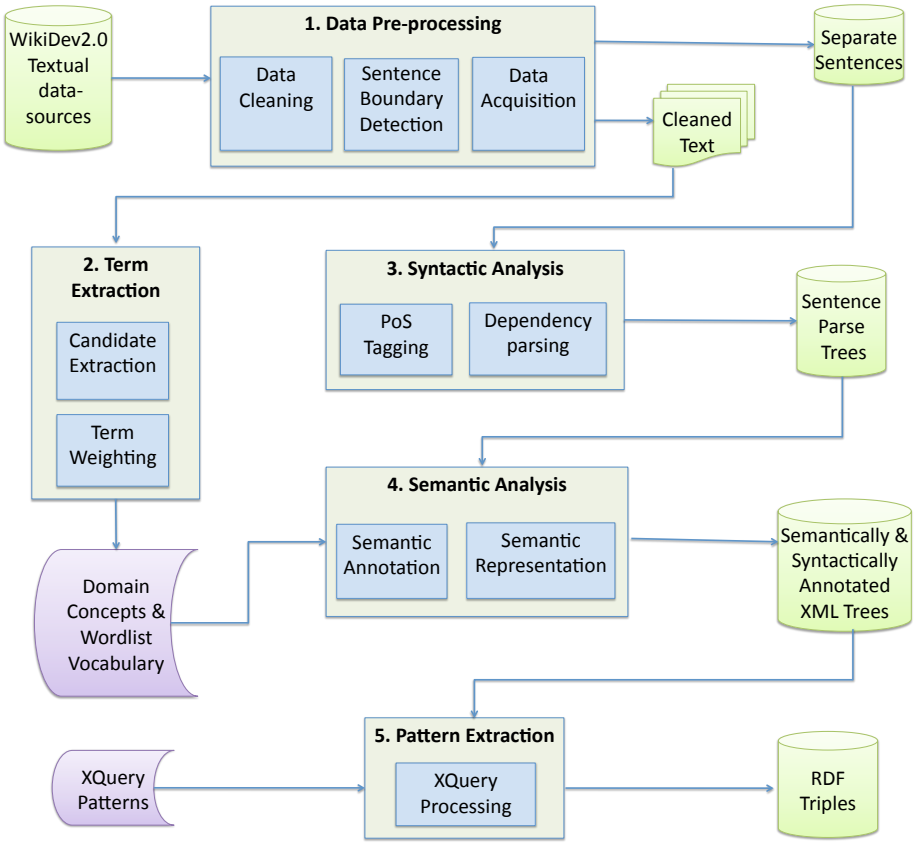


Figure 3.1: Tool Architecture

the data need some pre-processing and cleaning before any subsequent analysis can be applied.

Basically, data cleaning is defined as a process of normalizing texts by eliminating irrelevant non-text data, or any noise elements [38]. For example, in our corpora typical noise elements include pieces of source code inside the text, (wiki, html, and javadoc) markup tags, greetings and signatures in the email messages. Another essential part of any cleaning process in text analysis involves the identification and removal of duplicate data [38]. This step is important, especially when the data are used in statistical analysis such as frequency functions. Duplicate or redundant data may result in either misleading or probably, false results. For example, the quoted parts of email messages are common duplicate data.

3.1.1 General Data Cleaning

Our analysis used five different data sources from the software artifacts of Wikidev2.0. Before they could be analyzed further, these data had to be refined and abstracted from the raw data sources stored in the Wikidev2.0 database. Apparently, these data sources needed some general cleaning which was common among different kind of data sources. Poor spelling and punctuation, abbreviations, acronyms and domain-specific language can cause noise in all data sources. Eliminating all of the noise from our data sources was impossible; however, we tried to remove some of the typical elements of noises as follows:

- Removing non ascii characters;
- Removing pieces of source code;
- Removing html, xml, wiki and javadoc markup tags;
- Replacing http links with a corresponding text. For example, replacing the link "http://hypatia.cs.ualberta.ca/ucosp/index.php" with "ucospURL";
- Replacing the name of some tools and programming languages with a corresponding name. For example, replacing C++ with Cpp, or .Net with dotNet;
- Removing the extension form the name of files such as ".txt", ".java", ".php". They should be removed because the parser may consider them as the period for the end of sentence;
- Removing bullets or numberings;
- Eliminating arbitrary use of punctuation marks and capitalization, such as "{", "[", and ">";
- Fixing spelling errors in the name of some domain-specific terms such as class names and table names. For instance, replacing "UML Handler" with "UMLHandler".

In addition to the above types of generally applicable data cleaning, some kinds of data artifacts need some specific refinements which differ from those of the other artifacts. For example, transforming the data in chat logs and email messages into a refined format suitable for subsequent analysis, involves some particular cleanings. By considering the general and particular cleanings required for different data sources, we tried to refine the raw data source and extract the data fragments required for our analysis. In the following sub-sections, we describe the specific features and challenges of each data source, also identify the data fragments that we obtained from them.

3.1.2 Bug Reports

Bug reports are one of the relatively clean data sources, because the interaction between the bug reporters is relatively well recorded and formal. Bug reports need only general refinements such as, removing pieces of source code, html and xml tags, and http links.

Bugs or tickets are usually tracked by means of an issue/bug tracking service such as "bugzilla". The bug-tracking system typically records the identity of the bug reporter, the bug description, and the date of the bug report. In the bug-tracking system, the bug identifiers are assigned to a ticket with a unique identity.

Bug records often contain pieces of information, such as the reporter of the bug, its description, and other team members mentioned within the bug description.

3.1.3 SVN comments

Another rich and relatively clean source of information created during software development life-cycle is sub-version commits. Similar to the bug reports, they only need some general cleaning. Subversion control repositories include collections of files committed together in the repository and the associated comments. For each commit, the data elements that the sub-version system typically records include, source code files, the author of the commit, revisions, and the commit message. Commit messages are the comments that programmers write when they commit revisions to the source control repository, which is the most readily accessible source

of project artifacts.

The data elements we abstracted from SVN comments included, the author of the commit, other team members mentioned within the message of the commit, and the actual commit message.

3.1.4 Wiki Pages

Wiki pages are written by using a specific markup language which includes different markup tags to specify the structure and formatting of the page contents. For example, in wiki text, a bulleted list in typically designed using an "asterisk". Therefore, in addition to the general cleanup process, wiki pages also need some specific kinds of cleaning to remove wiki markup tags. In order to convert the wiki text into plain text, first the wiki text is transformed into HTML text. Then, the HTML text is changed to plain text by using a HTML parser. We used the dedicated HTML parser of the MediaWiki [2] for this purpose. Another feature of wiki pages is that each wiki page may have several revisions. We considered only the last revision for each wiki page, to prevent redundancy of the wiki data.

After cleaning the wiki pages, the data elements we derived from this data source included the author of the last revision, the users mentioned within the content of the last revision, and the content of the last revision as plain text.

3.1.5 Email messages

Compared to other software development artifacts, such as change-log commits or bug reports, mailing lists are less structured. However, these lists contain valuable information about the development process, design decisions, and developers' characteristics. The email messages have the following specific features. Firstly, email messages contain duplicate and invalid data, stored in raw formats, which threaten the applicability of text analyzing approaches and need further processing. In email messages, users usually refer to the text of previous participants by quoting their messages. However, the additional text may not be desirable for text analysis approaches, as it is redundant information. Quoted messages typically begin with one

[2] http://www.mediawiki.org/wiki/MediaWiki

or more ">" signs at the beginning of a line (one for each level of quotation), and they can be easily removed automatically. Secondly, email messages also contain "signature" elements at the end of the text. Signatures are typically used as a sign of identity and to indicate the end of a message, and contain stereotypical phrases like "thank you" and "best regards". Signatures may contain a variety of artifacts, such as contact information, text graphics, and quotes. Many free email services may also add advertisements as a signature in their messages. As a result, the information included in the signature part of emails is often repetitive and unrelated to the message and should be removed. Finally, email messages are point-to-point messaging and therefore target particular recipients. As a result, the data we extracted from the mailing lists included the author of the message, the direct recipient(s), the team members mentioned within the body of the message, and the body of the message. In the mailing list of Wikidev2.0, the recipients of the each email message are all the users. However, in some messages the author specifies the name(s) of specific recipient(s). We extracted the possible specific recipient(s), from the beginning of the message.

3.1.6 IRC Chat Logs

Similar to email messages, chat logs are more unstructured and allow discussions on a wider range of topics than bug reports or commit logs. Chat logs have several unique properties. Chat messages may be written by users with a virtual identity or nick-name. Chat messages have no editorial modifications, therefore misspellings in them are more frequent than in edited text. Chat messaging has a specific style and vocabulary and its own method for expressing emotions by using emotion icons such as ":-)" and ":-(". Emotion icons are sequences of punctuation marks that represent feelings such as happiness, anger, and depression. Repetition of some characters in a word can also be used as a means of expressing emotions (e.g. "greaaatt!"). Moreover, in chat messages, the use of conscious misspellings such as "cya everyone" is frequent.

Like email messages, the chat logs are point-to-point messages which target particular recipients. That means, each chat message usually has specific recipient(s)

and a specific sender. In chat messaging, the names of the particular recipients usually appear at the beginning of the message.

After we removed possible noise elements from the chat messages, the data we extracted from each of them contained the author of the message, the direct recipient(s), the team members mentioned within the body of the message, and the content of the message itself.

Table 3.1 summarizes the required data cleanings and abstracted data fragments for each data source.

Table 3.1: Data cleaning and acquisition

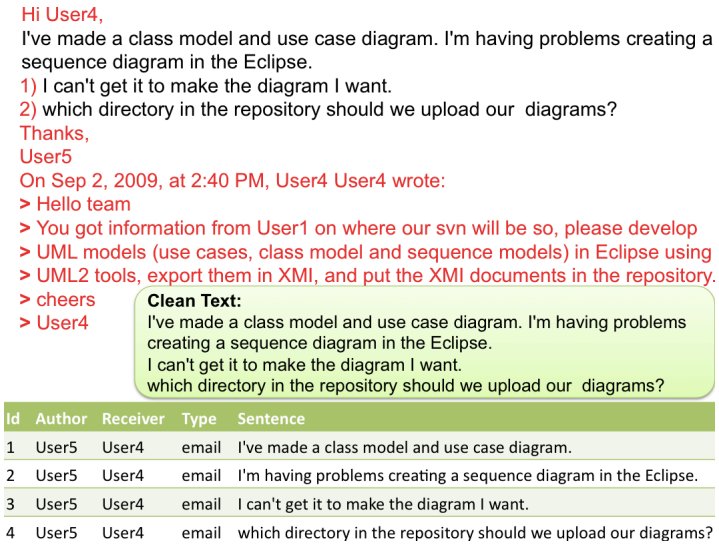
Data Source	**Abstracted Elements**	**Required Cleanings**
Bug reports	author, bug description	general cleaning
SVN commits	author, commit message	general cleaning
Wiki pages	author, page content	general cleaning, wiki markup cleaning
Email messages	author, body of the message, receiver(s)	general cleaning, removing greeting, signature and quoted part
Chat messages	author, message content, receiver(s)	general cleaning, removing emotion icons and resolving nick-names

In addition to common grammatical and punctuation mistakes, yet another systematic type of noise appeared in our data sources. We realized that in many of the sentences especially the SVN comments and bug descriptions, the author of the message did not include the subject of the sentence. For example instead of stating "I created a table", the author may state "created a table". By analyzing these sentences as they were, we would have missed the information included in those messages. In order to resolve this problem, we tried to recognize sentences with a missing subject and to add the author of the sentence as a subject in front of the sentences with no subject. The sentences missing a subject were the non-imperative sentences which had a verb but no subject. In order to distinguish between the sentences missing a subject and the imperative sentences, we considered one of these two features for the imperative sentences. The first feature is that the imperative sentences may contain the word "please". Secondly the tense of the verb in the

imperative sentence is simple present.

After we cleaned the data sources and acquired the data elements, we divided the textual data into individual sentences, by using a "Sentence Splitter" to detect sentence boundaries [31]. The separate sentences, along with the other data elements, then served as input to the syntactic analyzer for further analysis. The data elements associated with each separate sentence included "sentence id", "data type", "author", "receiver", and "sentence".

Figure 3.2, presents sample results of the data cleaning and acquisition process on an email message.



Id	Author	Receiver	Type	Sentence
1	User5	User4	email	I've made a class model and use case diagram.
2	User5	User4	email	I'm having problems creating a sequence diagram in the Eclipse.
3	User5	User4	email	I can't get it to make the diagram I want.
4	User5	User4	email	which directory in the repository should we upload our diagrams?

Figure 3.2: Sample Result of Data Preprocessing on an Email message

3.2 Term Extraction

The need to extract and manage domain specific terms has become increasingly important in recent years [17]. Term extraction, aims at the recognition of domain-specific terms from the corpora of a certain domain. Domain terms are the linguistic representation of the domain-specific concepts in the text. Terms may consist of a single word form, called a single-word term, or more than one word form, called a multi-word term. The most common compound nouns that could serve as multi-word terms include the adjective-noun and the noun-noun phrasal compounds [17].

In general, distinguishing terms from regular words in the text is not an easy

task. This task becomes even more complicated in the software domain for the following reasons.

- Most available terminological vocabularies for our specific domain are still far from being complete.
- The software terminologies change over time, and the terminological vocabulary should be updated appropriately.
- The software terminological vocabularies consist of a large number of concepts in various categories, including programming languages, software design and architecture, algorithms, development tools, data structures, software quality and maintenance.
- The lexicon of a software project tends to be project-specific and is often unique.

We propose a semi-automatic approach, in order to identify software technical terms and populate the terminological vocabulary of our specific domain. The proposed term extraction approach is based on two tasks: candidate term extraction and term weighting. The candidate extraction task tries to identify a list of candidate terms containing single words. Single-word candidates are simply defined by taking the list of all words that occur in the text but do not appear in a standard "stop-word" list or "noise" words. Stop-words are extremely common words in the text which do not provide any useful information, such as "about", "then", and "even".

The second task, term weighting, takes the list of candidate terms as input and ranks them as domain related terms and regular terms. In order to find out if the candidate terms are domain-specific terms, this task calculates a term weight for each word in the list of candidate terms. Then it ranks them based on the assigned weights and decides whether the weighted words are domain terms if their weights are higher than a threshold. The words weights are usually measured based on a statistical scale. We consider the frequency of the terms in the text as the statistical measure to calculate the weights of candidate terms. The basic underlying idea

is that interesting and meaningful terms for the domain are likely to occur with relatively high frequencies in the corpus of the domain [29].

The following formula shows the straightforward measure used to calculate the weight for each candidate term. This measure uses the number of all terms in the text in order to normalize the weight [29].

$$CandidateTerms = \{W1, W2, ..., Wi, ...\} \tag{3.1}$$

$$TermWeight(W_i) = \frac{f(W_i)}{\sum_j f(W_j)} \tag{3.2}$$

where $f(W_i)$, is the number of occurrences of term W_i in the text.

In our work we utilized TAPoR, the Text Analysis Portal for Research, to identify candidate terms and their weights for our dataset. TAPoR, is a web-based application developed by digital humanists. TAPoR exemplifies a lightweight analysis of natural-language text, focusing only at the "superficial" lexical level of the text. It provides a suite of lexical-analysis tools [2]. The tools offered by TAPoR include (a) word counts and lists, (b) word co-occurrence (i.e., keyword-in-context), (c) word-clouds visualizations, (d) words' collocations within sentences and paragraphs, and (e) pattern extraction. TAPoR has had several deployments around the world and is used to analyze the lexical properties of texts and collections.

We deployed the TAPoR word-frequency service to identify the candidate terms and calculate term frequency as their weight. In this service, TAPoR reads the input data from a text file, excludes stop-words and generates a list of all words sorted by the word frequency. TAPoR can also list words by applying an inflectional word stemmer. We considered the word stems to calculate their frequencies, because it helps to measure the occurrences of different lexical forms of a term as a single frequency count. Hence, a single term is derived from different lexical forms of a term, instead of several terms. For example, without stemming, the terms *diagram* and *diagrams* are treated as different items with separate term frequencies, while by using stemming, they are considered as a single term *diagram* with one frequency count.

After producing TAPoR word frequency results, we identified all the words whose weights (i.e., frequencies) were higher than a threshold, and assigned them to one of our domain concepts (or semantics), such as project tasks, project artifacts, and the names of the tools. The threshold were considered a frequency value that domain terms rarely appear with that frequency.

The frequency produced relatively good results, and its application to the corpora was relatively simple. For example, in our 26,464 word software corpus (without stop words), *xml* appeared 129 times, *uml* 123 times, and *php* 94 times. Of course, not all domain terms exhibited high frequencies. For example, *MySQL* appeared 5 times, and *Perl* only one time. Low frequency terms cause problems for statistical approaches to find them. In order to add some important terms with low frequency to our domain vocabulary, we augmented the vocabulary by using some standard dictionary related to the software domain such as the IEEE Computer Society's keywords in the "Software/Software Engineering" category [3] and "Software Engineering Body of Knowledge (SWEBOK)" [4].

In addition to using TAPoR's most frequent words, we also considered a set of terms that are already part of the WikiDev2.0 database, such as the anonymized names of the teams' members, the names of the developed files, the programming languages the teams used, the name of the created tables, and the IDs of the teams' tickets and code revisions.

After aggregating these three sets of terms, the ones extracted by using TAPoR, those selected from the Wikidev2.0 database, and those from standard software vocabularies, we created a specific vocabulary, organized around a set of term categories, for the software-development domain. Our domain vocabulary included the following concepts and their corresponding sample terms as depicted in Table 3.2.

We used the created domain vocabulary in order to extract domain-specific terms from each sentence. After we recognized a word as a domain-specific term in a sentence, we capitalized the word. The reason for this change is helping the parser to assign correct PoS tags to the domain terms, in the sentences. For example the

[3] http://www.computer.org/portal/web/publications/acmsoftware

[4] http://www.computer.org/portal/web/swebok/html/appendixd

Table 3.2: Created Domain Vocabulary, Including the Concepts and the Corresponding Sample Terms.

Concepts	**Sample Terms**
Users	User1, User2, ... , User9
Programming Languages	Java, PHP, XML, ...
Tools	Eclipse, Bugzilla, IBMJazz, ...
Tickets	ticket1, ticket2 , ...
Revisions	Revision1, Revision2 , ...
Action Verbs	create, debug, implement, fix, make, ...
Project Tasks	visualization, documentation, user interface, testing, ...
Project Artifacts	class, method, table, database, script, ...
Table Names	tickets, diagrams, ticketchanges, ...
Class Names	XMIparser, XMLparser, Associations, ...

parser tagged the word “eclipse” as “verb” instead of “noun” in several sentences. Capitalizing domain terms helps the parser to annotate them correctly.

3.3 Syntactic Analysis

The goal of the syntactic analysis is to assign a syntactic structure to each sentence. A syntax tree or a parse tree is a structural model for presenting a sentence and represents the syntactic structure of a sentence according to the grammatical rules of the language.

Two different methods for representing the syntactic structure of a sentence can be used: constituency trees and dependency trees. These two types of trees assume different semantics for the syntactic-tree nodes and the edges between them [12].

The constituency tree is a phrase-structure representation, starting from constitute S and ending in the sentence words as the leaf nodes. Constituents (labelled in capital letters) are the phrase structures of the word sequences in the sentence. In constituency trees, each non-terminal node represents a constituent such as NP (for noun phrase), VP (for verb phrase), PP (for prepositional phrase), ADJP (for adjective phrase), and ADVP (for adverbial phrase) [30]. Figure 3.3 presents a sample constituency tree for the sentence “I used Java and Eclipse before”.

Rather than using phrase-structure representation, a dependency tree represents

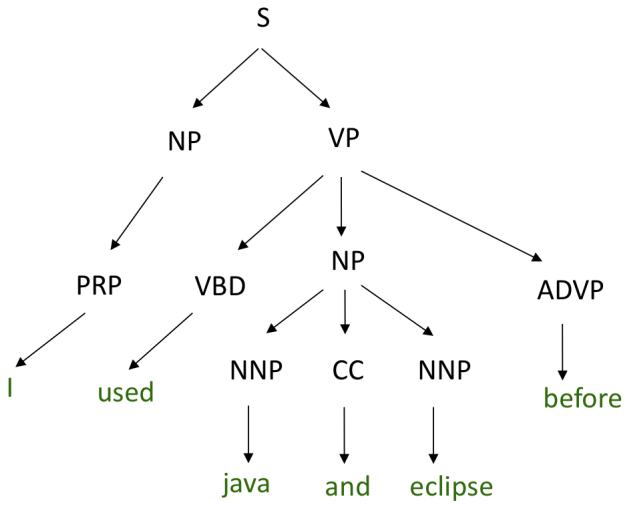


Figure 3.3: Constituency tree for the sentence ”I used Java and Eclipse before.”

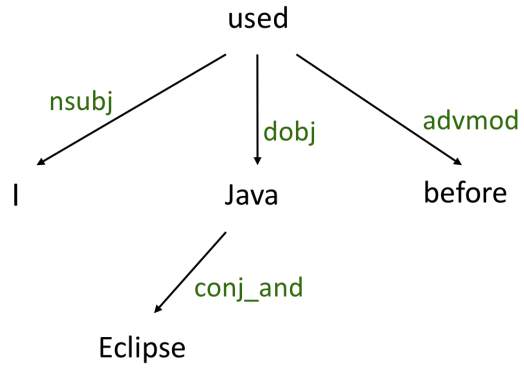


Figure 3.4: Dependency tree for the sentence “I used Java and Eclipse before.”

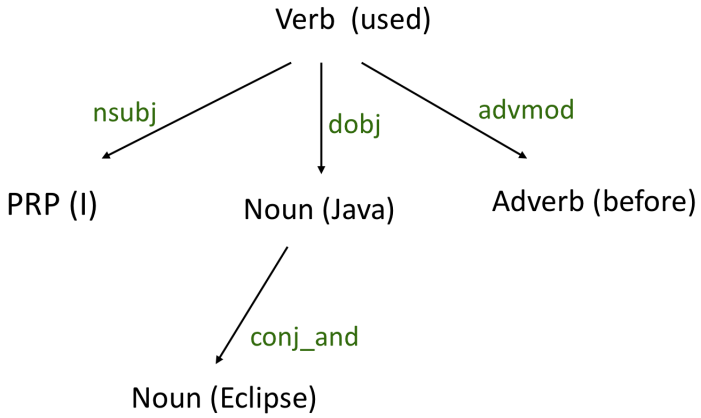


Figure 3.5: Augmented Dependency tree for the sentence “I used Java and Eclipse before.”

the grammatical relationships between the individual words in a sentence, as typed dependency relations, such as subject or direct-object. In a dependency tree, the nodes of the tree are sentence words, so that a dependency relation is established between a pair of the words in a sentence. One of the words in each dependency relation is the head or governing words, and the other is dependant (or subordinate) to the first one [14, 15]. Figure 3.4 presents a sample dependency tree for the sentence "I used Java and Eclipse before."

The dependency-tree representation is more effective in certain tasks than the constituency-tree representation. In information extraction applications such as relation extraction, matching the relations in a sentence is easier by using the dependency tree. Its use also simplifies semantic analysis tasks involving transforming the dependency tree in any semantic representation, such as conceptual graphs or semantic networks [14].

In spite of the apparent differences between constituency and dependency representations, both representations provide useful information on the syntactic structure, so that they can be combined. In fact, the model that we used as a structural model for representing the syntactic structure of our sentences is based on a dependency tree augmented with some constituency information. We enhanced the dependency tree, with the part-of-speech tag (PoS tag) of the word in each node. We chose the augmented dependency representation because we assumed that sentences containing similar information usually have the same structures in their dependency trees. Figure 3.5 presents a sample augmented dependency tree for the sentence "I used Java and Eclipse before."

For the task of syntactic analysis, including part-of-speech tagging and dependency parsing, we used the Stanford Parser [12, 31, 1]. Stanford parser is a very high performance probabilistic parser, which provides constituency parsing and dependency parsing. The separated sentences are parsed by the Stanford parser, whereby each word is assigned a part-of-speech tag which is a grammatical label such as "noun", "verb", "adjective" "adverb", and "preposition". Furthermore, the Stanford parser also provides the dependency relationships between pairs of the words in a sentence such as "subject", "object", and "possession". The Stanford

Syntactic tags:

I/PRP used/VBD java/NN and/CC Eclipse/NNP before/RB ./.

Parse Tree:

```
(S
    (NP (PRP I))
    (VP (VBD used)
        (NP (NNP Java)
            (CC and)
            (NNP Eclipse))
    (ADVP (RB before))) (. .)))
```

Dependency Relations:

nsubj (used-2, I-1)
dobj (used-2, Java-3)
conj_and (java-3, Eclipse-5)
advmod (used-2, before-6)

Figure 3.6: Stanford Parsing result for the sentence "I used Java and Eclipse before."

dependencies are represented as binary relations between two sentence words. The parser presents the dependency relations as triples including: the name of the relation, the governor (or head) and the dependent elements. Figure 3.6 presents the syntactic tags and the dependency relations created by the Stanford parser for the sentence "I have used Java and Eclipse before".

3.4 Semantic Analysis

The syntactic analysis described in the previous section, assigned a syntactic tag for each word and the grammatical relationships between the sentence word pairs. The next phase of our method is semantic analysis which consists of two main tasks. The first task annotates the domain-specific terms that exist in the sentence by using semantics. The second task represents the extracted syntactic and semantic information of each sentence by using a formal data model. The following subsections describe the two tasks of our semantic analysis.

3.4.1 Semantic Annotation

Semantic annotation is the task of assigning meaning to as many terms of the corpus as possible. This process attempts to clarify the domain-specific roles that these terms play. The main motivation behind this task is that the meaning of a sentence can be inferred based on the meaning of its components in their syntactic representation [27].

This semantic annotation phase was implemented as a process of attaching specific concepts, belonging to our enumerated domain vocabulary, to the terms in the sentence parse tree. This process led to these augmentation of the syntax tree of the sentence by using semantic attachments.

Our process to semantically annotate the terms relies on two steps. First, it identifies any nouns in the sentence parse tree by using part-of-speech tagging created by the syntactic analysis process. Secondly, it uses our domain vocabulary created by the term extraction task to annotate the terms of the sentence parse trees. Given the domain vocabulary and part-of-speech tagging, the semantic-annotation process associates each noun in the syntax tree of the sentence with a semantic concept from the domain vocabulary. Accordingly, a word in the syntax tree of the parsed sentences will be annotated if it is tagged as a noun and if it matched with any of the terms in the domain vocabulary.

The above process works for the words that appear as nouns in a sentence, but it may miss words that appear as pronouns. For instance, in the sentence "I used Java and Eclipse before", the pronoun "I" refers to a specific user, and should be annotated as a *developer*. To enable a pronoun to be annotated by the semantic concepts as well, we need the pronoun co-reference resolution.

Moreover, throughout the text, a single term is usually referred to in different ways, including pronominal references (e.g., "this class"). In order to find identical terms and references, a task of coreference resolution is required. Coreference occurs when two words in the text refer to the same term, and has two forms: nominal and pronominal coreferences. For the nominal coreference resolution, we considered the following conditions:

- For the names of users, first name, last name, and both refer to the same user. For example, “John”, “Smith”, and “John Smith” co-refer to the same user.

- For the name of the tools and programming languages, acronyms and full names may refer to the same tool or programming language. For example, the “SVN” and “Sub-Version System”, refer to the same tool, or “VB” and “Visual Basic” co-refer to the same language.

- A term can be an abbreviation of another term. For example “MS” is an abbreviation of “Microsoft”, and “DB” is abbreviation of “database”.

For the pronominal coreference resolution, detailed text analysis revealed that a few simple rules could resolve the vast majority of pronominal coreferences. For example, 80% of the occurrences of “he”,“she”,“his”, and “her”, referred to the closest person of the same gender in the same sentence [9]. Finding a resolution for the pronouns “it”, and “its” is more difficult because the number of possible proceeding nominals (i.e., entities that are referred to) is much higher compared with the other pronouns like “she” and “he”.

Our approach for the pronominal coreference resolution is very simple and is shown below.

- We divide the pronouns into three groups:

  - *First Person Pronouns* = {*I, we, me, us, my, our, mine, ours, myself, ourselves*}.
  - *Second Person Pronouns* = { *you, your, yours, yourself* }.
  - *Third Person Pronouns* = { *he, she, her, hers, him, his, herself and himself, them, their, theirs, themselves*}.

- We keep the author and the receiver for each sentence. The pronouns in the first group (i.e. “I, we, my”), refer to the author name and are annotated as *developer*. Accordingly, the second group pronouns (i.e. “you, your”), refer to the receiver’s name and are annotated as *developer*.

- In order to find the reference for the pronouns in the third group (i.e. “she, he, her”), we refer these pronouns to the proper individual nouns in the same sentence or in the preceding sentence, and annotate them as *developer*.

3.4.2 Semantic Representation

During the previous analysis, the syntactic and semantic information has been extracted from the sentences, including the syntactic tags of the words, the grammatical relationships between words, the domain-specific terms, and the semantic annotation of the terms, the next task involves representing the extracted syntactic and semantic information in a structured data model.

The semantic representation task has two main motivations. First, it maps unstructured sentences into a formal representation of their meaning, enabling us to run queries on them afterwards. Second, this formal representation can be used as the input for some subsequent analysis that produces richer and more meaningful information. In Section 3.5, we explain the use of subsequent analysis to extract more useful knowledge.

We decided to express the extracted information from the syntactic and semantic analysis in a data model using the Extensible Mark-up Language (XML). At this stage of the process, each sentence is transformed into a syntactically and semantically annotated tree, represented in XML. The annotated XML trees are stored in the Wikidev2.0 database in order to provide an automatic retrieval of the information by running queries towards them, and to do further analysis on them. The resulting XML representation is much more amenable to querying than the unstructured natural sentence. As shown in Figure 3.7, each such annotated XML tree includes the following pieces of information:

- Dependency relations between words, such as “dobj” between “Java” and “used”;
- Syntactic tags of the words, such as “NNP” for the word “Java”;
- Semantic annotation of the domain-specific terms, such as “language” for the term “Java”;

```
<S  Type="ticket-description"  ticketId="1"  sentId="127
Author="User1" Receiver="User2"> I used Java and Eclipse before.

  <Verb  stem="use" ID="1" PoS="VBN"  Relation="root"> Used
     <PRP ID="1"    PoS="PRP"  Relation="nsubj"
          semanticTag="Developer" Name="User1">  I </PRP>
     <Noun ID="2"  PoS="NNP"  Relation="dobj"
           semanticTag="language"> Java
        <Noun  ID="4"  PoS="NNP"  Relation="conj_and"
                  semanticTag="tool"> Eclipse </Noun>
     </Noun>
     <Adverb ID="5" PoS="RB" Relation="advmod"> before </
Adverb>
  </Verb>
</S>
```

Figure 3.7: The sentence "I used Java and Eclipse before." after parsing and semantic annotation.

- Stem of the verbs, such as "use" for the verb "used";
- Type of the sentence, such as "ticket-description";
- Author of the sentence, such as "User1"; and
- Recipient of the sentence, such as "User2".

Figure 3.7 shows the annotated XML tree resulting from the syntactic and semantic analysis of the sentence "I used Java and Eclipse before".

XML was selected as the data model for the semantic representation of sentences for two reasons. First, XML documents have a tree structure that starts at the root and branches to the leaves. Each node of the XML tree can contain some attributes, which provide additional information about that node. XML documents are often referred to as XML trees. Therefore, the XML tree structure is a good fit for representing annotated parse trees. Secondly, powerful query languages have been specifically designed to search across a variety of XML data and query them. By using XML Query languages, we can define queries and run them on the annotated parse trees represented by XML, in order to extract useful information from

them. In the following section, we will explain how to define some XML Queries to extract more useful information from annotated XML trees.

3.5 Pattern Extraction

Knowledge from the above sentence representation is extracted through pattern matching. The underlying hypothesis is that, there are grammatical patterns that occur in the information expressed in the sentences. In fact, the relations among the domain concepts are represented based on the predicate-argument(s) patterns which can be explored from the parse trees.

Accordingly, our pattern extraction method involves the matching of these "patterns" against the syntactically and semantically annotated XML trees of our sentences, in order to find semantic relations between domain-relevant terms. The output of this process contains instances of the significant relations among the domain concepts represented as RDF-style triples.

Each RDF triple expresses a semantic relation between two entities of the form $\langle Subject, Verb, Object \rangle$. Our pattern-extraction process attempts to recognize instances of these semantic relations, as "syntactic paths" connecting a verb in the sentence located in the proximity of two annotated domain entities. The relation verb along with the associated terms becomes "a relation triple", which can potentially indicate a relevant semantic relation. Essentially, the syntactic dependencies recognized by the Stanford parser provide the basis for the semantic relations between semantically annotated terms.

The proposed approach for the triple extraction traverses the annotated XML trees and analyzes the dependency relations and the semantic annotations in order to find predicate-argument patterns among the domain-specific concepts. In order to extract semantic relations, our approach looks only for the dependency relations with respect to a verb such as subject and object dependencies. The dependency relationships make writing and establishing extraction patterns much easier.

The pattern-extraction task is implemented as a rule-based process. We define the following three rules to search for the predicate-argument patterns that exist in

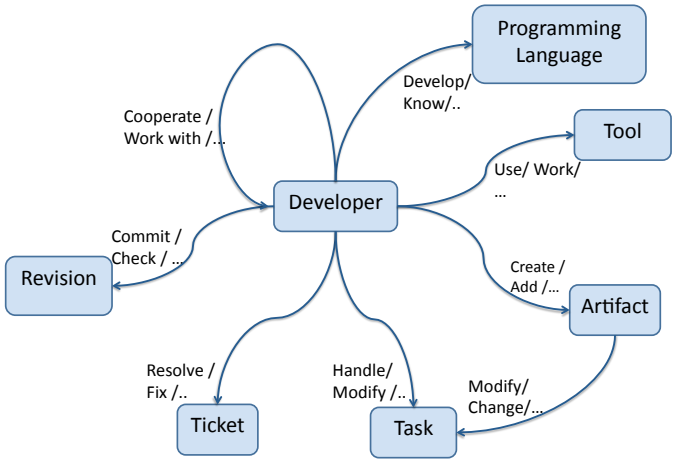


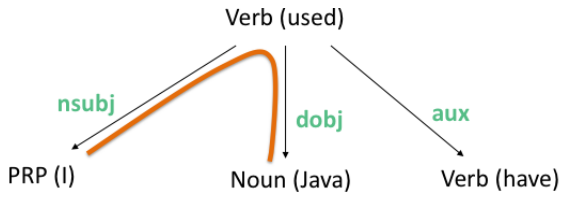
Figure 3.8: The semantic relations of the RDF triples produced by the pattern-extraction process.

the annotated XML trees and to extract triples from the annotated XML trees.

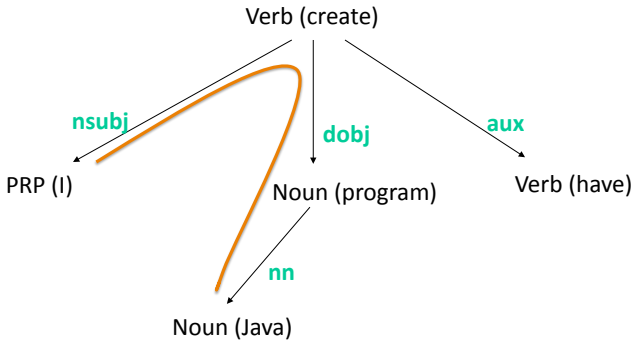
Rule 1: If a verb V is associated with two different entities E_i and E_j which satisfy the condition $[Subj(V, E_i) \wedge Rel(V, E_j)]$, then V is identified as a *relational verb* between entities E_i and E_j. Based on the dependency relations identified by the parser, $Rel(V, E_j)$ can be either an "object relation" or a "prepositional relation". For example, the sentence "I have used JUnit" contains the object relation $dobj(use, JUnit)$, while the sentence "I have worked with JUnit" contains the prepositional relation $prep\_with(use, JUnit)$.

Rule 2: If a verb V is associated with three entities E_i, E_j and E_k which satisfy the condition $[Subj(V, E_i) \wedge Rel_1(V, E_j) \wedge Rel_2(E_j, E_k)]$, then V is identified as a *relational verb* between the two entities E_i and E_k. Based on the dependency relations identified by the parser, the relation $Rel_1(V, E_j)$ can be an "object relation" or a "prepositional relation", and the relation $Rel_2(E_j, E_k)$ can be a "prepositional relation", a "conjunction relation", or a "noun-modifier relation."

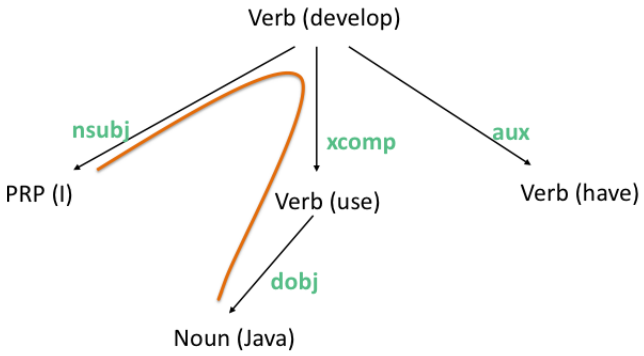
Rule 3: If a verb V_i is associated with two entities E_i, E_j and another verb V_j which satisfy the condition $[Subj(V_i, E_i) \wedge Rel_1(V_i, V_j) \wedge Rel_2(V_j, E_j)]$, then V is identified as a *relational verb* between the two entities E_i and E_j. Based on the dependency relations identified by the parser, the relation $Rel_1(V_i, V_j)$ can be a



Sentence “I have used Java”.



Sentence “I have created a Java program”



Sentence “I have developed using Java”

Figure 3.9: Dependency paths in the XML tree to represent Rule 1, Rule 2, and Rule 3.

“clause-modifier” or a “clause-complement”, and the relation $Rel_2(V_j, E_j)$ can be either an “object relation” or a “prepositional relation”.

Rule 1 matches with the relation in which the object component is the explicit child under the verb component in the dependency path of the XML tree. On the

other hand, Rule 2 and Rule 3 cover the kind of relations in which a noun modifier or a verb modifier exists between the verb and the object component. Intuitively, Rule 1 covers explicit relations directly expressed in the text, while Rule 2 and Rule 3 capture implicit relations, such as indirect objects and long-distance dependency relations, as depicted in Figure 3.9.

We utilized the above rules to find relevant semantic relationships between our domain-specific concepts. Essentially, we were looking for the triples of the type $\langle domain\_concept, semantic\_relation, domain\_concept \rangle$. Our target triples are shown in the model represented in Figure 3.8 as eight semantic relations between domain concepts. This figure presents the rich conceptual model of our domain that captures interesting relations between developers and software products.

As Figure 3.9 shows, each rule is associated with a specific predicate-argument pattern in the annotated XML trees. Based on the above three rules and our target triples, we defined a set of patterns, implemented by using the XQuery language. XQuery is a query language designed by the W3C to search through the variety of XML data, select the XML data elements of interest, and return the results. For more information about the XQuery language we refer the reader to reference [39].

Once the XQuery patterns had been created, we ran the defined XQueries toward the annotated XML trees in order to retrieve interesting relations as RDF triples. We have generated fifteen XQuery patterns, for extracting our target triples by using the defined rules. The answers returned by running the defined XQueries on annotated XMLs were the instances of our target triples depicted in Figure 3.8. The extracted triples were stored in a database in order to help users to efficiently query them.

Chapter 4

Evaluation

4.1 Experiments

This section describes the details of our experiment on the textual data stored in the Wikidev2.0 database. It also reports the experimental results from applying each stage of our method to these data sources.

4.1.1 The Input Dataset

In this thesis, we used five different sources of textual information collected through the WikiDev2.0 tool as our input dataset. The resources currently integrated into WikiDev 2.0 include (a) subversion control repositories, (b) ticket information as bug reports, (c) wiki pages, (d) mailing list archives, and (e) Internet Relay Chat (IRC) logs.

The information of these resources is imported from a bug-tracking system, a dedicated mailing list, internet relay chat logs, and SVN repository information. Subversion-control repositories include collections of files committed together in the repository and the associated comments. Bugs and ticket information are usually tracked by means of an issue or bug-tracking service such as bugzilla. We selected the change-set comments and bug descriptions as two textual data sources mainly about source code. Wikidev2.0 also collects archives of IRC from each team project to include a real-time communication data source along with the Email messages.

For our experiment, we chose to analyze the data from a team of nine devel-

opers from a project-based undergraduate software engineering class. This dataset included information about the requirements of the software system, distribution of the tasks among developers, the internal design, and the functionalities of the various modules in this system.

First, we collected raw data from five data sources, including 39 records from tickets, 134 records from mailing list archives, 297 records from SVN change-sets, 621 records from wiki pages, and 2,584 records from chat logs. In total, we collected 3,675 text records from the above data sources which resulted in 5,495.8 KB of plain text. Then we analyzed these raw unstructured textual data based on the five main stages of the methodology discussed in Chapter 3. The following section describes the results of our experiment for each stage.

4.1.2 Experimental Results

The whole experiment consisted of several phases, in which we used different components of the developed tool.

1. **Data Cleaning and Preprocessing.** First, our data sources were cleaned from any "noise" elements, such as redundant data, pieces of source code inside the text, (wiki, html, and javadoc) markup tags, greetings, signature and quoted parts from email messages. Also, possible punctuation errors in the names of domain-specific terms were corrected. After applying the cleanup process and eliminating noisy data, we divided the cleaned textual records into separate sentences by using a sentence splitter. In total, we collected the 5,265 sentences from all of the input data sources which resulted in 537 KB of plain text. The first row in Table 4.1, summarizes the results of this process on our five input data sources. To each sentence, we attached the following information: (a) a sentence id, (b) the author of the sentence, (b) the receiver of the sentence (only for email and chat messages), (c) the origin of the sentence (svn, ticket, email, irc, or wiki), and (d) the actual sentence text. We used this set of sentences and information fragments for the syntactic and semantic analysis and triple-extraction task.

2. **Term Extraction.** During the term extraction task, we populated the domain vocabulary for the dataset. We used the TAPoR word-frequency service to identify the domain terms from our corpus which were project tasks, project artifacts, and the names of the tools and programming languages. The TAPOR lexical analysis showed that our data sources contain 2918 unique words other than those in the stop list, 24550 words other than those in the stop list, and 56145 words in total including the stop words. We also considered a set of terms that are already part of the WikiDev2.0 database, including the anonymized names of the teams members, the names of the developed files, programming languages the team used, and the names of created tables. The domain vocabulary that we created, includes the corresponding terms for these concepts: Users, Programming Languages, Tools, Project Tasks, Project Artifacts, Table Names, and Class Names.

3. **Syntactic and Semantic Analysis.** We applied syntactic parsing and semantic annotation to each separate sentence and generated an annotated XML tree for each sentence. These processes generated 180 XML trees from the tickets, 441 XML trees from the mailing list archives, 361 XML trees from the SVN change-sets, 938 XML trees from the wiki pages, and 2,862 XML trees from the chat logs. In total, these analyses produced 4,782 XML trees annotated syntactically and semantically, and 483 un-parsed text fragments. In most of the un-parsed cases, the text fragment was not even similar to a sentence and therefore, an XML tree could not be extracted out of it. Most of these useless text fragments consisted only of nouns without any predicate, such as "probably not", or "good idea!". Some of them started with a verb and were missing a subject, such as "will do" and "edited". Some other reasons why an XML tree could not be extracted were that the text fragment content was just noise, or that it was part of a sentence without the main words and only attributes remained, or that it was an informal chat expression such as "cya everyone". However, we tried to improve the sentences were missing a subject, by adding the author of the sentence as the missing subject. We modified 1034 sentences from all data sources, by adding the author of the

sentence as the missing subject.

4. **Pattern Extraction.** Finally, by applying pattern extraction to the annotated XML trees, a total of 1,605 triples were extracted. The sentences that include information about these triples were 945 sentences. The third row of Table 4.1, summarizes the results of this stage on the input data sources. An extracted triple representing a subject-predicate-object tuple, should be treated as correct if it has the correct and full subject, predicate and object. The valid triples constitute instances of the eight semantic relations of the conceptual model of the domain that capture interesting relations between developers and software products.

We followed the above steps and ran the developed tool on Wikidev2.0 data sources. Table 4.1 shows more details about the results of our experiment.

Table 4.1: Syntactic/Semantic Analysis Results for the Wikidev2.0 data sources

	SVN commit	Ticket	Email	IRC chat	Wiki page	Total
Number of Sentences	377	190	480	3,095	1,123	5,265
Number of Annotated XML trees	361	180	441	2,862	938	4,782
Number of Triples	208	77	158	904	258	1,605
$\langle developer\ use/...tool \rangle$	12	1	23	58	13	107
$\langle developer\ write/...language \rangle$	10	5	13	115	24	167
$\langle developer\ work/...task \rangle$	7	0	7	97	13	124
$\langle developer\ create/...artifact \rangle$	161	62	96	540	197	1056
$\langle developer\ fix/...ticket \rangle$	16	3	3	19	0	41
$\langle developer\ check/...revision \rangle$	2	5	0	7	0	14
$\langle developer\ workwith/...developer \rangle$	0	1	16	68	11	96

A closer look at the results of our experiments in Table 4.1 shows that the relation $\langle developer - predicate - artifact \rangle$ is the most common relation in our dataset. The relations $\langle developer - predicate - language \rangle$ stands in the second place, and the relations $\langle developer - predicate - task \rangle$ and $\langle developer - predicate - tool \rangle$, with a small difference go to the third place. The results also show that most of the $\langle developer - predicate - developer \rangle$ relations come from communication artifacts, which are Email messages and IRC chats. The last seven rows show significant se-

mantic relations between domain-specific terms. From these semantic relations we could find useful pieces of information as the following:

- **Expertise of developers:** Triples about programming languages or tools showed that User6 and User9 used mostly PHP, MySQL and PostgreSQL, while User5 and User7 used Flex, Eclipse, and UML2. Some sample sentences are (*User9 tried to run the Php code*, *User5 has started learning Php*).

- **Responsibility of developers:** We looked for the triples about the project tasks. We found that User5 worked mainly on testing, User8 worked on "documentation and testing", User7 worked on "user interface", User9 worked on "designing", and User5, User6, and User7 worked together on "screencast". Some sample sentences are, (*User7 neaten up the UI along the side of UML display*, *User8 take care of documentation*).

- **Developer's contributions to the project:** By searching for the triples about developed classes, we found out that User5 and User6 worked with "XMIParser", User6 and User9 worked on "UMLHandler", User6 worked on "UMLViewer", and User8 worked with "Wikiroamer" and "WikiviewFactor". Two sample sentences from this group are, (*User5 handled associations in XMIParser*, *User8 changes wikiroamer and wikiviewfactor*).

- **Developer's relationships:** The triples including the names of two developers show that User4 had the most relationships and worked with all developers, and that User1 and User3 were in second and third place, respectively. User5 worked mainly with User6, and User9 worked with User6 and User8. Sample sentences include, (*User6 modify User5 's Php file to allow ...*, *User6 and User9 should focus on preparing parser.*).

4.2 Evaluation

To determine the performance of our approach, we evaluated the quality of the extracted information. This section describes the evaluation method that we employed to assess the results of the above experiment.

4.2.1 Empirical Evaluation Method

Evaluation is a crucial component in the field of information extraction from text. This task becomes even more complicated in the case of specific domains, where no standards benchmark corpora are collected, the standards should be used are unclear, and no baseline is available for comparison. An ideal evaluation method for the text analysis method proposed in this thesis involves a manual comparison with respect to an existing manually annotated corpora. Applying this evaluation method involves two steps: the first step is to develop a set of manually created relations by a domain expert. Then the automatically created relations are evaluated against the manually-created ones by human evaluators based on some metrics such as precision and recall.

Our work in this thesis identifies domain-specific terms and their relationships expressed in a sentence. To measure the accuracy of our extraction mechanism, we manually evaluated the extracted entity-relationship-entity triples. The applied evaluation method is based on comparing the triples created automatically by the proposed approach with the reference triples suggested by a domain expert from a sample of sentences selected randomly.

In the evaluation method, the system is evaluated with respect to precision, recall and the F-measure. Precision is the proportion of triples predicted by the system which are correct. Recall is the proportion of correct triples which are predicted by the system. Finally, the F- measure computes the harmonic mean of precision and recall, and can be used as a comprehensive indicator of combined precision and recall values. The Precision, Recall and F-measure are computed by using the following equations:

$$Precision = \frac{Retrieved\ Triples \cap Relevant\ Triples}{Retrieved\ Triples} \tag{4.1}$$

$$Recall = \frac{Retrieved\ Triples \cap Relevant\ Triples}{Relevant\ Triples} \tag{4.2}$$

$$F = \frac{2 * Precision * Recall}{Precision + Recall} \tag{4.3}$$

The following steps describe the detailed scenario for the evaluation method applied to our experimental results [11]:

1. A sample of sentences was selected out of the five data sources. The idea was to find a sample set of informative sentences that reasonably written and contained useful information about the project such as someone (possibly the author of the sentence) knew a tool, reported a bug, or modified a class. These sentences were selected by a domain expert and were considered as a test set for our dataset.

2. A domain expert manually extracted all possible triples after reading and browsing the selected sub-set of sentences. The domain expert was the teaching assistant of the course project and he was quite a professional on the domain. At this point, we obtained a sub-set of *entity-relation-entity* triples. Manually extracted triples were used as a reference set of triples for our evaluation.

3. The triple-extraction tool was run on the collection of sentences extracted from all data sources and created a large set of *entity-relation-entity* triples. From these extracted triples, only the triples related to the selected sentences in the test set were considered for the evaluation process. At this point, we had a sub-set of triples automatically extracted by the developed tool.

4. The triples extracted automatically were compared with the reference triples suggested by the domain expert based on equality or synonymy. The comparison was accomplished by using the evaluator judgments, including myself and another evaluator. This comparison resulted in

- True Positives (TP), i.e., triples found by the tool and by the domain expert.
- False Positives (FP), i.e., triples found by the tool but not by the domain expert.
- False Negatives (FN), i.e., triples identified by the domain expert, but not found by the tool.

5. We computed the prior precision and recall by making a comparison with the reference set of triples. Prior precision is the proportion of extracted triples that match with reference triples. Prior recall is the proportion of reference triples that match with the extracted triples. They were computed as follows:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \tag{4.4}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{4.5}$$

   The above formulas for calculating the precision and recall are the same as the ones represented in 4.1 and 4.2. They are only rephrased by using true positive, false positive, and false negative values.

6. After we calculated the prior precision and recall, the extracted triples not matching with the reference triples were submitted to the domain expert for posterior evaluation.

7. The expert then had the opportunity to suggest some of the non-matching triples as relevant and therefore increase the set of correct triples. Two different augmentations were possible: strict and relaxed. In the strict augmentation, a triple became relevant only if it should have been in the list of reference triples but it had been neglected to be. In the relaxed augmentation, an extracted triple might not necessarily be relevant for the application domain, but became relevant only if the domain expert judged it as a correct and meaningful relation based on the context.

8. Posterior precision and recall were computed after adding the new triples that were considered as correct triples.

In our experiments, we applied the above procedure to measure the precision and recall. The results of our evaluation are described in the following section.

4.2.2 Evaluation Results

We applied the evaluation strategy outlined in section 4.2.1, for the extracted triples. Firstly, a domain expert selected about 4% of the sentences out of the whole dataset as our test set, which resulted in 191 informative sentences. Secondly, the domain expert manually extracted all possible triples from the test set to produce our reference set of the triples. Next, the proposed method was applied to the whole dataset to extract all of triples. Finally, the evaluators compared the triples extracted automatically with the reference triples extracted manually, in order to measure the accuracy of the proposed approach.

Some sample triples are presented in Table 4.2 to show the process of comparing the extracted triples with the reference triples. This comparison resulted in the number of "true positive", "false positive", and "false negative" triples, as depicted in Table 4.3.

Table 4.2: Samples of Comparing Extracted Triples with Reference Triples

Extracted Triples			Reference Triples			Evaluation Results		
subject	verb	object	subject	verb	object	TP	FP	FN
User2	create	script	User2	create	a script to populate the changesets and changesetdetails tables	1		
User2	populate	changesetdetails changesets				1		
User2	populate	tables				1		
User1	add	ticketchanges ; ticket script	User1	add	ticketchanges to ticket script	1		
User2	make	modifications ; table schemas	User2	made	modifications to table schemas	1		
User1	cluster	java code	User1	add	user3 's clustering java code and a script to run it incrementally		1	
User1	cluster	script					1	
			User2	fix	casing issues in the gvapi scripts			1
User7	clean	wikimapflexproject UI	User7	clean up	UI in wikimapflexproject	1		
User6	add	uml	User6	add	support for the uml: Usage relations (relations between packages)		1	
User5	adapt	XMI parser	User5	adapt	XMI parser for eUML2 files	1		
User5	adapt	files					1	

Once the comparison was finished, we computed the prior and posterior precision and recall, using the method we described in the previous section for the evaluation. Table 4.4 presents the overall results of the evaluation. This table also presents the separate results of the evaluation for each data source. The sample set

Table 4.3: Comparison Results for the Extracted Triples

		True Positive	**False Positive**	**False Negative**
Prior Results	SVN commit	70	19	64
	Bug reports	12	10	11
	Email message	5	2	1
	Chat log	27	6	18
	Overall	**114**	**37**	**94**
Posterior Results	SVN commit	75	14	64
	Bug report	14	8	11
	Email message	5	2	1
	Chat log	30	3	14
	Overall	**124**	**27**	**90**

of sentences was not selected from each data source separately. In fact, the sample set was selected from the overall sentences of five data sources. Therefore the sample set contains erratic number of sentences from each data source. For example, it contains a few sentences from the email messages and more sentences from the SVN commits.

Table 4.4: Evaluation Results of the Tool for each Data Source

		Precision	**Recall**	**F-measure**	**Number of Selected Sentences**
Prior Results	SVN commit	79%	52%	63%	122
	Bug report	55%	52%	54%	24
	Email message	71%	83%	76%	5
	Chat log	82%	69%	69%	40
	Overall	**75%**	**55%**	**64%**	191
Posterior Results	SVN commit	84%	54%	66%	122
	Bug report	62%	54%	58%	24
	Email message	71%	83%	76%	5
	Chat log	91%	68%	78%	40
	Overall	**82%**	**58%**	**68%**	191

The overall evaluation results presented in table 4.4 shows that 82% of the triples extracted by the developed tool are correct. This table also shows that the tool can extract 58% of the relevant triples from the input data sources. The following section describes the major reasons for the missing and incorrect triples in our experiment.

4.3 Discussion

In this chapter, we evaluated the proposed framework, based on precision and recall. We tried to verify the quality of the extracted triples against high-quality triples hand-built by a domain expert. This evaluation confirmed that (1) a deeper linguistic analysis leads to good results, and (2) good results can be achieved by using state-of-the-art, off-the-shelf techniques.

The results of the evaluation process (see Table 4.4) indicate better precision than recall. By verifying the results of the evaluation (see Table 4.3 and 4.2), we identified the following reasons for the incorrect and missing triples:

- The errors were mostly due to mistakes in the output of the linguistic analysis tools such as PoS tagger and parser. A common mistake was that, verbs at the beginning of the sentence were often mistaken for nouns, thus causing a lower recall. For example, the verb "add" in the sentence "added non extension directory to repo along with sample UML diagram .", was parsed as an adjective for the noun phrase "non extension directory". Another type of parsing mistake was the sentence, "User1 added user3 's clustering java code and a script", which the parser parsed as "User3 is clustering java code and script."

- A second source for errors was spelling, punctuation and grammatical mistakes. We observed that short textual descriptions of software artifacts were characterized by a low grammatical quality. This problem affected the results from the NLP tools. Obviously, NLP tools perform worse on unedited text than on well-written texts. For example, in the sentence "mimetype to html", a missing space existed between the word "mime" and "type". An example of a spelling error is the word "paramaters" in the sentence "cluster script to handle new paramaters ".

- Sentences without subject were another reason for low recall. Although our tool tried to add a subject for these sentences, some sentences were interpreted in another way by the parser such that the tool could not recognize

them as sentences without a subject. For instance, the sentence "updated cluster script to handle new parameters" has no subject; however, the parser parsed "updated cluster" as a subject and "script" as a verb.

- In our data sources, some sentences were not complete, and the tool could not extract any information out of them. This problem affected the recall of the extracted set. For example, the following is an incomplete sentence: "for testing on the web server."

- A common source of error was that the created domain vocabulary (i.e., the list of terms) was not complete and thus caused a lower recall. Apparently, the quality of the term-extraction process had a direct influence on the quality of the system. For example, it did not include the term "pidgin", and the tool could not extract any triple out of the sentence: "I am using pidgin, it 's free and open source."

- Another reason for the low recall was that the current "extraction patterns" did not cover complex sentences, such as "I am also about half done making a database diff tool for us to compare databases and see what has changed which is going to be part of our testsuite (team parser)."

In order to improve the results, we should consider increasing the recall of the "term extraction" process even at the cost of its precision. Another way to improve the recall would be to extend our original set of "extraction patterns" to cover more complex sentences.

Chapter 5

Conclusions and Future Work

Software development involves communication among developers and coordinating work with others. These activities lead to different kinds of artifacts such as email messages, IRC chat logs, bug descriptions, SVN commit messages, and wiki pages. The communications among software developers contain valuable information about their activities, the issues the team members faced during their work, and the decisions they made. Hence, extracting this information can provide valuable insight into the collaboration of the team members, the relative contributions of each member to the project, and the relation of the various software artifacts to the project requirements.

In this section, we discuss the contributions of the research conducted in the context of this thesis and the more important conclusions that can be drawn from it. We also present some ideas for future work, based on our findings.

5.1 Conclusions and Contributions

In this thesis, we proposed and validated an approach for lexical, syntactic, and semantic analysis of the textual data produced during the life-cycle of a software project. The main underlying motivation was to extract useful pieces of information from textual data sources in the software domain and represent them by using a structured and formal data model that could be queried and automatically explored by end users. The extracted information can be used for several purposes. It can be combined with source code information to better support various software

maintenance tasks and activities. It also can be used for answering questions about different aspects of the software system and its development process.

In order to extract valuable knowledge from heterogeneous textual data, our text analysis system integrates state-of-the art computational linguistic techniques with domain-specific knowledge. The system contains five major steps: data cleaning and pre-processing, term extraction, syntactic analysis (i.e., PoS tagging and parsing), semantic analysis (i.e., semantic annotation, and semantic representation), and pattern extraction.

The extracted information is represented as RDF-style triples. These triples constitute an instance of a rich conceptual model of the domain that captures interesting relations between developers and software products. The triples are stored in a database in order for users to efficiently query them. For outside users of the software, the extracted information can provide insight into the software development practices in order to answer questions such as, "Who is working on what?", "What classes have the developers been working on?", "What have the developers been doing?", "Who has made changes to a class?", and "Who fixed a bug?".

We applied the proposed text-analysis approach to extract information from five different sources of textual information collected through the WikiDev2.0 tool. The resources currently integrated into WikiDev 2.0 are (a) subversion control repositories, (b) ticket information or bug reports, (c) wiki pages, (d) mailing list archives, and (e) IRC chat logs.

To get an insight about the performance of the information produced by the tool, we verified the quality of the extracted triples against high-quality hand-built triples by a domain expert. We evaluated the results of this experiment, based on precision and recall. This evaluation resulted in 82% for the precision, 58% for the recall, and 68% for the F-measure.

The results of our experiment and the evaluation stage allow us to draw the following conclusions.

- A deeper linguistic analysis leads to good results. The dependency relationships based method seems to perform better than the surface lexical analysis. Firstly, the dependency relations increases the performance of the triple ex-

traction from the corpus. Secondly, the richer dependency information makes writing and establishing extraction patterns much easier.

- Good results can be achieved by using state-of-the art, off-the-shelf techniques. The text analysis process uses a number of standard NLP methods. These methods include first, cleaning the textual input data and dividing them into individual sentences by using a sentence splitter to detect sentence boundaries. Then, running a statistical parser and Part-of-Speech (PoS) tagger that assigns syntactic labels (e.g., noun, verb, and adjective) to each sentence word. Finally, the grammatical relationships between pairs of words in each sentence are created by using a dependency parser.

- Short textual descriptions of software artifacts are characterized by having low grammatical quality and using a specific language that make them complicated for automatic analysis. In general, analyzing software artifacts tends to be very project-specific, and the lexicon of a software project is often unique. Therefore, software domain vocabularies (i.e., word lists) should be tuned appropriately.

- The information extracted by using our approach (i.e., the extracted triples and annotated xml trees) can potentially be useful for many software engineering activities such as reverse engineering, traceability, program comprehension, software reuse, software maintenance and recovery. The extracted triples help us to understand who is working on what and how his/her work affects other group members. Such knowledge can also be relevant to maintainers.

- Our approach is domain-independent; thus, the developed tool is indeed applicable in other domains.

This thesis makes the following contributions to the field.

The first characteristic and strength of the approach is that it deals with multiple heterogeneous textual sources. Indeed, the system contains different steps for dealing with unstructured data sources. In particular, the challenge we address here

is not so much the development of novel methods, but rather, the improvement of scalability in order to deal with and explore a wider range of textual data sources.

The second strength of our approach is that it relies on unsupervised methods for analyzing textual sources around software artifacts, and thus does not need manual rules or training sets. Our text analysis system integrates natural language processing techniques and statistical methods, with domain-specific knowledge, in order to understand the semantics conveyed by the textual artifacts. Within the text analysis system, we made use of a number of standard NLP tools for the task of syntactic analysis, such as sentence splitter, pos tagger, and dependency parser. We also generated domain-specific vocabulary based on the linguistic information and frequency of the terms.

Compared to other existing approaches, the strength of our technique is that, it provides a deep analysis of textual artifacts, rather than a lightweight analysis that focuses only at the superficial level. We applied a thorough lexical, syntactic and semantic analysis in an integrated manner to five different sources of software artifacts, to extract valuable knowledge about various aspects of the software life-cycle. We proposed a term extraction and semantic annotation method to obtain information about individual software terminologies mentioned in the text. The developed system also detects predicate-argument structures in order to extract semantic relations between the identified terms. Our approach represents the extracted semantic information as RDF-style triples.

5.2 Future Work

In this thesis, we proposed an approach for analyzing the textual data from software projects to extract valuable information. Although, we covered a variety of issues on this topic, we envision that some improvements can be done in the context of enhancing the method. Many directions for future research on the ideas presented in this thesis, can be considered. Some of them are outlined below.

Extracted triples can be used for diverse purposes, from answering question about different aspects of the software system to software maintenance applications.

One possible future work would be developing a domain-specific query language (based on our underlying conceptual model represented in Figure 3.8) for flexible question-answering on the project lifecycle.

Future studies can also investigate how the information from more traditional repository mining approaches can be combined with and complemented by information extracted from textual data, and whether the text can provide further evidence to support the information inferred by using these other approaches.

As another improvement, the term extraction method could be extended to increase the list of terms in the domain vocabulary (see Section Term Extraction in Chapter 3). Our term extraction method could be improved by applying multiple-word term extraction algorithms, and by using methods that consider synonymy (e.g., WordNet synset). Also, augmenting the vocabulary by including entities from the source code would be a straightforward method for improvement.

The pattern extraction method could be extended by adding more lexical-based extraction patterns to complement the current high coverage patterns (see Section Pattern Extraction in Chapter 3). Extending our extraction patterns may lead to the definition of more extraction rules. In this thesis, we defined three rules, but according to our evaluation results, our rules do not cover some complicated sentences.

Another possible improvement would be to extend our underlying conceptual model to cover more RDF-triple patterns (see Figure 3.8). This extension might happen when a new concept is added to our domain vocabulary, or when a new type of relationship between existing concepts is discovered.

Finally, an interesting extension to our work would be visualizing the extracted information to represent the interesting relations between developers and software products in a graphical model.

Bibliography

[1] The stanford parser: A statistical parser. http://nlp.stanford.edu/software/lex-parser.shtml, 2010.

[2] Tapor: Text analysis portal for research. http://taporware.mcmaster.ca/ taporware/textTools/, 2011.

[3] Anupriya Ankolekar, Katia Sycara, James Herbsleb, Robert Kraut, and Chris Welty. Supporting online problem solving communities with the semantic web. In *Proceedings of the 15th international conference on World Wide Web (WWW 2006), Edinburgh, Scotland*, pages 575–584. ACM Press, 2006.

[4] Ken Bauer, Marios Fokaefs, Brendan Tansey, and Eleni Stroulia. Wikidev 2.0: discovering clusters of related team artifacts. In *CASCON '09: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 174–187, New York, NY, USA, 2009. ACM.

[5] Andrew Begel, Yit P. Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE '10, Cape Town, South Africa, 1-8 May 2010*, pages 125–134, New York, NY, USA, 2010. ACM.

[6] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143, New York, NY, USA, 2006. ACM.

[7] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Software engineering - product quality - part 1: Quality model. Technical report, 2001.

[8] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[9] Kalina Bontcheva, Marin Dimitrov, Diana Maynard, Valentin Tablan, and Hamish Cunningham. Shallow methods for named entity coreference resolution. In *Proceedings of the 9th conference on Traitement Automatique des Langues Naturelles, TALN 2002*, June 2002.

[10] Kalina Bontcheva and Marta Sabou. Learning ontologies from software artifacts: Exploring and combining multiple sources. In *Proceeding of the 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE)*, 2006.

[11] Paul Buitelaar, Philipp Cimiano, and Bernardo Magnini, editors. *Ontology Learning from Text: Methods, Evaluation and Applications*, volume 123 of *Frontiers in Artificial Intelligence*. IOS Press, July 2005.

[12] Marie catherine De Marneffe, Bill Maccartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *In LREC 2006*, 2006.

[13] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. Gate: A framework and graphical development environment for robust nlp tools and applications. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL'02), Philadelphia, PA, USA*, 2002.

[14] Marie-Catherine de Marnee and Christopher D. Manning. Stanford typed dependencies manual.

[15] Marie-Catherine de Marneffe and Christopher D. Manning. The stanford typed dependencies representation. In *Proceedings of the Workshop on Cross-framework and Cross-domain Parser Evaluation, COLING'08*, pages 1–8, Stroudsburg, PA, USA, Aug 2008. Association for Computational Linguistics.

[16] Neil A. Ernst and John Mylopoulos. On the perception of software quality requirements during the project lifecycle. In *REFSQ*, pages 143–157, 2010.

[17] Katerina T. Frantzi and Sophia Ananiadou. Automatic term recognition using contextual cues. In *In Proceedings of 3rd DELOS Workshop*, 1997.

[18] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 175–184. ACM, 2010.

[19] Maryam Hasan, Eleni Stroulia, Denilson Barbosa, and Manar Alalfi. Analyzing natural-language artifacts of the software process. In *Early Research Achievement track of the 26th IEEE International Conference on Software Maintenance (ICSM'2010), Timisoara, Romania, September 12-18, 2010*, 2010.

[20] Abram Hindle, Neil A. Ernst, Michael W. Godfrey, Richard C. Holt, and John Mylopoulos. Automated topic naming to support analysis of software maintenance activities. In *The 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, Hawaii*, Vancouver, 2011. in press.

[21] Abram Hindle, Daniel M. Germán, Michael W. Godfrey, and Richard C. Holt. Automatic classification of large changes into maintenance categories. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*, pages 30–39, 2009.

[22] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM.

[23] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Release pattern discovery: A case study of database systems. In *23rd IEEE International Conference on Software Maintenance, ICSM 2007, October 2-5, 2007, Paris, France*, pages 285–294, 2007.

[24] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. What's hot and what's not: Windowed developer topic analysis. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 339–348. IEEE, 2009.

[25] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Software process recovery using recovered unified process views. In *Proceedings of the International Conference on Software Maintenance (ICSM 2010)*, 2010.

[26] Shih-Kun Huang and Kang min Liu. Mining version histories to verify the learning process of legitimate peripheral participants. In *MSR*, 2005.

[27] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall, second edition, February 2008.

[28] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, 2007.

[29] Kyo Kageura and Bin Umino. Methods of automatic term recognition: A review. In *Terminology 3(2)*, pages 259–289, 1996.

[30] Dan Klein and Christopher D. Manning. Natural language grammar induction using a constituent-context model. In *Neural Information Processing Systems, NIPS*, pages 35–42, 2001.

[31] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *In Proceedings of the 41st Meeting of the Association for Computational Linguistics*, pages 423–430, 2003.

[32] L. Lopez-Fernandez, G. Robles, and J. M. Gonzalez-Barahona. Applying social network analysis to the information in cvs repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 101–105, 2004.

[33] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, New York, NY, USA, 2008. ACM.

[34] Denys Poshyvanyk. Using information retrieval to support software maintenance tasks. *Software Maintenance, IEEE International Conference on*, 0:453–456, 2009.

[35] Denys Poshyvanyk and Andrian Marcus. Using information retrieval to support design of incremental change of software. In *Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering*, ASE '07, pages 563–566, New York, NY, USA, 2007. ACM.

[36] Peter C. Rigby and Ahmed E. Hassan. What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list. In *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*, page 23, 2007.

[37] Juergen Rilling, Yonggang Zhang, Wen Jun Meng, René Witte, Volker Haarslev, and Philippe Charland. A Unified Ontology-Based Process Model for Software Maintenance and Comprehension. In *Models in Software Engineering: Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *LNCS*, pages 56–65. Springer Berlin/Heidelberg, 2007.

[38] Jie Tang, Hang Li, Yunbo Cao, and Zhaohui Tang. Email data cleaning. In *Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, KDD '05*, pages 489–498, New York, NY, USA, 2005. ACM Press.

[39] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

[40] Ren Witte, Yonggang Zhang, and Juergen Rilling. Empowering software maintainers with semantic web technologies. In *European Semantic Web Conference*, pages 37–52, 2007.

[41] Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald Gall. Supporting developers with natural language queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 165–174, 2010.

[42] Takashi Yoshikawa, Shinpei Hayashi, and Motoshi Saeki. Recovering traceability links between a simple natural language sentence and source code using domain ontologies. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 551–554, 2009.

[43] Liguo Yu, Srini Ramaswamy, and Chuanlei Zhang. Mining email archives and simulating the dynamics of open source project developer networks. In *4th International Workshop on Enterprise Modelling and Simulation, Montpellier, France, June 16-17, 2008*, 2008.

[44] Yonggang Zhang, Ren Witte, Juergen Rilling, and Volker Haarslev. An ontology-based approach for traceability recovery. In *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM'06)*, pages 36–43, 2006.