ASSESSING THE QUALITY FACTORS FOUND IN IN-LINE DOCUMENTATION WRITTEN IN NATURAL LANGUAGE: THE JAVADOCMINER

Ninus Khamis

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science (Software Engineering)
Concordia University
Montréal, Québec, Canada

April 2011

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Ninus Khamis**

Entitled: **Assessing the Quality Factors found in In-line Documentation Written in Natural Language: The JavadocMiner**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr. Hovhannes A. Harutyunyan

_____ Examiner
Dr. Benjamin C. M. Fung

_____ Examiner
Dr. Peter Grogono

_____ Supervisor
Dr. Juergen Rilling

_____ Supervisor
Dr. René Witte

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

Assessing the Quality Factors found in
In-line Documentation Written in Natural Language:
The JavadocMiner

Ninus Khamis

An important software engineering artifact used by developers and maintainers to assist in software comprehension and maintenance is source code documentation. It provides the insight needed by software engineers when performing a task, and therefore ensuring the quality of documentation is extremely important. In-line documentation is at the forefront of explaining a programmer's original intentions for a given implementation. Since this documentation is written in informal natural language, ensuring its quality needs to be performed manually. In this works, we present an effective and automated approach for assessing the quality of in-line documentation using a set of heuristics, targeting both the quality of language and consistency between source code and its comments. Our evaluation is made up of three parts: We first apply the JavadocMiner tool to the different modules of two open source applications (ArgoUML and Eclipse) in order to automatically assess their intrinsic comment quality. In the second part of our evaluation, we correlate the results returned by the analysis with bug defects reported for the individual modules in order to examine connections between natural language documentation and source code quality. And finally, we compare the comment quality results generated using our JavadocMiner with the quality assessments performed manually by undergraduate and graduate computer science students.

Acknowledgments

With the utmost gratitude, I acknowledge the guidance, enthusiasm, and inspiration I received from my supervisors, Dr. Juergen Rilling and Dr. René Witte. Their patience and persistence has made it possible for me to excel as a research student, and more importantly, as an individual. I would have been lost without them.

Equal thanks go to the many people who have taught me over the years: my high school teachers (especially Mr. Wagner), my undergraduate teachers at Toronto (especially Ravinder Singh, Kanti Akhtar, Dr. Pajkowski, and Denise Simanic), and my graduate teachers (especially Dr. Haarslev).

I also take with me many cherished moments and good memories from my colleagues at the Ambient Software Evolution Group (ASEG), as well as the Semantic Software Lab (SSL). Together we created a stimulating and fun environment to work in.

Lastly, I would like to express my deepest appreciation to my parent's Shelmon and Vivian, and siblings Raymond and Jessica for their enormous support, infinite patience, and unwavering belief in me. To them I dedicate this thesis.

Contents

List of Figures

List of Tables

List of Accronyms

SSL	Semantic Software Lab
ANNIE	A Nearly New Information Extraction
API	Application Programming Interface
AST	Abstract Syntax Tree
DL	Description Logic
GATE	General Architecture for Text Engineering
GIF	Graphics Interchange Format
IDE	Integrated Development Environemnt
IE	Information Extraction
HLT	Human Language Technologies
HTML	HyperText Markup Language
JAPE	Java Annotation Pattern Language
JAR	JAVA Archive
JPEG	Joint Photographic Experts Group
MCC	McCabe's Cyclomatic Complexity
OSS	Open Source Project
OWL	Web Ontology Language
POS	Part of Speech
PR	Processing Resource
SLOC	Source Lines of Code
NLP	Natural Language Processing
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

Chapter 1

Introduction

This thesis is concerned with generating a corpus from source code and source code comments, using linguistic analysis to assess the quality of in-line documentation based on a set of heuristics, and providing users with instant results of the analysis. Within this chapter we discuss the motivation behind our efforts and give an overview of the structure of this thesis.

1.1 Motivation

Over the last decade, software engineering processes have constantly evolved to reflect cultural, social, technological, and organizational changes. Among these changes is a shift in the development processes from document-driven to agile development, which focuses mainly on software development rather than documentation. This ongoing paradigm shift leads to situations where source code and its comments often become the only available system documentation capturing program design and implementation decisions. Software maintenance and evolution (SME) is an important sub-field of software engineering. For software systems with a long lifetime, SME-activities account for 50-70% of the total software life-cycle costs [Kos10]. As stated in Lehman's first law [LPR98], a software application must continually evolve to reflect different requirements as they emerge; otherwise, the application eventually becomes unusable. An important aspect of software engineering is the process of reading and trying to understand source code in order to perform software maintenance tasks [dSM09]. When developers and maintainers find it difficult to comprehend source code, SME

tasks become increasingly prone to error [dSM09].

A significant amount of software engineering artifacts contain information written in informal natural language, i.e., version control commit messages or bug reports. Source code comments are essential when trying to perform software comprehension and maintenance tasks. Studies have shown that the effective use of comments "can significantly increase a program's comprehension" [NLC03], yet the amount of research focused towards the quality assessment of in-line documentation is limited [PTZ09].

Traditional software engineering metrics, such as Source Lines of Code (SLOC) or McCabe's Cyclomatic Complexity (MCC), are of little or no use when attempting to measure the quality of source code comments. Since in-line documentation is written using informal natural language, the only means of assessing the quality of in-line documentation is by performing time-consuming manual code checks.

Another motivation for our focus on in-line documentation is its close proximity to source code. This enables us to perform additional analysis that focus on the consistency between code and documentation, which is known to often degrade due to changes in source code not being reflected in their comments.

1.2 Contribution

The challenging nature of SME has driven researchers to seek solutions that facilitate software maintenance by *(1)* examining new ways to analyze and interpret software engineering data and *(2)* transferring the results achieved to future professionals [Kos10]. Creating solutions that assist in software maintenance and evolution tasks by studying the software's readability has been the effort of researchers in the past [EM82, BWKG05, YWA05, JH06, dSM09]. Within this work, we assess the quality of in-line documentation found in source code based on a number of quality factors. To evaluate our approach, we apply our analysis on multiple versions of a software project, to analyze how the quality of the documentation increases or decreases over time. We also attempt to correlate the different measures with reported bug defects, in order to determine which of the measures can be used to identify potential problem areas of a software project. Finally, we re-establish traceability links between different software engineering artifacts, a task manually performed by developers when

trying to understand unfamiliar code. We also show how existing software engineering tools can be enriched using results obtained from Natural Language Processing (NLP) analysis.

Throughout this writing, the major aspects of our work is divided into two major categories: *(1)* Generating a corpus from source code and and source code comments to be used as input for NLP systems and *(2)* analyzing the quality of source code comments in relation to the different quality factors. The section concerned with source code comment analysis is further split into three subsections: *(1)* the internal (natural language (NL)) quality analysis of source code comments, *(2)* code/comment consistency (i.e., how well the comments match the described source code segments), and finally *(3)* the generation of traceability links between the different parts of a Javadoc comment and other software engineering artifacts, such as version control and issue tracking systems.

Research Observations: In recent years, the field of Natural Language Processing (NLP) has enabled the implementation of a number of robust analysis techniques that can assist users in content analysis [PL08]. While other domains already benefit from NLP applications, the analysis of software engineering artifacts written in natural language continues to be undersold.

Our work demonstrates how NLP techniques can be used to automate the quality assessment of in-line documentation. We also illustrate how potential problem areas of a source code implementation can be identified by assessing the quality of the code's documentation. We also use ontology models to automatically establish traceability links between the different software engineering artifacts. A task manually performed by developers and maintainers when attempting to modify unfamiliar source code.

1.3 Structure of the Thesis

This thesis is divided into nine chapters. In this chapter we discussed the motivation behind our efforts and we also described how NLP can be used to assist in evaluating the quality of in-line documentation.

In Chapter 2 we cover the background material related to our work. Namely, the importance of in-line documentation quality on software comprehension and maintenance. We also discuss the fundamentals of NLP, and how they are used to perform

linguistic analysis. We present an application of NLP that uses algebraic expression to generate a readability index capable of measuring the complexity of text written in natural language. Finally, we end the chapter with a discussion on the use of ontology models to represent information.

Chapter 3 covers the related work, where we compare similar systems focused on analyzing source code comments. We begin by evaluating the different applications concerned with building a corpus from source code and source code comments, by comparing the output generated by each tool. The evaluation of past efforts concerned with the analysis of source code and source code comments is categorized into three major sections: *(1)* the quality analysis of source code comments, *(2)* the consistency analysis between source code and source code comments, and finally *(3)* the elicitation of traceability links between source code comments and the various software engineering artifacts such as version control and issue tracking systems.

In Chapter 4, we discuss the requirements analysis of the different components that make up our system. Based on the quality assessments provided by previous work, we define and detail the set of quality factors that will be used in our quality assessment of in-line documentation written in natural language.

The principal approach used to create our JavadocMiner application is discussed in Chapter 5. We begin by explaining the process of building a corpus from source code and source code comments, followed by a discussion on how we plan to satisfy the set of comment quality assessment requirements. We end the chapter by explaining the design of the source code comment ontology used to model the set of concepts and relations found in Javadoc, and how the ontology can be linked with other software engineering artifacts.

The implementation of the various components that make up the JavadocMiner system is discussed in Chapter 6. The chapter begins with an overview of the entire system; we then describe the implementation of our Semantic Software Lab Javadoc Doclet, which we use to generate the input documents needed for our analysis. We also discuss the NLP framework that was used to implement the core of our JavadocMiner system. We later cover the process of exporting the annotations created by our JavadocMiner NLP pipeline to an OWL model using a component developed by us called the OwlExporter [WKR11].

The evaluation of our system is covered in Chapter 7. We begin the chapter by

discussing the means of generating a corpus using two open source projects. We then discuss how the JavadocMiner was used to analyze the quality of in-line documentation found in different versions of the two open source projects. Further experiments discussed in the chapter also include correlating the results returned by the JavadocMiner with bug defects reported using the project's issue tracker system, and comparing the results generated by the JavadocMiner with the manual assessment conducted by under and graduate students.

Chapter 8 is dedicated to a summary of the work discussed herein, and a preview of future work. Chapter 9 lists accepted and currently being reviewed publications pertaining to our work.

Chapter 2

Background

In this chapter, we discuss the background material related to our work, in particular the impact of in-line documentation quality on software maintenance, followed by an overview and definition of the field of NLP, and finally, the use of ontology models to represent knowledge.

2.1 Source Code Comments and Impact on Software Maintenance

With millions of lines of code written every day, the importance of good documentation cannot be overstated. Well-documented software components are easily comprehensible and therefore, maintainable and reusable. This becomes especially important in large software systems [LB85]. Since in-line documentation comes in contact with the various stakeholders of a software project, it needs to effectively communicate the purpose of a given implementation to the reader. However, the only means of assessing the quality of in-line documentation currently is through performing time-consuming manual code reviews.

The efforts of developers and maintainers are constantly being shifted to other software projects, and as a result, documentation becomes the only means of communicating a developer's original intentions. Without documentation, future developers and maintainers run the risk of making dangerous assumptions about the source code, scrutinizing the implementation, or even interrogating the original developer if possible [Kot00]. Developers should prepare these comments when they are coding, and

update them as the programs change. There exist different types of guidelines for in-line documentation, often in the form of programming standards such as GNU[1], and GDSG[2]. In general, each program module should contain detailed descriptions, purpose, and rationale for the module [Kot00]. Such comments may also include references to subroutines and descriptions of conditional processing. Comments for specific lines of code may also be necessary for unusual coding. For example, an algorithm (formula) for a calculation may be preceded by a comment explaining the source of the formula, the data required, the result of the calculation, and how the result is used by the program.

Writing in-line documentation is a painful and time-consuming task [Kot00], which often gets neglected due to release or launch deadlines. With such deadlines pressuring the development team, it becomes increasingly important to prioritize their tasks. Since customers are mostly concerned with the functionality of an application, implementation and bug fixing tasks receive a higher priority compared to documentation tasks. Furthermore, finding a balance, describing all salient program features comprehensively and concisely is another challenge programmers face while writing in-line documentation [Zok02]. Ensuring development programmers use the facilities of the programming language to integrate comments into the code, and to update those comments, is an important aspect of software quality. Even though the impact of poor quality documentation is widely known, there are few research efforts focused towards the automatic assessment of in-line documentation quality [SDZ07].

A survey conducted in [FL02] aimed at determining the reasons for documentation not being maintained at the same rate as changes to the source code. The participants of the survey "agree that documentation tools should seek to better extract knowledge from core resources", such as the system's source code. The study found that the developers preferred documentation generating tools that are closely integrated with source code, thereby reducing the amount of effort needed to document the software system.

[1]GNU Coding Standard, http://www.gnu.org/prep/standards/standards.html
[2]GDSG Coding Standard, http://library.gnome.org/devel/gdp-style-guide/stable/

2.1.1 In-line Documentation and Javadoc

Literate programming was suggested in the early 1980's by Donald Knuth [Knu84] in order to combine the process of software documentation with software programming. Its basic principle is the definition of program fragments directly within software documentation. Literate programming tools can further extract and assemble the program fragments as well as format the documentation. The extraction tool is referred to as *tangle* while the documentation tool is called *weave* [Knu84].

Single-source documentation, like Javadoc [Kra99], also fall into the category of documents with inter-weaved representation. Contrary to the literate approach, documentation is added to source code in form of comments that are ignored by compilers. Given that programmers typically lack the appropriate tools and processes to create and maintain documentation, it has been widely considered as an unfavourable and labour-intensive task within software projects [Bro83]. Documentation generators currently developed are designed to lessen the efforts needed by developers when documenting software, and have therefore become widely accepted and used. In Figure 1, we show an example of a constructor documented using Javadoc.

```
/**
* Manages the event changes of elements within a UML model,
* and uses the {@link ActivityGraphsHelper} helper.
* @author Bob Tarling
* @param source The bean that fired the event.
* @param propertyName The programmatic name of the property
*                     that was changed.
* @param oldValue The old value of the property.
* @param newValue The new value of the property.
* @param originalEvent The event that was fired internally
*      in the Model subsystem that caused this.
*/
public AttributeChangeEvent(Object source, String propertyName,
        Object oldValue, Object newValue, EventObject originalEvent)
```

Figure 1: A Javadoc Comment for an ArgoUML Constructor

In order for humans and compilers to differentiate between source code and documentation, a specific documentation or programming syntax has to be used. Javadoc comments added to source code are distinguishable from normal comments by a special comment syntax (`/**`) as shown in Figure 1. The Javadoc tool also provides an

API to implement custom extraction and transformation routines [Kra99]. A generator (similar to the weave tool within literate programming) extracts these comments and transform the corresponding documentation into a variety of output formats, such as HTML, LaTeX, or PDF. The description immediately following the special comment syntax is known as a *doc comment*.

Using the Java source code and source code comments, the Javadoc tool generates API documentation in HTML. In Figure 2, we show a small section of an API document generated using the Javadoc tool.

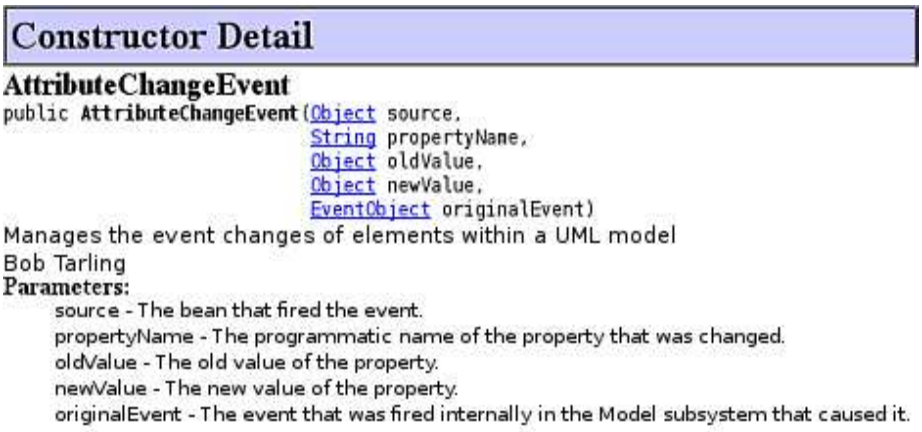

Figure 2: Documentation Generated using Javadoc for an ArgoUML Constructor

Most tools also provide specific tags within comments that influence the format of the documentation produced or the way documentation pages are linked. In terms of Javadoc comments, there are two types of tags used for formatting:

Block Tags: These are placed only in the tag section that follows the doc comment. Block tags are denoted using the annotation "@tag-name".

In-line Tags: These can be placed anywhere in the doc comment and in the comments of the block tags. In-line tags are denoted using the "@tag-name" annotation.

In Tables 1 and 2 we list and describe the most commonly used block and in-line tags found within a Javadoc comment.

Even during early stages of implementation, the Javadoc tool can be used to process pure stubs (classes with no method bodies), enabling the comment within the stub to explain what future plans hold for the created identifiers.

Table 1: Block Tags Commonly Used in a Javadoc Comment

Tag	Description
@author <value>	Used to specify the main contributor of the design or implementation of the system.
@version <value>	Specifies the version of the implementation using the Source Code Control System (SCCS). This feature however is not supported by current versioning systems such as Concurrent Version Systems (CVS) or Apache Subversion (SVN).
@param <value> <description>	Specifies the parameter name, and describes the purpose of a parameter included in the parameter list of a constructor or method. The Java convention calls for the "first noun in the description to be the data type of the parameter."
@return <type> <description>	Specifies the return type and gives a description on what the constructor or method is returning.
@throws <type> <description>	Specifies the thrown type and description for methods that contain any checked exceptions.
@see <value>	Directs the reader to a different part of the application that contains the local declaration of an identifier.

Table 2: Inline Tags Commonly Used in a Javadoc Comment

Tag	Description
{@link <url>}	Converts the text to an HTML hyperlink pointing to the documentation of a given class.

Different types of comments are used to document the various types of identifiers. A class comment should provide insight on the high-level knowledge of a program, for example, which services are provided by the class, and which other classes make use of these services [NLC03]. A method comment, on the other hand, should provide a low-level understanding of its implementation [NLC03].

2.1.2 Javadoc Writing Guidelines

When writing comments with the Javadoc tool, there are a number of quality guidelines described in the Java[3] specification that need to be followed to ensure the tool

[3]Java, http://www.java.com/en/

is being used effectively[4]. The specifications include syntactic details, such as:

1. A Javadoc comment must appear directly before a class, field, method or constructor declaration.

2. Each method parameter must be documented using the "@param" tag. The tag is followed by the name (not data type) of the parameter, followed by a description of the parameter.

3. Having an explicit "@return" tag documenting the return type of a method makes it easier for someone to find the return value quickly.

4. The checked exceptions of a method must be documented using the "@throws" tag. The tag is followed by the data type of the exception.

Internal NL quality details are also described in the specifications such as:

1. When documenting a certain method using Javadoc comments, the descriptions need to begin with verb phrases. For example, "Gets the label of this button." (preferred) "This method gets the label of this button." (avoid).

2. Avoid the use of abbreviations when writing source code comments.

3. Add description beyond the API name. If the in-line documentation merely repeats the identifier's name in sentence form, the reader will not be able to gain any useful information. For example, the comment "Sets the tool tip text." for the method "setToolTipText(String text)" adds no additional information that the reader could not have gathered by looking at the method's identifier.

4. Use a third person (declarative) rather than second person (prescriptive) writing style when creating in-line documentation. for example, "Gets the label." (preferred) "Get the label." (avoid).

5. Class/interface/field descriptions can omit the subject and simply state the object. E.g., "A button label." (preferred) "This field is a button label." (avoid).

[4]How to Write Doc Comments for the Javadoc Tool, http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

6. Use "this" instead of "the" when referring to an object created from the current class. For example: "Gets the toolkit for this component." (preferred) "Gets the toolkit for the component." (avoid).

Considering the entire set of guidelines included in this specification, it becomes often difficult for a developer, documentation writer and even conformance tester who are manually checking the Javadoc comment quality, to recall all guidelines while writing/assessing Javadoc comments. Existing Javadoc comment quality analysis tools such as the *Doc Check Doclet*[5], and Checkstyle[6] provide a quality assessment based on the syntactic guidelines. However, more analysis can potentially be applied on Javadoc comments, measuring factors such as comprehension, efficiency, and usefulness.

2.2 Foundations of Natural Language Processing

Naturally occurring text, either spoken or written, can be of any language, mode, genre, etc. The text is used by humans to communicate with one another [Lid01]. Natural Language Processing (NLP), a sub field of Artificial Intelligence (AI) and Computational Linguistics (CL), aims to accomplish human like language processing using computations. When trying to understand text, humans apply a number of language analyzing processes. NLP attempt to represents those processes using a *range of computational techniques* [Lid01]. Chronological *levels of linguistic analysis* are needed when performing NLP. The multiple levels are executed in sequence to process language. Each level generates additional information that succeeding levels may or may not use. What differentiates one NLP service from another is the combination of levels the system uses, which depends on the type of linguistic analysis the language engineer is interested in performing. When stating that the goal of NLP is to *accomplish human-like language processing*, the word "processing" is not to be confused with "understanding". The field of NLP was originally referred to as Natural Language Understanding (NLU) in the early days of AI. It is well agreed today that while the goal of NLP is true NLU, that goal has not yet been accomplished [Lid01].

[5]Doc Check, http://www.oracle.com/technetwork/java/javase/documentation/index-141437.html

[6]Checktyle, http://checkstyle.sourceforge.net/

A full NLU System would be able to:

1. Paraphrase an input text
2. Translate the text into another language
3. Answer questions about the contents of the text
4. Draw inferences from the text

While NLP has made serious strides over the years to accomplish goals 1 to 3, the fact that NLP systems cannot, draw inferences from text without the help of humans, NLU still remains the goal of NLP [Cho03].

Natural Language Processing Application: As mentioned earlier, most NLP applications are assembled using levels of linguistic analysis. Due to their complexity, NLP systems require the use of many different subsystems, therefore nearly all NLP systems are built in a modular fashion. Each of the subsystems or modules concentrate on one specific task. Modules that perform common linguistic analysis tasks often re-emerge in different NLP system. New modules need to be implemented to perform a specific type of NLP analysis. Some common linguistic analysis tasks that are found in many different NLP systems are:

Tokenizing: Annotating tokens of a text according to their symbolic structure (see Appendix C.2).

Sentence Splitting: Segmenting the tokens (or "annotates" above) of a text into sentences based on their boundaries (see Appendix C.3).

Part-of-Speech Tagging: Tags tokens of a text with corresponding parts of speech (e.g., Hepple Tagger C.4).

Named Entity Recognition: Detecting elements within text such as, for example, `persons`, `organizations`, and `locations` using Gazetteer lists, rule-based, or machine learning based, extraction techniques (see Appendix C.5).

Noun and Verb Phrase Chunking: Tagging noun (see Appendix C.7) and verb (see Appendix C.8) phrases found within a sentence using sentence parse trees and regular expressions.

Coreference Resolution: Identifying the entities that re-appear in different parts of a text and linking them together using coreference chains.

2.3 Readability Measures

The term "readability" is described as the ability to determine the level of education needed by an individual to understand a given block of text using a set of computations [dSM09]. In the early twentieth century, linguists conducted a number of studies that used people to rank the readability of text [SDZ07]. Such studies require significant resources and therefore, can often not be applied in the context of source code comments.

Extensive work was then put towards mapping a block of text to an algebraic value that corresponds to its readability. In the 1980s, the number of readability formulas was around 200 [DuB06]; however, amongst the the widely used measures were the "Flesch Reading Ease Formula" (1943), Gunning's "Fog Index" (1952), and "Flesch-Kincaid" (1975) [dSM09].

Originally used by a number of U.S. government agencies such as the DoD and IRS to analyze the readability of their technical documents [SDZ07], some of the readability formulas are currently integrated into coding standards such as the GNOME Documentation Library[7].

Some of the major factors that affect the quality of text, and it's readability are:

Legibility: At the surface level, readability is concerned with the visual perception of the typeface and layout [DuB06, Har00].

Comprehension and Retention: This was the focus of "classic readability studies", aiming at matching the knowledge of vocabulary an individual requires to understand and quickly memorize a block of text. Factors within this category include sentence structure, technical knowledge, and the level of reasoning needed to understand the text [DuB06].

Persistence and Efficiency: The 1950s were a time when older manufacturing industries had little demand for advanced readers, and new technologies required

[7]GNOME, http://library.gnome.org/devel/gdp-style-guide/stable/

workers with higher reading proficiency. As a result, the focus of "the new readability" studies were concerned with reading persistence and speed [DuB06].

Common to these readability formulas is that they use various lexical and grammatical features as input, and the output, or readability index, can be described as a value that corresponds to a text's reading difficulty or the grade level [HCTE08]. Furthermore, all readability measures use "the sum total (including all the interactions) of all elements within a given piece of printed material that affect the success a group of readers have with it. The success is the extent to which they understand it, read it at an optimal speed, and find it interesting." [EC49].

Some of the most commonly used readability measure used today are:

Flesch Reading Ease Score: Integrated in word processors such as Microsoft Word, Lotus WordPro, and Google Docs [dSM09], the Flesch Reading Ease Score rates text on a 100 point scale. The higher the value, the easier a text is to read. The formula correlates 0.70 with the 1925 McCall-Crabbs reading tests [MP82], and 0.64 with the 1950 version of the same tests [DuB06]. A "standard" and optimal score for the Flesch measure would range from 60–70 [DuB06]. A score between 90–100 would indicate that the block of text could be understood by an 11 year old and would therefore be overly simplified [DuB06]. A 100 word sample is required to accurately compute a score using the Flesch metric. As we observe in Chapter 7, most comments do not exceed an average of 25 words. Making this readability measure unsuitable for analyzing source code comments.

Fog Index: Robert Gunning had found that magazines, newspapers and business papers used writing styles that were not **direct** and **straight forward**, but rather were full of "fog" and included unnecessary complexity. He realized that much of the reading problem was actually attributed to a writing problem [DuB06]. As a result, he developed the Fog Index, which indicates the number of years of formal education a reader would need to understand a block of text. The measure uses the average sentence length as a grammatical feature and the number of words with more than two syllables. Gunning developed his formula using a 90% correct-score with the McCall-Crabbs reading tests [DuB06]. An optimal score for the Fog Index would be between 8–11. A score above 17 would indicate a block of text that is understandable by graduate level individuals [DuB06].

Flesch-Kincaid Readability Formula: This metric was developed in 1975 by Kincaid, Fishburne, Rogers and Chissom to be used by the U.S. navy to improve the readability of their technical documents [SDZ07]. The measure was created to translate the 0–100 Flesch Reading Ease score into a U.S. grade level. However, the Flesch-Kincaid grade level score is not a simple conversion of the Flesch score, but rather was re-designed to analyze the readability based on persistence and efficiency readability factors [DuB06]. The measure uses a linear combination of mean number of syllables per word and the mean number of words per sentence. It is similar to the Fog Index; however, it calculates a finer grain measure of word length [dSM09]. Based on the observations made by linguists in the past, an optimal score for the Flesch-Kincaid readability index would range from 8–10 [DuB06]. The Flesch-Kincaid Grade Level measure is also widely used, and is implemented in word processors such as Microsoft Word.

Some important factors to remember when mechanically applying formulas to predict the level of complexity of text are:

- The widespread availability of computational power and of readability formulas have contributed to the misapplication and misinterpretation of readability measures [Kla00].
- Scores produced by readability measures are not meant to be interpreted as highly accurate values of the text that they measure, but rather as guides that provide "quick, easy help in the analysis and placement" of text [DuB06].
- Readability formulas can be considered as "screening devices that provide probability scores" for text that is otherwise not easily represented [Kla00].

Put into context of our work, the term readability refers to software readability, which can be described as a programmers ability to "understand the utilization, control flow and procedures written in source code" using the in-line documentation [dSM09]. When creating in-line documentation such as Javadoc comments, writers need to be direct and straight forward, excluding any pointless complexity in the writing.

The open source project GNOME[8] uses the Flesch, Fog, and Kinciad readability

[8]GNOME, http://www.gnome.org/

measures as part of their documentation guide. The readability metrics are used as a "quick assessment of the density of the technical documents". The goal of the GNOME Documentation Standard Guideline (GDSG)[9], is to provide the project's contributors with a framework to write good and consistent documentation, so that users "can expect certain structures and conventions" in the documentation.

2.4 Knowledge Representation using Ontologies

Ontologies offer the formal, and explicit representation of a shared conceptualization needed to model a domain of discourse [Gru93]. As a result, ontologies have recently become the focus of many researchers attempting to model a large amount of information using a formal representation [Bei10].

Using concepts and relationships, ontologies provide the machine translatable constructs needed to represent knowledge. A key benefit to creating ontology models is that they provide a non-proprietary formal language that facilitates knowledge sharing [MCHS09]. An ontology model is effective in representing a large amount of information using a small number of axioms (individuals and relationships). The semantically rich model provides users with a high-level conceptualization of the information, while at the same time allowing them to focus on specific parts of the model. A semantically rich ontology model also provides the formal language capable of linking related information spanning across boundaries. The process of linking ontologies using concept, instance and relationship assertions is called ontology alignment [Bei10].

Revision control and issue tracking systems are some of the many tools that are currently being used to perform software development and maintenance tasks. Software engineering repositories can be either hosted on individual computers or distributed across multiple locations shared using a network. Because of the important and often large amount of information stored in these repositories, researchers are constantly looking for ways to analyze this data to determine a project's maturity, recover architectural thoughts, perform impact analysis, and re-establish traceability links [MRBW10, KBT07, HKST06, ZWRH06]. One of the key challenges when trying to analyze software repositories is the lack of a common representation. Modelling,

[9]GDSG, http://library.gnome.org/devel/gdp-style-guide/stable/

and linking the different artifacts using ontologies is a "prerequisite for interoperability, and unhampered semantic navigation and search" [Bei10].

Ontology population from text is also becoming increasingly important for NLP applications. As a majority of the world's knowledge is encoded in natural language text, automating the population of these ontologies using results obtained from NLP analysis of documents [Cim06] has recently become a major challenge for NLP applications. In the biomedical NLP (bio-NLP) domain, ontologies are being used to support information extraction and semantic search applications [Bei10]. Populated from natural language texts, they offer significant advantages over traditional export formats, such as plain XML. The development of text analysis systems have been greatly facilitated by modern NLP frameworks (e.g., The General Architecture for Text Engineering (GATE)).

Using an OWL model enables us to reuse existing ontologies. OWL models also support importing and extend other ontologies. This feature also makes OWL scalable, because it enables the construction and maintenance of distributed knowledge bases.

Past research on the use of ontologies focus mostly on the conceptualization of the domain, and less about providing an automated means of ontology population.

Web Ontology Language (OWL): The Web Ontology Language (OWL) became a World Wide Web Consortium (W3C) standard in 2004, as a successor to earlier ontology languages, such as DAML+OIL. The aforementioned formal language used by OWL ontologies is called *Description Logic*, or *DL*. Though more expressive than propositional logic, DL concentrates on the decidable fragment of First Order Logic (FOL). The DL language defines the set of concepts, as well as object and datatype properties needed to build an OWL model [FBMNPS07]. An ontology that only contains concepts and relationships is known as a taxonomic representation of the domain of discourse, or *T-Box*. For example, *Wine*, *WineColour*, and *Region* might be concepts that appear in a Wine Ontology[10]. Here, *hasColour* might be an object property relation [FBMNPS07] linking *Wine* and *WineColour* together, and *locatedIn* would be a relationship between *Wine* and *Region*. The concepts and relations discussed thus far would make up the T-Box of the ontology. Apart from the T-Box, DL based languages also use a set of assertions, or *A-Box*, to create statements reflecting

[10]Wine Ontology, http://oaei.ontologymatching.org/tests/102/onto.html

the domain of discourse. Instance assertions make it possible for statements, such as: "Porto" is a type of wine, "Red" is a type of colour and "Portugal" is an instance of the *Region* concept. Furthermore, relationship assertions make it possible to have statements such as "Porto" has the wine colour "Red" and "Porto" is located in the "Portugal" region.

Ontologies modelled using DL can also take advantage of the reasoning services provided by a DL reasoner such as Racer [HM01], Pellet [SPG+07], or FaCT++ [TH06]. Visualizations and queries using SPARQL [PS08] can also be applied on a given knowledge base.

2.5 Summary

In this chapter, we provided the background information related to our work. In Chapter 3, we compare similar efforts related to the different parts of this thesis, namely generating a corpus using source code and source code comments and the analysis of source code comments using different quality factors.

Chapter 3

Related Work

In this chapter, we discuss related work separately for the two major aspects of our work: *(1)* Generating a corpus from source code and source code comments to be used as input for NLP systems, and *(2)* analyzing the quality of source code comments in relation to the different quality factors. The section concerned with source code comment analysis is further split into three subsections: *(1)* the internal quality analysis of source code comments, *(2)* consistency analysis between code and its natural language comment, and finally *(3)* the elicitation of traceability links between source code and the various software engineering artifacts, such as version control and issue tracking systems.

3.1 Corpus Generation from Source Code

The fact extraction and transformation tool JavaML is capable of representing Java source code using an XML representation to support "powerful querying" capabilities [Bad00]. Including Javadoc comments as part of the source code analysis was not included until JavaML 2.0 [ADB04]. However, when looking at the generated XML documents, we found that the main focus of JavaML 2.0 was associating `docComments` with related code structures (i.e., Class, Field and Method), and less about representing the different parts of a Javadoc comment. Along with providing insufficient information, we also found that JavaML included redundant information as part of their analysis. Services such as tokenizing the content found in in-line documentation were included, a feature that can easily be added in our quality assessment of Javadoc

comments. Similar tools, such as JavaCC [Kod04] or Japa[1], also provided little or no support for Javadoc comments.

After determining that current fact extraction and transformation tools were unable to assist in the analysis of Javadoc comment, we quickly turned our attention to transformation tools that make use of the Javadoc extraction tool[2], known as doclets [Kra99]. A number of Javadoc doclets exist that can generate XML files using source code and Javadoc information, such as the `xml-doclet`,[3] Mavens's `XMLDoclet`,[4] and finally the `jeldoclet`[5]. However, when looking at the schema generated by these doclets, we observed that they were not necessarily designed for generating a corpus to be used within NLP applications.

For example, the `xml-doclet` marks up information using only XML tags and elements and does not make use of XML attributes to represent information. As mentioned earlier, XML attributes are interpreted by NLP frameworks as features of an annotation.

The doclet that generates a schema that best satisfies our requirements is the `jeldoclet`. The `jeldoclet` however does not attempt to differentiate between the different types of comments (i.e., Calss, Field, and Method), which could minimize the descriptiveness of the corpus. The `jeldoclet` also does not capture the information provided by Javadoc when a certain class implements or extends another class, as shown in Figure 13 on page 46. The source data being represented and the output format is the same for all XML generating doclets, and the XML documents generated using the doclets mentioned herein can be loaded within an NLP framework. However, how the information is marked-up can drastically change the number of annotations, features and entities that are created, which can have a cascading effect on the rest of processing resource within the NLP application.

Having the most number of annotations, features or entities as a result of how the information is marked up within an XML document is not necessarily beneficial. Providing a schema that enables NLP frameworks to differentiate between what is an annotation, feature, and entity is important when generating an XML document that

[1]Japa, http://www.java2s.com/Open-Source/Java-Document/IDE/tIDE/japa.parser.htm/

[2]Javadoc, http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

[3]XML-Doclet,http://code.google.com/p/xml-doclet/

[4]Maven Doclet,http://maven.apache.org/maven-1.x/

[5]jeldoclet,http://jeldoclet.sourceforge.net/

is to be used as a corpus. None of the existing doclets that we examined were capable of doing so. For example, since the `xml-doclet` marks up all the information using XML tags only, no features are created when the document is loaded within an NLP framework, and the increased number of annotations would actually have a negative impact on the amount of work needed by the language engineers to make use of the generated corpus.

3.2 Quality Analysis of Source Code and Source Code Comments

In this section, we discuss past efforts interested in the analysis of source code and source code comments based on three categories defined throughout this thesis.

3.2.1 Internal Analysis of Source Code Comments

There has been effort in the past that focused on analyzing source code comments, for example, in [BW08] human annotators were used to rate excerpts from Jasper Reports, Hibernate, and jFreeChart as being either "More Readable", "Neutral" or "Less Readable". The authors developed a "Readability Model" that consists of a set of features, such as the average and/or the maximum 1) line length in characters; 2) identifier length; 3) identifiers; and 4) comments, represented using vectors. The heuristics used in the study were mostly quantitative in nature and based their readability scale on the length of the terms used, and not necessarily the complexity of the text as a whole. The authors also made no attempt to measure how up-to-date the comments were with the source code they were explaining.

The authors of [PTZ09] manually studied approximately 1000 comments from the latest versions of Linux, FreeBSD and OpenSolaris. Part of their study was to see how comments can be used in developing a new breed of bug detecting tool, and how comments that use cross-referencing can be used by editors to increase a programmer's productivity by decreasing navigation time. The work attempts to answer questions such as: *(1)* what is written in comments; *(2)* whom are the comments written for or written by; *(3)* where are the comments located; and *(4)* when were the comments written. Results from the study showed that 22.1% of the analyzed comments clarify

the usage and meaning of integers, 16.8% of the examined comments explain implementation, for example, which function is responsible for filling a specific variable and, 5.6% of source code comments describe code evolution such as cloned code, deprecated code and TODOs. The purpose of the study was to classify the different types of in-line documentation found in software and not necessarily assess their quality.

An empirical study of the usefulness of `Class` and `Method` comments on program understanding was investigated in [NLC03]. Part of the effort was also concerned with "what recommendations and guidelines for commenting should be taught and used in beginning Java programming courses" [NLC03]. The authors prepared a questionnaire consisting of 10 questions: 5 focusing on high-level questions, such as what the program does or how one class relates to another, and 5 low-level questions, such as how a certain method is suppose to behave. The answers to the questions were given in the `Class` and `Method` comments. The questionnaire was given to a total of 103 students with varying GPAs and Java experience. Based on the results of the questionnaire, the authors were able to conclude that `Method` comments had more of an impact on program comprehension compared to `Class` comments. Both the authors of [PTZ09] and [NLC03] made no attempt of providing an automated quality assessment, nor was there any major correlations made with other software engineering artifacts.

The authors of [FSH+08] introduce an algorithm that extracts verb information from comments and verb direct object pairs from method identifiers (e.g., `open` → `File`). The authors state that the "particularities of source code structure" hinder a search engine's ability to return relevant information. For example, a search for "add auction" will not find the related method "addEntry" that "adds auctions to a list". The implemented algorithm is able to elicit related information by analyzing the method's parameter type, which in the case of "addEntry" is "AuctionInfo". While the efforts of the authors were mostly focused towards enhancing search capabilities, as we explain in Chapter 4, we are more concerned with identifying method comments that do not add additional value over what could be comprehended using the method identifier.

The only work that we know of that focuses on automatically analyzing quality of API documentation generated by Javadoc was done by the authors of [SDZ07]. The

authors implemented a tool called `Quasoledo` that measures the quality of documentation with respect to its completeness, quantity and readability. Here, we extend the works of [SDZ07] by introducing new quality assessment metrics. We also analyze each module of a software project separately, allowing us to observe correlations between the quality of in-line documentation and bug defects. Both of the efforts mentioned above focus mostly on the evolution of in-line documentation and whether they co-change with source code, and not necessarily on the quality assessment of in-line documentation. None of the efforts mentioned in this section put nearly as much emphasis on correlating the quality of in-line documentation with reported bug defects as we do in this thesis.

3.2.2 Code/Comment Consistency Analysis

Automatically analyzing comments written in natural language to detect code-comment inconsistencies was the focus of [TYKZ07]. The authors explain that such inconsistencies may be viewed as an indication of either bugs or bad comments. The authors present a tool called `iComment` that 1) applies Part of Speech Tagging (POS) on comments, 2) uses statistics to determine the most predominant terms of a comment, 3) uses Decision Tree Learning to generate models from a small set of manually annotated comments, and 4) uses program analysis techniques to detect inconsistencies between code and comments. The tool was applied on 4 large Open Source Software projects: Linux, Mozilla, Wine and Apache, and it detected 60 comment-code inconsistencies, 33 new bugs and 27 bad comments.

Finding regularities in source code and source code comments using Zipf's Law [Zip32] was the focus of [Zha08] and [PP09]. A "lexical analyzer" used to estimate the length of software was implemented in [Zha08]. The analyzer applies "Software Science Estimation", "Magnitude of Relative Error" and Zipf's Law on source code and comments. The authors applied the tool on *Jena*, *Protégé*, *Ant* and nine other software systems. Some of the results of the study found are the most frequently occurring keyword used in *Jakarta Tomcat* is "Public". The authors also argue that because "String" was the most frequent identifier for *JENA*, could act as an indication for needed optimizations. The authors also found that the identifiers commonly used by the different developers were "org", "i", "e", "name", etc. In order to observe the evolution of a single software project, the "lexical analyzer" was applied to multiple

versions of Tomcat. The authors observed that Zipf's Law also held for the multiple versions with no major differences in the rank of words used in the lexicon.

The work described in [AHM+09] defines a Source Code Vocabulary (SV) as being the union of Class Name, Attribute Name, Function Name, Parameter Name and Comment Vocabularies. The work uses a combination of existing tools like `diff` to answer questions; such as how the vocabularies evolve over time, what type of relationships exist between the individual vocabularies, are new identifiers introducing new terms, and finally what do the most frequent terms refer to.

An automated approach in mining abbreviation and acronym expansions from source code and source code comments to enhance software maintenance was the focus of [HFB+08]. Domain specific terms used in comments are often abbreviated in the identifiers. The authors used the example of the word "number", which appeared 4,314 times in the Java 2 Platform, while its abbreviation "num" occurred 5,226 times. The authors argue that "concern location" and source code traceability using software tools could be improved by integrating abbreviation expansion techniques. An algorithm was proposed that *(1)* identifies if a token is a non-dictionary word and *(2)* search as for a potential long form for the given short form using the set of mined words. Fifteen open source projects were manually inspected; some of the observations made by the authors are that dictionaries that include terms from the software engineering domain were hard to find and that some abbreviations also had multiple potential long form candidates. Given such challenges, the authors developed a set of algorithms capable of identifying the short form abbreviations and use regular expressions to determine the potential long form. Although certain aspects of this work might resemble some of the heuristics which we propose herein; however, our work differs in the application of the tools. More specifically, the analysis conducted by the authors of [HFB+08] aims at providing users with more precise search tools capable of including abbreviations, whereas we are more interested in detecting abbreviations as a deterrent of good comment quality.

None of the works mentioned in this section attempted to analyze the quality of Javadoc comments used to generate API documentation. Unlike source code comments that describe a given implementation within a method body, and are used by developers and maintainers, Javadoc API documentation is used by other stakeholders, such as projects managers and conformance testers. Other tools that fall under

the code/comment consistency analysis of in-line documentation are the Doc Check Doclet, and Checkstyle discussed in Chapter 3. Both tools provide quality analysis that are purely syntactic in nature, which can be easily manipulated to give incorrect results (e.g., copying and pasting comments).

3.2.3 External Analysis of Source Code Comments

One of the first research efforts concentrated on creating a "software repository data exchange format" using OWL was [KBT07]. Due to the groups interest in software evolution, the `EvoOnt` knowledgebase modelled information from the version control and issue tracker repositories. The same group created an extension of SPARQL called `iSparql`, which they use to query their knowledge base. iSparql does not introduce any additional syntax, but rather considers the notion of "virtual triples" that are not matched in the ontology graph, but are bound to a resource using SPARQL and joined by iSparql using the different similarity measures. EvoOnt along with iSparql were then used to conduct common software engineering tasks, such as: code evolution visualization, analyzing commit messages and bug reports, and clone detection.

The authors of [HKST06] link data from the different software engineering repositories to identify the components of a software system that can be reused in other projects based on details such as applicability and licensing. Using their approach, the authors are also able to identify developers that have expertise in building systems from a given domain. We are unaware of similar work attempting to link in-line documentation for the purpose of enriching a software engineering knowledge base, built using ontologies.

3.3 Summary

Common to all Java fact extraction and transformation tools that we analyzed is that they provide an Abstract Syntax Tree (AST) representation of the source code. However, most of these parsers either *(i)* did not include sufficient information regarding Javadoc comments, or *(ii)* produced an output that was not designed to be used as input for NLP applications, and therefore could not assist in the analysis of Javadoc comments.

Efforts interested in analyzing source code comments in the past [PTZ09, AHM+09,

TYKZ07, BW08, PTZ09] focused mostly on implementation level documentation describing an algorithm, rather than API level documentation discussing the overall responsibility of a `Class` or `Method`. In cases where the source code is not made available (i.e., closed source software), implementation level documentation may only be available to internal stakeholders of the software project. Individuals outside of the organization attempting to re-use some of the services provided by the source code binaries would therefore need to use the API level documentation generated using tools such as Javadoc. Another common application of source code comment analysis researchers were interested in was geared more towards the enhancement of results returned by search engines [HFB+08, FSH+08], and less about measuring the quality of source code comments using different quality factors. Part of the focus for our study attempts to observe the adverse effects of comment quality on source code quality, using factors such as reported bug defects. Evaluating the quality of comments found in different versions of a software system also allow us to also observe if the quality of source code comments increase or decrease over time. Existing comment analysis tools [BW08, SDZ07] also based their assessments on mostly quantitative and syntactic quality factors. Features such as identifier and comment length, as well as the ratio between undocumented and documented identifiers were used as a measure for comment quality. For our application, we are also concerned with measuring the internal NL quality of source code comments. Using NLP services such as POS tagging, noun and verb group chunking, we can analyze the semantic structure of source code comments. This will us to measure quality factors such as writing style and the usefulness of the source code comment. The internal NL quality analysis of Javadoc comments conducted by [SDZ07] relied mainly on the Flesch-Kincaid readability index to measure comment quality. However, as discussed in Chapter 2, the Flesch-Kincaid readability formula was designed to measure the readability of text based on persistence and efficiency; although these are important for in-line documentation, the readability formula is unable to measure equally important factors such as comprehension and retention. Moreover, there are other aspects of comment quality such formulas are incapable of analyzing, such as active vs. \passive voice, and descriptive vs. \prescriptive writing style, as discussed in greater detail in Chapter 4. Due to their limitations, analyzing texts using readability measures alone can be seen as a misuse of the formulas [Kla00, DuB06].

There have been previous attempts to link source code documentation with either bug defects [TYKZ07] or version control systems [SDZ07, AHM+09]. However, we are unaware of efforts attempting to establish traceability links between source code comments and other software engineering artifacts to facilitate the software development and maintenance process.

Chapter 4

Requirements Analysis

Having discussed and summarized the related work material in Chapter 2, we now consider the requirements of our system. More specifically, we define and detail the list of quality factors that we use to analyze the qaulity of Javadoc comments written in natural language.

4.1 Generating a Corpus from Source Code

Requirement#1.1: Generating Input Documents. Before being able to process source code documents, the code has to be transformed into a more abstract representation. The format of the input documents used for NLP analysis has a large impact on the whole application, and is therefore addressed early on [KWR10]. In some cases an NLP application may, for example, benefit more from the rigid representation of an XML format, versus an application used to process Internet content and designed to analyze HTML documents. Given the large amount of information that exists in source code documents, the JavadocMiner requires a representation capable of modelling the entire information found in source code and source code comments using a format that will facilitate NLP analysis.

4.2 Comment Syntax Analysis

Prior to applying semantic quality assessments on Javadoc comments, we begin our internal NL quality analysis using simple NLP services such as creating tokens, as

well as detecting the use of noun phrases and verb groups within Javadoc comments.

Requirement#2.1: Detecting Number of Words within a JavadocComment. The amount of words contained within a comment can act as an indicator of how much information was provided to document the source code [SDZ07]. As a preliminary means of analyzing the internal quality of comments, the JavadocMiner must compare the amount of information included in similar types of comments, i.e., `Class`, `Method` or `Field`. If a class was assessed as containing a large amount of documentation in comparison to other classes, it could be an indicator for a class with too much responsibility, and a candidate for potential refactoring.

Requirement#2.2: Detecting Use of Abbreviations. Abbreviations such as "GIF" and "JPEG" are easily forgotten after their introduction; however, they are crucial to the understanding of in-line documentation. Included in this analysis are hard to read abbreviations, such as "WYSIWYG", which "can make a reader feel dyslexic" [Kla00]. Abbreviations can reflect the domain specific understanding of a certain community, making it difficult for others to understand. Including abbreviations in technical documentation can hinder a person's ability to comprehend the content. According to the Javadoc guidelines discussed in Chapter 2, the use of abbreviations in comments should be avoided [Kra99]. As a means of monitoring the amount of abbreviations included within a Javadoc comment, the JavadocMiner shall identify and count the abbreviations being used within source code comments.

Requirement#2.3: Detecting Noun and Verb Phrase Usage. Detecting the use of noun and verb phrases provides a basic assessment of the use of well-formed sentences within in-line documentation.

4.3 Internal Comment Quality Assessment

Traditional software engineering metrics such as Source Lines of Code (SLOC) or McCabe's Cyclomatic Complexity (MCC) are of little or no use when attempting to measure the semantic quality of source code comments [SDZ07]. Without adequate measures for comment quality, developers and documentation writers can freely create in-line documentation with the possibility of scrutinizing the readability of the source code implementation due to bad or insufficient documentation quality.

Important quality factors of readability are the comprehension, retention, efficiency, and persistence of text [DuB06]. Such characteristics of text also need to be applied to in-line documentation, which needs to effectively and efficiently explain the motivation for a given source code implementation to future developers and maintainers.

The JavadocMiner shall also be able to measure the readability of in-line documentation based on the different factors that impacts a developer's and maintainer's ability to comprehend and retain the in-line documentation. Along with analyzing the readability of text using readability measures such as the FOG and Kincaid readability index, our JavadocMiner system shall also assess source code comments based on quality factors such as writing style, and voice. An evaluation of text other readability measures are incapable of performing [Kla00, DuB06].

To ensure the proper application and coverage of readability measures, we list and define the set of features used to measure the readability of source code comments (Table 3).

Table 3: JavadocMiner Comment Readability Criterion

Feature	Requirement	FOG	KINCAID	SPW	PWS
Calculate Comprehension and Retention	3.1	✓			
Calculate Efficiency and Perseverance	3.2		✓		
Detect Second Person Writing Style	3.3			✓	
Detect Passive Writing Style	3.4				✓

Requirement#3.1: Calculating Comprehension and Retention. Text with optimal readability levels results in greater and more complete retention, as well as greater acceptability (attractiveness) [DuB06]. Simplifying documentation also allows individuals to focus more on the newly introduced technical terms, and less on trying to understand complicated writing [DuB06]. The source code vocabulary consists of *class names*, *attribute names*, *function name*, *parameter names* and *comments* [AHM+09], which contain many technical terms from the application domain. To assist in software comprehension and maintenance tasks, it is important for Javadoc comments to be easily understood by developers and maintainers. The ability to quickly recall the different components of an application is an equally important software engineering task [Pfl98]. The JavadocMiner shall provide users with a quality assessment of source code comments based on the comprehension and retention of text found in the Javadoc comments.

Requirement#3.2: Calculating Efficiency and Perseverance. "New readability studies" [DuB06] were more concerned with reading efficiency and persistence (or perseverance), and less about comprehension. Studies have shown that better readability increases reading efficiency and perseverance by more than 80% [DuB06]. With more important tasks at hand, developers and maintainers require Javadoc comments that are efficient in explaining the implementation of an application. Maintaining optimal and consistent levels of comment quality based on the efficiency and perseverance quality factors may result in the added productivity of developers and maintainers. To ensure Javadoc comments maintain satisfactory quality levels based on the following readability factors, the JavadocMiner shall assess the quality of source code comments using readability formulas designed to measure the efficiency of text.

Requirement#3.3: Detecting Prescriptive Writing Style. Often found in documents from the legal or medical domain, prescriptive language is used by authors attempting to share a personal opinion or belief. Prescriptivists explain how you *ought* to do something, where descriptivists explain how to actually *do* it [HP02]. A third person descriptive writing style is much more formal than first or second person, and should therefore be used when writing technical documentation [Kra99]. To ensure developers and documentation writers use an acceptable writing style when documenting their software [Kra99], the JavadocMiner shall detect sentences within source code comments that use a second person prescriptive writing style.

Requirement#3.4: Detecting Passive Voice Writing Style. In-line documentation is meant to be clear, short and to the point; therefore, passive writing should generally be avoided in Javadoc comments. An active voice writing style is more "direct and vigorous" compared to a passive voice [JW00]: "Formats the passed string" is, for example, much more direct than "In charge of formatting the string passed to it". Both examples are equally understandable; however, the former achieves the same goal with less written content. Measures discussed this far are incapable of detecting the writing style of text. The JavadocMiner shall detect comments that use a passive writing style, and provide the users with recommendations on how the comments may be improved.

4.4 Code/Comment Consistency Analysis

Over time the quality of source code comments can degrade, due to continuous changes to the source code not being reflected in the comments. Existing Javadoc analysis tools such as DocCheck and Checkstyle focus mostly on code/comment type of analysis. We chose to include syntactic analysis in our JavadocMiner system to provide users with full coverage of Javadoc comment analysis as shown in Table 4, eliminating the need to use additional tools offering similar quality assessments.

```
/**
* Manages the event changes of elements within a UML model,
* and uses the {@link ActivityGraphsHelper} helper.
* @author Bob Tarling
* @param source The bean that fired the event.
* @param propertyName The programmatic name of the property
*                     that was changed.
* @param oldValue The old value of the property.
* @param newValue The new value of the property.
* @param originalEvent The event that was fired internally
*      in the Model subsystem that caused this.
*/
public AttributeChangeEvent(Object source, String propertyName,
        Object oldValue, Object newValue, EventObject originalEvent)
```

Figure 3: A Javadoc Comment for an ArgoUML Constructor

Requirement#4.1 Detecting Undocumented Identifiers. Without sufficient documentation, developers modifying unfamiliar source code run the risk of introducing a fault. The JavadocMiner shall measure the completeness of a class in terms of documented identifiers. In order for a Javadoc comment to be properly associated with an identifier and therefore included in the generated API documentation, it must appear directly before the class, field, method or constructor declaration. In the initial phase of our code/comment quality assessment, the JavadocMiner will identify source code that do not have Javadoc comments associated with them. In Figure 3, we show an example of a constructor taken from ArgoUML, documented using a Javadoc comment. The comment includes the `docComment` explaining the responsibility of the constructor. Also included are the author's name, and parameter comments describing what each parameter is being used for.

```
package org.argouml.model;

import java.util.EventObject;

/**
 * A change event due to change in an attribute of a model element
 * (eg the name of a model element has changed).
 *
 * @author Bob Tarling
 */
public class AttributeChangeEvent extends UmlChangeEvent {
    /**
     * Constructor.
     *
     * @param source The bean that fired the event.
     * @param propertyName The programmatic name of the property
     *            that was changed.
     * @param oldValue The old value of the property.
     * @param newValue The new value of the property.
     * @param originalEvent The event that was fired internally
     *      in the Model subsystem that caused this.
     */
    public AttributeChangeEvent(Object source, String propertyName,
            Object oldValue, Object newValue, EventObject originalEvent) {
        super(source, propertyName, oldValue, newValue, originalEvent);
    }

    /**
     * The UID.
     */
    private static final long serialVersionUID = 1573202490278617016L;
}
```

Figure 4: A Completely Documented ArgoUML Class

Requirement#4.2: Detecting Classes With Insufficient Documentation. Another important task of our syntactic analysis of Javadoc comments is detecting classes that are under-documented. The JavadocMiner must process each class separately in order to identify the ratio between documented identifiers vs. \undocumented identifiers within a class. For the ArgoUML class shown in Figure 4, the JavdocMiner will assign a value of 1, indicating that all three of the identifiers (class, constructor, and field) are documented using Javadoc comments.

Requirement#4.3: Detecting Undocumented Method Return Types. When documenting the return type of a method, the `@return` block tag must begin with the correct type being returned (e.g., String), followed by the doc comment discussing what is being returned. The JavadocMiner system must detect methods that have return types

```
/**
* A sequence diagram can accept all classifiers . It will add them as a new
*  Classifier Role with that classifier as a base. All other accepted figs
* are added as is .
* @param object The object to accept
* @return true if the diagram can accept the object , else false
* @see org.argouml.uml.diagram.ui.UMLDiagram#doesAccept(java.lang.Object)
*/
@Override
public boolean doesAccept(Object objectToAccept) throws ObjectAcceptException
```

Figure 5: An Incomplete ArgoUML Method Comment

that are not being documented. In Figure 5, we show an example of an ArgoUML method that returns the type "boolean", which is incorrectly documented using a Javadoc comment that begins with the value of "true", rather than the actual return type.

Requirement#4.4: Detecting Undocumented Method Parameters. When documenting the parameter list of a method, the `@param` block-tag should begin with the correct name of the parameter being documented, followed by the doc comment discussing the parameter. The JavadocMiner must detect parameters that do not have a parameter comment associated with them. Either because *(i)* documentation for the parameter was not included, or *(ii)* a change to the name of the parameter in the parameter list does not reflect that of the associated comment. In Figure 5, we show an example of an ArgoUML method that contains a parameter called "objectToAccept", which is out of sync with the Javadoc parameter comment explaining it.

Requirement#4.5: Detecting Undocumented Thrown Exceptions. When documenting the exceptions thrown by a method, the `@throws` or `@exception` block tags must begin with the correct type of the exception being thrown (e.g., IOExcpetion) followed by the doc comment explaining the exception itself. The JavadocMiner system must detect methods that throw exceptions that are not being documented. In Figure 5 we show an example of an ArgoUML method that throws an exception of the type "ObjectAcceptException" but does not have a Javadoc comment associated with it.

Requirement#4.6: Detecting Methods With Insufficient Documentation. Studies have shown that due to developers and maintainers constantly modifying a computer program, modifications to the source code often don't reflect their comments [FWGG09]. For in-line documentation describing a method to be considered complete, it should document all aspects of the method. Similar to the ratio between undocumented vs.

\documented identifiers within a class, the JavadocMiner will provide users with a ratio between the number of items within a method that should be documented using the method block tags discussed in Requirements #3.3–3.5, identifying the different parts of a method that are out of sync with the comment(s) discussing it. For the method comment example in Figure 5, the JavadocMiner will generate a value of 0.25, indicating a method that contains a method comment but is out of sync with the return type, parameter list, and exception it throws.

```
/**
* Gets popup menu
*
* @return Popup menu
*/
private JPopupMenu getPopupMenu()
```

Figure 6: A Method Comment with no Added Value

Requirement#4.7: Detecting Under-Documented Method Comments. Developers mostly interested in writing source code find creating and reading source code comments a highly unfavourable task [FWG07]. Therefore, to help software engineers realize the importance of source code comments, in-line documentation must add value to the readability of the source code implementation. The purpose of writing in-line documentation is to add value beyond what an individual is able to gather from looking at just the declaration names. In Figure 6, we show an example of an ArgoUML method called "getPopUpMenu", which is described using a comment that adds no value beyond the API name. As part of the syntactic analysis, the JavadocMiner will detect comments that provide insufficient documentation compared to what could be gathered using the API name.

4.5 Representing the JavadocMiner Results

Requirement#5.1: Ontology Representation. The final step of the JavadocMiner must export the results of the language service to a repository that enables users to view, analyze, and query the data. When attempting to provide users with information from an analysis, we need to consider how to model the information using a robust and scalable representation.

Requirement#5.2: Traceability Links. Most software projects undergo a brief development period, followed by a much longer maintenance period, where efforts are focused towards adopting new contexts and requirements [KCA06]. During this maintenance period, software developers spend a considerable amount of time searching and exploring the different software engineering artifacts in an attempt to understand the unfamiliar code. Software engineering knowledge is typically distributed across multiple artifacts and repositories. Automatically linking the different artifacts using common software engineering concepts eliminates the need for developers and maintainers to manually search for specific information from the different SE repositories. A process that interrupts developers from more important tasks such as writing code [KCA06]. Along with modelling the results of the analysis, the JavadocMiner will also establish traceability links between Javadoc comments and other software engineering artifacts, such as version control and issue tracking systems.

4.6 Tool Integration

Requirement#6.1: Embedding the Results. For the results generated by the analysis of our tool to be of use to end users, there needs to be a separation of concern where the language service provided by our JavadocMiner runs in the background of existing software engineering tools. Extending existing tools lessens the impact of having to learn new technologies, thus increasing the possibility of the JavadocMiner being accepted and used. Integrated Development Environments (IDEs), build servers and version control systems are just a few of the many tools used to perform software engineering tasks that can be enriched using results produced by the JavadocMiner analysis.

4.7 Summary

Existing Javadoc quality assessment tools provide an analysis that are purely syntactic in nature with minimal regard to analyzing the semantic quality of the documentation. In Table 4 we provide a comparison using the different categories of analysis provided by the JavdocMiner with the similar tools such as Quasoledo [SDZ07], Doc Check Doclet, and Checkstyle.

According to Oracle's website[1], future error checks using their Doc Check Doclet will include identifying comments that do not add any value e.g. "Returns the component name" for a method called "getComponentName"–this is a feature already included in our JavadocMiner system (Requirement #3.7).

[1]Errors Identified by the Sun Doc Check Utility, http://java.sun.com/j2se/javadoc/doccheck/docs/DocCheckErrors.html

Table 4: A Comparison between Different Javadoc Analysis Tools

Type	Feature	REQ.	JavadocMiner	Quasoledo	Doc Check Doclet	Checkstyle
Comment Syntax	Words Per Javadoc Comment	2.1	✓	✓		
	Refrain from using abbreviations in comments	2.2	✓			
	Number of Noun Phrases and Verb Groups	2.3	✓			
Internal NL	Calculate Comprehension and Retention	3.1	✓	✓		
	Calculate Efficiency and Perseverance	3.2	✓			
	Detect Prescriptive writing style	3.3	✓			
	Detect Passive voice writing style	3.4	✓			
Code/Comment	Missing Comments	4.1	✓	✓	✓	✓
	Any Javadoc Comment	4.2	✓	✓	✓	✓
	Method missing "@return" tag	4.3	✓	✓	✓	✓
	Method missing "@param" tag	4.4	✓	✓	✓	✓
	Method missing "@throws" tag	4.5	✓	✓	✓	✓
	Documentable Item Ratio	4.6	✓	✓		
	Comment must add value beyond API Name	4.7	✓			
External	Ontology Representation	5.1	✓			
	Traceability Links	5.2	✓			
Embedded	Tool Integration	6.1	✓			✓

Chapter 5

Design

In this chapter, we cover the design decisions taken when developing the JavadocMiner. We start by identifying the different system components, discuss the process of marking up source code to be used as input for our quality assessment, followed by a description of the metrics that were used to analyze the quality of in-line documentation. We then give an example of the type of results produced by our analysis and finally discuss the design of the ontology that is used to store the results.

5.1 System Components

The first component of our JavadocMiner system is in charge of generating a corpus from Javadoc information. For reasons discussed in the following section, we decided on using an XML representation for the input documents of our NLP application. The first component of our JavadocMiner system is in charge of generating XML documents using information found in Javadoc. Once the set of documents (corpus) has been generated, it can be used as input for our NLP application, as shown in Figure 7.

As mentioned in Chapter 2, NLP application pipelines are assembled using levels of linguistic analysis, with each level adding information to the document. An NLP application will typically *(1)* reuse resources that perform common linguistic tasks, and *(2)* design specialized resources specific to an application. The results of an NLP application can be exported in a number of different formats such as an XML file, database or ontology. For our JavadocMiner system, generated results are exported

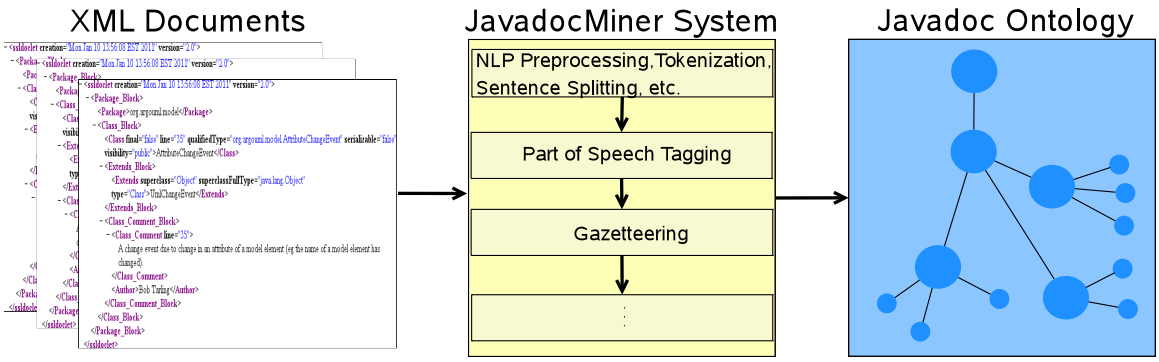

Figure 7: JavadocMiner Overview

to an ontology, since it allows for a semantically rich of a large amount of information using small number of axioms [WKR11]. Users are also able create inferences by applying reasoning services on the knowledgebase, and retrieve information from the knowledge base using a standard query language.

For now, we have identified the major components in our JavadocMiner architecture, giving us a general idea on how the system will be able to assess the quality of documentation found in source code. In the remaining parts of this chapter, we will provide a more detailed design description of each component.

5.2 Corpus Generation from Source Code: The SSL Javadoc Doclet

As mentioned in Chapter 3, the tool that provides the most information regarding the constructs found in Javadoc comments is the Javadoc tool.

```
<H2>
<FONT SIZE=1>org.argouml.model</FONT>
<BR>
Class  AttributeChangeEvent
</H2>
```

```
<Package_Block>
<Package>org.argouml.model</Package>
        <Class_Block>
          ......
        </Class_Block>
</Package_Block>
```

Figure 8: Documentation Generated Using Javadoc for an ArgoUML Package

The Javadoc tool is capable of extracting all of the Javadoc related information, and Java source code related information up to the method level (not including the method body itself), which is sufficient for the assessment of Javadoc comment quality.

Javadoc's standard doclet generates API documentation using the HTML format; while this is convenient for human consumption, automated NLP applications require a more structured XML format as we discuss in the rest of this section. For our purpose, we found that the XML format is much more versatile than HTML, and therefore ideal for our type of NLP analysis. XML enables users to: *(1)* define custom tags and attributes to mark up the information of the XML elements [Ray03] and *(2)* define a well-formed XML structure that is easily processed by NLP applications. With HTML, we are limited to pre-defined tags such as `<p>` or `<head>`, and predefined attributes such as `font-size`, as well as a standard HTML structure that encloses information using the pre-defined tags, such as `<body>` and `<table>` [GP03]. Such tags found in HTML are designed to be rendered by a browser for human consumption [Av08], but are less suitable for automatic processing in an NLP framework. For example, in Figure 8 we show the same Java package taken from the ArgoUML OSS project [BF09] represented using HTML (left) and XML (right). The HTML representation not only contains additional irrelevant information, but also uses HTML tags that do not reflect the semantics of the information they mark-up. This is not the case for the XML representation, which only includes relevant information and uses custom defined tags that reflect the source code and source code comment information such as, `Package_Block, Class_Block`.

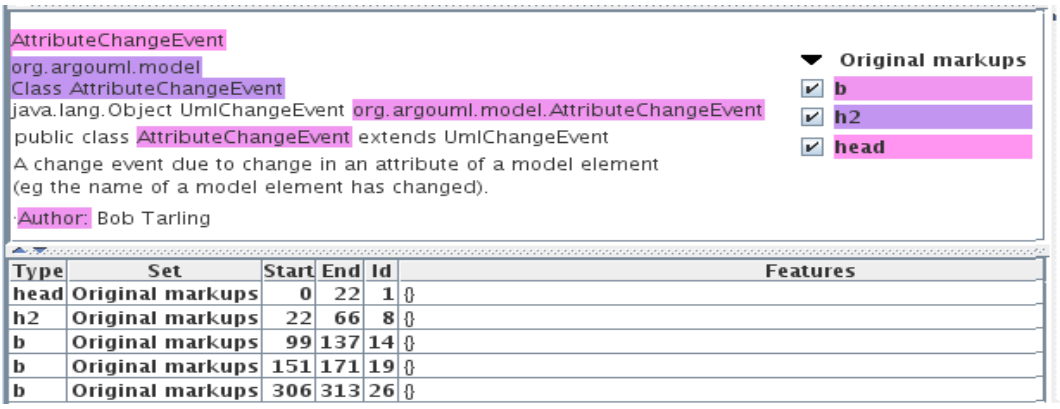

Figure 9: HTML Generated Documentation Loaded within an NLP Framework

When loading HTML documents generated using the standard doclet into an NLP framework (Figure 9), the elements of an HTML tag are interpreted as being entities

of an annotation. For example, the Java package "org.argouml.model" is interpreted as being of the type `h2`. This is because Javadoc's standard doclet extraction tool marked up the package using the `<h2></h2>` tags. As a result, additional processing is required in order to identify the entity as being a package.

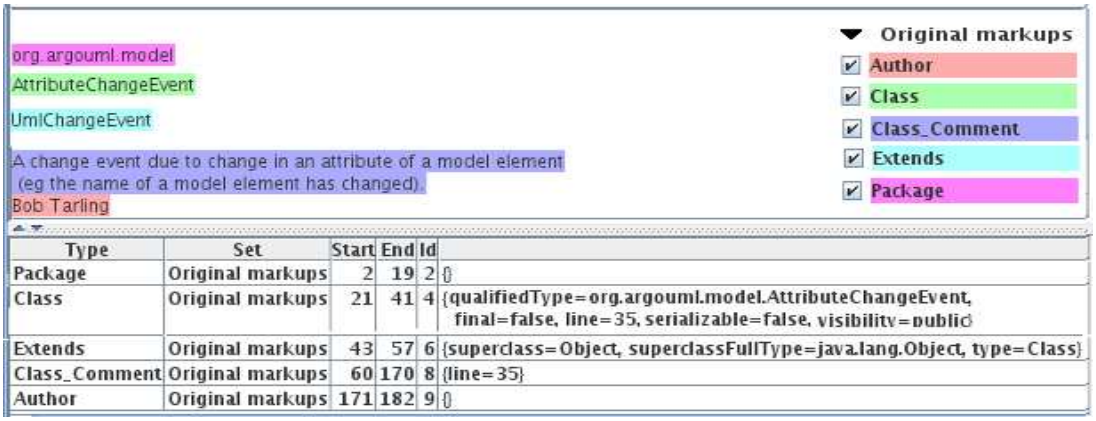

Figure 10: XML Generated Documentation Loaded within an NLP Framework

Compare this with an XML document (Figure 10), where the elements of the XML tags coincide with the encapsulated entity, clearly identifying them as being of type `Package`, `Class`, `Class_Comment`, and `Author`.

To satisfy Requirement #1.1, we designed our own doclet capable of generating XML documents. The SSL Javadoc Doclet [KWR10] converts Java source code and Javadoc comments into an XML representation, thereby creating a corpus that the JavadocMiner NLP service can efficiently analyze. The SSL Javadoc Doclet enables us to *(i)* control what information from the source code will be included in the corpus, and *(ii)* mark up the information using a schema that NLP applications can easily process.

The SSLDoclet uses a schema that maintains the relationships found in source code and represents the information using a combination of XML tags, attributes and elements. In Figure 11, we show how the relationships found in the sample source code are modelled (note: XML elements and attributes were omitted for readability purposes).

```
<Abstract_Class_Block>
        <Abstract_Class/>
                <Extends_Block>
                        <Extends/>
                        <Extends_Comment/>
                </Extends_Block>
                <Class_Comment_Block>
                        <Class_Comment/>
                        <Author/>
                </Class_Comment_Block>
                <Constructors>
                        <Constructor_Block>
                                <Constructor/>
                                        <Constructor_Comment_Block>
                                                <Constructor_Comment/>
                                        </Constructor_Comment_Block>
                                        <Parameter_Block>
                                                <Parameter/>
                                                <Parameter_Comment/>
                                        </Parameter_Block>
                        </Constructor_Block>
                </Constructors>
        ...
</Abstract_Class_Block>
```

Figure 11: SSLDoclet Schema

5.2.1 Marking Up Source Code

The SSL Javadoc doclet is able to model both the syntactic and semantic information found in Java source code, such as:

- Parent/Child relationships between generalized and specialized classes.
- The package an interface or abstract class belongs to.
- Fields, constructors and methods of a class.
- The types, modifiers (private, public, protected), and constant values of the fields.
- The return types, parameter list, and thrown exceptions of a method.

In Figure 12, we show an abstract class declaration taken from ArgoUML, and the same class is shown in Figure 13 after converting it using the SSL Doclet. The information found in the abstract class is represented using the `<Package>` and `<Extends>`

```
package org.argouml.notation.providers;

import java.beans.PropertyChangeListener;
import org.argouml.model.Model;
import org.argouml.notation.NotationProvider;

/**
 * This abstract class forms the basis of all Notation providers
 * for the text shown in the Fig that represents the CallState.
 * Subclass this for all languages.
 *
 * @author mvw@tigris.org
 */
public abstract class CallStateNotation extends NotationProvider
```

Figure 12: An Abstract Class Declaration taken from ArgoUML's Source Code

tags to model the package the abstract class belongs to, and the `super class` that it extends, respectively. The parameter list that belongs to the method "intialiseListener" are modelled using the `<Parameter>` XML tag as illustrated in Figure 14.

Both Figures 13 and 14 demonstrate how our doclet is able to represent more information, compared to the standard doclet, effectively using a well formed XML representation. For example, we now also know that the parent of the "CallStatNotation" is "Object", and that the listener parameter of the "intialiseListener" method has the type "PropertyChangeListener".

5.2.2 Marking Up Source Code Comments

Our SSL Javadoc Doclet is also designed to mark up the natural language information found in a Javadoc comment, such as the docComment, block, and in-line tags.

Along with the source code information discussed earlier, Figure 12 also shows an example of a Javadoc comment that includes a `docComment` and uses the "@author" block tag. In Figure 13, we show how Javadoc comments are marked up using the `<Extends_Comment>` tag, which contains the comment belonging to a super class. Additionally, the figure shows how the class comment, belonging to "CallStatNotation", is represented using the `Class_Comment`, and `Author` XML tags.

To conclude, even though there exists a number of XML generating doclets that can be downloaded from the net, however we feel that our SSLDoclet differs from the rest due to its ability to generate XML output using a schema that is optimized

```
<Abstract_Class_Block>
<Abstract_Class>CallStateNotation</Abstract_Class>
<Package>org.argouml.notation. providers </Package>
<Extends_Block>
        <Extends superclass=Object
         qualifiedType =org.argouml.notation . NotationProvider
                  superclassFullType =java.lang . Object
                           type=AbstractClass>
         NotationProvider
         </Extends>
         <Extends_Comment>
                  A class that implements this abstract class manages a
                  text shown on a diagram. This means it is able to
                  generate text that represents one or more UML objects.
                  And when the user has edited this text , the model may be
                  adapted by parsing the text .
                   Additionally , a help text for the parsing is provided ,
                  so that the user knows the syntax .
         </Extends_Comment>
</Extends_Block>
<Class_Comment_Block>
         <Class_Comment>
                  This abstract class forms the basis of all Notation
                   providers for the text shown in the Fig that represents
                  the CallState .
                  Subclass this for all languages .
         </Class_Comment>
         <Author>mvw@tigris.org</Author>
</Class_Comment_Block>
```

Figure 13: A Section of a Corpus Generated Using ArgoUML Source Code

for further NLP processing, which is an application scenario not targeted by existing efforts.

5.3 Preprocessing Phase

Before being able to apply NLP services on Javadoc comments, we apply a preprocessing stage, in-charge of filtering some of the content found in the documentation. Included in the preprocessing phase are things such as splitting up the string "getToolTip", for a comment describing a method using the Java naming convention, or "PersonDAO" for a comment describing the class 'Person Data Access Object". Though not important for determining the writing style of a comment, such strings would alter the results of the Fog and Kincaid readability measures. Whenever the

```
<Method_Block>
        <Method modifier=public  visibility =public
                signature=(java.beans.PropertyChangeListener,java.lang.Object)>
                 initialiseListener </Method>
        <Method_Comment_Block>
        <Method_Comment>
                 Initialise the appropriate model change listeners for the given
                modelelement to the given listener . Overrule this when you need
                more than listening to all events from the base modelelement.
        </Method_Comment>
        </Method_Comment_Block>
        <Parameter_Block>
        <Parameter fulltype=java.beans.PropertyChangeListener
                type=PropertyChangeListener>
                 listener </Parameter>
        <Parameter_Comment>the given listener</Parameter_Comment>
        </Parameter_Block>
        <Parameter_Block>
                <Parameter fulltype=java.lang.Object type=Object>
                modelElement</Parameter>
        <Parameter_Comment>
                the modelelement that we provide notation for
        </Parameter_Comment>
        </Parameter_Block>
</Method_Block>
```

Figure 14: A Section of a Corpus Generated Using an ArgoUML Method

string "getToolTip" appears in the comment, we use regular expression to split the string into "get Tool Tip". Additional content, such as HTML tags, hyperlinks and in-line tags, are also filtered out before applying the NLP analysis.

5.4 JavadocMiner Quality Assessments

The goal of our JavadocMiner tool is to enable users to *(1)* automatically assess the quality of source code comments, *(2)* provide users with recommendations on how the Javadoc comment may be improved, and *(3)* export the in-line documentation and results of the quality assessment to an ontology to support further analysis such as querying, reasoning services, and linking the data with other software engineering artifacts.

For our JavadocMiner system, we focus mostly on readability measures that are based on statistical models, and avoid using measures which use stop words to calculate the complexity of text. Measures such as the Dale-Chall Readability Formula [EC49] developed in 1948 uses a list of 3000 "easy words" to calculate the

readability of text. As was observed by [HFB+08], such dictionaries are often not maintained frequently, and more importantly they do not consider commonly used terminology used in computer science.

5.4.1 Comment Syntax

The following metrics measuring simple quality factors, provide the initial means of assessing the internal quality of in-line documentation.

Words Per Javadoc Comment Metric (WPJC)

Originally proposed by [SDZ07], the WPJC metric calculates the average number of words found in Javadoc comments (Requirement #2.1). After applying NLP prepossessing services, such as segmenting the generated documents into sentences, and sentences into tokens, the WPJC measure is in charge of counting the number of words found in each Javadoc comment, and dividing it by the number of Javadoc comments.

$$\text{WPJC} = \frac{\text{\# of Words in a Javadoc Comment}}{\text{\# of Javaodc Comments}}$$

Abbreviation Count Metric (ABB)

Using a list of abbreviations from a plain text file, the ABB metric is designed to detect and count the number of abbreviations used within Javadoc comments (Requirement #2.2). Abbreviated strings within the in-line documentation that match the entities from the list are annotated using features specifying that the string is an abbreviation.

Token, Noun and Verb Phrase Count Metrics (TNVC)

For each Javadoc comment within a Java class, we calculate the number of tokens, noun phrases and verb groups the comment contains using the TNVC metric (Requirement #2.3). The JavadocMiner must first apply noun and verb phrase chunking [CMBT02] services on in-line documentation to assist in additional quality assessments, for example, determining the writing style of a Javadoc comment.

5.4.2 Internal NL Comment Analysis

The following metrics also focus on the natural language quality of the in-line documentation itself. The metrics aim at assessing the semantic quality of the documentation, which greatly impacts a reader's ability to understand the technical documentation.

Calculating Comprehension and Retention

Using the "classic readability" [DuB06] measures, we assess the quality of Javadoc comments based on their ease of understanding and recall (Requirement #3.1). The core of the Fog Index is sentence length and word length. The weights assigned to the two parameters are designed to measure the ease of comprehension and retention of text:

$$\text{Fog} = 0.4(\text{ASL} + \text{HW})$$

Where:
ASL = Average sentence length using number of words.
HW = Number of words with more than two syllables

Calculating Efficiency and Perseverance

To satisfy Requirement #3.2, we use the Flesch-Kincaid Readability Formula to assess the quality of Javadoc comments based on the efficiency and perseverance readability factors. Regardless of the similarity with the original Flesch readability measure (i.e., the use of sentence length, and word length), the two formulas are weighted differently. The Flesch-Kincaid index actually correlates inversely to the traditional measure. The same block of text would be analyzed as having a higher Flesch Reading Ease Score compared to the Flesch-Kincaid Readability Formula. By lowering the weights, the Flesch-Kincaid formula is designed to focus more on the efficiency rather than comprehension of text.

$$\text{Flesch}-\text{Kincaid} = (0.39 \times \text{ASL}) + (11.8 \times \text{ASW}) - 15.19$$

Where:
ASL = Average sentence length using number of words.

ASW = Average number of syllables per word.

The JavadocMiner not only provides default threshold values based on Chapter 2, but also allows users to define custom threshold levels for the different readability indices. For sentences that return a readability index above or (as in the case of the Fog metric) below the threshold, the system generates a warning identifying these sentence(s).

Second Person Writing Style Metric (SPW)

By analyzing the part-of-speech of sentences found in Javadoc comments, the SPW metric is in charge of identifying comments that use a prescriptive writing style (Requirement #3.3). For each sentence within the Javadoc comment, we iterate through each token, identifying sequences where a present participle verb is followed by a determiner and finally a proper noun (e.g., "gets the label"). Once such sequences are found, we compare the stem of the verb (e.g., "get") with the original string. Sequences where the verb groups of the n-gram match the stem are identified by the JavadocMiner as using a 2nd person prescriptive writing style. For comments that are detected as using a prescriptive writing style, the JavadocMiner extracts the n-gram, converts it to the correct writing style using the stem of the words, and presents users with recommendations on how the comment can be improved

Passive Writing Style Metric (PWS)

Readability measures, such as FOG and Kincaid, are unable to determine the writing style for a given block of text (e.g., active vs. \passive). Using a rule-based verb chunker, the JavadocMiner can provide information regarding the tense, type, and voice of the verb group. The PWS metric detects sentences that use a passive voice writing style (Requirement #3.4).

5.4.3 Code/Comment Consistency Analysis

The metrics introduced in this section are designed to analyze in-line documentation and their consistency in documenting the actual source code. Because the Javadoc extraction tool is capable of detecting identifiers that contain no documentation, the

SSLDoclet is also incharge of detecting identifiers that are not documented using a Javadoc comment (Requirement #4.1).

Any Javadoc Comment Metric (ANYJ)

To compute the ratio of identifiers with Javadoc comments compared to the total number of identifiers (Requirement #4.2), we use the ANYJ metric [SDZ07]. ANYJ can be used to determine which classes provide the least amount of documentation, and could therefore be most prone to a newly introduced fault in the source code which would lead to a failure in the program (i.e., bug).

$$\text{ANYJ} = \frac{\text{Declarations With Any Javadoc Comment}}{\text{Total Number of Declarations}}$$

SYNC Metrics (PSYNC/RSYNC/ESYNC)

The following metrics detect methods that are documenting parameters, return types, and thrown exceptions that are no longer valid (e.g., due to changes in the code):

RSYNC: To identify methods with return types that contain outdated documentation (Requirement #4.3), we perform string comparison between the value of the return string indicated in the comment and the actual return type of the method.

PSYNC: To identify the method parameters that contain outdated documentation (Requirement #4.4), we perform a string comparison between the value of the parameter name indicated in the comment and the parameter name as it appears in the parameter list.

ESYNC: Identifying methods that throw exceptions that have no or out of date documentation (Requirement #4.5) is made possible by using a string comparison between the value of the exception string indicated in the comment and the actual exception type thrown by the method.

The "parseAssociationEnd method" in Figure 15 is an example of a method that contains a return type, parameter list, and exception that is consistent with the in-line documentation used in the `@return`, `@param`, and `@throws` block tags.

Documentable Item Ratio Metric (DIR)

To satisfy Requirement #4.6, we designed the DIR metric. Originally proposed by [SDZ07], the DIR metric takes into account the use of Javadoc block tags to document a constructor or method.

```
/** The following method parses the associations of a class diagram.
* @return String              A String association is returned
* @param role                 The AssociationEnd <em>text</em> describes.
* @param text                 A String on the above format.
* @throws ParseException      When is detects an error in the role string.
*                             See also ParseError.getErrorOffset().
*/
protected String parseAssociationEnd(Object role, String text) throws ParseException
```

Figure 15: An Example of a Javadoc Method Comment

In Figure 15, we show an example of a Javadoc comment for the "parseAssociationEnd" method that is completely documented using the block tags. The results of the DIR metric is the ratio between the parts of a method that should be documented versus the parts that actually were.

$$\text{DIR} = \frac{\text{Documented Items}}{\text{Documentable Items}}$$

Added Readability Value Metric (ARV)

The ARV metric detects documentation that adds no value beyond what can be understood using the API name (Requirement #4.7) by conducting an n-gram comparison between the identifier (e.g., "getsTheLabel"), and the docComment (e.g., "Gets the label"). After *(1)* splitting the identifier name using regular expressions designed to process the Java naming convention [DD07] for a `Class`, `Method`, etc., and *(2)* taking the stem of each word found in both the identifier and the comment, we perform a string comparison between the two. If the strings are an exact match then the JavadocMiner informs the user on the bad quality of the comment. Future plans for this requirement include analysis for comments that are not an exact match however, still add little value (e.g., "In charge of getting the label.").

5.5 The Javadoc Output Ontology

Results from the previous automated linguistic analysis on in-line documentation are reported by the JavadocMiner in the form of annotations and feature lists as shown in Figure 16.

Type	Set	Start	End	Id	
Class_Block		31	791	84	{AVGFLESCH=70.88, AVGFOG=5.19, AVGKINCAID=5.73}
Class		32	45	85	{ANYJ=66.67, AVGABBCOUNT=0, AVGFLESCH=70.88, AVGFOG=5.19, AVGKINCAID=5.7
Class_Comment		56	135	89	{ABBCOUNT=0, NumberOfNouns=4, NumberOfTokens=15, NumberOfVerbs=3, classN
Author		136	141	90	{className=Author, corefChain=[null], instanceName=Ninus, kind=Class, representat
Method_Comment		206	276	105	{ABBCOUNT=0, NumberOfNouns=4, NumberOfTokens=11, NumberOfVerbs=2, classN
ReadabilityAnalysis		206	276	916	{FleschExplanation=Sentence is 49.9pts. below threshold., FleschMetric=19.100006, K
Method_Comment		299	353	111	{ABBCOUNT=0, NumberOfNouns=5, NumberOfTokens=11, NumberOfVerbs=1, classN
Method_Comment		387	442	118	{ABBCOUNT=0, NumberOfNouns=5, NumberOfTokens=11, NumberOfVerbs=1, classN
Method_Comment		473	525	126	{ABBCOUNT=0, NumberOfNouns=4, NumberOfTokens=10, NumberOfVerbs=1, classN

Figure 16: Annotations Created by the JavadocMiner

Although annotations and feature lists are readable to a domain expert (i.e., language engineer), it would be difficult for a more general audience to use the information. For the result to be useful for software engineers, the generated output needs to be exported to a format such as XML or a database, allowing the information to be further queried and analyzed.

In order for the quality assessment provided by the JavadocMiner to be of use, the data needs to be stored using a persistent storage. In the following section we specify the response structure by means of the Web Ontology Language (OWL) using OWL constructs (Requirement #5.1). Even though other formats are possible, representing information in OWL is beneficial due to it's ability to provide a common language that enables the interoperability of different knowledge bases represented using OWL. Another advantage to using OWL is the non-proprietary standard that is not available when using database technologies.

A large number of source code and in-line documentation related concepts and relationships exist within Javadoc generated documentation. In Figure 17, we show some of the relationships that exist between the different source code and in-line documentation concepts, and in Figure 18 are some of the relationships that exist within the source code itself. The domain-specific Javadoc ontology is complemented by a domain-independent NLP ontology, which models commonly used concepts in NL

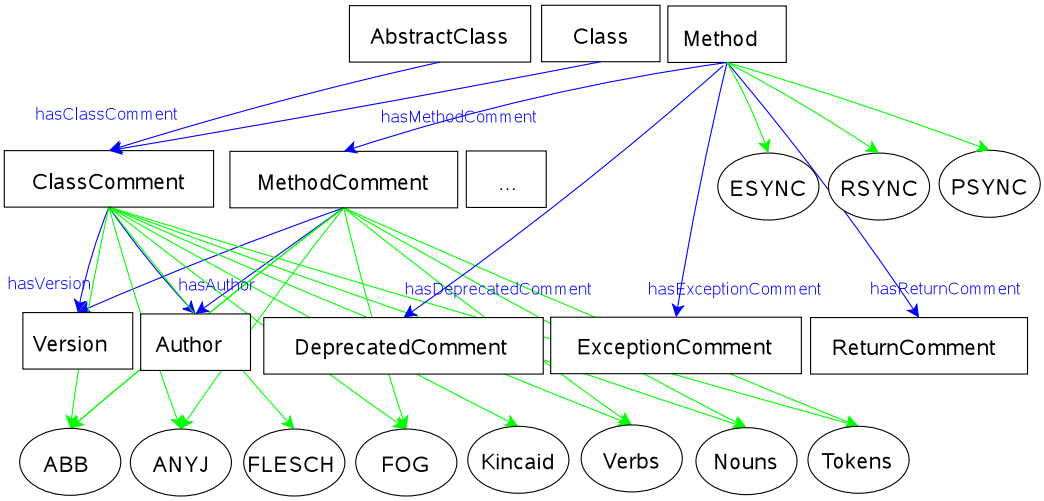

Figure 17: Ontology Showing Relationships found in JavadocComments

such as `Document, Sentence, NP,` and `VP`, as shown in the taxonomic representation (T-Box) in Figure 18. Some of the NLP ontology relationships are defined in Table 6.

An existing Javadoc ontology is available for download from the Semantic Web Search engine SWOOGLE[1] however, the ontology did not reflect the current version of Javadoc, and was therefore missing some of the current in-line and block tags introduced in the later versions of Javadoc.

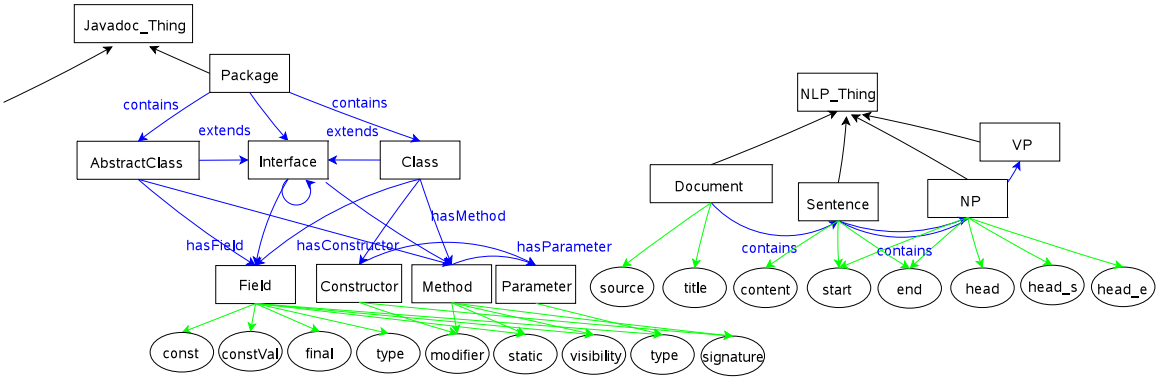

Figure 18: Ontology Showing Relationships found in Source Code

The Javadoc ontology models both source code related concepts such as `Class, Field, Method` and `Exception`, and in-line documentation related concepts such as `Method Comment, Author` and `Version`. The ontology represents the relationships

[1]SWOOGLE, http://swoogle.umbc.edu/

Table 5: Relationships and Concepts found in the Javadoc Ontology

Object Property	Domain	Range	Description
belongsToPackage	Interface ∪ Abstract Class ∪ Class	Package	The Package a given Interface or Class belongs to
hasConstructor	Interface ∪ Abstract Class ∪ Class	Constructor	Constructors contained within an Interface or Class
hasConstructorComment	Constructor	Comment	The comment that belongs to a Constructor
hasAuthor	Comment	Author	The author of a specific comment
hasVersion	Comment	Version	The version of the comment

between the concepts using a number of object properties, some of which are shown in Table 5.

Table 6: Relationships and Concepts found in the NLP Ontology

Object Property	Domain	Range	Description
hasSentence	Document	Sentence	The sentences found in a Javadoc document
hasNP	Sentence	NP	Noun phrases found in a sentence
hasVP	Sentence	VP	Verb phrases found in a sentence

Finally, relationships can be created between the two ontologies that allow for the linking of instances across ontology boundaries. For example, an `appearsIn` relationship can be used to link the segments or sentences of a document with the comment they appear in. Not included in Table 5 or 6 are the inverse properties, such as `isAuthorOf` or `isSentenceOf`, which a reasoner uses to create additional inferences, and that also allow for additional ways for querying the ontology. Modelling the Javadoc domain using ontologies allows us to query, reason, and create cross links with other software engineering artifacts represented using OWL models, and thereby contribute to a rich knowledge base incorporating other software artifacts [RWSC08].

5.5.1 External Traceability Links Generation

When attempting to find the different revisions and issues that belong to a given `Class`, developers and maintainers must manually query the different repositories (e.g., version control, and issue trackers) in search of the information. Having the different software engineering artifacts represented using OWL models allows us to establish external traceability links between the different repositories (Requirement #5.2).

Using multiple ontologies to represent the information within a domain such as software engineering or bioinformatics, requires aligning the ontologies as a "prerequisite for interoperability, and unhampered semantic navigation and search" [Bei10]. Various methods have been proposed to to perform ontology alignment (i.e., string-based, and structural) [Bei10]. Because our knowledge base uses common concepts in software engineering such as `File`, `Class`, and `Issue`, and the naming conventions are similar between the different ontologies, we use a string based comparison to seek out the common concepts. Cross boundary relations are then created linking the individuals together. The linked ontologies (Figure 19) allows developers and maintainers to query the entire knowledge base using a single access end-point.

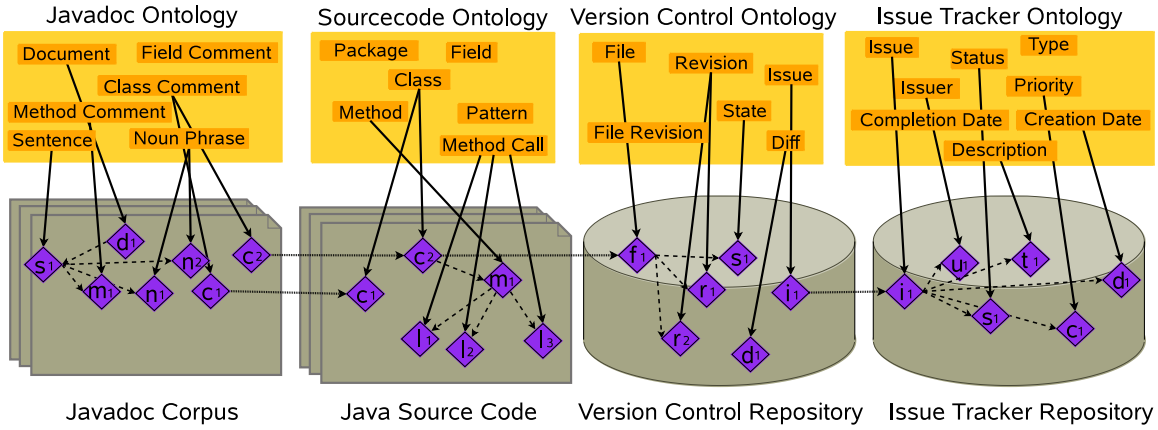

Figure 19: Traceability Links Create Between Different Software Engineering Artifacts

For example, an individual attempting to perform a software maintenance task on a class *c2* can query the knowledge base for information regarding *(1)* the documentation that is available for the class (javadoc ontology), *(2)* the pattern(s) the class may be a part of (source code ontology), *(3)* the different changes class *c2* has undergone from the time of its initial commit (version control ontology), and finally *(4)* the issues that have been reported for the class in the past (issue tracker ontology). This is just one of the many contexts a user can query the knowledge base for.

5.6 Summary

In this chapter, we have presented the design decisions for the main components that make up the JavadocMiner system. For the corpus generation task, we produce a corpus using source code and source code comments. The generated corpus is then used as input for the JavadocMiner, which analyzes the quality of the in-line documentation using a set of metrics. Finally, the results generated by the JavadocMiner are exported to OWL models for inferencing and querying services, as well as establishing traceability links with other software engineering artifacts using common concepts.

Chapter 6

Implementation

In this chapter we describe the implementation of the JavadocMiner application, which is created using the General Architecture for Text Engineering (GATE) [CMBT02] framework. The JavadocMiner uses some of the standard components included within GATE, as well as components developed by us. We first introduce the implementation of our Javadoc SSL doclet, and then discuss briefly the GATE framework, followed by a description of the JavadocMiner. The chapter will conclude with details on the exporting of entities and relationships into an OWL model using our OwlExporter, and linking the data with other software engineering artifacts.

6.1 System Overview

The major components that make up our JavadocMiner system are: *(i)* the SSL Javadoc Doclet used to generate a corpus from source code and source code comments; *(ii)* an NLP application that analyzes the documents in the corpus; and *(iii)* OWL ontologies that are used to store the results of the NLP analysis.

To summarize, our JavadocMiner tool measures the completeness and readability of in-line documentation based on the design decisions discussed in Chapter 5. The JavadocMiner is also capable of providing users with recommendations on how to improve a Javadoc comment.

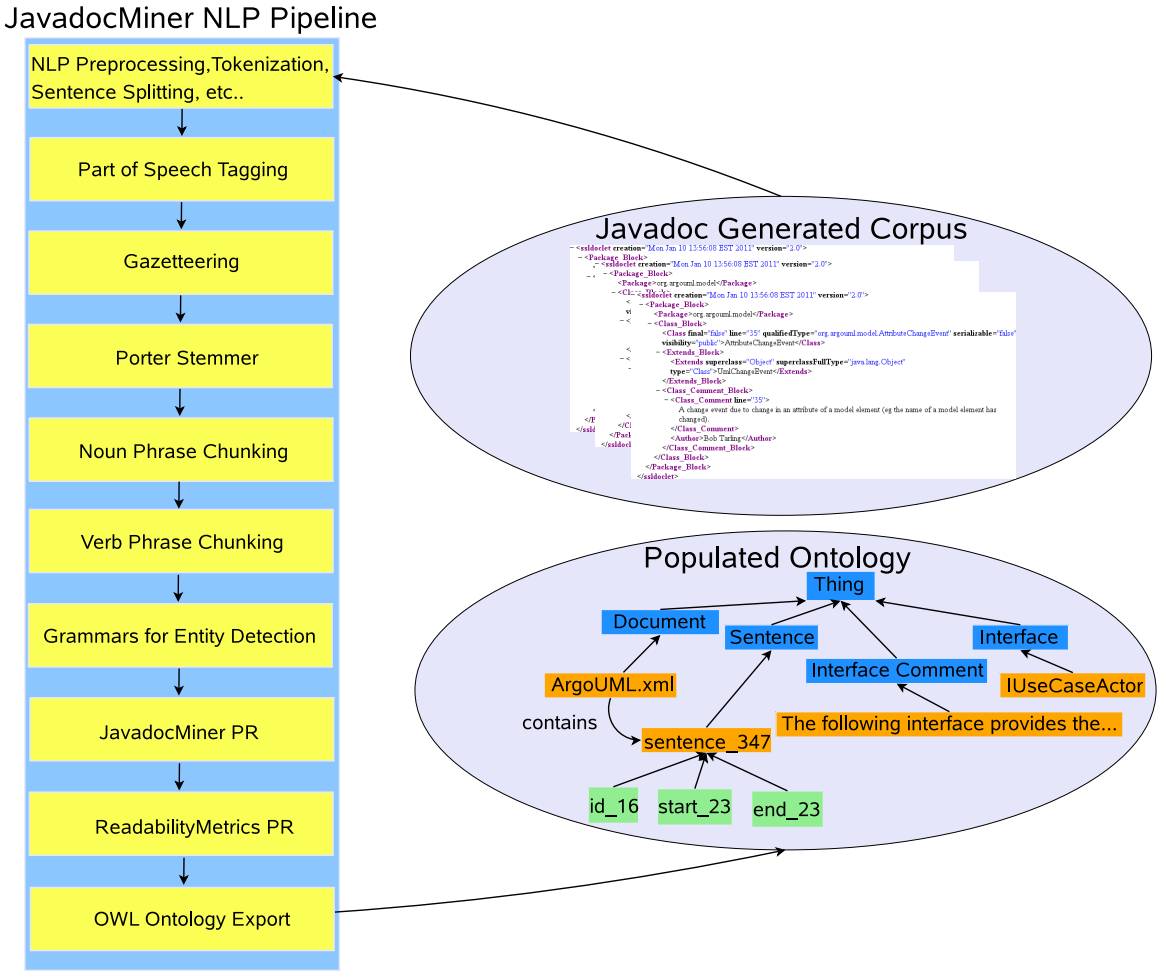

Figure 20: Overview of the JavadocMiner System Components

6.2 The SSL Javadoc Doclet

The Javadoc library is in charge of extracting information from the source directory and providing an interface to a set of objects that are created as a result of the static source code analysis. Transformation of the static source code analysis into the desired output (i.e., XML) is then made possible by developing a custom doclet that uses the Javadoc API library.

Our SSL Javadoc Doclet is an extension to the Javadoc tool [Kra99], which is implemented as a Javadoc plugin. The doclet is compiled into a Java Archive (JAR), and can be passed as a command line, MAKE or ANT parameter to the Javadoc tool. In Figure 21 we show an example of a Javadoc ANT task that indicates *(i)* the path and the name of the doclet, *(ii)* the path to the source directory, *(iii)* the name of

```
<target name="docs" depends="jar">
        <javadoc docletpath = "${doclet.dir}/
                                        ${ant.project.name}.jar"
        doclet          = "${doclet}"
        sourcepath        = "${src.dir}"
        packagenames    = "info.semanticsoftware.doclet"
        additionalparam  = "-J-Xmx256m"
        />
</target>
```

Figure 21: Javadoc Ant Task that accepts the SSLDoclet as a Parameter

the package in the source directory that needs to be processed, and finally *(iv)* any other additional parameters, for example, to increase the default Java heap space.

6.3 GATE Environment

GATE provides a framework for creating language processing software. The framework is implemented using the Java programming language, and can therefore run on any platform that includes the Java Virtual Machine [CMB+10]. As part of the GATE framework, a development environment is also distributed. The development environment is built on top of the GATE framework and includes a graphical interface for developing and editing language analysis components, as well as tools for visualizing and evaluating the generated results. A detailed discussion of the GATE framework and its different component can be found in [CMB+10]. The GATE framework includes a set of default components known as *resources* that provide common language analyzing tasks discussed in Chapter 2. Each component can be divided into three main categories:

Visual Resources: represent visualization and editing components that participate in GUIs.

Language Resources: represent entities such as lexicons, corpora or ontologies.

Processing Resources: represent entities that are primarily algorithmic, such as parsers, generators or ngram modellers.

The set of resources integrated in GATE are called CREOLE (a Collection of REusable Objects for Language Engineering). All the resources within CREOLE are packed as Java Archives (JARs) and imported into the GATE framework as plug-ins.

Language Resources. The language resources in GATE include a *Corpus*, *Document*, and *Annotation*. They provide the input and output of an NLP service [CMB+10]. The corpus consists of a set of documents, which is made up of text that is processed using a set of annotations. An annotation is assigned a type that may contain a set of *features*. Features are a set of names and values created for an annotation by the different language analyzing components of a service.

Processing Resources. There exist a number of standard PRs in GATE, which perform common language analyzing tasks, such as parts-of-speech tagging. Such resources are seen as being application independent and could be used to process text from, for example, bioinformatics or software engineering.

To summarize, an application running in the GATE environment typically consists of:

- A set of Visual resources for extended text processing being run in the GATE graphical user interface.
- A corpus, i.e., a set of documents, being processed.
- A set of Processing resources, standard or implemented as CREOLE by a user.

The PRs are executed in a sequence defined by the ordering of the components in a GATE pipeline. The output generated by preceding resources may be used as input for seceding resources. Furthermore, the results provided by the analysis of each component enriched the information of the document using *annotations*.

6.4 The JavadocMiner NLP Application

The core JavadocMiner NLP application is implemented as a GATE pipeline. In this section, we give a description of the Processing Resources used within our JavadocMiner system. The JavadocMiner pipeline re-uses components from A Nearly-New Information Extraction system (ANNIE) [CMBT02] shipped with GATE and components developed by us to be used specifically for the JavadocMiner. The components used in order of their execution, are (Figure 20):

Document Reset PR: If a pipeline is used to process the same document twice, this component ensures that the previous results are removed.

Annotation Set Transfer: This PR is in charge of transferring annotation types from one annotation set (e.g. Default markup), to an annotation set that is used for processing (e.g. To be processed).

ANNIE English Tokenizer: The tokenizer identifies words, space-tokens, numbers, and punctuation as well as other symbols (Appendix C.2).

Javadoc Sentence Splitter: This component prepares the different parts of a Javadoc comment for the default sentence splitter provided by ANNIE.

ANNIE Sentence Splitter: This component divides the text into sentences trying to identify the start of a new sentence by considering abbreviations and tokens (Appendix C.3).

ANNIE POS Tagger: Part-of-speech tagging is performed by the Hepple tagger [Hep00] included in the GATE distribution (Appendix C.4).

ANNIE Gazetteer: This component provides a list lookup to identify entity names in the text. The component is used for tagging tokens with their semantic categories (Appendix C.5). We also use the gazetteer list to detect abbreviations included in a Javadoc Comment.

Stemmer: The stemmer is in charge of producing the reduced form or stem of an inflected word. GATE uses the Porter stemmer for English (Appendix C.6).

Multilingual Noun Phrase Extractor: The Noun Phrase Extractor (MuNPEx)[1] is a base NP chunker, i.e., it does not deal with any kind of conjunctions, appositions, or PP-attachments. It is implemented as a JAPE transducer and can make use of previously detected named entities (NEs) to improve chunking performance (Appendix C.7).

ANNIE VP Chunker: This transducer, implemented in JAPE, identifies verb groups and annotates them with tense, voice, type, etc. information (Appendix C.8).

JavadocMiner PR: We implemented a GATE processing resource component called the JavadocMiner PR that contains the set of metrics specific to Javadoc comment analysis described in Chapter 5. The JavadocMiner PR is implemented

[1]Multi-Lingual NP Chunker (MuNPEx), http://www.semanticsoftware.info/munpex

using the Java programming language [DD07], and takes advantage of the different design patterns such as the *builder pattern*, enabling the GATE plug-in to be easily extended to include additional Javadoc quality analysis.

ReadabilityMetrics PR: For analyzing the readability of a given document, we implemented an application independent component that contains the readability metrics described in the design chapter. The component can be used in any NLP service where the quality of a document needs to be measured. The ReadabilityMetrics PR makes use of an existing library[2] to calculate the readability of text using readability measures, and our own implementation in charge of analyzing the writing style of a block of text. The readability PR is also implemented using the Javadoc programming language.

Using the results generated by these components, the JavadocMiner is also able to give recommendations on how the comment can be improved based on this analysis, and identifies the verb group that was detected as using a passive voice as shown in Figure 22.

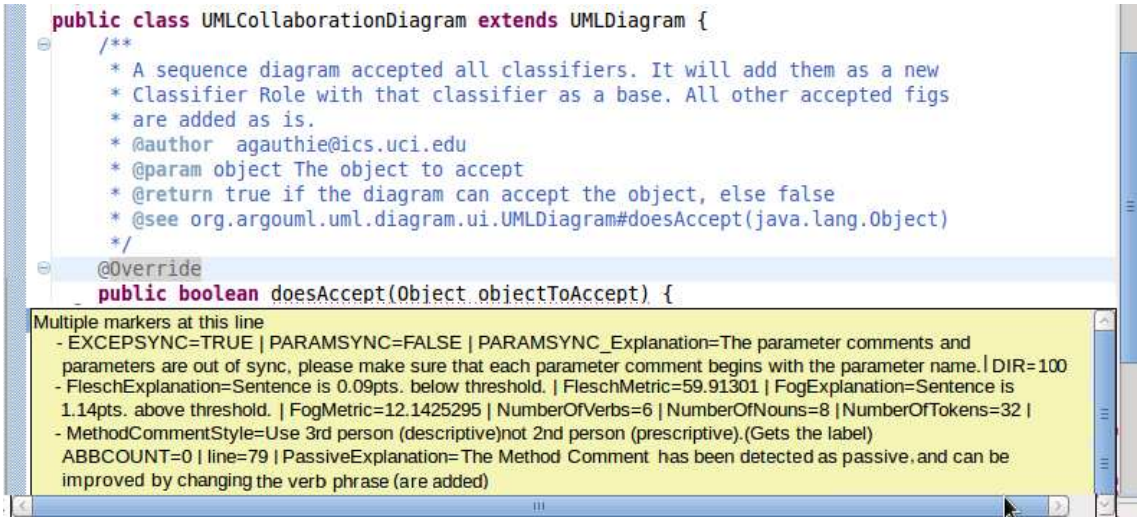

Figure 22: A Method taken from the ArgoUML OSS assessed using the JavadocMiner

In Figure 28 we show an example of the JavadocMiner NLP pipeline processing the API documentation taken from ArgoUML project.

[2]Readability Metrics Java Implementation, http://www.representqueens.com/fathom/

6.5 Javadoc Ontology

As discussed in Chapter 5, we use ontologies for the modelling of the various entities and relationships [FBMNPS07] extracted from Javadoc. In this section, we discuss how the results provided by the quality assessment created by our JavadocMiner system is exported to an ontology using an application independent GATE component developed by us. We also illustrate, how common concepts found in the Javadoc ontology are linked with concepts found in different software engineering repositories like, version control, or issue trackers.

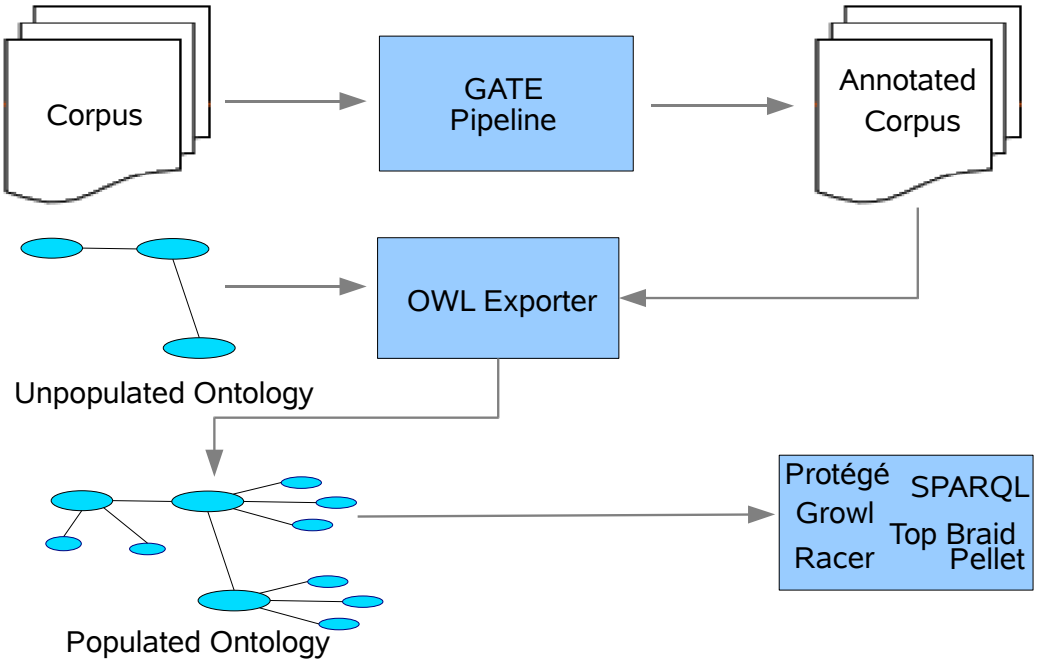

Figure 23: Ontology Population from Text

6.5.1 Ontology Population Using the OwlExporter

The Javadoc and NLP ontologies created thus far contain only concepts and relationships. Before it can be of help to a software engineer, it needs to be instantiated in order to obtain a knowledge base. Due to the large number of facts being extracted from these repositories, an automated approach for "ontology population" [Cim06] is required based on the results of an NLP analysis step (Figure 23).

In general, designing an ontology's taxonomy (T-Box), and populating it using

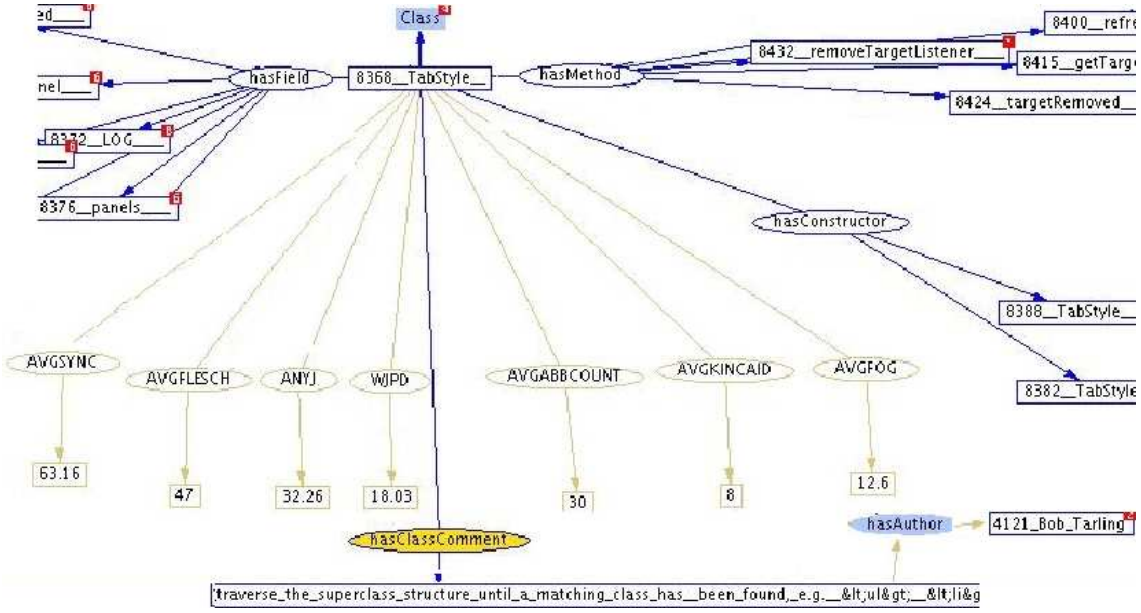

Figure 24: An Excerpt from the Javadoc Ontology

concept and relationship assertions (A-Box) [FBMNPS07] is a complicated and time consuming task that requires the expertise of an ontology engineer. The Web Ontology Language Exporter (OwlExporter)[3] [WKR11] is a component implemented by us that provides an automated, portable, and simplified means of populating an ontology using the results provided by an NLP service. Using our OwlExporter component, we export the entities and relationships that are created by our JavadocMiner system as OWL instances and relationships in the Javadoc and NLP ontologies. For example, the relationships where a class that implements a certain interface which contains a certain comment that is written by a specific author are all exported to the Javadoc and NLP ontologies. In Figure 24, we show an excerpt of the populated Javadoc ontology.

Exporting the results created by our JavadocMiner application to an ontology enables users to create `SPARQL` [PS08] queries to extract the asserted and inferred knowledge of the model. For example, conformance testers can quickly identify the modules within an application that do not follow organizational standards and thus return poor quality figures. In Figure 25, we show the results of a `SPARQL` query

[3]Web Ontology Language Exporter (OWLExporter), `http://www.semanticsoftware.info/owlexporter`

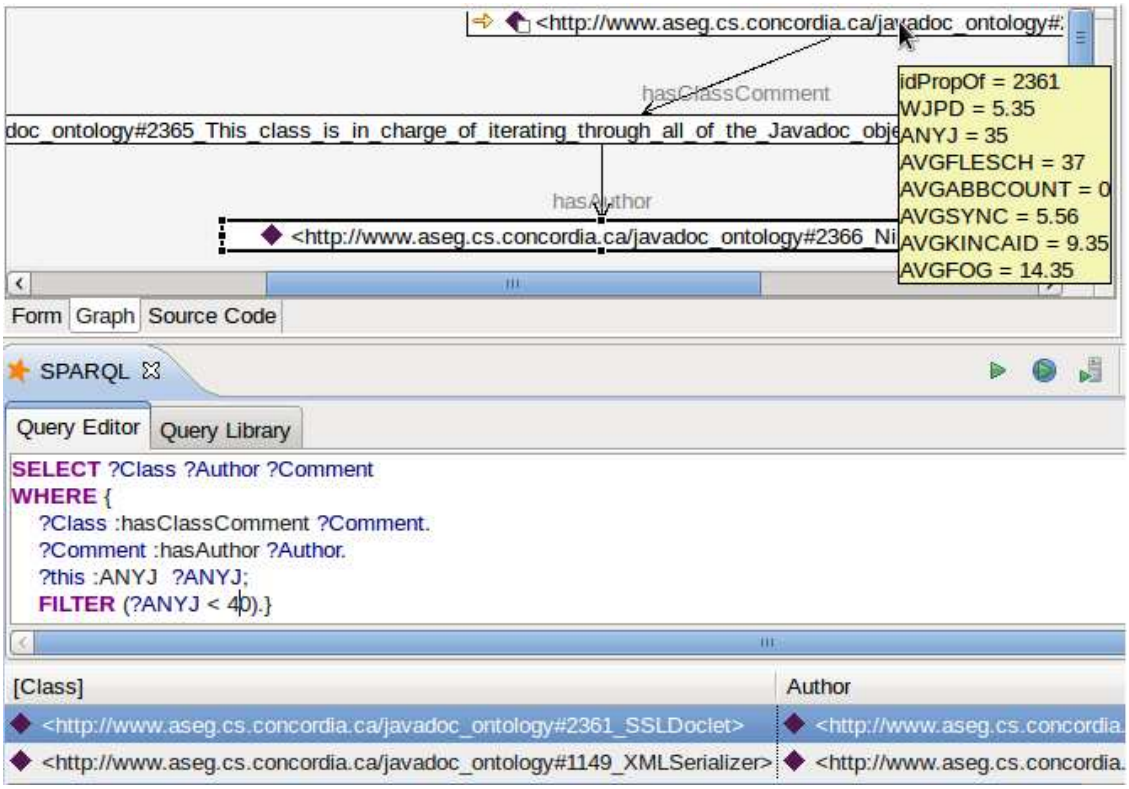

Figure 25: Results of a SPARQL Query on the NLP-Populated Source Code Comment Ontology

that returns classes that were assessed by our JavadocMiner as having low quality documentation. Also returned by the query are the authors that created the Javadocs for the class.

6.5.2 Linking Software Engineering Data

As mentioned in Chapter 5 Section 5.6.1, the process of ontology alignment is to create a mapping between two or more ontologies. More specifically, when two individuals of a concept are detected as being similar, an equivalence relationship is created linking the two instances together [Bei10].

In Figure 26, we show the "StereoTypeUtility" instance found in the version control, Javadoc, and source code ontologies linked together using the bi-directional relationship assesrtions [FBMNPS07] "hasCrossLink". Aligning ontologies in this manner enables us to focus on the information found in the individual ontology, as

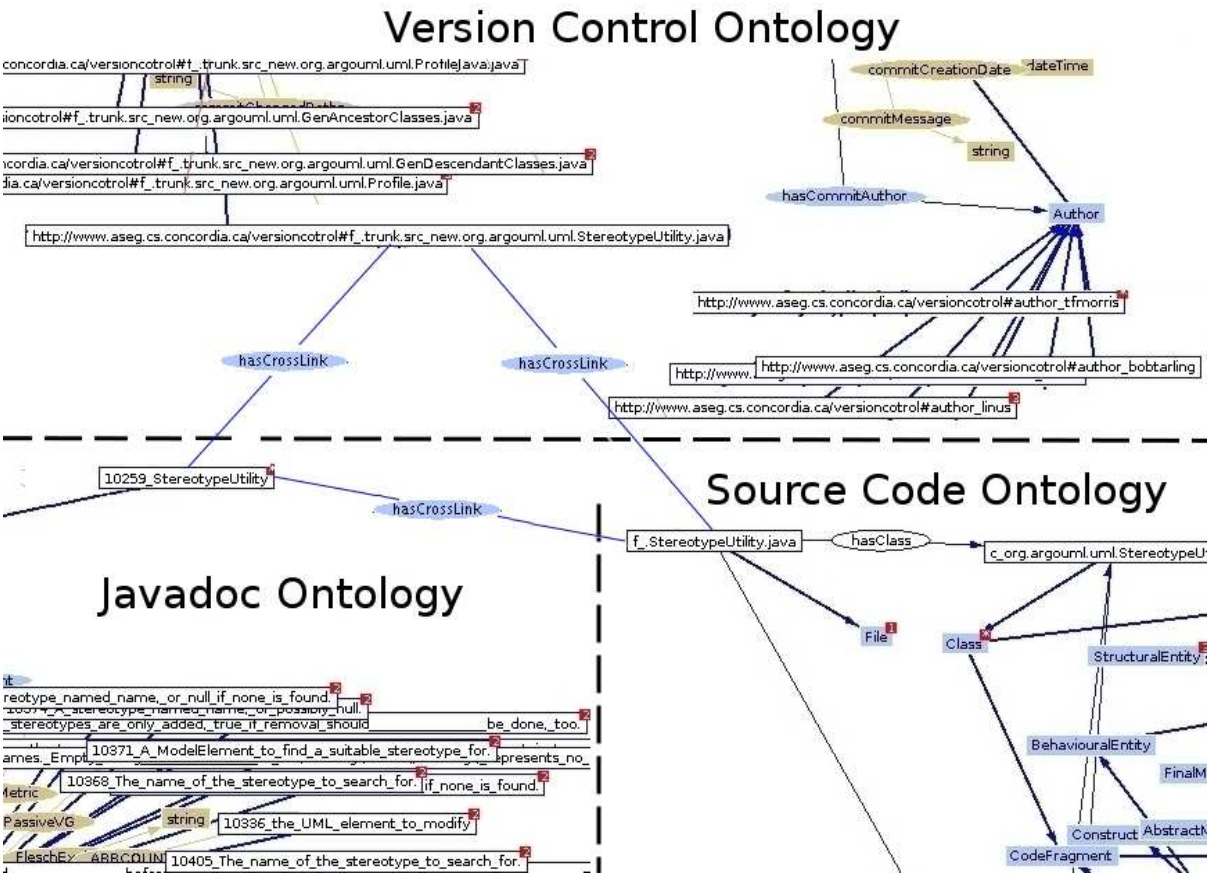

Figure 26: Ontologies Linked Using the hasCrossLink relationship

well as the entire knowledge base.

Using `SPARQL` queries, the entire KB comprised of multiple software engineering artifacts can be queried using a single end-point. In Figure 27, we show a cross artifact query that includes information from the source code, Javadoc and version control ontologies. More specifically, we queried the knowledge base for all Javadoc related commits (as indicated by the commit message), for the class comment belonging to the "StereoTypeUtility" class.

Figure 27: Cross Artifact SPARQL Query

6.6 Summary

In this chapter, we discussed the implementation of our component-based JavadocMiner system. The pipeline is assembled using existing ANNIE PRs, and PRs developed by us such as the ReadabilityMetrics PR. In Figure 28, we show the JavadocMiner pipeline loaded within GATE. The list of components are visible on the left side, in the middle we see a document generated using the SSL Javadoc Doclet, and highlighted is the feature list for the "MethodComment" and "Readabilityanalysis" annotations. Finally, on the right side are the various annotations of the text.

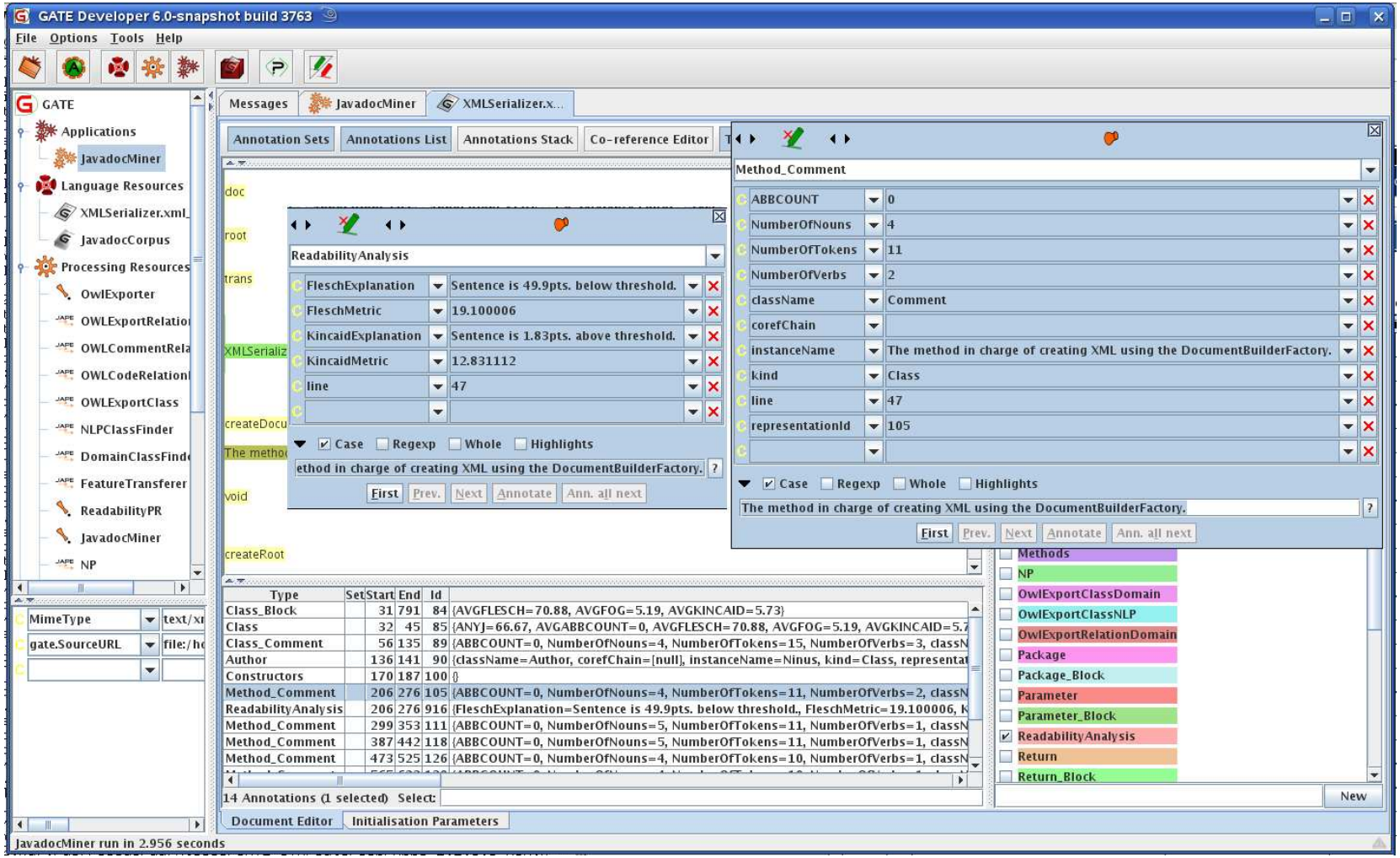

Figure 28: Screenshot of the JavadocMiner system running in GATE Developer

Chapter 7

Evaluation

In this chapter, we discuss how the JavadocMiner was applied on two open source projects to analyze the consistency of source code with comments and the quality of comments in these systems. We begin our analysis by benchmarking the amount of time it takes the SSL Javadoc Doclet to generate a corpus from the source directory. We then evaluate the results of our study by examining how the quality of comments evolved in time between the different versions. In the second part of our evaluation, we attempt to correlate the results obtained from our analysis with bug statistics from each open source project.

7.1 Data

For our case study we required software projects that were:

- Open source, and implemented using the Java programming language.
- Actively maintained projects with at least three major releases.
- Managing source using an issue tracker, such as Redmine[1] or Bugzilla[2].

We also wanted to examine projects that enforced either an organization-specific or public coding standard (e.g., GNU[3]). This enables us to identify the parts of the source code that do not conform to such standards.

[1]Redmine, http://www.redmine.org/
[2]Bugzilla, https://bugzilla.mozilla.org/
[3]GNU, http://www.gnu.org/

Table 7: Assessed Open Source Project Versions, Release Dates, Number of Reported Bugs

Project Version	Release Date	Number of Bug Defects
ArgoUML v0.24	02/2007	46
ArgoUML v0.26	09/2008	54
ArgoUML v0.28.1	08/2009	48
Eclipse v3.3.2	06/2007	176
Eclipse v3.4.2	06/2008	413
Eclipse v3.5.1	06/2009	153

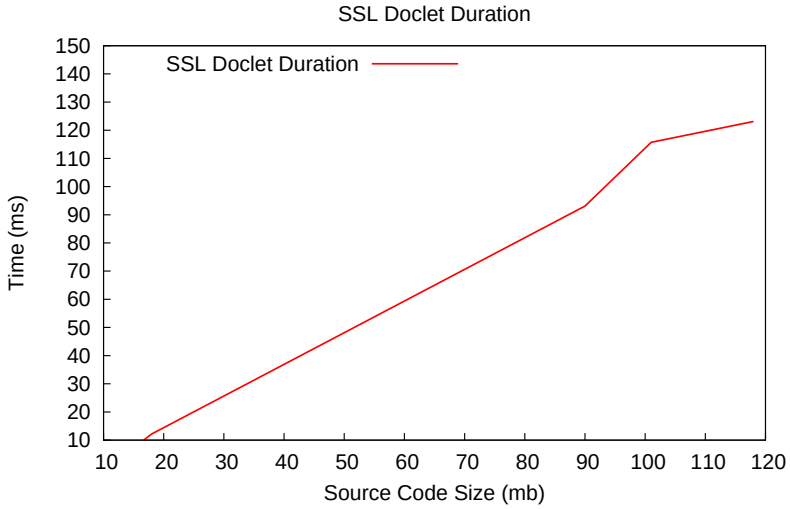

Figure 29: Reported Bugs for ArgoUML and Eclipse OSS

The two projects we selected that fit our requirements are ArgoUML[4], a UML modeling tool, and the Eclipse IDE[5]. Both projects are mature and active projects, which have a significant amount of historical data available. For our study, we were also interested in monitoring how the quality of Javadoc comments for the two projects evolved over time, and therefore applied the JavadocMiner on the last three major releases of each project. In Table 7, we show the versions of the projects that were part of our quality assessment.

[4]ArgoUML, http://argouml.tigris.org/
[5]Eclipse, http://www.eclipse.org/

7.2 Generating a Corpus using Open Source Software

We evaluated the performance of our SSL Javadoc doclet in order to assess the time needed for creating a corpus from source code. In Table 8, we show the time required to process different versions of the ArgoUML and Eclipse open source projects.

Table 8: Open Source Project Versions, Lines of Code (LOC), Number of Comments and Identifiers, and Process Duration for ArgoUML and Eclipse

Project	LOC	Number of Comments	Number of Identifiers	Duration (sec.)
ArgoUML v0.24	250,000	6,871	13,974	3.4
ArgoUML v0.26	600,000	6,875	14,262	8.9
ArgoUML v0.28.1	800,000	7,168	14,789	12.2
Eclipse v3.3.2	7,000,000	32,172	158,009	93.1
Eclipse v3.4.2	8,000,000	33,919	163,238	115.7
Eclipse v3.5.1	8,000,000	34,360	165,945	123.1

In Figure 29, we show the linear time of which it takes the SSL Javadoc Doclet to generate a corpus for the data set used in our evaluation. Making our SSL doclet an efficient tool for transforming Java source code documents into an XML representation.

7.3 Assessing the Quality of In-Line Documentation found in Open Source Software

To assist in interpreting the data and finding correlations between the different measures and software engineering artifacts, such as reported bug defects, we separated the ArgoUML and Eclipse projects into their major modules; for ArgoUML: Top Level, View & Control, and Low Level; and for Eclipse: Plugin Development Environment (PDE), Equinox, and Java Development Tools (JDT).

The quality of the in-line documentation found in each module was assessed separately for a total of 43,025 identifiers and 20,914 comments from ArgoUML, and 487,192 identifiers and 100,451 comments from Eclipse. As part of our evaluation, we continue by finding the amount of bug defects that were reported for each version of

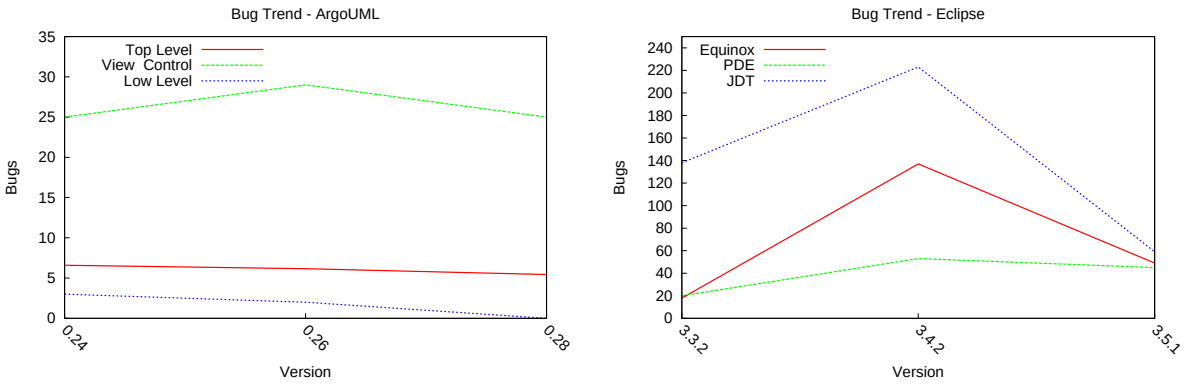

Figure 30: Reported Bugs for ArgoUML and Eclipse OSS

the modules for ArgoUML (Figure 30, left), and for Eclipse (Figure 30, right), using the issue tracker systems used by each project.

The Pearson product-moment [RN88] correlation coefficient measure was then applied to the data gathered from the quality assessment and issue tracker systems to determine the varying degrees of correlation between the individual heuristics and bug defects.

7.4 Quality Analysis

When looking at the code/comment consistency trends for ArgoUML (Figure 31, Top), and for Eclipse (Figure 32, top), we found that the modules that were thoroughly documented and consistent with the source code, are the Low Level module in ArgoUML, and the PDE module in Eclipse.

In terms of the readability measures for ArgoUML (Figure 31, bottom), and Eclipse (Figure 32, bottom), the Low Level and PDE modules maintained readability levels that were in-between the other modules for two of the three versions used in our assessment.

We believe that the reason for these Low Level and PDE modules outperforming the rest of the modules in every heuristic is the fact that are both base libraries used throughout the programs. For example, Eclipse is a framework that is extended using plug-ins that use the services provided by the PDE API module. The Eclipse project is separated into API and internal non-API packages, and part of the Eclipse coding policy states that all API packages must be properly documented [SDZ07].

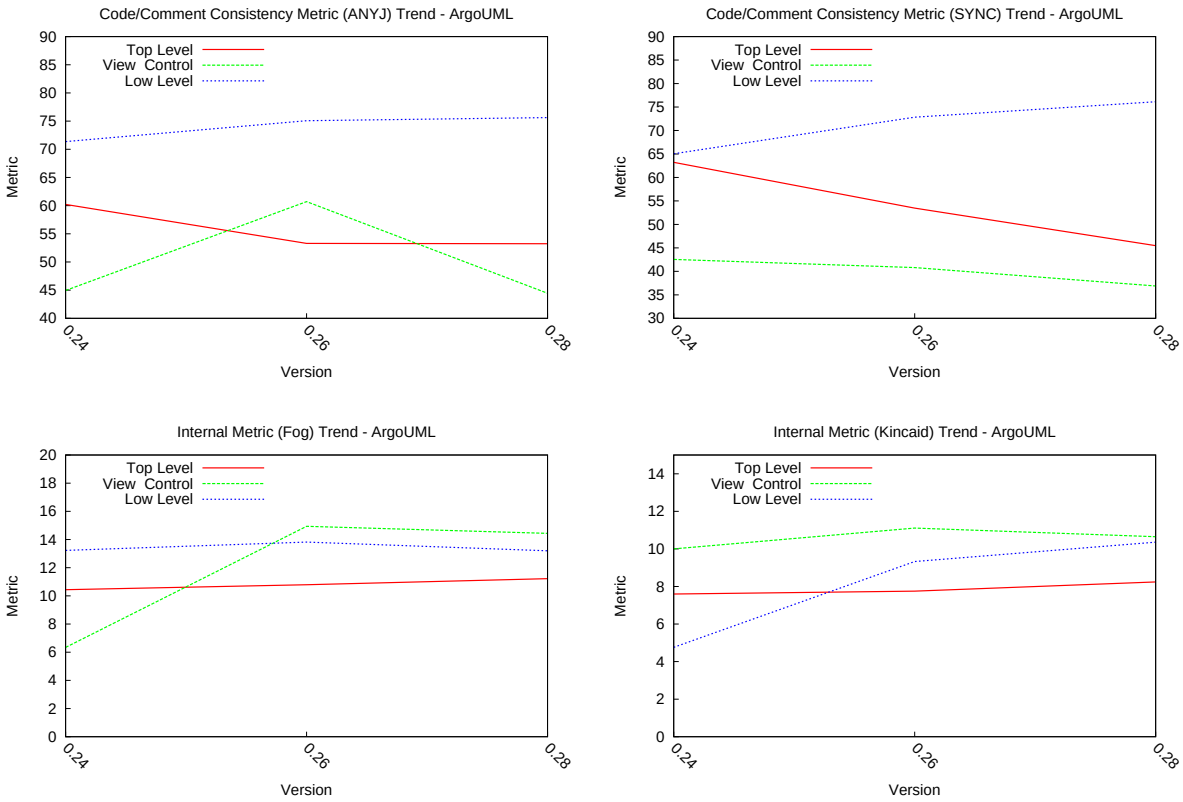

Figure 31: ArgoUML Charts for Code/Comment and Internal (NL Quality) Metrics

Comment-Bug Correlation

Also as part of our analysis, we correlated the results of our study with another software engineering artifact, bug defects. By examining the amount of reported bug defects for each version of the modules (Figure 30), we correlated the quality of comments found in source code with bug defects. Doing so enables us to determine if potential problem areas can be identified early by analyzing the in-line documentation.

As we observed earlier, the modules that performed best in our quality assessment also had the least amount of reported bug defects, and vice versa for the modules that performed poorly. In order to determine how closely each metric correlated with the number of reported bug defects, we applied the Pearson product-moment correlation coefficient [CKC68] on the data gathered from the quality assessment and the number of reported bug defects (Table 9).

The correlation and coefficient results showed ANYJ, SYNC, ABB, Tokens, and WPJC as being amongst the strongly correlated measures. With a correlation of at

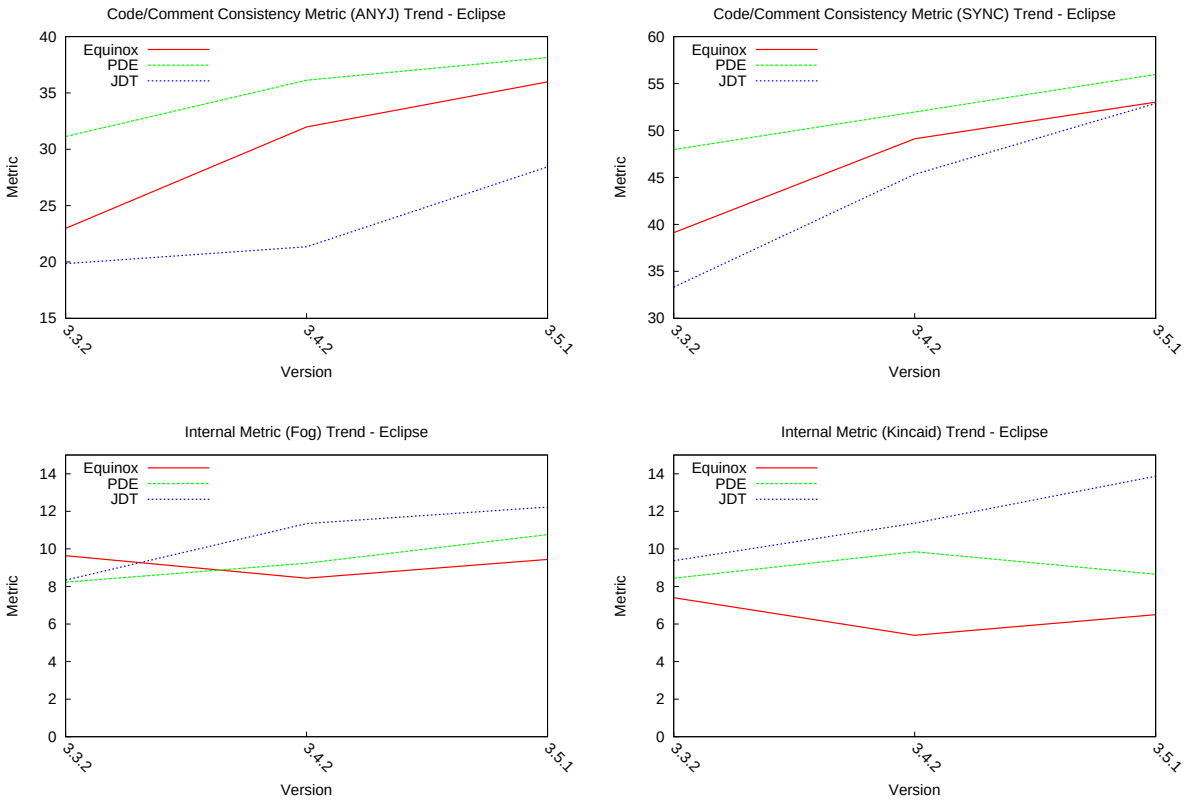

Figure 32: Eclipse Charts Code/Comment and Internal (NL Quality) Metrics

Table 9: Pearson Correlation Coefficient Results for ArgoUML and Eclipse

Project	ANYJ	SYNC	ABB	FOG	KINCAID	TOKENS	WPJC	NOUNS	VERBS
ArgoUML	0.99	0.98	-0.94	0.32	0.79	0.89	0.91	0.98	0.87
Eclipse	0.97	0.89	-0.86	0.37	0.84	0.88	0.86	0.91	0.73

least 80%. In the case of ABB a negative correlation, that is, the higher ABB, the more amount of reported bugs.

To visualize the correlation between reported bug defects and quality assessments for the different modules, we plot the number of reported bug defects (X-Axis) against the values returned by some of the code/comment and internal NL quality metrics (Y-Axis): see Figure 33 for ArgoUML and Figure 34 for Eclipse.

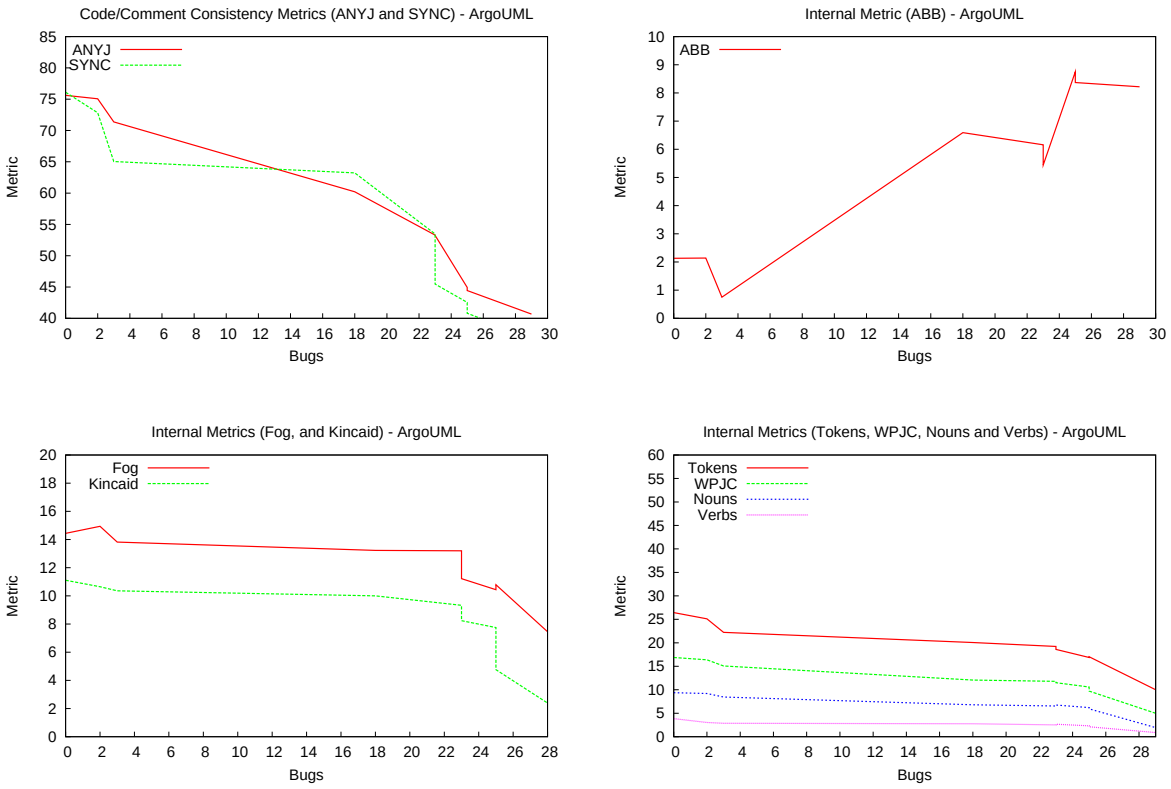

Figure 33: Code/Comment Consistency and NL Quality Metrics vs. \Bugs – ArgoUML

Human Assessment of Javadoc Comments

As part of our efforts to compare the quality assessment made by our JavadocMiner system with that of human intuition, the JavadocMiner was evaluated against annotations manually created by a group of students. For our case study, we asked 14 students from an undergraduate level computer science class (COMP 354) and 27 students from a graduate level software engineering course (SOEN 6431) to evaluate the quality of Javadoc comments randomly selected from ArgoUML. For our survey we selected a total of 110 Javadoc comments: 15 Class and Interface comments, 8 Field comments, and 87 Constructor and Method comments. Before participating in the survey, the students were asked to review the Javadoc guidelines[6] discussed in

[6]Javadoc, http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

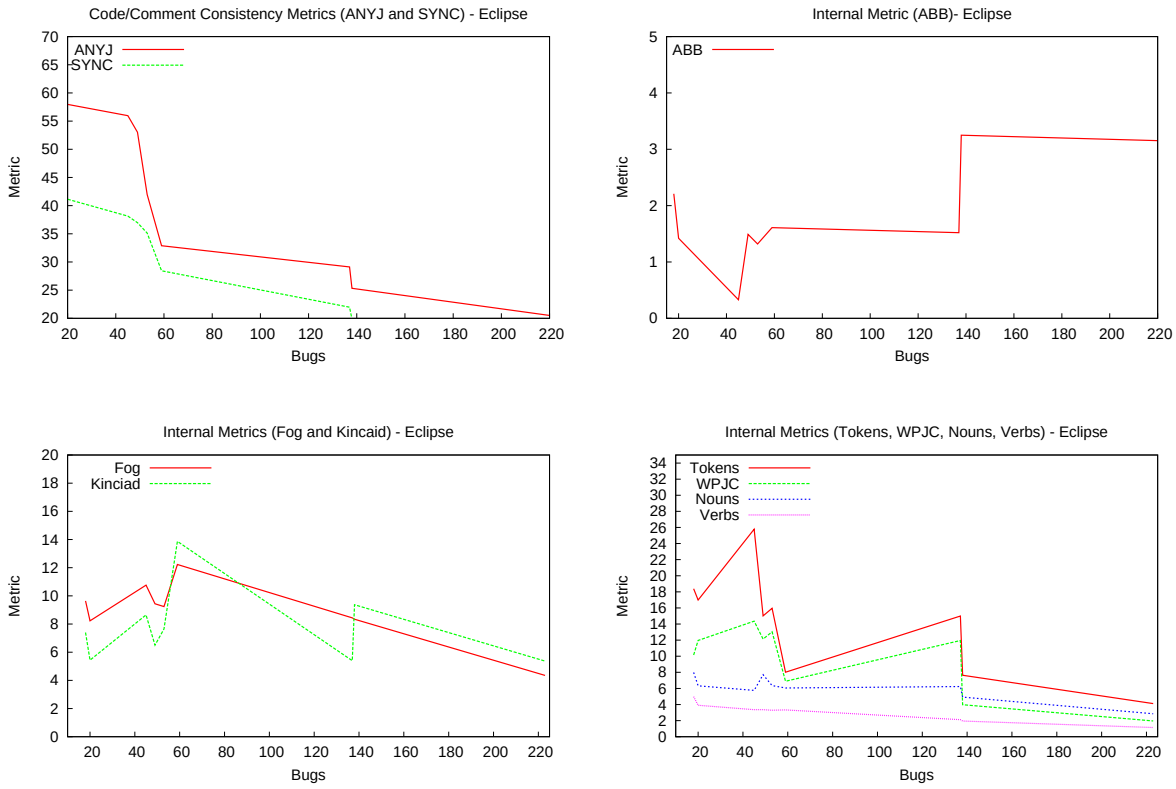

Figure 34: Code/Comment Consistency and NL Quality Metrics vs. \Bugs – Eclipse

Chapter 2. The students had to log into the free online survey tool Kwik Surveys[7] using their student IDs, ensuring that all students complete the survey only once. As shown in Table 10, most of the students participating had at least 3 years of general programming and at least 1-2 years of Java programming experience.

Table 10: Years of General and Java Programming Experience of Students

	General			Java		
Course	**0 Years**	**1-2 Years**	**3+ Years**	**0 Years**	**1-2 Years**	**3+ Years**
COMP 354	11%	31%	58%	7%	61%	32%
SOEN 6431	02%	22%	76%	10%	49%	41%

As part of the case study, the students were asked to evaluate the comments as being either *Very Poor, Poor, Good, or Very Good* as shown in Figure 35. The students were also asked to provide a short description justifying their assessment.

[7]Kwik Surveys, http://www.kwiksurveys.com/

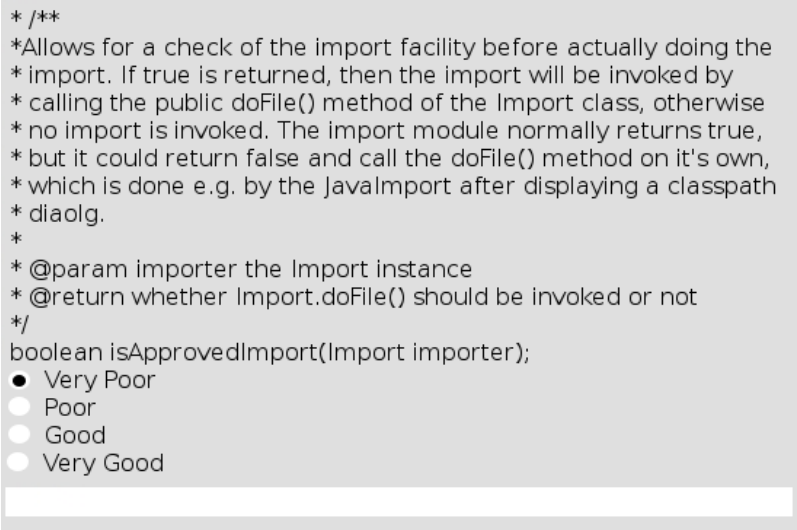

Figure 35: A Sample Question from the Survey

From the 110 manually assessed comments, we selected a total of 67 comments: 5 *class and interface comments*, 2 *field comments*, and 60 *constructor and method comments*, where participants strongly agreed ($\geq 60\%$) as them being of either good (39 comments) or bad (28 comments) quality. While comparing the students manual evaluations of method comments with some of the NL measures of the JavadocMiner (Table 11), we found that the comments that were evaluated negatively contained half as many words (14), compared to the comments that were evaluated as being good. Regardless of the insufficient documentation of the bad comments, the readability index of Fog and Kincaid indicated text that contained a higher density, or more complex material, which the students found hard to understand.

Table 11: Method Comments Evaluated by Students and the JavadocMiner

Student Evaluation	Avg. Number of Words	Avg. Fog	Avg. Kincaid
Good	28.03	12.63	14.15
Bad	14.79	13.98	12.66

In order to evaluate if students were capable of assessing Javadoc comments based on some of the syntactic quality factors, we included methods that contained: *(1)* parameter lists, *(2)* return types and *(1)* thrown exceptions, which needed to be documented using the appropriate Java annotation. While reviewing the survey results, we found that most of students failed to analyze the consistency between the code

and comments as shown in Figure 36. Our JavadocMiner also detected a total of 8 abbreviations being used within comments, which none of the students mentioned.

Justify:
3, +C, +D, +F //Comment indicates how process the error.
parameters are specified,but control flow should not be mentioned
2, +d
missing @param monitor
1, -B, +E
1, +B, -C, +D, +E, +F
1, +B, -C, +D, +E, +F
1, -C, +D, -E
2, +B, -C, -D
1, +C, +E

Figure 36: A Sample Answer from the Survey

Finally, for twelve of the 39 comments that were analyzed by the students as being good, 12 of them were not using a declarative third-person writing style, a detail all of the students also failed to mention. From our case study, it is obvious that humans are incapable of analyzing in-line documentation based on all of the quality factors without the help of an application.

7.5 Summary

In this chapter we illustrated how the JavadocMiner was used to assess the quality of comments found in two open source projects. We began our evaluation by benchmarking the amount of time it took the SSL Javadoc Doclet to generate a corpus from source code and source code comments. We found that the Doclet is able to convert the Java documents into an XML representation in linear time. We continued by analyzing the quality of comments found in different versions of ArgoUML and Eclipse, and observed how the quality of source code comments increased, decreased or was flat over time. To correlate the quality of comments with reported bug defects we *(1)* applied the Pearson product-moment correlation coefficient measure to both the results returned by the JavadocMiner and reported bug defects, and *(2)* plotted the values of the results with the number of reported bug defects for the different modules. The results of that study identified which of the metrics had a strong correlation to bug defects, and could therefore be used to identify parts of an implementation that is most prone to error due to low quality documentation. As a means of comparing

the JavadocMiner with human intuition, we asked students to evaluate the quality of randomly selected comments. What we found was, most students were unable to assess the Javadoc comments given the different quality factors. An indication of the need to automate the process. In Chapter 8 we discuss the conclusion of our work, and what future plans we hold for our JavadocMiner system.

Chapter 8

Conclusions and Future Work

In this thesis, we discussed challenges, the software engineering domain faces, when assessing the quality of source code comments written in natural language. We presented an approach that automatically assesses the quality of Javadoc documentation found in software, implemented in the JavadocMiner tool. We also showed how the JavadocMiner can be used to identify modules that may contain a higher number of bug defects, due to poor and out dated in-line documentation. Regardless of the current trends in software engineering and the paradigm shift from documentation to development, we have shown how potential problem areas can be minimized by maintaining source code that is sufficiently documented using good quality up-to-date source code comments. We demonstrated how the JavadocMiner was used to assess the quality of in-line documentation found in two existing open source projects. We also correlated each of the heuristics with bug defects, and found some of the heuristics had a stronger correlation to bug defects than others.

8.1 Future Work

The readability measures used to analyze the quality of in-line documentation found in source code make use of simple proxies in order to perform grammatical and lexical complexity analysis. Fairly simple features were often employed due to the lack of computational power [HCTE08]. Such features "exhibit high bias" due to their assumption that grammatical complexity is based on sentence length or number of syllables. More recent approaches to reading difficulty implement more sophisticated

models that make use of the growth in computational power [HCTE08]. For example, the authors of [CTC04] created a readability measure using a smoothed unigram model to predict the reading difficulty of web pages. To satisfy the 100 word sample needed by the model, we plan on merging all the comments found in a given `class`, generating a readability index for the entire document. Machine learning techniques are often used in a variety of text classification problems [SO05]. The data from our case study, where we asked students to manually assess the quality of in-line documentation, can also be used to train a machine learning algorithm such as Support Vector Machines (SVMs). Using an SVM algorithm, we can classify the quality of in-line documentation based on the set of features that we currently have, such as NP, VG, writing style, and the Flesch-Kinkaid readability index. Separate classifiers would be used to assess the different types of in-line documentation i.e. class, field and method comments. By using supervised learning techniques, the bias assumptions made by the readability measures would be reduced.

An important challenge in software engineering research is the integration of tools to assist software developers in performing SME tasks efficiently and effectively [KCA06]. The focus of researchers in the past has always been about applying more sophisticated NLP analysis to assist in software engineering, and less about tool integration [KCA06]. For example, the Doc Check Doclet quality analysis tool is designed to be executed using command line, Make, or ANT, and users must also provide Javadoc parameters that influence the output of the analysis. An added task that can be seen as unfavourable, compared to the JavadocMiner that can be invoked from within existing tools such as Eclipse and Hudson.

A central concept in the design of new tools is how they can be tightly integrated into existing tools used for software engineering. Version control systems, issue trackers, mailing lists, and IDEs are just some of the many tools used in industry to facilitate software development. The analysis provided by such tools are mainly syntactic in nature (e.g., making sure an issue contains: issue date, author, severity, status, and description), and simple forms of NL quality analysis (e.g., spell-checking).

Many tools currently being used employ a framework architecture that is easily extended using plug-ins. This is especially true for tools used for software development.

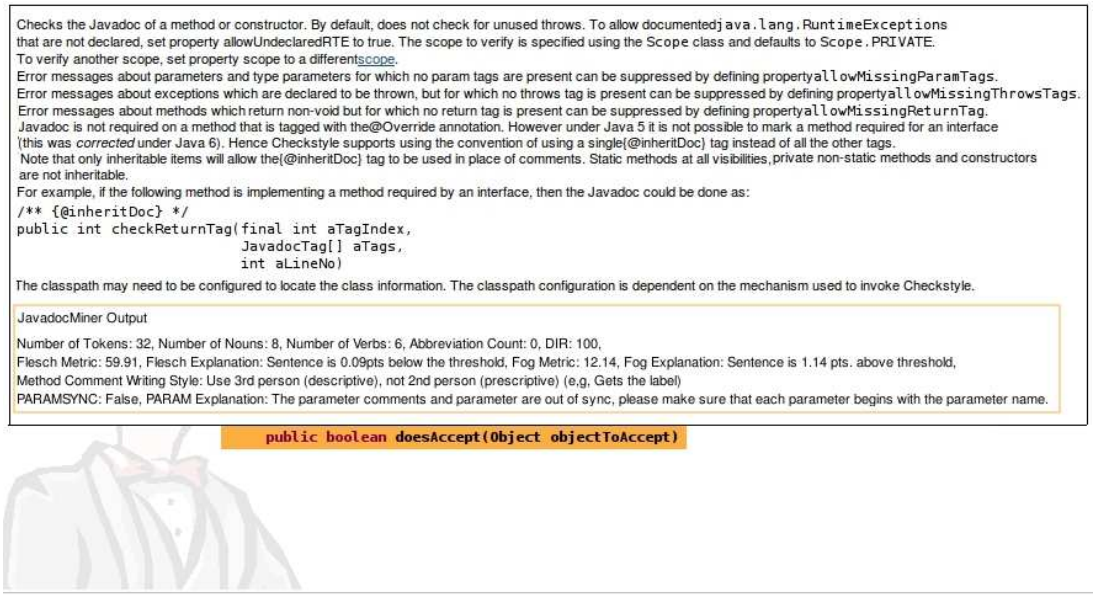

Figure 37: The JavadocMiner Output Included in Hudson

Tools such as Eclipse[1], Redmine[2], or Hudson[3] all provide an interface that enables new features to be added. In [WSKR11] we discussed how the JavadocMiner NLP service was integrated into the Eclipse IDE. This allows developers and maintainers to perform an analysis on their source code comments while still in the Eclipse environment, and receive instant feedback on the quality of comments. This is far more efficient than introducing an entirely new tool that would require users to navigate away from their development environment to perform a quality assessment on their Javadoc comments.

Build servers are also tools commonly used to facilitate software engineering. The tool is in charge of including all dependencies needed by a software system, and building the entire source tree. The task is invoked based on a specified time or event (i.e., file commits using the versioning system). Hudson is a continuous integration server used to manage the quality of source code being committed by the individual stakeholder into a versioning system. It is also implemented using a framework architecture, enabling users to extend the analysis provided by the tool with their own

[1]Eclipse, http://www.eclipse.org/
[2]Redmine, http://www.redmine.org/
[3]Hudson, http://hudson-ci.org/

solutions. Developers have already integrated the Checkstyle analysis tool mentioned in Chapter 7 into the Hudson build server. As part of our future works, we plan on enriching the analysis provided by Hudson and Checkstyle (Requirement #6.1) by including the analysis provided by the JavadocMiner into the Hudson plug-in, as shown in Figure 37.

Chapter 9

Publications

9.1 Accepted / Published

- Ninus Khamis, René Witte, and Juergen Rilling. Automatic Quality Assessment of Source Code Comments: The JavadocMiner. In proceedings of *The 15th International Conference on Applications of Natural Language to Information Systems (NLDB 2010)*, June 23–25, Cardiff, UK. (Acceptance rate: 30%)

- Ninus Khamis, Juergen Rilling, and René Witte. Generating an NLP Corpus from Java Source Code: The SSL Javadoc Doclet. *New Challenges for NLP Frameworks*, Workshop at LREC 2010, pp.41–45, May 22, 2010, Valletta, Malta.

- René Witte, Ninus Khamis, and Juergen Rilling. Flexible Ontology Population from Text: The OwlExporter. In proceedings of *The Seventh International Conference on Language Resources and Evaluation (LREC 2010)*, pp.3845–3850, May 19–21, 2010, Valletta, Malta.

- René Witte, Bahar Sateli, Ninus Khamis, and Juergen Rilling. Intelligent Software Development Environments: Integrating Natural Language Processing with the Eclipse Platform. To appear at *The 24th Canadian Conference on Artificial Intelligence (AI 2011)*, May 25-27, 2011, St. John's, Canada

Bibliography

[ADB04] Ademar Aguiar, Gabriel David, and Greg Badros. Javaml 2.0: Enriching the markup language for java source code. In *XML: Aplicações e Tecnologias Associadas (XATA 2004)*, 2004.

[AHM+09] Surafel Lemma Abebe, Sonia Haiduc, Andrian Marcus, Paolo Tonella, and Giuliano Antoniol. Analyzing the Evolution of the Source Code Vocabulary. *Software Maintenance and Reengineering, European Conference on*, pages 189–198, 2009.

[Av08] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, 2 edition, March 2008.

[Aza89] S. Azar. *Understanding and Using English Grammar*. Prentice Hall Regents, 1989.

[Bad00] Greg J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):159–177, 2000.

[Bei10] Elena Beisswanger. Exploiting relation extraction for ontology alignment. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *International Semantic Web Conference (2)*, volume 6497 of *Lecture Notes in Computer Science*, pages 289–296. Springer, 2010.

[BF09] Panuchart Bunyakiati and Anthony Finkelstein. The Compliance Testing of Software Tools with Respect to the UML Standards Specification - The ArgoUML Case Study. In Dimitris Dranidis, Stephen P. Masticola, and Paul A. Strooper, editors, *AST*, pages 138–143. IEEE, 2009.

[Bro83] Ruven E. Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[BW08] Raymond P. L. Buse and Westley R. Weimer. A metric for software readability. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130, New York, NY, USA, 2008. ACM.

[BWKG05] Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. *SIGSOFT Softw. Eng. Notes*, 30(5):177–186, 2005.

[Cho03] G Chowdhury. Natural language processing. *Language*, 39(1):n/a–n/a, 2003.

[Cim06] Philipp Cimiano. *Ontology Learning and Population from Text: Algorithms, Evaluation and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[CKC68] Frederick E. Croxton, Sidney Klein, and Dudley J. Cowden. *Applied general statistics / Frederick E. Croxton, Dudley J. Cowden and Sidney Klein*. Pitman, London :, 3rd ed. edition, 1968.

[CMB+10] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. *Developing Language Processing Components with GATE Version 6 (a User Guide)*. University of Sheffield, December 2010. For GATE version 6.1.

[CMBT02] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Annual Meeting of the ACL*, 2002.

[CMT00] H. Cunningham, D. Maynard, and V. Tablan. JAPE: a Java Annotation Patterns Engine (Second Edition). Research Memorandum

CS–00–10, Department of Computer Science, University of Sheffield, November 2000.

[Cob99] C. Cobuild, editor. *English Grammar*. Harper Collins, 1999.

[CTC04] Kevyn Collins-Thompson and James P. Callan. A language modeling approach to predicting reading difficulty. In *HLT-NAACL*, pages 193–200, 2004.

[DD07] Harvey M. Deitel and Paul J. Deitel. *Java*. Prentice Hall, 7th ed edition, 2007. xliv, 1579 p. ; 24cm. 2 CD-ROMS(4 3/4 in.) : ill.

[dSM09] Luis Carlos dos Santos Marujo. REAP em Português, July 2009.

[DuB06] William H. DuBay. *Smart language: Readers, Readability, and the Grading of Text*. Impact Information, 2006.

[EC49] Dale Edgar and Jeanne Chall. The concept of readability. 1949.

[EM82] James L. Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Commun. ACM*, 25:512–521, August 1982.

[FBMNPS07] Diego Calvanese Franz Baader, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2007.

[FL02] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, DocEng '02, pages 26–33, New York, NY, USA, 2002. ACM.

[FSH+08] Z. P. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker. Analysing source code: looking for useful verb-direct object pairs in all the right places. *Software, IET*, 2(1):27–36, February 2008.

[FWG07] Beat Fluri, Michael Würsch, and Harald Gall. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *WCRE*, pages 70–79, 2007.

[FWGG09] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Control*, 17(4):367–394, 2009.

[GP03] Charles F. Goldfarb and Paul Prescod. *The XML Handbook, Fifth Edition*. Prentice-Hall PTR, 2003.

[Gru93] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition Academic Press Inc. 5(2)*, 1993.

[Har00] Gretchen Hargis. Readability and computer documentation. *ACM Journal of Computer Documentation*, 24(3):122–131, 2000.

[HCTE08] Michael Heilman, Kevyn Collins-Thompson, and Maxine Eskenazi. An analysis of statistical models and features for reading difficulty prediction. In *Proceedings of the Third Workshop on Innovative Use of NLP for Building Educational Applications*, EANL '08, pages 71–79, Morristown, NJ, USA, 2008. Association for Computational Linguistics.

[Hep00] Mark Hepple. Independence and commitment: Assumptions for rapid training and execution of rule-based POS taggers. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL-2000)*, Hong Kong, October 2000.

[HFB+08] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 79–88, New York, NY, USA, 2008. ACM.

[HKST06] Hans-Jörg Happel, Axel Korthaus, Stefan Seedorf, and Peter Tomczyk. KOntoR: An Ontology-enabled Approach to Software Reuse. In *IN: PROC. OF THE 18TH INT. CONF. ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING*, 2006.

[HM01] Volker Haarslev and Ralf Möller. RACER System Description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning:*

First International Joint Conference (IJCAR) 2001, volume 2083 of *Lecture Notes in Computer Science*, page 701, Siena, Italy, June18-23 2001. Springer-Verlag.

[HP02] Rodney D. Huddleston and Geoffrey K. Pullum. *The Cambridge Grammar of the English Language*. Cambridge University Press, April 2002.

[JH06] Zhen Ming Jiang and Ahmed E. Hassan. Examining the evolution of code comments in PostgreSQL. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 179–180, New York, NY, USA, 2006. ACM.

[JW00] W. Strunk Jr and E. B. White. *The Elements of Style*. Allyn & Bacon, 4th edition, 2000.

[KBT07] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. Mining Software Repositories with iSPAROL and a Software Evolution Ontology. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.

[KCA06] Andrew J. Ko, Michael J. Coblenz, and Htet H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, December 2006.

[Kla00] George R. Klare. Readable computer documentation. *ACM J. Comput. Doc.*, 24:148–168, August 2000.

[Knu84] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[Kod04] Viswanathan Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Software*, 21(4):70–77, 2004.

[Kos10] Jussi Koskinen. *Software Maintenance Fundementals*. Taylor & Francis Group., 2010.

[Kot00] Jeffrey Kotula. Source Code Documentation: An Engineering Deliverable. *Technology of Object-Oriented Languages, International Conference on*, 0:505, 2000.

[Kra99] Douglas Kramer. API documentation from source code comments: a case study of Javadoc. In *SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153, New York, NY, USA, 1999. ACM.

[KWR10] Ninus Khamis, René Witte, and Juergen Rilling. Generating an NLP Corpus from Java Source Code: The SSL Javadoc Doclet. In *New Challenges for NLP Frameworks*, Valletta, Malta, 05/2010 2010. ELRA.

[LB85] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Lid01] E.D. Liddy. *Natural Language Processing*. Marcel Decker, Inc, NY, 2nd edition, 2001.

[LPR98] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of Evolution Metrics on Software Maintenance. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 208–, Washington, DC, USA, 1998. IEEE Computer Society.

[MCHS09] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, and Ulrike Sattler. Representing Ontologies Using Description Logics, Description Graphs, and Rules. *Artificial Intelligence*, 173(14):1275–1309, 2009.

[MP82] Douglas R. McCallum and James L. Peterson. Computer-based readability indexes. In *Proceedings of the ACM '82 conference*, ACM '82, pages 44–48, New York, NY, USA, 1982. ACM.

[MRBW10] Christoph Müller, Guido Reina, Michael Burch, and Daniel Weiskopf. Subversion Statistics Sifter. In George Bebis, Richard D. Boyle, Bahram Parvin, Darko Koracin, Ronald Chung, Riad I. Hammoud, Muhammad Hussain, Kar-Han Tan, Roger Crawfis, Daniel Thalmann,

David Kao, and Lisa Avila, editors, *ISVC (3)*, volume 6455 of *Lecture Notes in Computer Science*, pages 447–457. Springer, 2010.

[NLC03] E. Nurvitadhi, Wing Wah Leung, and C. Cook. Do class comments aid Java program understanding? In *Frontiers in Education (FIE)*, volume 1, Nov. 2003.

[Pfl98] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, 1998.

[PL08] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Foundation and Trends in Information Retrieval*, 2(1-2):1–135, 2008.

[PP09] D. Pierret and D. Poshyvanyk. An empirical exploration of regularities in open-source software lexicons. pages 228–32, Piscataway, NJ, USA, 2009.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. Technical report, 1 2008.

[PTZ09] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers Taxonomies and characteristics of comments in operating system code. In *ICSE '09*, pages 331–341, Washington, DC, USA, 2009. IEEE Computer Society.

[Ray03] Erik T. Ray. *Learning XML*. O'Reilly & Associates, Sebastopol, California, 2nd edition, September 2003.

[RN88] J. L. Rodgers and W. A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42:59–66, 1988.

[RWSC08] Juergen Rilling, René Witte, Philipp Schuegerl, and Philippe Charland. Beyond Information Silos – An Omnipresent Approach to Software Evolution. *International Journal of Semantic Computing (IJSC)*, 2(4):431–468, December 2008. Special Issue on Ambient Semantic Computing.

[SDZ07] Daniel Schreck, Valentin Dallmeier, and Thomas Zimmermann. How documentation evolves over time. In *IWPSE '07: Ninth international*

workshop on Principles of software evolution, pages 4–10, New York, NY, USA, 2007. ACM.

[SO05] Sarah E. Schwarm and Mari Ostendorf. Reading Level Assessment Using Support Vector Machines and Statistical Language Models. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 523–530, 2005.

[SPG+07] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, June 2007.

[TH06] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *In Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006*, pages 292–297. Springer, 2006.

[TYKZ07] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, New York, NY, USA, 2007. ACM.

[Wit] René Witte. Multi-lingual noun phrase extractor.

[WKR11] René Witte, Ninus Khamis, and Juergen Rilling. Flexible Ontology Population from Text: The OwlExporter. In *Int. Conf. on Language Resources and Evaluation (LREC)*, Valletta, Malta, 05/2010 2011. ELRA.

[WSKR11] René Witte, Bahar Sateli, Ninus Khamis, and Juergen Rilling. Intelligent Software Development Environments: Integrating Natural Language Processing with the Eclipse Platform. In *Canadian Conference on Artificial Intelligence (Canadian AI)*, Newfoundland and Labrador, Canada, 05/2011 2011.

[YWA05] Annie T. T. Ying, James L. Wright, and Steven Abrams. Source code that talks: an exploration of Eclipse task comments and their implication to repository mining. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[Zha08] Hongyu Zhang. Exploring Regularity in Source Code: Software Science and Zipf's Law. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 101–110, Washington, DC, USA, 2008. IEEE Computer Society.

[Zip32] G. K. Zipf. Selective Studies and the Principle of Relative Frequency in Language, 1932.

[Zok02] David M. Zokaites. Writing Understandable Code. 2002.

[ZWRH06] Yonggang Zhang, René Witte, Juergen Rilling, and Volker Haarslev. An Ontology-based Approach for the Recovery of Traceability Links. In *3rd Int. Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006)*, Genoa, Italy, October 1st 2006.

Appendix A

JavadocMiner Pipeline

A detailed list of the GATE PRs used in our JavadocMiner application can be found in Table 12. In bold are the components developed by us.

Table 12: Processing resources of the JavadocMiner pipeline

Processing function	**GATE resource name**	**Control input**
DocumentResetPR	Document Reset PR	
AnnotTransfer	Annotation Set Transfer	Standard PR used to transfer the annotations from Default markup to the processing AS.
Tokenization	ANNIE English Tokenizer	Standard tokenization rules for English text.
JavadocSplit	JAPE Transducer	Grammar to identify Javadoc sentences.
ANNIESentence-Splitter	ANNIE Sentence Splitter	Standard sentence splitting rules for sentences.
Part of speech tagging	ANNIE POS Tagger	Brill tagger trained on general English text.

Continued on next page ...

Continuation of Table 12 ...

Processing function	**GATE resource name**	**Control input**
Gazetteer	ANNIE Gazetteer	Gazetteer list used for detecting commonly used abbreviations in English text.
Stemmer	Stemmer PR	Rule based component, in charge of taking the stem of the word.
JavadocMiner	JavadocMiner PR	Contains the set of metrics specifically used to analyse Javadoc comments.
ReadabilityMetrics	ReadabilityMetrics PR	Contains the set of metrics used to analyze english text.
FeatureTransferer	JAPE Transducer	Grammar to transfer features from the inside annotation set to the outside annotation set.
DocInfo	JAPE Transducer	Creates document information such as sourceURL, start and end offsets.
Verb Phrase chunking	ANNIE VP Chunker	Grammer to identify Verb Phrases
Noun Phrase chunking	JAPE Transducer	Grammar to identify Noun Phrases
DomainClassFinder	JAPE Transducer	Identifies the domain terminology annotated by the previous PRs. Needed by the OwlExporter to create instances in the domain ontology model.
NLPClassFinder	JAPE Transducer	Identifies the NLP terminology annotated by the previous PRs. Needed by the OwlExporter to create instances in the NLP ontology model.
OwlExportClass	JAPE Transducer	Creates the temporary annotation type needed by the OwExporter to create instances in the ontology model.

Continued on next page ...

Continuation of Table 12 ...

Processing function	**GATE resource name**	**Control input**
CodeRelationFinder	JAPE Transducer	Identifies the source code relationships annotated by previous PRs. Needed by the OwlExporter to create source code object and datatype relationships.
CommentRelation-Finder	JAPE Transducer	Identifies the in-line documentation relationships annotated by previous PRs. Needed by the OwlExporter to create comment object and datatype relationships.
NLPRelationFinder	JAPE Transducer	Identifies the NLP relationships annotated by previous PRs. Needed by the OwlExporter to create NLP object and datatype relationships.
DomainNLP-Relation-Finder	JAPE Transducer	Identifies the relationships between the NLP and domain annotations created by previous PRs. Needed by the OwlExporter to create object and datatype relationships between the NLP and domain ontologies.
OwlExporter	OwlExporter PR	The PR that is in charge of exporting the instances and relationships identified by the pipeline to the related ontology.

Appendix B

Components Developed for the JavadocMiner

B.1 SSLDoclet Parameters

The run-time parameters accessible within GATE for the SSLDoclet component, together with the default values, are shown in Table 13.

Table 13: Default parameter settings for the SSLDoclet component

Parameter Name	**Type**	**Default**	**Comment**
corpus	Corpus		The GATE corpus needed to store the generated input documents.
appendCorpus	Boolean	false	Specifies whether all Java files processed by the doclet get appended to the corpus.
doclet	String		Specifies the doclet to be used when generating the Javadocs.
docletPath	URL		Specifies the location of the doclet to be used.
debugFlag	Boolean	false	Specifies whether or not to print debugging messages while running the component.

B.2 JavadocMiner Parameters

The run-time parameters accessible within GATE for the JavadocMiner component, together with the default values, are shown in Table 14.

Table 14: Default parameter settings for the JavadocMiner component

Parameter Name	**Type**	**Default**	**Comment**
corpus	Corpus		The GATE corpus needed to store the generated input documents.
inputASName	String		Specifies the annotation set that the user would like to process using the JavadocMiner component.
metricList	ArrayList	POS	Specifies the list of JavadocMiner metrics that the user would like to run. Possible options are (POS, MethodCompleteness, CommentAverage, ClassCompleteness, AbbreviationCount, MethodCommentStyle).
debugFlag	Boolean	false	Specifies whether or not to print debugging messages while running the component.

B.3 ReadabilityMetrics Parameters

The run-time parameter accessible within GATE for the ReadabilityMetrics component together with the default values are shown in Table 15.

Table 15: Default parameter settings for the Readability-Metrics component

Parameter Name	**Type**	**Default**	**Comment**
inputASName	String		Specifies the annotation set that the user would like to process using the ReadabilityMetrics component.
outsideAnnotation	Set	Document	Specifies the annotation(s) that the application will append features to, specifying the average figures returned by the analysis on the insideAnnotationSet. For the JavadocMiner application the values are (AbstractClass_Block, Class_Block, Interface_Block).
insideAnnotaiton	Set	Sentence	Specifies the annotation(s) that the application will use to apply the set of metrics on. For the JavadocMiner application the values are (Field_Comment, Method_Comment, Constructor_Comment).
runFleschMetric	Boolean	true	Specifies whether or not to run the Flesch readability metric on the corpus.
fleschThreshold	double	65	Specifies the the threshold the user would like to set for the metric. If the sentence has a Flesh readability index below the threshold, a feature is created stating the difference between the two values.
runFogMetric	Boolean	true	Specifies whether or not to run the Fog readability metric on the corpus.

Continued on next page ...

Continuation of Table 15 ...

Parameter Name	**Type**	**Default**	**Comment**
fogThreshold	double	60	Specifies the threshold the user would like to set for the metric. If the sentence has a Fog readability index above the threshold, a feature is created stating the difference between the two values.
runKincaidMetric	Boolean	true	Specifies whether or not to run the Kincaid readability metric on the corpus.
kincaidThreshold	double	11	Specifies the threshold the user would like to set for the metric. If the sentence has a Kincaid readability index above the threshold, a feature is created stating the difference between the two values.
debugFlag	Boolean	false	Specifies whether or not to print debugging messages while running the component.

Appendix C

Generic GATE Components used for the JavadocMiner

C.1 JAPE Transducer.

JAPE stands for Java Annotation Patterns Engine and provides finite state transduction over text annotations based on regular expressions [CMT00]. A grammar written in JAPE is compiled into a transducer that consists of a set of phases, each of which consists of a set of pattern/action rules [CMB+10].

C.2 Tokenizer

The tokenizer annotates tokens of a text according to their symbolic structure. Therefore, it creates the "`Token`" Annotations with the Features "`orth`" and "`kind`". It is kept simple, so that it is on the one side flexible enough for all kinds of different tasks and on the otherside very efficient [CMB+10]. It leaves the more complex work to the JAPE Transducers (see Section C.1).

C.3 Sentence Splitter

The resource is a cascade of finite-state transducers, which segments text into sentences. Eventually, it creates the Annotation "Sentence" attached to the sentence boundaries, which is used by other Processing Resources. Each sentence is annotated

with the type `Sentence`. Each sentence break (such as a full stop) is also given a Split annotation. This has several possible types: ".", "punctuation", "CR" (a line break) or "multi" (a series of punctuation marks such as "?!?!". The sentence splitter is domain- and application-independent [CMB+10].

C.4 Part-Of-Speech Tagger

A special resource for tagging tokens of text with corresponding parts of speech. The Hepple Tagger uses a lexicon and a rule-set, obtained as the result of machine learning on a large corpus taken from the Wall Street Journal. The resource extends the "Token" Annotation with the Feature "pos", which has a value representing a part-of-speech of the current token. [CMB+10].

C.5 Gazetteer

This component is used for tagging tokens with their semantic categories. It creates the "Lookup" Annotation over certain tokens with the "majorType" and "minorType" features. Values of these Features define major and minor semantic categories of the token. The resource utilizes a set of list files: each file containing a set of names that have a certain type; and the definition file (lists.def), which attaches the "majorType" and "minorType" values to each names file [CMB+10].

C.6 Stemmer

The stemmer plugin, "Stemmer_Snowball", consists of a set of stemmers PRs for the following 11 European languages: Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Portuguese, Russian, Spanish and Swedish. These take the form of wrappers for the Snowball stemmers freely available from `http://snowball.tartarus.org`. Each Token is annotated with a new feature 'stem', with the stem for that word as its value. The stemmers should be run as other PRs, on a document that has been tokenised [CMB+10].

C.7 Multi-lingual Noun Phrase Extractor

MuNPEx [Wit] is a base NP chunker, i.e., it does not deal with any kind of conjunctions, oppositions, or PP-attachments. It is implemented using JAPE and can make use of previously detected named entities (NEs) to improve chunking performance [CMB+10].

C.8 Verb Group Chunker

The rule-based verb chunker is based on a number of grammars of English ([Cob99], [Aza89]). 68 rules were developed for the identification of non recursive verb groups. The rules cover finite ('is investigating'), non-finite ('to investigate'), participles ('investigated'), and special verb constructs ('is going to investigate') [CMB+10].