Assessing the quality factors found in in-line documentation written in natural language: The JavadocMiner

Ninus Khamis, Juergen Rilling, René Witte*

Department of Computer Science and Software Engineering, Concordia University, Montréal, Canada

Abstract

An important software engineering artefact 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 this 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 natural language, ensuring its quality so far needed to be performed manually. In this paper, 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 two 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.

Keywords: Text Mining, Software Engineering, Source Code Comments, Automated Quality Analysis, Javadoc, Ontology

1. Introduction

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 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. A multitude of other software engineering artefacts also contain a significant amount of information written in informal natural language, e.g., 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" [1], yet the amount of research focused towards the quality assessment of in-line documentation is limited [2]. Recent advancements in 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. Furthermore, the close proximity of in-line documentation to the source code enables us to develop additional heuristics to also assess the consistency between code and documentation, which is known to often degrade due to changes in source code not being reflected in their comments. Results of the NLP analysis are then exported into OWL ontologies, which allow to query, reason, and cross-link them with other software engineering artefacts, contributing towards the creation of a rich knowledge base incorporating multiple software repositories [3].

*Corresponding Author, Concordia University, 1455 Boulevard de Maisonneuve O., H3G 1M8 Montréal, QC, Canada

Email addresses: `ni_kham@cse.concordia.ca` (Ninus Khamis), `rilling@cse.concordia.ca` (Juergen Rilling), `rwitte@cse.concordia.ca` (René Witte)

2. Background

In this section, we discuss background relevant for the presented work, in particular *Javadoc* and the impact of in-line documentation on software maintenance.

2.1. Source Code Comments and their 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 [4]. Since in-line documentation comes in contact with 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 available is through performing time-consuming manual code checks.

Any well-written computer program should contain a sufficient number of comments to permit people to understand it. Development programmers 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, like the GNU Coding Standards.[1] In general, each program module should contain comments that capture a module's logic, purpose and rationale. Such comments may also include references to subroutines and descriptions of conditional processing. Specific comments for specific lines of code may also be necessary for unusual coding. For example, an algorithm (or formula) for a calculation may be preceded by a comment explaining its source, the data required, and the result of the calculation and how the result is used by the program.

Writing in-line documentation is often seen as a painful and time-consuming task that is easily neglected due to release or launch deadlines. 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. Ensuring that development programmers use the facilities of the programming language to integrate comments into the code and update those comments is an important aspect of software quality. Even though the impact of poor quality documentation is widely known, only very limited research exists that focuses on the automated assessment of in-line documentation [5].

```
/**
* 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

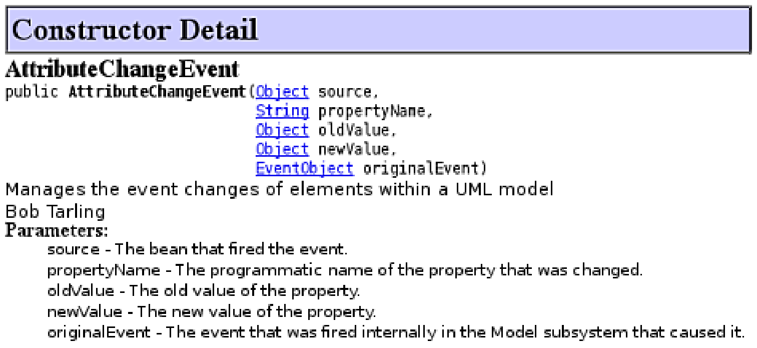
2.2. In-line Documentation and Javadoc

Literate programming was suggested in the early 1980's by Donald Knuth [6] 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

[1]GNU Coding Standards, http://www.gnu.org/prep/standards/standards.html

weave. In order to differentiate between source code and documentation, a specific documentation or programming syntax has to be used.

Single-source documentation, like Javadoc [7], also falls into the category of documents with inter-weaved representation. Contrary to the literate approach, Javadoc 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 [8]. Documentation generators are designed to lessen the efforts needed by developers when documenting software, and have therefore become widely accepted and used. Javadoc is a well-known tool that generates API



Constructor Detail

AttributeChangeEvent

```
public AttributeChangeEvent(Object source,
                            String propertyName,
                            Object oldValue,
                            Object newValue,
                            EventObject originalEvent)
```

Manages the event changes of elements within a UML model

Bob Tarling

Parameters:

source - The bean that fired the event.
propertyName - The programmatic name of the property that was changed.
oldValue - The old value of the property.
newValue - The new value of the property.
originalEvent - The event that was fired internally in the Model subsystem that caused it.

Figure 2: Documentation generated using Javadoc for an ArgoUML Constructor

documentation in HTML from Java source code and its comments [7]. Figure 1 shows an example of a Javadoc comment taken from the open source project ArgoUML, with the corresponding generated documentation in Figure 2.

2.3. How To Write Javadoc Comments

Javadoc comments added to source code are distinguishable from normal comments by a special comment syntax (`/**`). A generator (similar to the weave tool within literate programming) can extract these comments and transform the corresponding documentation into a variety of output formats, such as HTML, $\LaTeX$, or PDF. Most tools also provide specific tags within comments that influence the format of the documentation produced or the way documentation pages are linked.

Different types of comments are used to document the different types of identifiers. For example, a class comment should provide insight on the high-level knowledge of a program, e.g., which services are provided by the class, and which other classes make use of these services [1]. A method comment, on the other hand, should provide a low-level understanding of its implementation [1]. Even during early stages of implementation, the Javadoc tool can process pure stubs (classes with no method bodies), enabling the comment within the stub to explain what future plans hold for the created identifiers.

When writing comments for the Javadoc tool, there are a number of guideline specifications that need to be followed to ensure the tool is being used effectively.[2]
These specifications include details such as:

- Class/interface/field descriptions can omit the subject and simply state the object.
- Method descriptions need to begin with verb phrases.
- Use third person, declarative, rather than second person, prescriptive.
- Do not include any abbreviations when writing source code comments.

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

3. NLP Corpus Generation from Source Code

Before we can analyse Javadoc comments, we need to discuss the transformation of source code and its comments into a representation format suitable for automated NLP analysis.

3.1. Javadoc Doclets

The Javadoc tool provides both the implementation needed to extract information from the source directory and the library used to develop custom transformation tools, called *Doclets* [7]. Doclets provide an interface to the set of objects that are created as a result of the static source code analysis.

Javadoc's standard Doclet generates API documentation using the HTML format; while this is convenient for human consumption [9], automated NLP analysis applications can benefit from a more structured XML format: With HTML, we would be 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>` [10]. For processing in an NLP framework, an XML format where the semantics are encoded in explicit markup is much more versatile than HTML, as it enables us to: *(i)* define custom tags and attributes to mark up the information of the XML elements [11] and *(ii)* define a well-formed XML structure that is easily processed by NLP applications. Generating these structured XML documents from Javadoc information is possible by developing a custom Doclet that uses the Javadoc library.

```
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 3: An Abstract Class Declaration taken from ArgoUML's Source Code

The standard Doclet also provides a limited amount of checks that are mainly syntactic in nature. However, more analyses can potentially be applied on Javadoc comments, measuring factors such as completeness, code/comment consistency and readability.

3.2. Marking Up Source Code

The *Semantic Software Lab Doclet* (SSLDoclet)[3] [12] 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 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; and
- the return types, parameter list, and thrown exceptions of a method.

[3]SSLDoclet, http://www.semanticsoftware.info/javadoclet

```
<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 4: A Section of an NLP Corpus, generated from Source Code using an Abstract Class Declaration

As an example, consider the code in Figure 3, which shows an *Abstract Class* declaration taken from the open source project ArgoUML [13]. Figure 4 shows how our Doclet represents the information found in the abstract class using the `<Package>` and `<Extends>` tags to model the package the abstract class belongs to and the superclass that it extends. An example of how the parameter list of the "intialiseListener" method are modeled using the XML tag `<Parameter>` is illustrated in Figure 5.

```
<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 5: A Section of a Corpus Generated from Javadoc Comments

Both Figure 4 and 5 demonstrate how our Doclet is able to represent more information effectively using XML attributes, compared to the standard HTML output. For example, we now also know that the parent of the "Call-StatNotation" superclass is Object, and that the "listener" parameter of the "intialiseListener" method has the type "PropertyChangeListener".

3.3. *Marking Up Source Code Comments*

Our SSLDoclet is also able to mark up the natural language information found in Javadoc comments, such as the docComment, block, and in-line tags. Figure 3 also includes an example of a Javadoc comment that includes a docComment and uses the @author in-line tag. Figure 4 shows how the 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.

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 6, we show how the relationships found in the sample source code are modeled. We eliminated the XML elements and attributes 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 6: SSLDoclet Schema

The information generated using the SSLDoclet could have been modeled in a number of different ways; however, it is important to keep in mind that when a corpus is loaded within an NLP framework, the XML tags are typically interpreted as annotations, the XML elements are interpreted as entities of the annotation they belong to, and finally the XML attributes are interpreted as features of the annotation.

4. Improving Software Quality through Automatic Comment Analysis: The JavadocMiner

The goal of our JavadocMiner tool is to enable users to 1) automatically assess the quality of source code comments and 2) export the in-line documentation and the results returned from this analysis to an ontology. The JavadocMiner is also capable of giving users recommendations on how a Javadoc comment may be improved based on the "How to Write Doc Comments for the Javadoc Tool" guidelines and the metrics discussed in this section. These metrics are grouped into two categories, *(i)* internal, i.e., natural language (NL) quality only and *(ii)* code/comment consistency, i.e., how well the comments match the described source code segments. A summary of all heuristics is presented in Table 1.

4.1. *Internal (NL Quality) Comment Analysis*

We first describe the set of heuristics targeting the natural language quality of the in-line documentation itself.

4.1.1. *Token, Noun and Verb Count Heuristic (TNVC)*

For each Javadoc comment within a Java class, we calculate the number of tokens, noun phrases (NPs) and verb groups (VGs) the comment contains, based on standard noun and verb phrase chunking. The results are later used by additional quality assessments, such as SPW and PWS, in order to detect the writing style of a comment.

Table 1: Summary of Comment Quality Analysis Heuristics

Type	Name	Scope (I/E)	Output
Internal	TNVC	Javadoc Comments	Number of tokens, nouns and verbs
	WPJC	Javadoc Comments	Number of words
	ABB	Javadoc Comments	Number of abbreviations
	FOG	Javadoc Comments	Fog readability index
	FLESCH	Javadoc Comments	Flesch readability index
	KINCAID	Javadoc Comments	Kincaid readability index
	SPW	Javadoc Comments	Explanation on how the second person writing style may be improved
	PWS	Javadoc Comments	Explanation on how the passive writing style may be improved
Code vs. Comment	DIR	Methods + Method Comments	Ratio from 0 to 1 indicating the completeness of a method's documentation
	ANYJ	Javadoc Comments	Ratio from 0 to 1 indicating the completeness of a class' documentation
	SYNC	Methods + Method Comments	Boolean (True/False), indicating the sync between the different parts of a method, and its documentation
	ARV	Methods + Method Comments	Boolean (True/False), indicating whether the comment adds value beyond what can be understood using the identifier

4.1.2. Words Per Javadoc Comment Heuristic (WPJC)

Originally proposed by [5], the WPJC metric calculates the average number of words found in Javadoc comments. After applying NLP preprocessing 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. As an average, this number is then divided by the total number of Javadoc comments within a class.

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

4.1.3. Abbreviation Count Heuristic (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. 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.

4.1.4. Readability Heuristics (FOG/FLESCH/KINCAID)

In the early twentieth century, linguists conducted a number of studies where people were asked to rank the readability of text [5]. A number of formulas were implemented that analyse the readability of text [14], for example:

The Fog Index: Developed by Robert Gunning, it indicates the number of years of formal education a reader would need to understand a block of text. It is defined as:

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

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

Flesch Reading Ease Level: Rates text on a 100-point scale. The higher the score, the easier a text is to read. An optimal score would range from 60 to 70.[4] It is defined as:

$$\text{Flesch} = 206.835 - (1.015 \times \text{ASL}) - (84.6 \times \text{ASW})$$

[4]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 [14].

Where:
ASL = Average sentence length using number of words.
ASW = Average number of syllables per word.

Flesch-Kincaid Grade Level Score: Translates the 100 point scale of the Flesch Reading Ease Level metric to a U.S. grade school level.

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

Where:
ASL = Average sentence length using number of words.
ASW = Average number of syllables per word.

These readability formulas are also used by a number of U.S. government agencies, such as the DoD (Department of Defence) and IRS (Internal Revenue Service), to analyse the readability of their documents [5].

4.1.5. Second Person Writing Style Heuristic (SPW)

When writing Javadoc comments, a third person declarative rather than second person prescriptive writing style should be used [7]. By analysing the part-of-speech found in Javadoc comment sentences, the SPW metric identifies comments that use a prescriptive writing style. 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. When detecting such cases, the JavadocMiner uses the verb group within the n-gram being processed to generate recommendations on how the comment may be improved. This is achieved by converting the verb to its plural form and replacing it with the original verb within the processed n-gram.

4.1.6. Passive Writing Style Heuristic (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 group 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. For sentences within Javadoc comments that have been detected as being passive, the JavadocMiner then provides explanations to the user on how the Javadoc comment can be be improved by identifying the passive verb group.

4.2. Code/Comment Consistency Analysis

The following heuristics analyse in-line documentation in relation to the source code being documented.

4.2.1. Documentable Item Ratio Heuristic (DIR)

The DIR metric was originally proposed by [5]. For in-line documentation describing a method to be considered complete, it should document all its aspects. For example, for methods that have a return type, contain parameters, or throw exceptions, the "@return", "@parameter" and "@throws" in-line tags must be used. In Figure 7, we show an example of a Javadoc comment for the `parseAssociationEnd` method that is completely documented using the in-line tags.

The result of the DIR metric is the ratio between the parts of a method that should be documented versus the parts that actually were:

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

```
/** 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 7: An Example of a Javadoc Method Comment

4.2.2. Any Javadoc Comment Heuristic (ANYJ)

To compute the ratio of identifiers with Javadoc comments compared to the total number of identifiers, we use the ANYJ metric [5].

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

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 due to insufficient documentation, which would lead to a failure in the program (i.e., a bug).

4.2.3. SYNC Heuristics (RSYNC/PSYNC/ESYNC)

The following heuristics detect methods that are documenting return types, parameters and thrown exceptions that are no longer valid – for example, due to changes in the code not being reflected in the documentation.

RSYNC. The return type of a method is documented using the "@return" in-line tag. The in-line tag must begin with the correct name of the type being returned, followed by the doc comment explaining the return type. To identify methods with return types that contain outdated documentation, we perform a string comparison between the value of the return string indicated in the comment and the actual return type of the method.

PSYNC. When documenting the parameter list of a method, the "@param" in-line tag should begin with the correct name of the parameter being documented, followed by the doc comment explaining the parameter. The JavadocMiner identifies method parameters that contain outdated documentation by performing 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. Documenting exceptions thrown by a method is done using the "@throws" or "@exception" in-line tags. The documentation must begin with the correct names of the exceptions being returned, followed by the doc comment explaining the exception itself. Identifying methods that throw exceptions that have no or out of date documentation is achieved with a string comparison between the value of the exception string indicated in the comment and the actual exception type thrown by the method.

```
/**
* 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 8: An Example of a Partially Documented Method

As an example, consider the code shown in Figure 8: Based on the SYNC heuristics, the JavadocMiner would assign a value of "FALSE" to RSYNC, PSYNC, and ESYNC because: *(i)* the parameter comment begins with the value "object", and not "objectToAccept", *(ii)* the return comment begins with the possible value being returned ("true"), and not the return type itself, and finally *(iii)* the method throws an exception that is not documented.

4.2.4. Added Readability Value Heuristic (ARV)

The ARV metric detects documentation that adds no value beyond what can be understood using the API name by conducting an n-gram comparison between the identifier (e.g., "getsTheLabel"), and the docComment (e.g., "Gets the label"). After *(i)* splitting the identifier name using regular expressions designed to process the Java naming convention [15] for a `Class`, `Method`, etc., and *(ii)* 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. For the example in Figure 9, the JavadocMiner would assign a

```
/** Get the label .
* @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 getTheLabel(Object role , String text) throws ParseException
```

Figure 9: An Example of a Javadoc Comment that adds little or no Value

value of false for the ARV heuristic, meaning the method comment does not add any readability value.

The heuristics described in this section are summarized in Table 1, which also shows which parts of the Javadoc information is extracted and what each heuristic returns as result of the analysis.

4.3. An Ontology for Source Code Comments

We now discuss how to represent the results from the heuristics described above when running them on a concrete software system. To capture the rich semantics between the code and its resulting metrics, we export both into an ontology that can subsequently be queried or linked with other knowledge sources [3].

Ontology models allow for a formal conceptualization of terms and relations that define the vocabulary of a given domain [16]. Ontologies have become a major tool for developing semantically rich applications. Ontology models are capable of representing a large amount of information using a small number of axioms (individuals and relationships) [17]. In terms of knowledge representation, ontologies provide a non-proprietary common language with open world assumption capabilities.

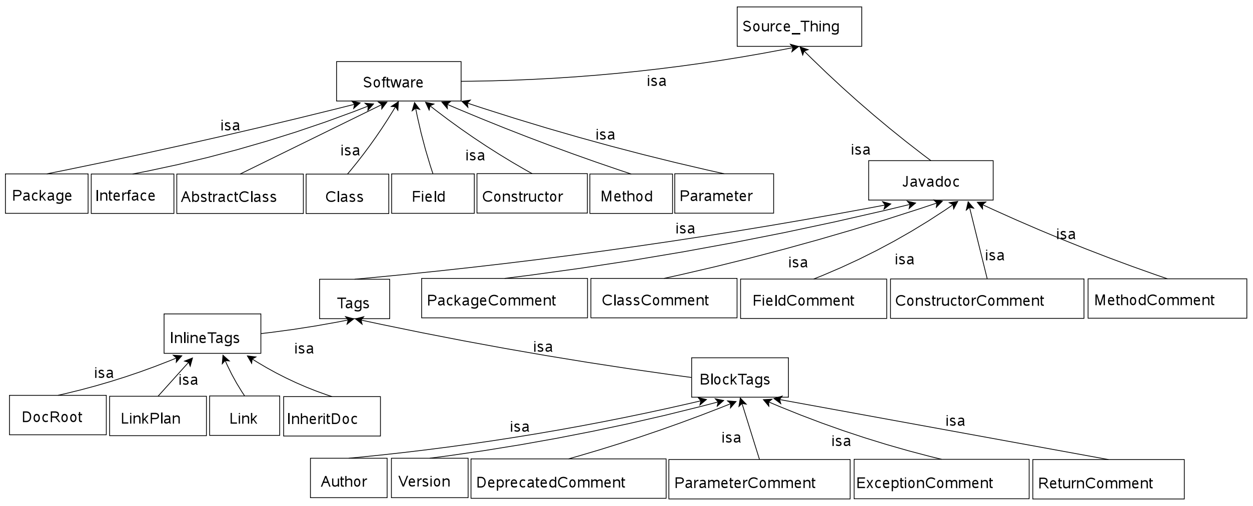


Figure 10: The Javadoc Ontology

Existing applications of ontologies in software engineering have typically focused on defining a domain of discourse and their relations. For example, [18] uses ontologies to model the collaborative nature of software engineering. An

OWL model also enables users to take advantage of visualizations, SPARQL queries, and reasoning services provided by a Description Logic reasoner such as Racer [19], Pellet [20], or FaCT++ [21].

4.3.1. Ontology Design

A large number of source code and in-line documentation related concepts and relationships exist within Javadoc generated documentation. Our Javadoc ontology models both source code related concepts, such as *Class*, *Field*, *Method* and *Exception*, and in-line documentation related concepts, such as *MethodComment*, *Author* and *Version*. Figure 10 shows the taxonomic representation of the major concepts modeled in our Javadoc ontology.

Table 2: Concepts and their Relations in the Javadoc Ontology

Object Property	**Domain**	**Range**	**Description**
belongsToPackage	Abstract Class ⊔ Class ⊔ Interface	Package	The Package a given Interface or Class belongs to
hasConstructor	Abstract Class ⊔ Class ⊔ Interface	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

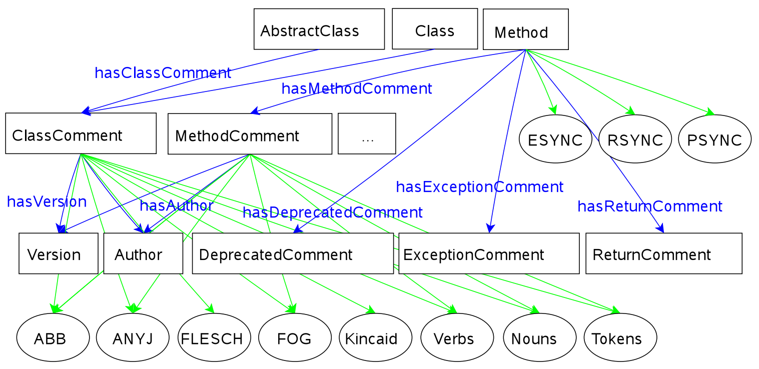


Figure 11: Relationships found in Javadoc Comments

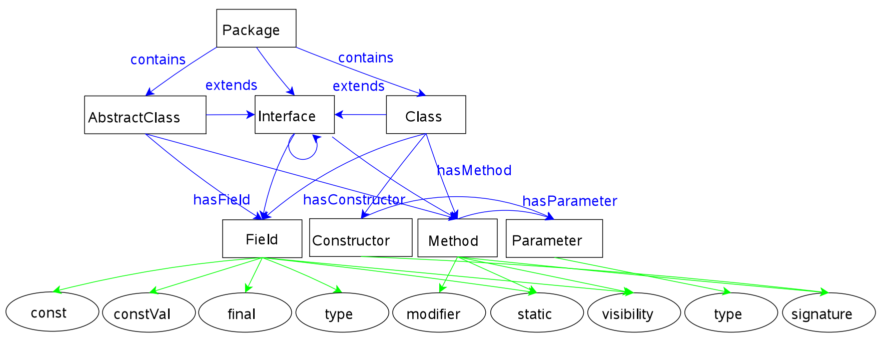


Figure 12: Relationships found in Source Code

The various relationships found in an auto-generated Javadoc are represented in our ontology using object properties [22]; a selected number of these are defined in Table 2. In Figure 11, we show some of the relationships that exist between the different source code and in-line documentation concepts and in Figure 12 some of the relationships that exist within the source code itself.

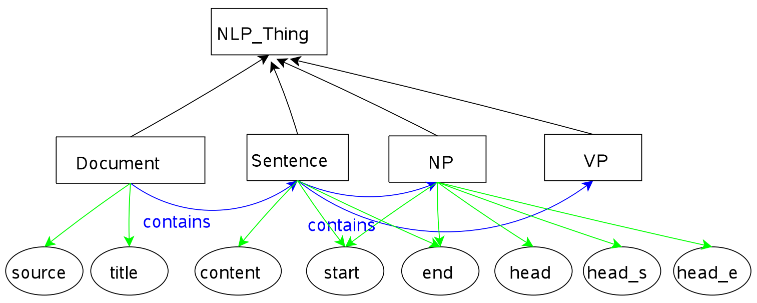


Figure 13: The NLP Ontology

The domain-specific Javadoc ontology is complemented by a domain-independent NLP ontology, which models commonly used concepts in language engineering such as *Document*, *Sentence*, *NP*, and *VP*, as shown in Figure 13. Table 3 lists the major relationships found in the NLP ontology.

Not included in Tables 2 and 3 are the inverse properties, such as “isAuthorOf” or “isSentenceOf”, which a reasoner uses to create additional inferences, providing additional ways of querying the ontology.

Table 3: 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 link their instances together. For example, an “appearsIn” relationship can be used to link the segments or sentences of a document with the comment they appear in. These links provide for further queries on the results of the JavadocMiner.

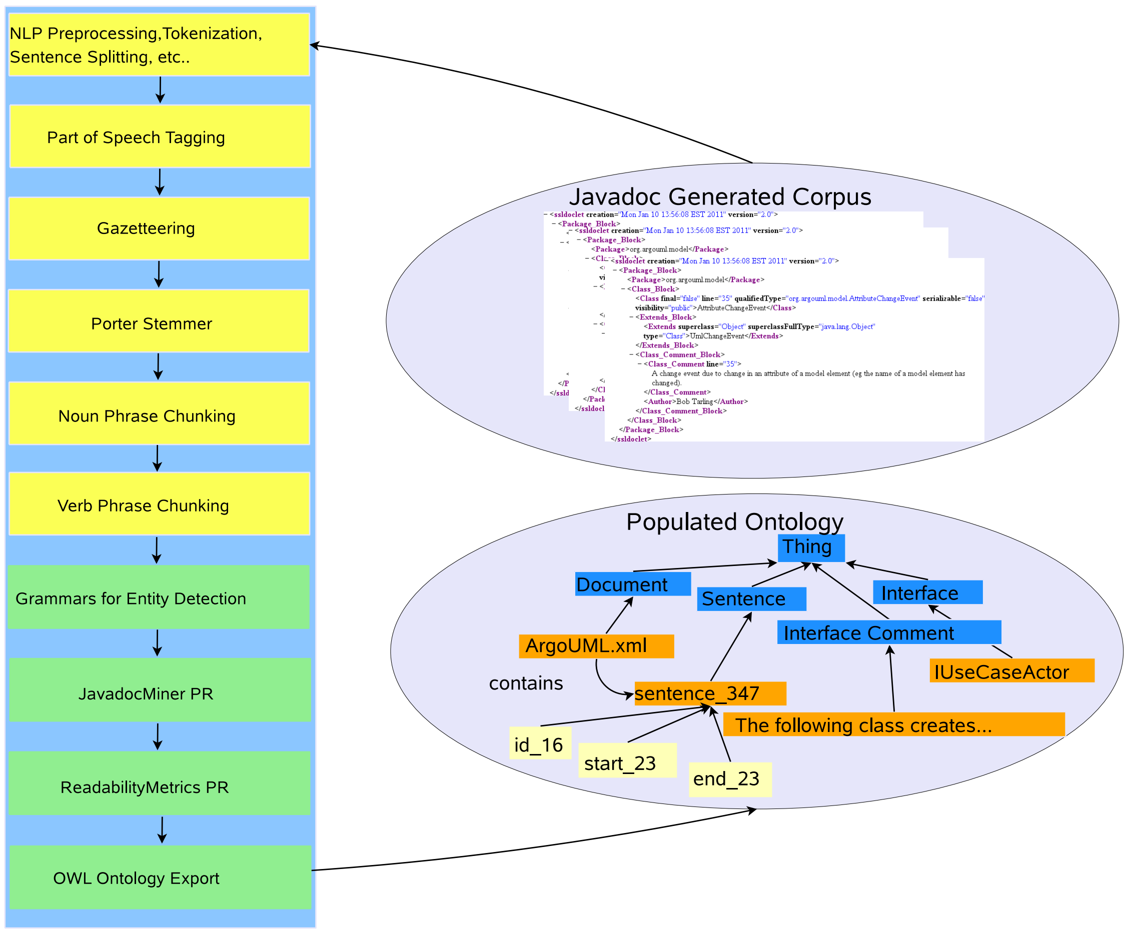
4.3.2. Ontology Population

The ontology created thus far contains only concepts. Before it can be of help to a software engineer, it needs to be instantiated to obtain a *knowledge base*. The large number of artefacts involved in this domain makes it infeasible to create instances manually. Hence, we need to automatically *populate* [23] the ontology based on the results of the NLP analysis.

In general, designing an ontology’s taxonomy (T-Box), and populating it using concept and property assertions (A-Box) [22] is a complicated and time consuming task that requires the expertise of an ontology engineer. Our JavadocMiner makes use of a GATE component called the OwlExporter [17] that provides an automated, portable, and simplified means of populating an ontology using the knowledge found in a text. The entities and relationships that are created by our JavadocMiner NLP application are exported to the Javadoc and NLP ontologies as OWL instances and relationships, as shown in Figure 14.

5. Implementation

In this section, we discuss how we implemented the quality analysis heuristics presented in the previous section (cf. Table 1). We begin by describing the system architecture as a whole and then focus on the different components that make up the JavadocMiner NLP service. We conclude the section with an explanation of how we integrated our NLP pipeline with the software IDE *Eclipse*, which allows a programmer to access the results of an NLP analysis directly during software development.



```
- <ssldoclet creation="Mon Jan 10 13:56:08 EST 2011" version="2.0">
  - <Package_Block>
      <Package>org.argouml.model</Package>
    - <Class_Block>
        <Class final="false" line="35" qualifiedType="org.argouml.model.AttributeChangeEvent" serializable="false"
        visibility="public">AttributeChangeEvent</Class>
      - <Extends_Block>
          <Extends superclass="Object" superclassFullType="java.lang.Object"
          type="Class">UmlChangeEvent</Extends>
        </Extends_Block>
      - <Class_Comment_Block>
        - <Class_Comment line="35">
            A change event due to change in an attribute of a model element (eg the name of a model element has
            changed).
          </Class_Comment>
          <Author>Bob Tarling</Author>
        </Class_Comment_Block>
      </Class_Block>
    </Package_Block>
</ssldoclet>
```

Figure 14: The JavadocMiner Pipeline for analysing the Quality of Source Code Comments

5.1. *Overall System Architecture*

NLP application pipelines are assembled using levels of linguistic analysis, with each level adding information to the document. An NLP application will typically *(i)* reuse resources that perform common linguistic tasks and *(ii)* require specialized resources specific to an application domain. The Java-based *General Architecture for Text Engineering* (GATE) [24] provides a framework for creating language processing software. As part of GATE, a development environment is also distributed. This development environment is built on top of the GATE Embedded framework and includes a graphical interface, GATE Developer, 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 components can be found in [24].

In GATE, analysis pipelines are built by assembling individual components, called *processing resources* (PRs), which are then sequentially executed on the documents in a corpus. Results from one PR are stored as *annotations* on the document, which can be read by subsequent components.

5.2. *The JavadocMiner NLP Application*

Before the NLP pipeline can process some source code, it is first converted by the SSLDoclet into an XML representation as discussed in Section 3. The resulting output documents (an example is shown in Figure 15) are then fed into the JavadocMiner pipeline.

Figure 14 shows the individual components that make up our JavadocMiner pipeline. In yellow are the general NLP processing resources provided by GATE [24] and in green are the PRs implemented by us.

```
<Methods>
<Method_Block>
<Method abstract="false" final="false" interface ="false" line ="55" modifier="public"
signature ="(java.lang.Object)" static ="false" synchronized="false" synthetic ="false"
 visibility  ="public">doesAccept</Method>
<Method_Comment_Block>
<Method_Comment line="55">A sequence diagram can accept classifiers. It  will  add them as a ne
  Classifier  Role with that  classifier  as a base. All  other accepted  figs
 are added as is.</Method_Comment>
<Auithor_Comment>agauthie@ics.uci.edu</Auithor_Comment>
<See_Comment>org.argouml.uml.diagram.ui.UMLDiagram#doesAccept(java.lang.Object)  *</See_C
</Method_Comment_Block>
<Return_Block>
<Return>void</Return>
<Return_Comment>true if the diagram can accept the object,  else  false </Return_Comment>
</Return_Block>
<Parameter_Block>
<Parameter fullType="java.lang.Object" type="Object">objectToAccept</Parameter>
</Parameter_Block>
</Method_Block>
</Methods>
```

Figure 15: The ArgoUML Method doesAccept represented using the SSLDoclet

5.2.1. *Preprocessing Stage*

Using components already provided by GATE, we perform general NLP pre-processing tasks, including tokenization, sentence splitting, and part-of-speech (POS) tagging [24].

5.2.2. *JavadocMiner PR*

We implemented a new GATE component called the *JavadocMiner PR*, which is responsible for performing the code/comment consistency analysis on Javadoc comments, such as *SYNC* and *SPW*.

```
for(Annotation I : this.getInputAS().get("Method_Block").getContained(
                t.getStartNode().getOffset(),
                t.getEndNode().getOffset())) {
  if(this.getInputAS().get("Parameter").getContained(
                  t.getStartNode().getOffset(),
                  t.getEndNode().getOffset()).size() !=
          this.getInputAS().get("Parameter_Comment").getContained(
                                  t.getStartNode().getOffset(),
                                  t.getEndNode().getOffset()).size())
          mFeature.put("PARAMSYNC", "FALSE");
          mFeature.put("PARAMSYNC_Explanation",
              "The parameter comments and parameters are out of sync,
              "please make sure that each parameter comment begins wit
              "the parameter name.");
          sync = false;
  }
  else{mFeature.put("PARAMSYNC", "TRUE");}
```

Figure 16: A Code Snippet of the SYNC Heuristic

In Figure 16, we show part of the Java implementation of the *SYNC* heuristic. The PR is designed to process the XML schema generated using the SSL Doclet. The heuristic starts by examining entities from the “Method_Block” annotation. It then determines if each “Parameter_Comment” is associated with a specific “Parameter”. If all parameters have been correctly documented, the JavadocMiner assigns a value of “TRUE” to “PARAMSYNC”; otherwise, “PARAMSYNC” is set to “FALSE” and an explanation is generated, which can be used to inform the user regarding the code/comment inconsistency. In a similar fashion, the SYNC heuristic determines the code/comment consistency of return comments and exceptions.

The JavadocMiner PR also includes the implementation needed to detect comments that add little or no value over what a reader can gather from looking at the identifier itself, as defined by the *ARV* metric, and computes the ratio between documentable method items vs. the items that have actually been documented (*DIR* metric).

5.2.3. *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 readability index of a document needs to be measured. The ReadabilityMetrics PR makes use of an existing library[5] to calculate the readability index. The ReadabilityMetrics PR also contains the implementation in charge of analyzing passive vs. active writing style as provided by the PWS heuristic.

```java
for (Annotation s : sentAS) {
    FeatureMap sFeature = Factory.newFeatureMap();
    AnnotationSet vgAS = new AnnotationSetImpl(
                    this.getInputAS().getContained(s.getStartNode().getOffset(),
                                   s.getEndNode().getOffset()).get(this.VG));

    for (Annotation v : vgAS) {
            if (v.getFeatures().get("voice").toString().compareToIgnoreCase("passive")==0) {
                    sFeature.put("WritingStyle", v.getFeatures().get("voice").toString());
                    sFeature.put("PassiveVG", this.getDocContent().getContent(
                                   v.getStartNode().getOffset(), v.getEndNode().getOffset())
                    sFeature.put("PassiveExplanation", "The " + s.getType().toLowerCase() +
                                   " has been detected as passive, and can be improved by "
                                   "changing the verb phrase " +
                                   this.getDocContent().getContent(
                                   v.getStartNode().getOffset(), v.getEndNode().getOffset())

                    this.createReadabilityAS(i, sFeature);
            }
    }
}
```

Figure 17: A Code Snippet of the PWS Heuristic

In Figure 17, we show a Java code snippet taken from the PWS module. The heuristic makes use of annotations created by previous processing resources such as the sentence splitter and verb group chunker. In case a passive writing style was detected, a suggestion for improvement is generated.

5.2.4. *OwlExporter PR*

The OwlExporter [17] is the final step in our pipeline. It is in charge of taking the annotations produced by the JavadocMiner and exporting it to the result ontology. The core idea of our OwlExporter is to take the annotations generated by an NLP pipeline and provide for a simple means of establishing a mapping between NLP and domain annotations on one hand and the *concepts* and *relations* of an existing NLP and domain-specific ontology on the other hand.

Using a few simple grammars written in the JAPE language [24] (see Figure 18 for an example), the OwlExporter is able to translate the annotations of a document as *individuals*, *datatype* and *object properties* of an ontology [22]. The resulting, populated ontology can then be used within any ontology-enabled tool for further querying, reasoning, visualization, or other processing. The relationships that exist between each of the mentioned entities are also represented using the OWL models. For example, a class name, its documentation, and the author's name are all exported to the Javadoc and NLP ontologies and linked. Figure 19 shows an excerpt of a populated Javadoc ontology.

5.3. *End-user Interfaces*

As discussed earlier, we also need to bring the results of the quality analysis to the software engineers, which are generally not familiar with text mining frameworks such as GATE. Towards this end, our implementation *(i)* integrates the NLP analysis pipeline into the Eclipse IDE for interactive use and *(ii)* facilitates the populated ontology for large-scale querying, reasoning, and further cross-linking with other software artifacts [3].

5.3.1. *The Semantic Assistants Eclipse Plug-in*

Eclipse is a multi-language software development environment, comprising an IDE and an extensible plug-in system. We implemented our own Eclipse plug-in called the Semantic Assistants Eclipse Plug-in [25] that acts as a client sending Web service requests to a Semantic Assistants [26] server (Figure 20).

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

```
Phase:  Javadoc_OWLExportClass
Input:  SWDomainClass SWNLPClass
Options:  control  =  all

Rule:   owlexportclass_rule

(
          {SWDomainClass} |
          {SWNLPClass}
)
: sw
-->
{
          AnnotationSet as  = (gate.AnnotationSet) bindings.get("sw");
          Annotation a  = (gate.Annotation)as. iterator ().next();

          a.getFeatures().put("kind",  "Class");
          a.getFeatures().put("corefChain",  null );

          if (a.getType().compareToIgnoreCase("SWDomainClass")==0) {
                    outputAS.add(as.firstNode (),  as.lastNode(),
                                         "OwlExportClassDomain",  a.getFeatures());
          }
          else
                    outputAS.add(as.firstNode (),  as.lastNode(),
                                         "OwlExportClassNLP",  a.getFeatures());
}
```

Figure 18: An Example JAPE Grammar Used to Export Domain and NLP Instances to the Ontology

The Semantic Assistants architecture is in charge of *(i)* invoking the different NLP services requested by the client, *(ii)* transporting the documents that need to be analysed to the service, and finally *(iii)* returning the results of the NLP service back to the client in the form of annotations.

The Eclipse client then maps the results returned by the analysis to the original documents, and displays the information to the user. In Figure 21, we show how recommendations made by our JavadocMiner on how a comment may be improved is displayed by the Semantic Assistants Eclipse plug-in. A software developer can now inspect these

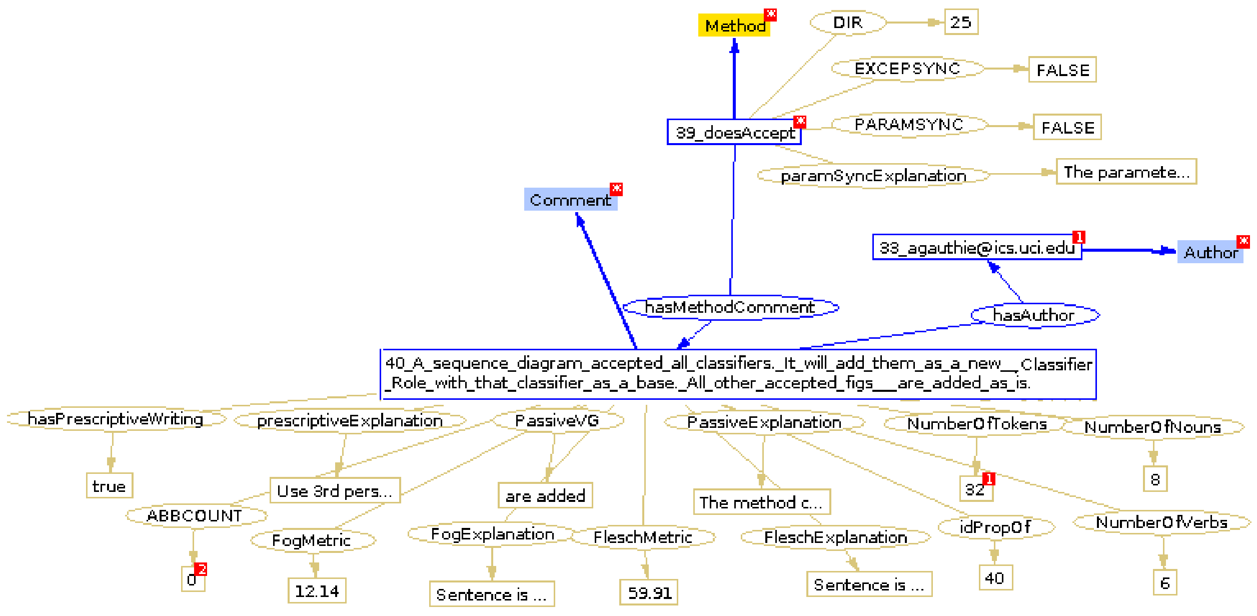


Figure 19: An Excerpt from a populated Javadoc Ontology

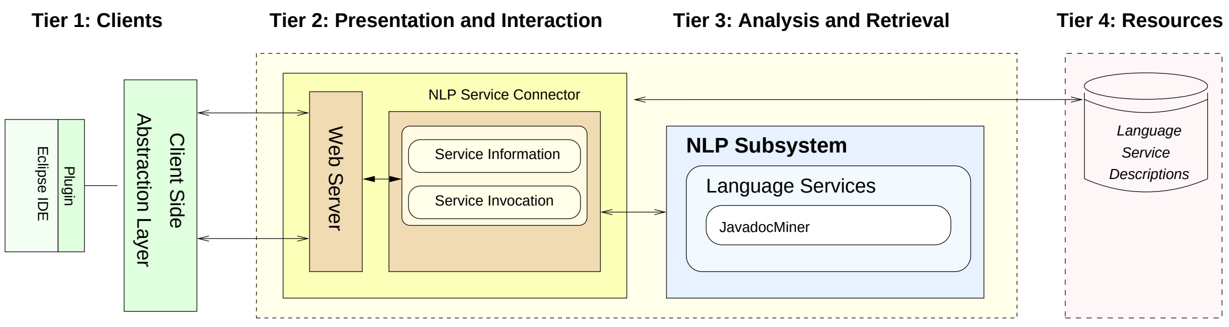
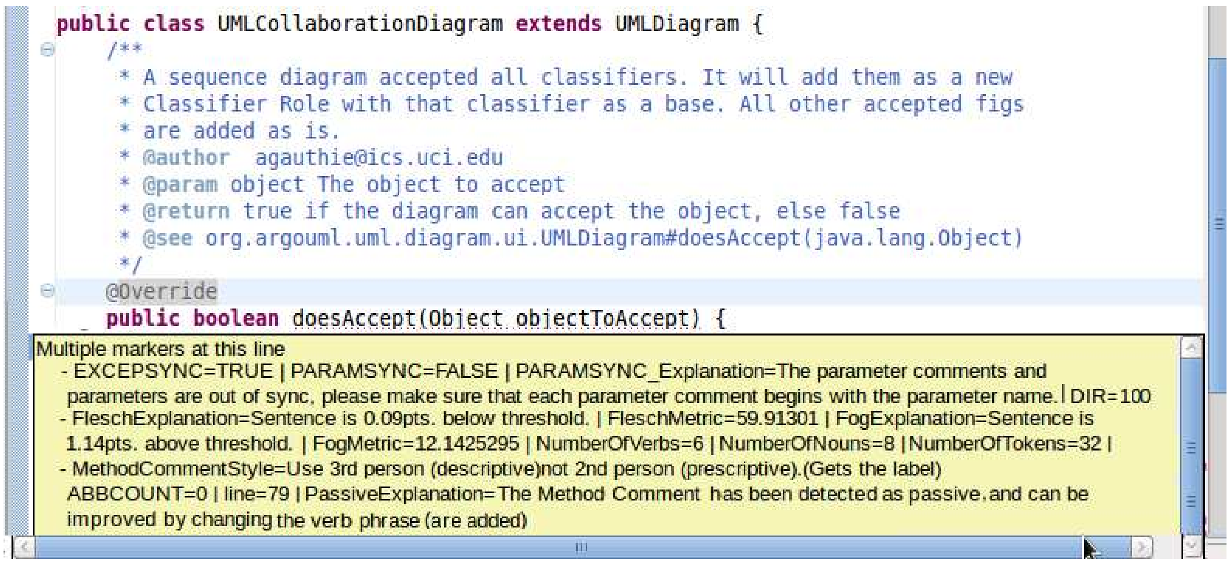


Figure 20: The JavadocMiner pipeline running within the Semantic Assistants Framework

quality analysis results directly within his code editor, in exactly the same way as he works with source code errors and warnings.



```
public class UMLCollaborationDiagram extends UMLDiagram {
    /**
     * A sequence diagram accepted 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.
     * @author  agauthie@ics.uci.edu
     * @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) {
```

```
Multiple markers at this line
  - EXCEPSYNC=TRUE | PARAMSYNC=FALSE | PARAMSYNC_Explanation=The parameter comments and
   parameters are out of sync, please make sure that each parameter comment begins with the parameter name.| DIR=100
  - FleschExplanation=Sentence is 0.09pts. below threshold. | FleschMetric=59.91301 | FogExplanation=Sentence is
   1.14pts. above threshold. | FogMetric=12.1425295 | NumberOfVerbs=6 | NumberOfNouns=8 |NumberOfTokens=32 |
  - MethodCommentStyle=Use 3rd person (descriptive)not 2nd person (prescriptive).(Gets the label)
   ABBCOUNT=0 | line=79 | PassiveExplanation=The Method Comment has been detected as passive.and can be
   improved by changing the verb phrase (are added)
```

Figure 21: The Semantic Assistants Eclipse Plug-in

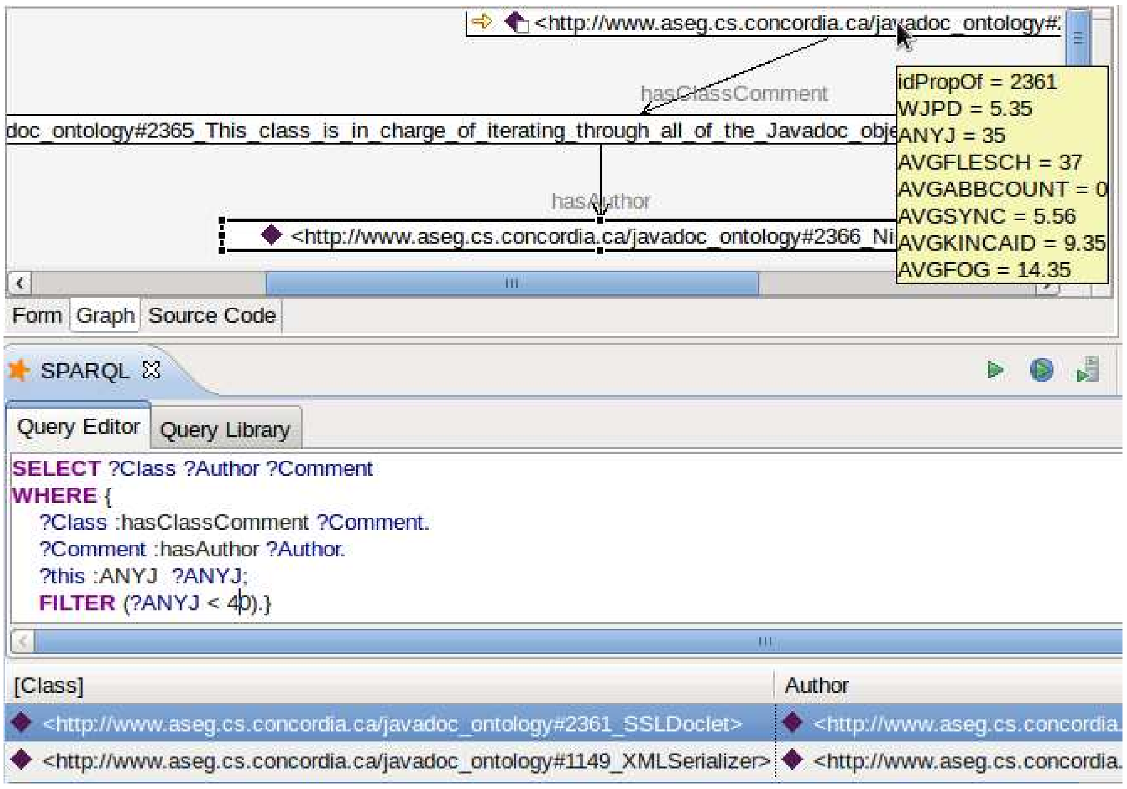
5.3.2. Ontology Application

Exporting the results created by our JavadocMiner application to an ontology enables users to *query* 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 low quality scores. In Figure 22, we show the results of a SPARQL query that returned classes that contained insufficient documentation as detected by the ANYJ metric. Also returned by the query are the authors that created the Javadoc documentation for the class.

Our approach of representing the source code analysis results in a standardized ontology format also facilitates further linking with other software knowledge repositories, including versioning systems and bug databases [3].

6. Evaluation

In this section, we discuss how the JavadocMiner was applied on two open source projects for an analysis of comment quality and their consistency with source code; we then evaluate the results of our study by examining how



<http://www.aseg.cs.concordia.ca/javadoc_ontology#
hasClassComment
doc_ontology#2365_This_class_is_in_charge_of_iterating_through_all_of_the_Javadoc_obj
hasAuthor
<http://www.aseg.cs.concordia.ca/javadoc_ontology#2366_Ni
Form
Graph
Source Code
SPARQL
Query Editor
Query Library

```
idPropOf = 2361
WJPD = 5.35
ANYJ = 35
AVGFLESCH = 37
AVGABBCOUNT = 0
AVGSYNC = 5.56
AVGKINCAID = 9.35
AVGFOG = 14.35
```

```
SELECT ?Class ?Author ?Comment
WHERE {
  ?Class :hasClassComment ?Comment.
  ?Comment :hasAuthor ?Author.
  ?this :ANYJ  ?ANYJ;
  FILTER (?ANYJ < 40).}
```

[Class]	Author
<http://www.aseg.cs.concordia.ca/javadoc_ontology#2361_SSLDoclet>	<http://www.aseg.cs.concordia.
<http://www.aseg.cs.concordia.ca/javadoc_ontology#1149_XMLSerializer>	<http://www.aseg.cs.concordia.

Figure 22: Results of a SPARQL Query on the NLP-Populated Source Code Comment Ontology

the quality of comments evolved between the different versions. In the second part of our evaluation, we correlate the results returned from our study with the bug statistics for each open source project, and show how the JavadocMiner can be used to measure the documentation quality of source code.

6.1. *Data*

We conducted a case study where the JavadocMiner was used to assess the quality of in-line documentation found in three major releases of the UML modeling tool ArgoUML[6] and the IDE Eclipse.[7] Three different versions of each project were checked out manually from the project's version control repository. By assessing each module separately, we are able to observe how the quality of documentation evolved over time. In Table 4, we show the versions of the projects that were part of our quality assessment.

Table 4: Assessed Open Source Project Versions, Release Dates, Lines of Code (LOC), Number of Comments, Identifiers and Bugs

Project Version	Release Date	LOC	Number of Comments	Number of Identifiers	Number of Bug Defects
ArgoUML v0.24	02/2007	250,000	6871	13,974	46
ArgoUML v0.26	09/2008	600,000	6875	14,262	54
ArgoUML v0.28.1	08/2009	800,000	7168	14,789	48
Eclipse v3.3.2	06/2007	7,000,000	32,172	158,009	176
Eclipse v3.4.2	06/2008	8,000,000	33,919	163,238	413
Eclipse v3.5.1	06/2009	8,000,000	34,360	165,945	153

[6]ArgoUML, http://www.ohloh.net/p/argouml/analyses/latest
[7]Eclipse, http://www.ohloh.net/p/eclipse/analyses/latest

6.2. *Experiments*

We split the ArgoUML and Eclipse projects into their three major modules, for ArgoUML: Top Level, View & Control, and Low Level; and for Eclipse: Plug-in 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. The complete quality assessment process for both open source projects took less than 3 hours. In the second part of our evaluation, we continued by finding the amount of bug defects that were reported for each version of the modules using the open source project's issue tracker system. The Pearson product-moment 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.

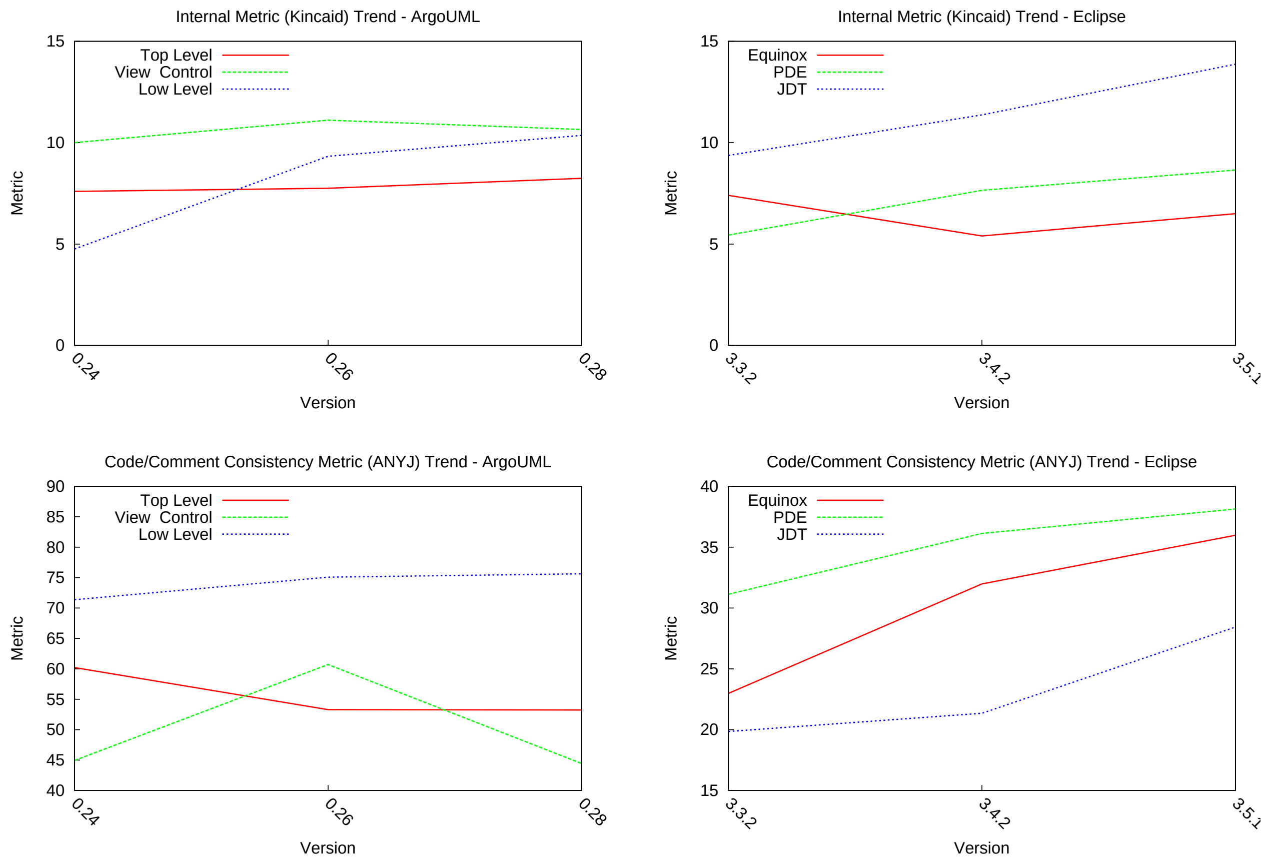


Figure 23: Charts for Code/Comment and Internal (NL Quality) Metrics

6.3. *Results and Analysis*

Results of our study indicated that the modules that performed best in our quality assessment were the Low Level module for ArgoUML (Figure 23, left side) and the PDE module for Eclipse (Figure 23, right side). The figure shows both results from an internal analysis (Kincaid), targeting the quality of the natural language itself (top) and the code/comment consistency as computed by the ANYJ heuristic (bottom). Looking at ArgoUML, the Kincaid measure for the View and Control as well as the Low Level modules remained relatively flat, versus the Low Level Module which continues in an upward direction. This can be interpreted as efforts geared towards improving the documentation of the project's core library. The same applies for the ANYJ measure, which indicated that the Low Level Module as being the most thoroughly documented module.

In terms of the Eclipse open source project, both the PDE and JDT modules showed increasing readability indexes; however, as mentioned before, a higher readability index does not necessarily indicate better readability: Based on the observations made by linguists in the past, an optimal score for the Flesch-Kincaid readability index would range from 8–10 [14]. Readability indexes can also be seen as a measure of text density, using the number of words and syllables as factors. In terms of the readability index as a measure of text density, a higher index does not necessarily mean more complex content, but also that the text contained less number of words with a similar number of syllables. Applied to the context of source code comments, the inline documentation "Gets the ToolTipModule object" would yield a Kincaid reading grade level of 13, where "This method is in charge of getting the tool tip module", would yield a value of 6. Both examples are equally understandable to a developer, with one containing less than half the number of words. Examining the comments found in the JDT module validated this observation, showing shorter comments as reported by WPJC, with increased number of syllables and denser comments on average as indicated by Kincaid. The PDE module showed readability measures between the 8–10 values for two of the three versions and a higher number of WPJC on average, compared to both the JDT and Equinox modules. For these reasons, we were able to conclude the PDE module as having the least dense in-line documentation and thoroughly documented source code as indicated by the ANYJ graph (Figure 23, bottom right).

6.3.1. Quality Analysis

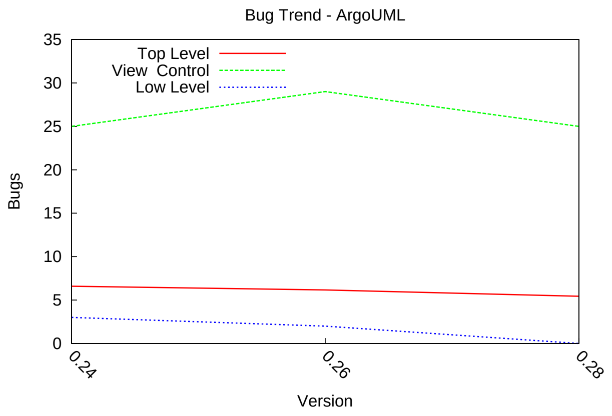
We believe that the reason for the Low Level module (ArgoUML) and the PDE module (Eclipse) outperforming the rest of the modules in every heuristic is that they are both the base libraries that every other module extends. For example, Eclipse is a framework that is extended using plug-ins that use the services provided by the PDE API module.

The readability results returned by the JavadocMiner indicate that the Low Level module contained in-line documentation that is less complicated compared to the View and Control module (Figure 23, top left), yet more complicated than the in-line documentation found in the Top Level module. The PDE module in Eclipse also returned similar Kincaid results compared to the other two Eclipse modules (Figure 23, top right).

In terms of the Code/Comment consistency metrics, the Low Level and PDE modules contained the most up-to date and thoroughly documented source code, compared to the rest of the modules. The Eclipse project is further separated into API and internal non-API packages, and part of the Eclipse policy states that all API packages must be properly documented [5].

6.3.2. Comment-Bug Correlation

As part of our efforts to correlate the results of our study with another software engineering artefact, we examined the amount of bug defects that were reported for each version of the modules for ArgoUML (Figure 24, left) and for Eclipse (Figure 24, right).



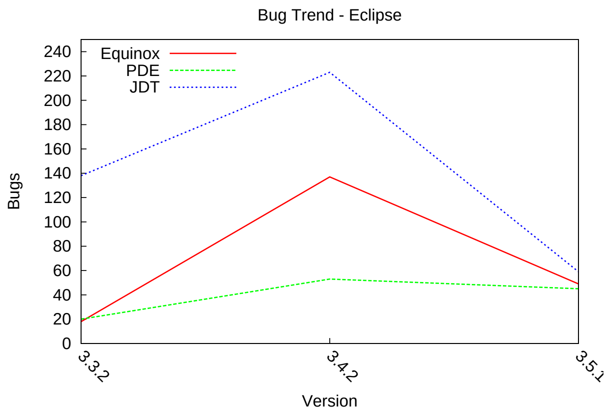


Figure 24: Charts for Code/Comment and Internal (NL Quality) Metrics

We correlated the quality of comments found in source code with bug defects in order to determine if potential problem areas can be identified early by analysing the in-line documentation. What we observed was that the modules that performed best in our quality assessment also had the least amount of reported 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 [27] on the data gathered from the quality assessment and the number of bug defects for both ArgoUML (Table 5), and Eclipse (Table 6). Also included in the bottom of each table are the correlation coefficient figures of the merged modules for each project.

Table 5: Pearson Correlation Coefficient Results for ArgoUML

Module	ANYJ	SYNC	ABB	FLESCH	FOG	KINCAID	TOKENS	WPJC	NOUNS	VERBS
Top Level	0.99	0.96	-0.98	0.32	0.80	0.74	0.89	0.96	0.98	0.89
View&Control	0.91	0.99	-0.87	0.37	0.77	0.81	0.88	0.86	0.91	0.73
Low Level	0.97	0.92	-0.93	0.31	0.82	0.84	0.84	0.82	0.99	0.88
total	0.99	0.98	-0.94	0.32	0.80	0.79	0.89	0.91	0.98	0.87

The correlation and coefficient results showed that the ANYJ, SYNC, ABB, Tokens, WPJC and Nouns heuristics strongly correlated with the number of bug defects, whereas the Flesch readability metric was among the least correlated. Even though Flesch-Kincaid translates the Flesch Reading Ease metric to a U.S. school level grade mathematically, they are two different types of computations and therefore a difference in the degree of correlation between the two metrics is possible.

Table 6: Pearson Correlation Coefficient Results for Eclipse

Module	ANYJ	SYNC	ABB	FLESCH	FOG	KINCAID	TOKENS	WPJC	NOUNS	VERBS
Equinox	0.98	0.90	-0.89	0.40	0.79	0.94	0.91	0.89	0.93	0.75
PDE	0.97	0.85	-0.83	0.34	0.71	0.82	0.81	0.82	0.90	0.71
JDT	0.95	0.81	-0.85	0.39	0.75	0.87	0.79	0.84	0.95	0.65
total	0.97	0.89	-0.86	0.37	0.77	0.84	0.88	0.86	0.91	0.73

In Figure 25 (top), we show the number of bug defects reported at the level of quality assessments returned by the ANYJ and ABB metrics, and in Figure 25 (bottom) for the readability and NLP metrics for ArgoUML. The same data is represented for Eclipse in Figure 26.

The charts illustrate the correlation between each metric and the number of bug defects; with the exception of the Flesch metric, which we previously determined as being the least correlated.

7. Related Work

In the following, we discuss related work, separately for the three major aspects of our work: Source code as input for NLP analysis, quality analysis of source code comments, and consistency analysis between code and its natural language comments.

7.1. Corpus Generation from Source Code

A number of other Doclets exist that can create XML files using Javadoc and information found in source code, such as the `xml-doclet`,[8] Mavens' `XMLDoclet`,[9] or the `jeldoclet`.[10] However, when looking at the schema generated by these Doclets, we observed that the Doclets 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.

A Doclet that generates a schema that closely resembles the SSLDoclet is the `jeldoclet`. The `jeldoclet`, however, does not attempt to differentiate between different types of comments, which reduces the descriptiveness of

[8]XML-Doclet, http://code.google.com/p/xml-doclet/
[9]Maven Doclet, http://maven.apache.org/maven-1.x/
[10]jeldoclet, http://jeldoclet.sourceforge.net/

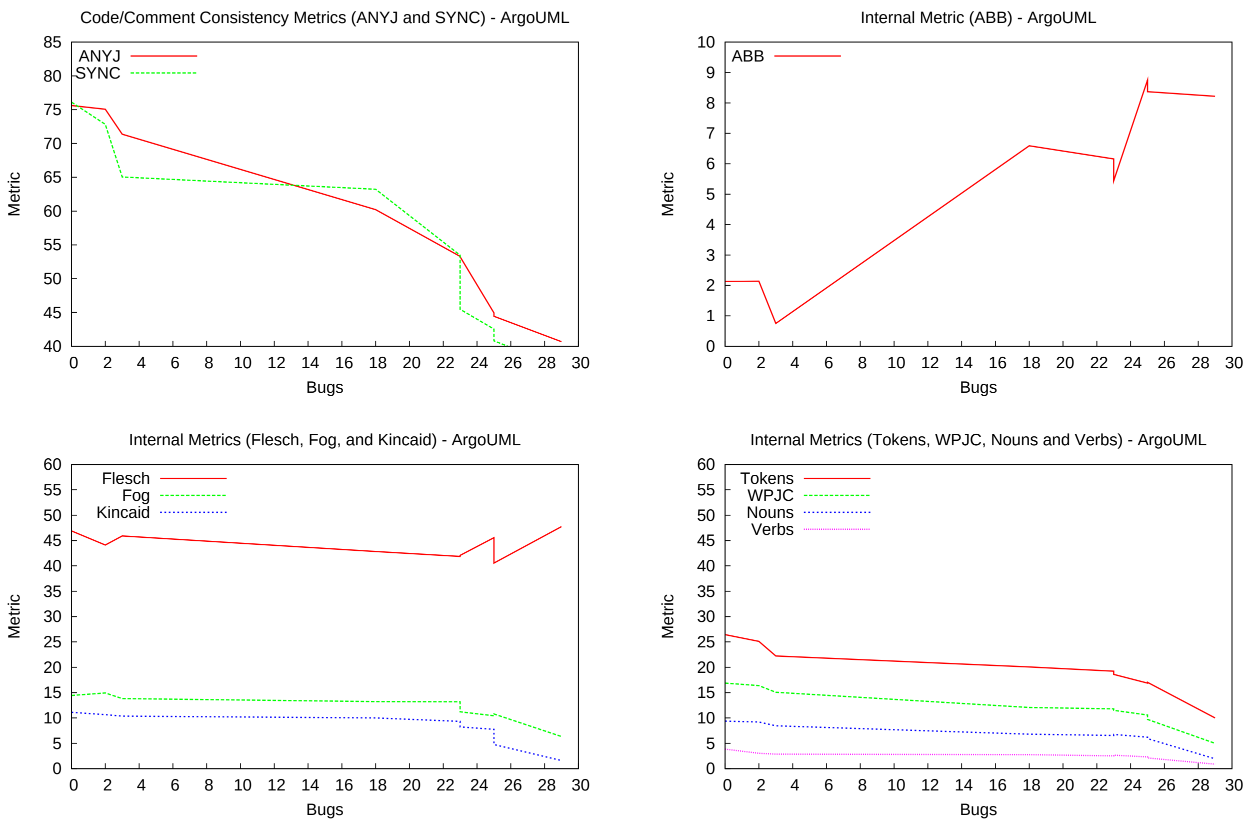


Figure 25: Code/Comment Consistency and NL Quality Metrics vs. Bugs – ArgoUML

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 4.

The source data being represented and the output format is the same for all XML generating Doclets; and the XML documents generated using these Doclets can be loaded within any NLP framework. But the information mark-up approach will not only drastically affect the number of annotations, features and entities that will be created but as a consequence also the NLP resources required for processing that information: Having the most number of annotations, features or entities as 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 an annotation, a feature, and an entity is important when generating an XML document that 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 number of annotations would exceed that of the SSLDoclet for the same amount of information. This will actually have a negative impact on the amount of work needed by the language engineers to make use of the generated corpus.

To conclude, even though there exist a number of XML generating Doclets that can be downloaded from the net, we feel that our SSLDoclet differs from the rest due to its ability to generate XML output using a schema that is optimized for further NLP processing, which is an application scenario not targeted by existing efforts.

7.2. Quality Analysis of In-Line Documentation

There has been effort in the past that focused on analysing source code comments, for example in [28] 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

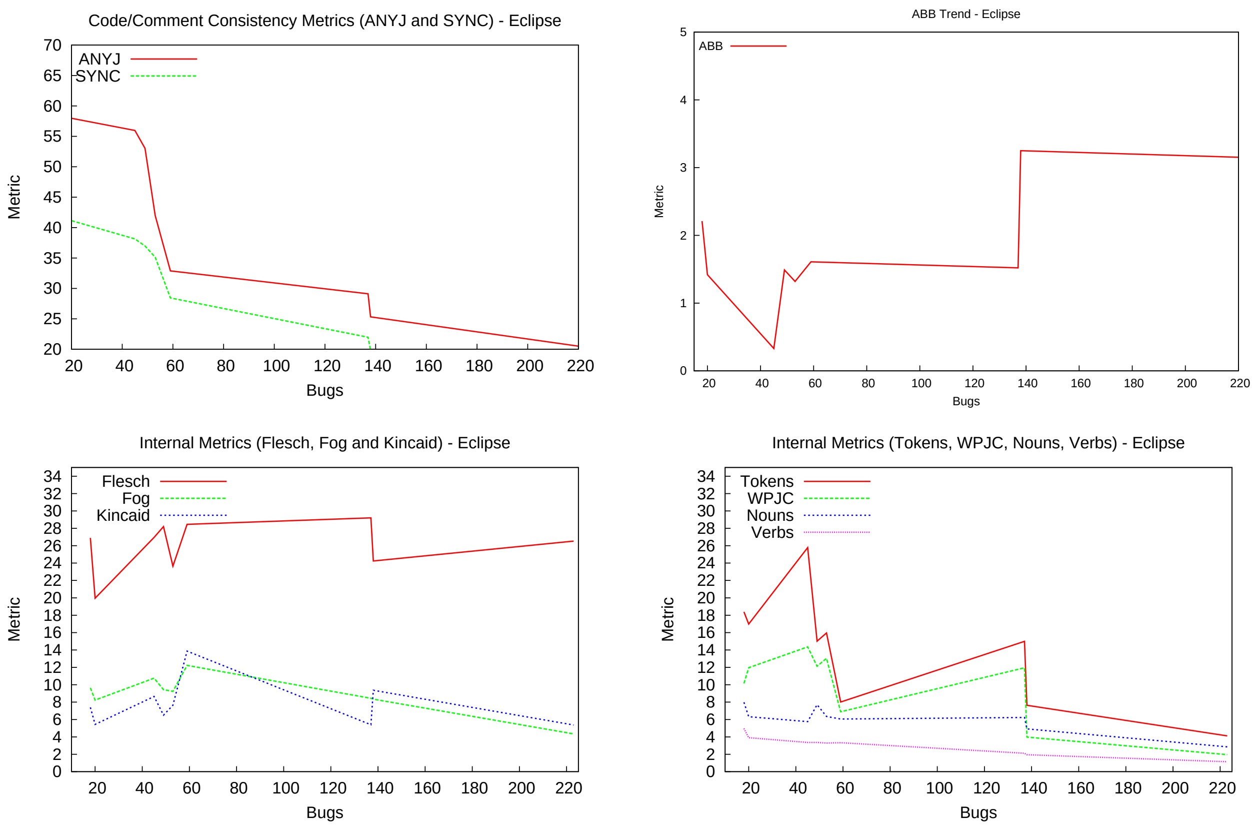


Figure 26: Code/Comment Consistency and NL Quality Metrics vs. Bugs – Eclipse

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 [2] 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, or 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 analysed 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 their study was to classify, rather than assess, the different types of in-line documentation found in software artifacts. The authors also made no attempt to automate the process, nor was there any major correlations made with other software engineering artefacts.

The only work we are aware of that focused on automatically analysing API documentation generated by Javadoc is [5]. The authors implemented a tool called Quasoledo that measures the quality of documentation with respect to its completeness, quantity and readability. Here, we extended the works of [5] by introducing new quality assessment metrics. We also analysed 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 was done in our study.

7.3. Code/Comment Consistency Analysis

Automatically analysing comments written in natural language to detect code-comment inconsistencies was the focus of [29]. The authors explain that such inconsistencies may be viewed as an indication of either bugs or bad comments. The authors implemented 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 four large open source software projects: Linux, Mozilla, Wine and Apache, and 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 [30] was the focus of [31] and [32]. A "lexical analyzer" used to estimate the length of software was implemented in [31]. 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. Results from the study include that the most frequently occurring keyword used in *Jakarta Tomcat* is *Public*. The authors also argued that because *String* was the most frequent identifier for Jena, it could act as an indication for needed optimizations. The authors also found that the identifiers commonly used by 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 [33] 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.

None of the work mentioned in this section attempted to analyse the quality of Javadoc comments used to generate API documentation. Source code comments that describe a given implementation within a method body mainly come in contact with the developers and maintainers of a software project. However, Javadoc API documentation also comes in contact with other stakeholders, such as managers and conformance testers.

8. Conclusions and Future Work

In this paper, we discussed the challenges facing the software engineering domain when attempting to manage the large amount of documentation written in natural language. We presented an approach that automatically assesses the quality of documentation found in software comments using our JavadocMiner system. We also showed how the JavadocMiner can be used to identify the 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.

Acknowledgments.

This research was partially funded by DRDC Valcartier (Contract No. W7701-081745/001/QCV). The authors would like to thank Bahar Sateli for implementing the Semantic Assistants Eclipse plug-in.

References

[1] E. Nurvitadhi, E. Nurvitadhi, C. Cook, Do class comments aid Java program understanding?, in: 33rd Annual Frontiers in Education (FIE'03), IEEE Computer Society, 2003, pp. T3C13–17.

[2] Y. Padioleau, L. Tan, Y. Zhou, Listening to programmers Taxonomies and characteristics of comments in operating system code, in: International Conference on Software Engineering (ICSE '09), IEEE Computer Society, Washington, DC, USA, 2009, pp. 331–341.

[3] J. Rilling, R. Witte, P. Schuegerl, P. Charland, Beyond Information Silos – An Omnipresent Approach to Software Evolution, International Journal of Semantic Computing (IJSC) 2 (2008) 431–468. Special Issue on Ambient Semantic Computing.

[4] M. M. Lehman, L. A. Belady (Eds.), Program evolution: processes of software change, Academic Press Professional, Inc., San Diego, CA, USA, 1985.
[5] D. Schreck, V. Dallmeier, T. Zimmermann, How documentation evolves over time, in: IWPSE '07: Ninth international workshop on Principles of software evolution, ACM, New York, NY, USA, 2007, pp. 4–10.
[6] D. E. Knuth, Literate Programming, The Computer Journal 27 (1984) 97–111.
[7] D. 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, ACM, New York, NY, USA, 1999, pp. 147–153.
[8] R. E. Brooks, Towards a Theory of the Comprehension of Computer Programs., International Journal of Man-Machine Studies 18 (1983) 543–554.
[9] G. Antoniou, F. van Harmelen, A Semantic Web Primer, The MIT Press, 2nd edition, 2008.
[10] C. F. Goldfarb, P. Prescod, The XML Handbook, Prentice-Hall, Upper Saddle River, New Jersey, 5th edition, 2004.
[11] E. T. Ray, Learning XML, O'Reilly & Associates, Sebastopol, California, 2nd edition, 2003.
[12] N. Khamis, R. Witte, J. Rilling, Generating an NLP Corpus from Java Source Code: The SSL Javadoc Doclet, in: New Challenges for NLP Frameworks, ELRA, Valletta, Malta, 2010.
[13] P. Bunyakiati, A. Finkelstein, The Compliance Testing of Software Tools with Respect to the UML Standards Specification – The ArgoUML Case Study, in: D. Dranidis, S. P. Masticola, P. A. Strooper (Eds.), AST, IEEE, 2009, pp. 138–143.
[14] W. H. DuBay, The Principles of Readability, Impact Information, 2004.
[15] H. M. Deitel, P. J. Deitel, Java, Prentice Hall, 7th edition, 2007.
[16] T. R. Gruber, Toward principles for the design of ontologies used for knowledge sharing, International Journal of Human-Computer Studies 43 (1995) 907–928.
[17] R. Witte, N. Khamis, J. Rilling, Flexible Ontology Population from Text: The OwlExporter, in: The Seventh International Conference on Language Resources and Evaluation (LREC 2010), ELRA, Valletta, Malta, 2010, pp. 3845–3850.
[18] P. Wongthongtham, E. Chang, T. Dillon, I. Sommerville, Development of a software engineering ontology for multisite software development, IEEE Transactions on Knowledge and Data Engineering 21 (2009) 1205–1217.
[19] V. Haarslev, R. Möller, RACER System Description, in: R. Goré, A. Leitsch, T. Nipkow (Eds.), Automated Reasoning: First International Joint Conference (IJCAR) 2001, volume 2083 of *Lecture Notes in Computer Science*, Springer-Verlag, Siena, Italy, 2001, p. 701.
[20] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, Web Semantics: Science, Services and Agents on the World Wide Web 5 (2007) 51–53.
[21] D. Tsarkov, I. Horrocks, FaCT++ description logic reasoner: System description, in: Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2006), Springer, 2006, pp. 292–297.
[22] D. C. Franz Baader, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider, The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2007.
[23] P. Cimiano, Ontology Learning and Population from Text: Algorithms, Evaluation and Applications, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
[24] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Roberts, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, W. Peters, Text Processing with GATE (Version 6), University of Sheffield, Department of Computer Science, 2011.
[25] R. Witte, B. Sateli, N. Khamis, J. Rilling, Intelligent Software Development Environments: Integrating Natural Language Processing with the Eclipse Platform, in: C. Butz, P. Lingras (Eds.), 24th Canadian Conference on Artificial Intelligence (Canadian AI 2011), volume 6657 of *LNAI*, Springer-Verlag, St. John's, Newfoundland and Labrador, Canada, 2011, pp. 408–419.
[26] R. Witte, T. Gitzinger, Semantic Assistants – User-Centric Natural Language Processing Services for Desktop Clients, in: 3rd Asian Semantic Web Conference (ASWC 2008), volume 5367 of *LNCS*, Springer, Bangkok, Thailand, 2009, pp. 360–374.
[27] F. E. Croxton, S. Klein, D. J. Cowden, Applied general statistics, Pitman, London, 3rd edition, 1968.
[28] R. P. L. Buse, W. R. Weimer, A metric for software readability, in: ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis, ACM, New York, NY, USA, 2008, pp. 121–130.
[29] L. Tan, D. Yuan, G. Krishna, Y. Zhou, /*icomment: bugs or bad comments?*/, in: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, ACM, New York, NY, USA, 2007, pp. 145–158.
[30] G. K. Zipf, Selective studies and the principle of relative frequency in language, 1932.
[31] H. 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, IEEE Computer Society, Washington, DC, USA, 2008, pp. 101–110.
[32] D. Pierret, D. Poshyvanyk, An empirical exploration of regularities in open-source software lexicons, in: 17th International Conference on Program Comprehension (ICPC 2009), pp. 228–232.
[33] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, G. Antoniol, Analyzing the Evolution of the Source Code Vocabulary, European Conference on Software Maintenance and Reengineering (2009) 189–198.