Automatic Quality Assessment of Source Code Comments: The JavadocMiner

Ninus Khamis, René Witte, and Juergen Rilling

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 insights that help software engineers to effectively perform their tasks, and therefore ensuring the quality of the documentation is extremely important. Inline 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 needs to be performed manually. In this paper, we present an effective and automated approach for assessing the quality of inline documentation using a set of heuristics, targeting both *quality* of language and *consistency* between source code and its comments. We apply our tool to the different modules of two open source applications (ArgoUML and Eclipse), and correlate the results returned by the analysis with bug defects reported for the individual modules in order to determine connections between documentation and code quality.

1 Introduction

"Comments as well as the structure of the source code aid in program understanding and therefore reduce maintenance costs." – Elshoff and Marcotty (1982) [1]

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 towards agile development, which focuses 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. Studies have shown that the effective use of comments "can significantly increase a program's comprehension" [2], yet the amount of research focused towards the quality assessment of in-line documentation is limited [3]. 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. In this work, we focus on using NLP services to support users in performing the time-consuming task of analysing the quality of in-line documentation. We focus mainly on in-line documentation due to its close proximity to source code. Additionally, we developed a set of

heuristics based on new and existing metrics to assess the consistency between code and documentation, which often degrades due to changes in source code not being reflected in their comments. Results of the NLP analysis are exported into OWL ontologies, which allow to query, reason, and cross link them with other software engineering artefacts.

2 Background

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

2.1 Inline Documentation and Javadoc

Literate programming [4] was suggested in the early 1980s by Donald Knuth 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*. In order to differentiate between source code and documentation, a specific documentation or programming syntax has to be used.

Single-source documentation also falls into the category of documents with inter-weaved representation. Contrary to the literate approach, documentation is added to source code in the 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 [5]. Documentation generators currently developed are designed to lessen the efforts needed by developers when documenting software, and has therefore become widely accepted and used. Javadoc [6] is an automated tool that generates API documentation in HTML using Java source code and source code comments. In Fig. 1, we show a small section of an API document generated using Javadoc.

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,

Node	**lastNode**() Get the node with the largest offset
Node	**nextNode**(Node node) Get the first node that is relevant for this annotation set and which has the offset larger than the one of the node provided.
boolean	**remove**(Object o) Remove an element from this set.

Fig. 1. Part of an API Documentation generated using Javadoc

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. Both Doxygen [7] and Javadoc also provide an API to implement custom extraction or transformation routines [6]. 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.

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 [2]. A *Method* comment, on the other hand, should provide a low-level understanding of its implementation [2]. The default Javadoc doclet 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, synchronization and readability.

2.2 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 [8]. 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. Currently, the only means of assessing the quality of in-line documentation is by performing time-consuming manual code checks. Any well-written computer program contains a sufficient number of comments to permit people to read 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.[1] In general, each program module contains a description of the logic, the purpose, and the rationale for the module. 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 (formula) for a calculation may be preceded by a comment explaining the source of the formula, the data required, and 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 that often gets neglected due to release or launch deadlines. With such deadlines pressuring the development team it becomes necessary to prioritize. 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

[1] See, e.g., http://www.gnu.org/prep/standards/standards.html

in-line documentation. Ensuring that 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 [9].

3 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.

3.1 Analysis Heuristics for Source Code Comments

In this section, we discuss the set of heuristics that were implemented to assess the quality of in-line documentation. The heuristics are grouped into two categories, *(i)* internal (NL quality only), and *(ii)* code/comment consistency.

Internal (NL Quality) Comment Analysis. We first describe the set of heuristics targeting the natural language quality of the in-line documentation itself.

Token, Noun and Verb Count Heuristic: These heuristics are the initial means of detecting the use of well-formed sentences within in-line documentation. The heuristic counts the number of tokens, nouns and verbs used within a Javadoc comment.

Words Per Javadoc Comment Heuristic (WPJC): This heuristic calculates the average number of words in a Javadoc comment [9]. It can be used to detect a Java Class that contains Fields, Constructors and Methods that are under- or over-documented.

Abbreviation Count Heuristic (ABB): According to "How to Write Doc Comments for the Javadoc Tool" [10] the use of abbreviations in Javadoc comments should be avoided; therefore, using "also known as" is preferred over "aka". This heuristic counts the number of abbreviations used within a Javadoc comment.

Readability Heuristics (FOG/FLESCH/KINCAID): In the early twentieth century linguists conducted a number of studies that used people to rank the readability of text [9]. Such studies require a lot of resources and are often infeasible for source code comments. A number of formulas were implemented that analyse the readability of text [11], 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.

```
/** 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
```

Fig. 2. An Example of a Javadoc Method Comment

Flesch Reading Ease Level: Rates text on a 100 point scale. The higher the scale the easier to read. An optimal score would range from 60 to 70.[2]

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

These readability formulas are also used by a number of U.S. government agencies such as the DoD and IRS to analyse the readability of their documents [9].

Code/Comment Consistency Analysis. The following heuristics analyse in-line documentation in relation to the source code being documented.

Documentable Item Ratio Heuristic (DIR): The DIR metric was originally proposed by [9]. 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.

Any Javadoc Comment Heuristics (ANYJ): ANYJ computes the ratio of *identifiers with Javadoc comments* to the *total number of identifiers* [9]. This heuristic can be used to determine which classes provide the least amount of documentation.

SYNC Heuristics (RSYNC/PSYNC/ESYNC): The following heuristics detect methods that are documenting *return types, parameters* and *thrown exceptions* that are no longer valid (e.g., due to changes in the code):

RSYNC: When documenting the *return type* of a *method* the *@return in-line tag* must begin with the correct name of the *type* being returned followed by the *doc comment* explaining the *return type*.

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*.

ESYNC: When documenting the *exceptions thrown* by a *method* the *@throws* or *@exception* in-line tags documentation must begin with the correct names of the *exceptions* being returned followed by the *doc comment* explaining the *exception* itself.

[2] 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 [11].

The `parseAssociationEnd` method in Fig. 2 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 in-line tags*.

3.2 Source Code Comment Ontology

The results of the heuristics described above are exported into an OWL ontology. Using an ontology enables us to model the 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. Using an ontology also enables users to take advantage of the reasoning services provided by a Description Logic reasoner such as Racer [12], Pellet [13], or FaCT++ [14]. Finally, visualizations and SPARQL queries can also be applied on the source code comment ontology.

The source code comment ontology models: *(1)* NLP related entities such as Sentence, NP, VP; *(2)* Source code identifiers such as Interface, Class, Field and Method; *(3)* Source code comments such as MethodComment or FieldComment; and *(4)* Entities such as Author and Version that can exist in source code comments.

Many relationships exist between each entity. For example, the relationships where a class: *implements* a certain interface, and *contains* a certain class comments that is written by a specific author. Also, fields and methods that also have their own comments are all modelled within the ontology. Fig. 3 shows an excerpt of our ontology.

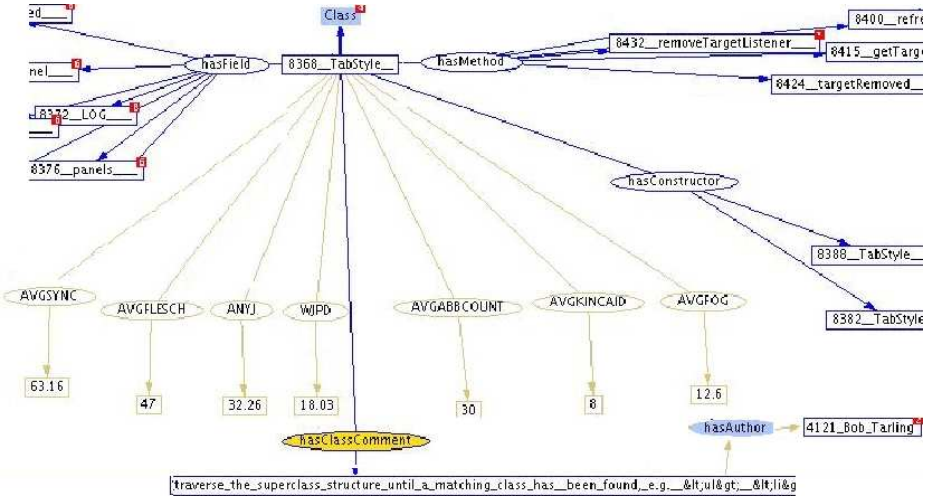

Fig. 3. A Excerpt from the Source Code Comment Ontology

4 Implementation

In this section, we discuss in detail the various parts that make up the JavadocMiner. We begin by covering the process of building a corpus from source code using a custom `doclet`.[3] We then explain the components used within our text mining application to process the corpus for the set of heuristics described above.

4.1 Corpus Generation from Javadoc

Javadoc's standard doclet generates API documentation using an HTML format. While this is convenient for human consumption, automated NLP analysis requires a more structured XML format. Generation of such an XML format is possible by developing a custom doclet using the Javadoc library, which provides access to the Abstract Syntax Tree (AST) generated from source code and source code comments. We implemented a custom doclet called the `SSLDoclet` [15],[4] which 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 our NLP application can process easily.

4.2 The JavadocMiner GATE Application

Our JavadocMiner is implemented as a GATE pipeline [16], which is assembled using individual *processing resources* (PRs) running on a corpus of documents.

Preprocessing Stage. Before running the JavadocMiner PR, we perform preprocessing tasks using components already provided by GATE, including tokenization, sentence splitting, and POS-tagging.

Abbreviation Detection. The abbreviations are detected using a gazetteering list that is part of ANNIE [16]. The list contains commonly used abbreviations within the English language.

JavadocMiner PR. We implemented a GATE processing resource component called the *JavadocMiner PR* that contains the set of heuristics described above to assess the quality of source code comments. Most of the heuristics were implemented by us; with the exception of the readability heuristics that makes use of an existing library [17].

OwlExporter PR. The OwlExporter [18] is the final step in our pipeline that is in charge of taking the annotations created by the text mining pipeline and exporting it to an ontology.

[3] See http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/overview.html
[4] SSLDoclet, see http://www.semanticsoftware.info/javadoclet

Table 1. Assessed Open Source Project Versions, Release Dates, Lines of Code (LOC), Number of Comments, Identifiers and Bugs

Project Version	Release Date	LOC	Num. of Comments	Num. of Identifiers	Num. 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

5 Application and Evaluation

In the section, we discuss how the JavadocMiner was applied on two open source projects for an analysis of comment quality and their consistency with the source code; we discuss the results gathered from our study, and finally show how we additionally correlated the NLP quality metrics with bug statistics.

5.1 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 modelling tool ArgoUML[5] and the IDE Eclipse.[6] In Table 1, we show the versions of the projects that were part of our quality assessment.

5.2 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 – *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. The complete quality assessment process for both open source projects took less than 3 hours. We continued by finding the amount of bug defects 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.

5.3 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 (Fig. 4, left side) and the *PDE module* for Eclipse (Fig. 4, right side).

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

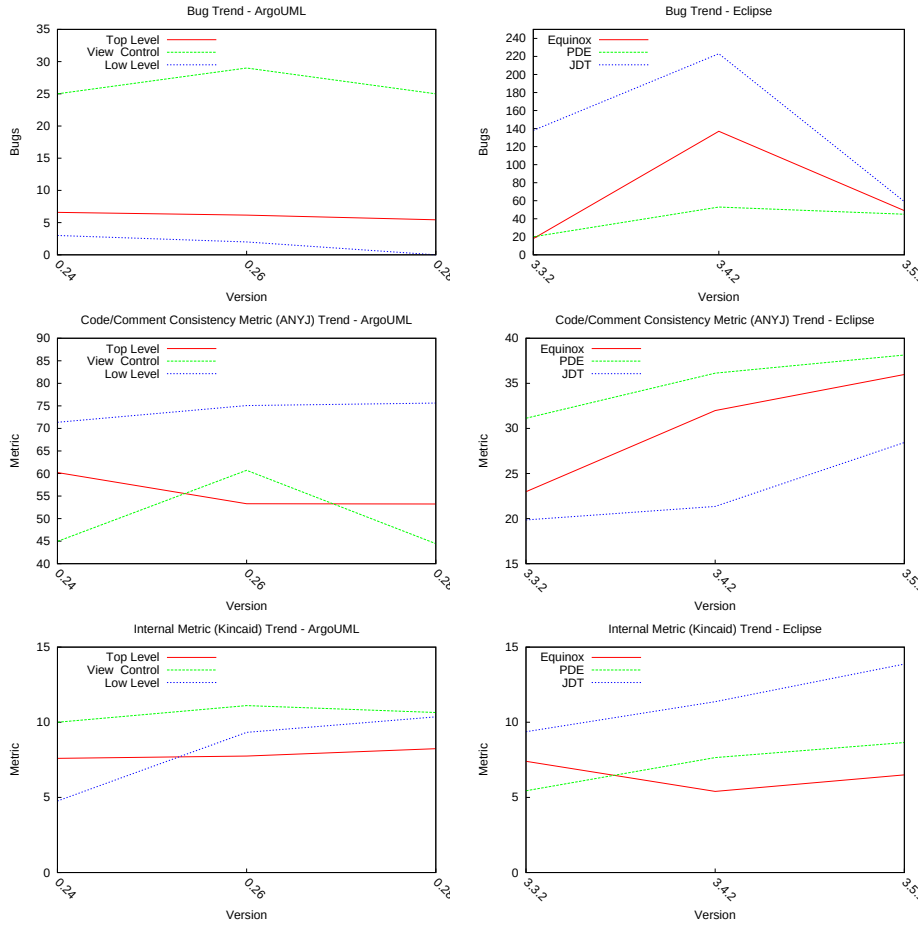

Fig. 4. Charts for Bug Trend, Code/Comment and Internal (NL Quality) Metrics

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 Eclipse project is separated into API and internal non-API packages, and part of the Eclipse policy states that all API packages must be properly documented [9].

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 (Fig. 4, bottom 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 (Fig. 4, bottom right).

Table 2. Pearson Correlation Coefficient Results for ArgoUML and Eclipse

Project	ANYJ	SYNC	ABB	FLESCH	FOG	KINCAID	TOKENS	WPJC	NOUNS	VERBS
ArgoUML	0.99	0.98	-0.94	0.32	0.80	0.79	0.89	0.91	0.98	0.87
Eclipse	0.97	0.89	-0.86	0.37	0.77	0.84	0.88	0.86	0.91	0.73

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 (Fig. 4, top left) and for Eclipse (Fig. 4, top right). We observed 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 on the data gathered from the quality assessment and the number of bug defects (Table 2). Fig. 4 also shows the quality assessments returned by ANYJ (Fig. 4, middle) and Kincaid (Fig. 4, bottom) for both ArgoUML and Eclipse.

The correlation and coefficient results showed that the ANYJ, SYNC, ABB, Tokens, WPJC and Nouns heuristics were strongly correlated to the number of bug defects, whereas the Flesch readability metric was amongst the least correlated. In Fig. 5 (top), we show the number bug defects reported at the level of quality assessments returned by the ANYJ and ABB metrics, and in Fig. 5 (bottom) for the readability and NLP metrics for ArgoUML. The same data is represented for Eclipse in Fig. 6.

Examining the charts showed a 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 using the correlation statistic.

6 Related Work

There has been effort in the past that focused on analysing source code comments, for example in [19] 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 [3] 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

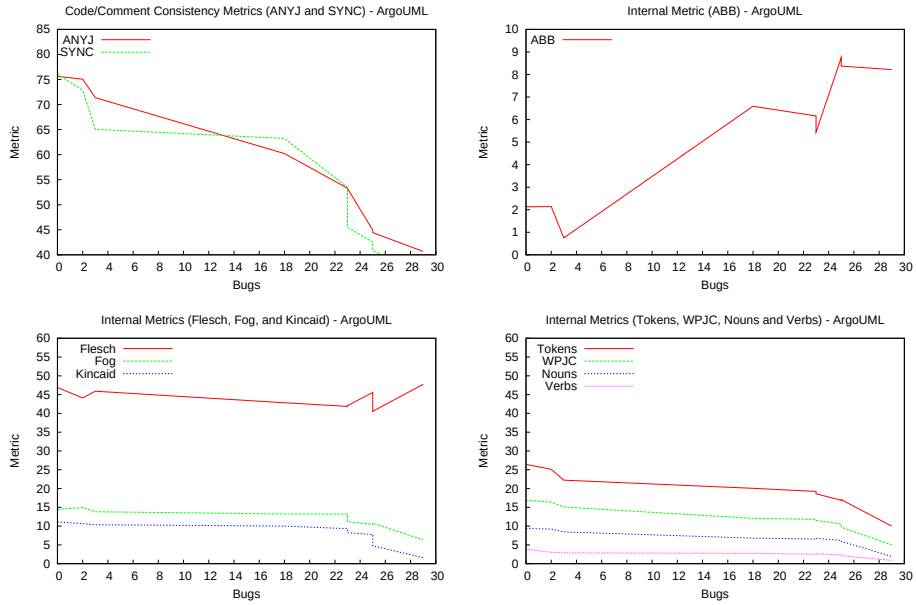

Fig. 5. Code/Comment Consistency and NL Quality Metrics vs. Bugs – ArgoUML

to answer questions such as 1) what is written in comments; 2) whom are the comments written for or written by; 3) where the comments are located; and 4) when the comments were 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, 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. The authors also made no attempt to automate the process, nor was there any major correlations made with other software engineering artefacts.

The work described in [20] 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.

The only work that we know of which focused on automatically analysing API documentation generated by Javadoc is [9]. 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 [9] by introducing new quality assessment metrics. We also analysed each module

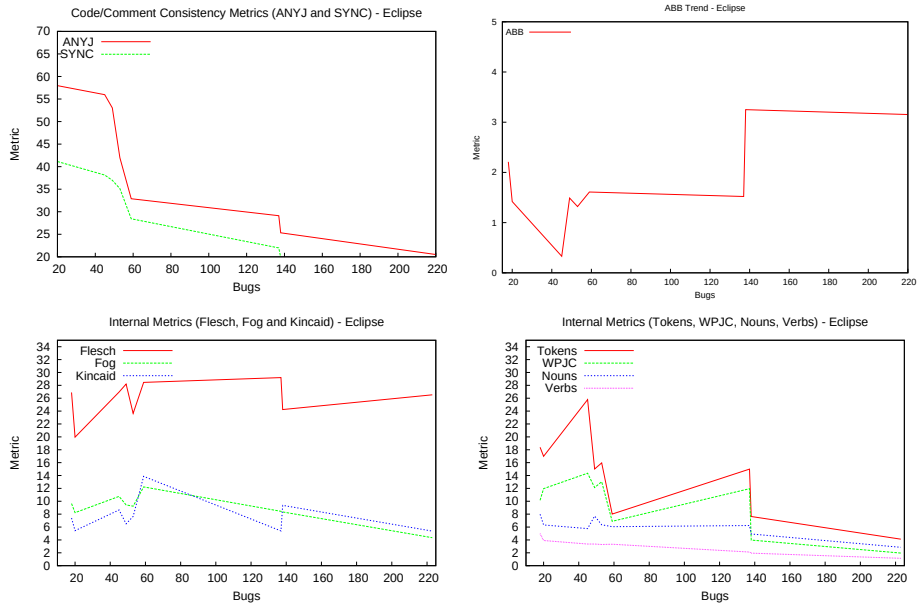

Fig. 6. Code/Comment Consistency and NL Quality Metrics vs. Bugs – Eclipse

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 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 to automatically assess the quality of documentation found in software. 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. Currently we are conducting a case study involving students and the quality assessment of 120 in-line documentation samples, in order to compare the results obtained from the JavadocMiner with the results of human intuition analysing the quality of in-line documentation.

Acknowledgements. This research was partially funded by DRDC Valcartier (Contract No. W7701-081745/001/QCV).

References

1. Fluri, B., Würsch, M., Gall, H.: Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In: WCRE. (2007) 70–79
2. Nurvitadhi, E., Leung, W.W., Cook, C.: Do class comments aid Java program understanding? In: Frontiers in Education (FIE). Volume 1. (Nov. 2003)
3. Padioleau, Y., Tan, L., Zhou, Y.: Listening to programmers Taxonomies and characteristics of comments in operating system code. In: ICSE '09, Washington, DC, USA, IEEE Computer Society (2009) 331–341
4. Knuth, D.E.: Literate Programming. The Computer Journal **27**(2) (1984) 97–111
5. Brooks, R.E.: Towards a Theory of the Comprehension of Computer Programs. International Journal of Man-Machine Studies **18**(6) (1983) 543–554
6. Kramer, D.: API documentation from source code comments: a case study of Javadoc. In: SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation, New York, NY, USA, ACM (1999) 147–153
7. van Heesch, D.: "Doxygen". http://www.stack.nl/~dimitri/doxygen/ (2010)
8. Lehman, M.M., Belady, L.A., eds.: Program evolution: processes of software change. Academic Press Professional, Inc., San Diego, CA, USA (1985)
9. Schreck, D., Dallmeier, V., Zimmermann, T.: How documentation evolves over time. In: IWPSE '07: Ninth international workshop on Principles of software evolution, New York, NY, USA, ACM (2007) 4–10
10. Sun Microsystems: "How to Write Doc Comments for the Javadoc Tool". http://java.sun.com/j2se/javadoc/writingdoccomments/
11. DuBay, W.H.: The Principles of Readability. Impact Information (2004)
12. Haarslev, V., Möller, R.: RACER System Description. In Goré, R., Leitsch, A., Nipkow, T., eds.: Automated Reasoning: First International Joint Conference (IJCAR) 2001. Volume 2083 of Lecture Notes in Computer Science., Siena, Italy, Springer-Verlag (June18-23 2001) 701
13. Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web **5**(2) (June 2007) 51–53
14. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: In Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006, Springer (2006) 292–297
15. Khamis, N., Witte, R., Rilling, J.: Generating an NLP Corpus from Java Source Code: The SSL Javadoc Doclet. In: New Challenges for NLP Frameworks, Valletta, Malta, ELRA (05/2010 2010)
16. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V.: GATE: A framework and graphical development environment for robust NLP tools and applications. In: Proc. of the 40th Anniversary Meeting of the ACL. (2002)
17. Ryan, K., Fast, G.: "Java Fathom". http://www.representqueens.com/fathom/
18. Witte, R., Khamis, N., Rilling, J.: Flexible Ontology Population from Text: The OwlExporter. In: Int. Conf. on Language Resources and Evaluation (LREC), Valletta, Malta, ELRA (05/2010 2010)
19. Buse, R.P.L., Weimer, W.R.: A metric for software readability. In: ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis, New York, NY, USA, ACM (2008) 121–130

20. Abebe, S.L., Haiduc, S., Marcus, A., Tonella, P., Antoniol, G.: Analyzing the Evolution of the Source Code Vocabulary. Software Maintenance and Reengineering, European Conference on (2009) 189–198