How Documentation Evolves Over Time

Daniel Schreck
Dept. of Computer Science
Saarland University
Saarbrücken, Germany
schreck@st.cs.uni-sb.de

Valentin Dallmeier
Dept. of Computer Science
Saarland University
Saarbrücken, Germany
dallmeier@cs.uni-sb.de

Thomas Zimmermann
Dept. of Computer Science
Saarland University
Saarbrücken, Germany
tz@acm.org

ABSTRACT

Good source code documentation, especially of programming interfaces, is essential for using and maintaining software components. In this paper, we present the QUASOLEDO tool that automatically measures the quality of documentation with respect to completeness, quantity, and readability. We applied our set of metrics to the ECLIPSE project, and benchmarked against the well-documented JAVA class library. The result of QUASOLEDO is a map of documentation quality in ECLIPSE, showing the best documentation for its core components. Additionally, we looked at the evolution of ECLIPSE and identified batch updates that caused jumps in documentation quality. For ECLIPSE, only 32.6% of all changes touched documentation.

1. INTRODUCTION

Unfortunately, much computer system documentation is difficult to understand, badly written, out-of-date or incomplete. – Ian Sommerville [10]

A survey of industrial software projects in 2002 found that satisfaction with documentation quality is low or even very low in 84% of the investigated development projects [12]. Projects with bad documentation are difficult to maintain and extend since original developers may no longer be available.

The results of the study also indicated that bad documentation is not a result of bad documentation processes, but of developers not adhering to the specified processes. A project manager that wants to assure good documentation quality therefore needs an easy means to assess the current level of documentation. For large projects, this can become a tedious task, which is why we would like to automate it.

The goal of our work was to *automatically assess the quality features of source code documentation* for computer programs. We focus on source code documentation since developers consider source tools like Javadoc as one of their most important documentation tools [6]. In this paper, we specifically explore the following three questions:

1. *How can we measure documentation quality?* (Sec. 2)
2. *Which parts of a program are documented well or poorly?* (Section 3)
3. *When and why in the evolution of a project does documentation quality change?* (Section 4)

In order to address these questions, we implemented a prototype named QUASOLEDO that analyzes comments in JAVA source code. We applied our approach to the source code of ECLIPSE, an industrial-sized open-source project with almost two million lines of code. With QUASOLEDO we drilled down several aspects of documentation quality to program parts that might need improvement of documentation. Furthermore, we automatically identified major evolution steps in the documentation of ECLIPSE. Such points in time are helpful to annotate the history of a project and build a journal of software evolution, including events such as documentation phases and refactorings.

2. DOCUMENTATION QUALITY

We now present the metrics (Section 2.1) and readability measures (Section 2.2) that we used to assess the quality of documentation. While not all aspects of documentation quality can be measured automatically, we believe that our set of measures can already greatly assist developers and managers.

In this paper, we focus on Javadoc comments that are embedded in the source code (see Listing 1 for an example) and used by the JAVADOC tool to generate HTML reference documents. The reference documents can be used internally by the developers of the software, but are typically intended for external developers that need to use the software API.

In the source code, each Javadoc comment starts with the string `/**` and precedes a class, method, or field. The text immediately at the beginning is used as the main description of the associated declaration. *Tags* provide details about certain aspects of an element such as parameters, return value, and exceptions.

2.1 Metrics

We arrange similar metrics into families. All metrics in a family use the same formula, but each takes a different set of source entities into account such as only classes, only methods or all entities. In this paper we concentrate on

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWPSE'07, September 3-4, 2007, Dubrovnik, Croatia.
Copyright 2007 ACM ISBN 978-1-59593-722-3/07/09 ...$5.00.

```
/** Splits this string around matches of a
      regular expression.
 * @param  regex  the delimiting regular
       expression
 * @return  the array of strings computed by
       splitting this string
 * @throws  PatternSyntaxException  if the
       regular expression's syntax is invalid */
public String[] split(String regex) {
  return this.split(regex, 0);
}
```

Listing 1: Example of a Javadoc comment

metrics that consider only methods, because most of the interaction with interfaces happens through method calls, making method Javadoc most important.

Completeness.
The ANYJ family computes the ratio of declarations with any Javadoc to the total number of declarations.

$$\text{ANYJ} = \frac{\text{declarations with any Javadoc comment}}{\text{total number of declarations}} \quad (1)$$

Listing 1 has a score of ANYJ = 1.0 because there is one method declaration and it has a Javadoc comment.

In order to have complete Javadoc, every method parameter must be documented with a @param, every return value with @return and every exception with @throws or @exception. We take these tags into account when we compute the *documented items ratio* (DIR). Here, an item includes possible tags, as well as classes, fields, and methods.

$$\text{DIR} = \frac{\text{documented items}}{\text{documentable items}} \quad (2)$$

The method in Listing 1 scores DIR $= \frac{3}{3} = 1.0$ because the method itself, the parameter and the return value each are documented. The tag @throws is not considered as the exception is not declared on the method.

Sometimes developers use traditional style comments in the same way that Javadoc is used—they put it immediately before a method declaration and explain the purpose of the method in the comment. That way the documentation is not found by tools and is thus of little use to users, but the developers can provide an internal specification of the method. To allow for detecting such documentation we have the $\text{ANYC}_{\text{method}}$ metric, which is the ratio of methods with any kind of non-empty comment preceding it, no matter if Javadoc or ordinary comment.

$$\text{ANYC}_{\text{method}} = \frac{\text{methods with any kind of comment}}{\text{methods}} \quad (3)$$

The Listing scores $\text{ANYC}_{\text{method}} = 1.0$ as the method has a Javadoc comment. The difference to $\text{ANYJ}_{\text{method}}$ is that a comment like "// Splits the String" before the method will result in $\text{ANYC}_{\text{method}} = 1$, but $\text{ANYJ}_{\text{method}}$ will be 0.

Note that the values of ANYJ, $\text{ANYC}_{\text{method}}$, and DIR are limited to the interval [0,1], where 1 is the best score.

Quantity.
The *words in the Javadoc per declaration* (WJPD) metrics measures the average number of words in Javadoc. If a declaration does not have Javadoc, it is counted with zero words. HTML tags in Javadoc content are ignored.

$$\text{WJPD} = \frac{\text{words in Javadoc of declarations}}{\text{declarations}} \quad (4)$$

Accordingly, Listing 1 scores WJPD $= \frac{29}{1}$.

A low number of words might indicate that the documentation gives too little information, while very high values might indicate documentation that is too verbose for typical usage. Ultimately, the difficulty of a method should be considered, because high values might also be because of a complex interface.

2.2 Readability

Source code documentation should be easily accessible to both users and developers.[1] It is important for users, because to skip reading could mean they make errors in interfacing the code; and it is important for developers, because to keep documentation and implementation in-sync they frequently need to check the documentation and want to do so quickly.

Since the early twentieth century, linguist researchers have striven to find out what makes a text difficult to read and understand. Testing text with users is the best way to find out, however, such tests are time-consuming and expensive. In 1949 Robert Flesch proposed his *Flesch Reading Ease* to assess text based on its structure and by now it "became the most widely used formula and one of the most tested and reliable" ([3], p. 21).

In 1975 Kincaid, Fishburne, Rogers, and Chissom adapted the Flesch Reading Ease and several other readability measures to a grade level scale and adjusted it with navy training documents and adult readers. The *Flesch-Kincaid Grade Level* is a U.S. grade level that the readers of a text should have at least:

$$\text{KINCAID} = 0.39 \times \frac{\#\text{words}}{\#\text{sentences}} - 11.8 \times \frac{\#\text{syllables}}{\#\text{words}} - 15.59 \quad (5)$$

The KINCAID formula is recommended by the U.S. Department of Defense, the Internal Revenue Service and other U.S. government agencies to achieve better readability of their documents [3]. The appropriate readability grade level depends on the intended audience, but should in general not be above college level (12th grade).

QUASOLEDO uses the UNIX tool STYLE [1] to calculate seven readability formulas. In this paper we will discuss the results for the KINCAID measure.

2.3 Aggregation

There are several *granularity levels* on which a JAVA program can be divided into parts, starting with the coarsest: product, module, package and class level. For reasons of brevity we only discuss the results for product and package level here. Results and more on product and package level can be found in [9].

Visibility levels are orthogonal to the granularity levels of JAVA program elements. These levels are in decreasing order: *public*, *protected*, *package private* and *private*. To indicate the visibility level, we add a superscript to the name of the metric, like in $\text{ANYJ}^{public}_{\text{method}}$. Keep in mind that just *public* and *protected* program elements are externally accessible. Consequently, HTML reference documents for exter-

[1] The term *user* refers to a developer using an API.

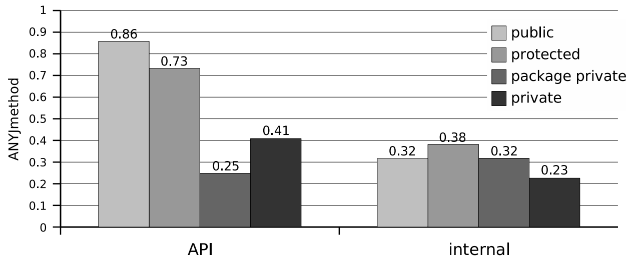

Figure 1: Public methods are documented more frequently ($ANYJ_{method}$)

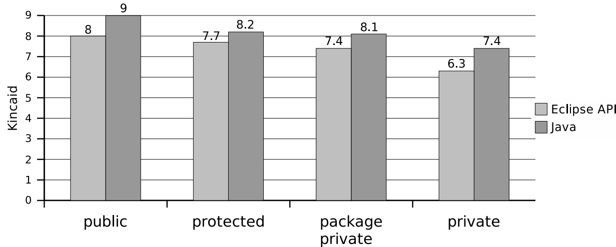

Figure 2: Readability of documentation for Eclipse API and Java (Kincaid)

nal developers are generated only from the Javadoc of those elements.

The measures in the previous sections were for single classes. In order to apply them to parts or all of a software product, we combine them as follows: For the *relative metrics* like ANYJ, DIR and WJPD we just add up numerator and denominator separately. This way aggregation is implicitly weighted according to the size of the classes' interfaces—classes with more declarations will influence the aggregated value more. The *readability scores* are calculated on the text extracted from all the Javadoc belonging to the class, package, module or product. The scores are never combined in any way, they are recalculated completely by the STYLE tool for each set of Javadoc comments. A new paragraph is started for every class, method or field description.

2.4 Discussion

Readability formulas are a valid way to assess readability, but they do not cover everything. Or with the words of G. Hargis [7]: "Technical writers have accepted the limited benefit that these measurements offer in giving a rough sense of the level of difficulty of material." In a similar way our selection of metrics and measures is far from being comprehensive, but covers many important aspects such as documentation that is incomplete, too brief, or too difficult. With today's technology automatic assessment, however, can only assist and not replace human analysis.

3. WHERE IS POOR DOCUMENTATION?

We applied our QUASOLEDO tool on ECLIPSE (snapshot of April 2006), a big open source product with over 1.6 million source lines of code, in order to identify the parts with good and poor documentation. As a benchmark for the quality,

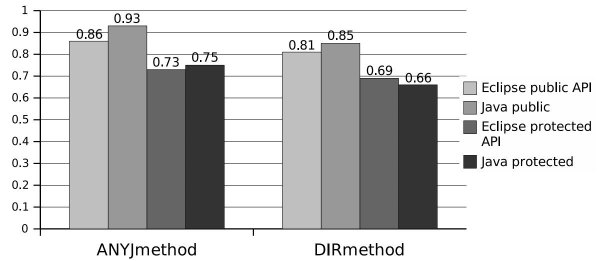

Figure 3: Benchmarking Eclipse API against Java ($ANYJ_{method}$ and DIR_{method})

we used the documentation of Sun Microsystem's implementation of the JAVA class library, version 5.0.

3.1 Documentation Quality in Eclipse

ECLIPSE is seperated into API and non-API packages because its modules typically have *internal packages* besides those meant for public use. As API and internal parts have different documentation needs, we discuss them separately.

The internal classes of ECLIPSE outnumber the API classes three to one; there are 11,336 internal, but only 3,753 API classes. Moreover, there are 103,920 internal methods, but only 42,013 API methods. Looking at the number of words in the Javadoc, the picture turns around, as only 758,049 words can be found in the internal Javadoc versus 1,544,101 words in API classes. So, while the internal part represents at least two thirds of ECLIPSE, it has only one third of the documentation volume.

In fact, to document all API but not the internal parts is proclaimed policy of the ECLIPSE developers [2]: "By their very nature, API elements are documented and have a specification, in contrast to non-API elements which are internal implementation details usually without published documentation or specifications." Further, for ECLIPSE only *public* and *protected* classes, methods and fields are considered as *API elements*. This fact is evident in Figure 1 which differentiates the values of $ANYJ_{method}$ by visibility and separated into internal and API packages: *public* and *protected* API methods have Javadoc more than twice as often as other methods.

The quantity of documentation also varies. While the average Javadoc method comment on a *public* API method has 48.8 words, the average *public* Javadoc comment in an internal package has just 17.8 words. Moreover, while Javadoc comments on internal methods have roughly the same average length regardless of their visibility (19.0 ± 1.5), the documentation quantity of API methods grows with increasing visibility from 22.1 for *private* methods to 48.8 for *public* methods.

3.2 Benchmarking Documentation Quality

In order to have a benchmark for the ECLIPSE scores, we compare them to scores obtained from the JAVA API of Sun Microsystem's JAVA version 5.0, i.e., the *public* and *protected* elements in the packages java and javax.

Figure 3 compares ECLIPSE and the JAVA class library on the two visibilities relevant for API documentation. The scores on the completeness metrics are close together. The JAVA library scores a bit higher in terms of *public* Javadoc;

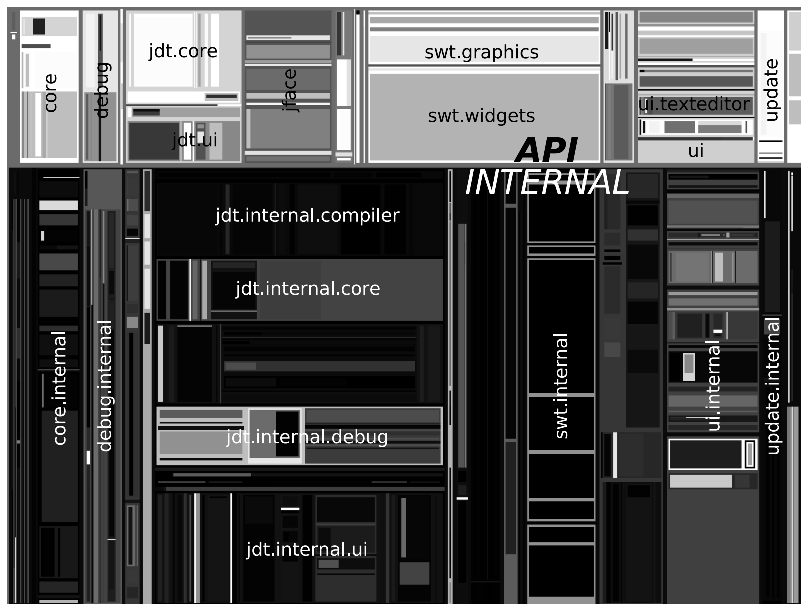

Figure 4: Treemap of documentation quality for Eclipse packages ($\mathbf{ANYJ}^{public/protected}_{\mathbf{method}}$)

its score for $\mathrm{DIR}_{\mathrm{method}}$ and $\mathrm{ANYJ}_{\mathrm{method}}$ is 4.6% and 7.3% higher, respectively. On *protected* Javadoc the picture is mixed; while ECLIPSE has a higher score on $\mathrm{ANYJ}_{\mathrm{method}}$, its score is lower on $\mathrm{DIR}_{\mathrm{method}}$. Thus, there are more methods in ECLIPSE that have some Javadoc, but there are fewer tags given.

In terms of documentation quantity the difference is similar. The JAVA API scores $\mathrm{WJPD}^{public}_{\mathrm{method}}$ =49.3 and ECLIPSE API scores 41.7. Similarly $\mathrm{WJPD}^{protected}_{\mathrm{method}}$ is 28 for JAVA and 23.2 for ECLIPSE API. When looking only at the average over the methods that have Javadoc, the difference is smaller: 53.2 versus 48.8 on *public* methods and 37.2 versus 31.7 on *protected* methods. To sum up, the ECLIPSE API has Javadoc on fewer elements, and the comments are shorter on average.

3.3 Readability of Documentation

Figure 2 shows the readability of ECLIPSE API and JAVA documentation broken down by visibility. Less visible program elements are easier to read for both ECLIPSE and JAVA. One reason could be that less visible elements often have short Javadoc that is less likely to contain long and complex sentences. However, as there is no visibility level where the scores differ by more than one grade level, the difference is negligible.

Comparing the ECLIPSE API and the JAVA class library documentation, it turns out that JAVA documentation is slightly more difficult to read than ECLIPSE documentation. This is rather surprising considering that the JAVA library documentation is used by more people than that of ECLIPSE. In Section 2.2 we argued that a KINCAID grade level below 12 is appropriate. Our results show that both the ECLIPSE API and JAVA API achieve that goal.

3.4 Mapping Documentation Quality

For assessing and comparing documentation quality between packages, we used *treemaps*. Treemaps are drawn in a rectangular area and represent entities through colored boxes. The size of a box corresponds to the size of the represented entity. In our case, the entities are packages and their size is the number of methods. The color of a box corresponds to the package's documentation quality, as determined by one of our metrics. Child packages are drawn inside their parent's box and enlarge it accordingly. Treemaps are often explored interactively, usually querying the label of a box by pointing the mouse at it. For the treemaps printed in this paper we labeled important packages by hand.

Figure 4 illustrates the completeness of documentation quality as measured by the $\mathrm{ANYJ}^{public/protected}_{\mathrm{method}}$ score for ECLIPSE packages. Packages with a score of one are represented through a white box; those with a score of zero through a black box. The scores in between are represented on a gray scale: the lighter the color, the better the score. Further, the larger a box, the more *public* and *protected* methods the package has. The upper quarter of the diagram contains the API packages, and the lower three quarters contain the internal packages.

Again we can observe the differences in documentation quality between API and internal packages. There are a few internal packages with good documentation quality in terms of $\mathrm{ANYJ}^{public/protected}_{\mathrm{method}}$, for example, the debugging packages in jdt.internal.debug.core and jdi.internal. In general, though, internal *public* and *protected* methods are seldom documented. However, not all API packages get top scores. Most of the less documented packages deal with GUI presentation, like jdt.ui, jface or ui. Particularly well documented are core, jdt.core, jdt.debug and update.

3.5 Discussion

The scores of our metrics for ECLIPSE reflect the known split of ECLIPSE into well documented API and poorly documented internals. This finding confirms that our metrics are suitable to measure differences in documentation quality.

For both ECLIPSE and the JAVA class library, Javadoc

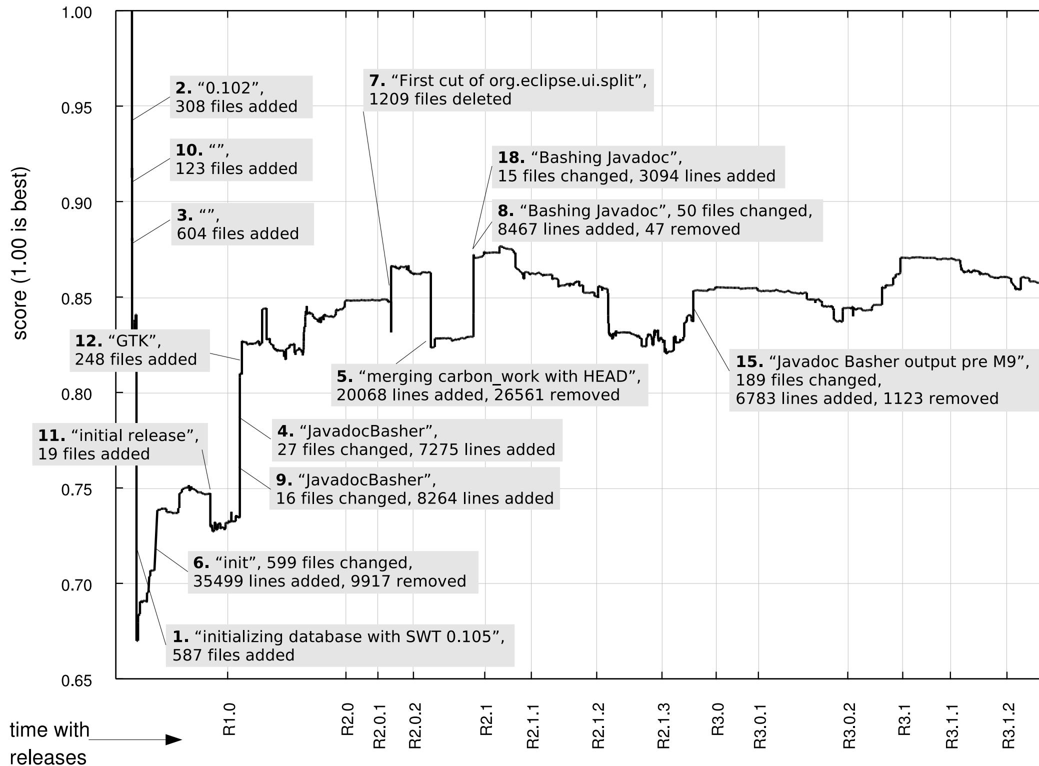

Figure 5: Annotated evolution graph of $\text{ANYJ}^{public}_{\text{method}}$ of Eclipse API packages

availability (ANYJ), tag usage (DIR) and verbosity (WJPD) all decrease with lower visibility. Further, Eclipse core modules are documented the best, both in terms of ANYJ and DIR; in contrast GUI packages are documented the worst. In general, we observed that for Eclipse the most important and central packages usually were documented best.

4. HOW DOCUMENTATION EVOLVES

In this section, we investigate the evolution of documentation quality in Eclipse. The Eclipse project has a fairly long development history of five years and over 100,000 changes. Our Quasoledo tool can either

- *sample* the history in defined intervals to obtain a quick overview or
- *replay* the entire development history of a program (commit by commit) to show a detailed overview how documentation quality evolves over time.

4.1 Evolution of Documentation Quality

In Figure 5 we show the evolution of the $\text{ANYJ}^{public}_{\text{method}}$ metric on API methods of Eclipse. Time is represented in the x-dimension, with releases marked on the x-axis. The first timepoint in the history is April 28, 2001; the last is April 8, 2006. Note that the y-axis starts at 0.65 as the score never drops below that value. Jumps in Figure 5 have been *automatically* annotated by Quasoledo with commit messages from the CVS archive from Eclipse. Therefore we can see not only when documentation quality changed, but also *why*.

In order to annotate the jumps, Quasoledo first identifies the widest jumps between the available sample data points. Next Quasoledo increases the sampling granularity to the level of commits and performs a linear search to identify the commit (and its message) that caused the jump. In case a wide jump consists of several smaller changes, Quasoledo reports the biggest of those changes.

In case the available data already includes all commits, we directly start with the linear search. We used this latter variant to obtain the data in the figure, because it ensures that no jumps are missed. However, the runtime cost to create the detailed data with all commits in the first place is considerable.

The text in the boxes in Figure 5 summarizes the properties of the commit. The bold number in each box is the rank in terms of the relative impact the commit has had on the $\text{ANYJ}^{public}_{\text{method}}$ score. The text in quotes ("...") is the comment provided by the CVS user when she committed the changes. In addition, we list the number of files affected by the commit or the number of lines changed. For the figure Quasoledo annotated the top 20 commits, but we do not present all of the in the graph. Some commits with empty comments, some merges and file additions are left out.

We can observe two different reasons for jumps in Figure 5: (1) additions or deletions of large quantities of files ("new/removed code") and (2) changes that affect many lines ("churned code").

Rank 1, 2, 3, and 10. The widest jumps occurred in the first few days of Eclipse, when more and more files were added to the CVS repository. While the first few files had Javadoc on all *public* API methods and thus Eclipse had $\text{ANYJ}^{public}_{\text{method}} = 1.0$, additions of more and more files with less Javadoc resulted in the score dropping to 0.67 within two weeks. Four commits, ranking first, second, third and

tenth added 1,622 files.

Rank 4, 9, 15 and 8, 18 ("JavadocBasher"). Another big change in the quality of documentation occurred three weeks after release 1.0. Two commits (ranked fourth and ninth in terms of impact), both with the comment "JavadocBasher", added more than 15,000 lines with 44,256 words to 43 files, only 9 lines were deleted. This means that the commits cannot just have improved existing Javadoc comments only, but mostly added comments to previously undocumented program elements, which explains the jump.

The term "JavadocBasher" appears not only when documentation is added, but also when it is changed. The commit with rank 15 affected a total 189 files, added 6,783 lines and removed 1,123 lines. Since CVS records changes as removals and additions, we validated by manual inspection that many of these lines were actually changed, e.g., spelling and punctuation was corrected and sentences were reworked.

Apparently, the message "JavadocBasher" describes commits that polish the Javadoc of recently developed code, which typically happen shortly before a release. This indicates that technical writers are responsible that Javadoc can be understood and complies with the HTML standards. In ECLIPSE, we identified two developers bashing Javadoc: "carolyn" checked in the commits with ranks 4, 9 and 15 and "veronika" the ones with ranks 8 and 18.

Rank 5. Later jumps in the graph were caused by merges from other branches, like in the commit with rank five. Its comment is "merging carbon_work with HEAD". Carbon is the name of a native GUI toolkit for the operating system MacOS. Thus, the changes affected the GUI code of ECLIPSE.

4.2 Co-changes of Documentation

We also checked how often Javadoc is changed along with the code (a so-called *"co-change"*) and how often the two are changed independently. Although, a change which only touches code and not the Javadoc may indicate aging of Javadoc, such changes are more likely internal optimizations, refactorings and the like. Changes that touch code and documentation, are likely to implement new features or change external functionality.

Methods.

For ECLIPSE we found that out of the 399,908 changes on methods, 15.3% changed only the Javadoc and 72.6% changed only the code. 12.1% changed both. These numbers indicate that code and documentation are updated separately. This process can be sub-optimal because developers have to understand code they or someone else has written earlier.

Commits.

Out of 71,313 commits, 2.1% changed only Javadoc, 67.4% changed only code and 30.5% changed both. The interesting number here is 2.1% which shows that there exist commits that exclusively touch documentation. Additionally, almost every third change touches documentation (32.6%).

Fixes.

We also correlated bug fixes with changes on Javadoc.

- *Bugfixes.* Of the 28,273 commits that were fixes, 255 (0.9%) changed only Javadoc, 68.7% changed only code and 30.4% changed both.
- *Non-Bugfixes.* In the 43,040 commits that were not fixes, changes affecting only the Javadoc are more frequent with 2.9%. About 66.5% of commits touched only code and 30.6% changed both.

These results show that concerning the maintenance of Javadoc, there is no clear difference between fixes and regular development changes. However, interesting is that some bugs (0.9%) can be fixed by only adjusting Javadoc. One example for this is ECLIPSE bug report 119638 "Typo in ToolItem.getControl". The corresponding fix just changed "items" to "item" and removed a dot at the end of the sentence.

```
252c252
< * the item when the items is a <code>SEPARATOR</code>.
---
> * the item when the item is a <code>SEPARATOR</code>
```

4.3 Discussion

Ideally, documentation of a program element should always be up-to-date, and be updated whenever its external behavior is changed. In reality, programs can and often do change while they are still in development. Developers do not want to waste their time writing documentation that is likely to be thrown away later. Therefore, they often keep developing new code until it runs and has been tested. Writing documentation is then deferred to a later point in time (in form of a *Javadoc bash*). It can become a problem, if much time passes between writing the code and writing its documentation, because developers tend to forget rationale and design after some time. Later then, developers need to invest much time into re-understanding the code. Thus, a reasonable policy would not require a program element documentation to be written at the same time as the code, but there should not be more than a few days passing in between. Additionally, having a *correction phase* for documentation before a release, where wording and other matters of secondary importance are handled (such as correcting HTML code or grammar), is reasonable.

5. RELATED WORK

As mentioned in the introduction, the problem of poor quality system documentation is widely known. Yet, there are few scientific papers dealing with automatic assessment on the specific problem of source code documentation. In the following paragraphs we present works with similar, yet different goals to ours.

The goal of Robles et al. [8] is to automatically evaluate documentation quality. They present a tool that can automatically assess the volume of documentation of open source software. The tool reports the amount of text documents as well as the amount of source code comments found in a software archive. In contrast to our work, the parts of the software that lack documentation cannot be pinpointed as easily with the reports.

Etzkorn et al. [5] collected metrics on object oriented software to judge its "reusability quality". Among other metrics, they also calculated a documentation quality metric from the comment density in C++ code which they used to categorize documentation into five buckets ranging from

excellent to unacceptable. On three C++ GUI libraries the categorization matched that of experts in 62.5% of the cases. The authors additionally found that experts also evaluated the quality of the identifiers used for class, method or field names. Consequently, experts considered some classes without comments to be documented, because they could understand everything from the identifiers. However, it must be considered that the software consisted of GUI packages, for which the concepts used are easy to grasp in many cases.

Today there is no technology for automatically *understanding* comments or identifiers. A step in that direction was made by Etzkorn et al. [4] with their tool CHRIS, which used a natural language parser. From identifiers and comment sentences CHRIS built a conceptual model of the functionality of each code entity. Their tool was the first to combine information from identifiers and comments into natural language reports of the functionality for each class.

Steensland and Dervisevic [11] did a case study at an international company that develops business software. They strove to improve comprehensibility and translatability of end-user documentation by enforcing style rules based on controlled language theory. Experienced technical writers at the company preferred documentation respecting these rules in 85% of the cases.

In recent years, integrated development environments (IDEs) evolved into assisting developers in writing and maintaining source code documentation. The popular ECLIPSE IDE is an example for this in the JAVA world. It can be configured to keep a list of declarations missing Javadoc or missing some tags. This list representation is useful when developers want to have documentation for all declarations, but it quickly grows unusable if they deliberately decide not to document certain parts of the code.

6. CONCLUSIONS

We introduce several metrics that assess quality features of source code documentation. Our prototype QUASOLEDO analyzes comments in JAVA source code. The tool presents results in space (e.g. for different modules in a project) and over time (the project history). We applied QUASOLEDO to ECLIPSE, an industrial-sized open-source project with almost two million lines of code.

Our findings indicate that there are strong differences in documentation quality across modules, especially for internal and non-internal-packages. This is in accordance with documentation guidelines of the ECLIPSE foundation, which confirms that our metrics are suitable to assess documentation quality.

QUASOLEDO found several strong jumps in documentation quality over the history of ECLIPSE. Commit messages for the changes that caused these jumps indicate that technical writers are responsible for maintaining documentation quality. Overall we found that, after an initial setup phase, documentation quality of Eclipse increases over time.

Our future work will concentrate on increasing the accuracy of our metrics, for example by including documentation inherited from interfaces and superclasses. Apart from that, we also want to integrate QUASOLEDO into an IDE such as ECLIPSE, to provide developers with a quick overview of where documentation needs to be improved.

For ongoing information on the project, see

`http://www.softevo.org/`

Acknowledgments. Our work on mining software repositories is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Thomas Zimmermann is additionally funded by the DFG-Graduiertenkolleg “Leistungsgarantien für Rechnersysteme”. We thank Michael Burch, University of Trier, Germany for providing his Treemap tool to create Figure 4.

7. REFERENCES

[1] L. Cherry and W. Vesterman. Writing Tools: The STYLE and DICTION Programs. Technical report, Bell Laboratories, Murray Hill, N.J., 1981.

[2] J. de Rivires. How to Use the Eclipse API. Technical report, Object Technology International, 2001. Available online at `http://www.eclipse.org/articles/article.php?file=Article-API-Use/index.html`; last visited on February 26th 2007.

[3] W. DuBay. The Principles of Readability. *Costa Mesa, CA: Impact Information*, 2004.

[4] L. Etzkorn, L. Bowen, and C. Davis. An approach to program understanding by natural language understanding. *Natural Language Engineering*, 5(03):219–236, 1999.

[5] L. Etzkorn, W. Jr., and C. Davis. Automated reusability quality analysis of OO legacy software. *Information & Software Technology*, 43(5):295–308, 2001.

[6] A. Forward and T. Lethbridge. The relevance of software documentation, tools and technologies: a survey. *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, 2002.

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

[8] G. Robles, J. M. G. Barahona, and J. L. Prieto. Assessing and Evaluating Documentation in Libre Software Projects. In *Workshop on Evaluation Frameworks for Open Source Software (EFOSS 2006)*, Como, Italy, 2006.

[9] D. Schreck. Quality of Source Level Documentation. Master's thesis, Department of Computer Science, Saarland University, Saarbrücken, Germany, 2007.

[10] I. Sommerville. *Software Engineering, Vol 2: The Supporting Processes*, chapter Software Documentation. Wiley-IEEE Press, 2002.

[11] H. Steensland and D. Dervisevic. Controlled Languages in Software User Documentation. Master's thesis, Linkping University, Department of Computer and Information Science, 2005.

[12] M. Visconti and C. Cook. An overview of industrial software documentation practice. *Computer Science Society, 2002. SCCC 2002. Proceedings. 22nd International Conference of the Chilean*, pages 179–186, 2002.

[13] T. Zimmermann. Fine-grained processing of CVS Archives with APFEL. *Proceedings of the 2006 OOPSLA workshop on eclipse technology exchange*, pages 16–20, 2006.