

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

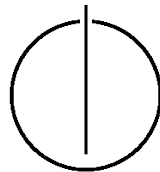
Lehrstuhl IV

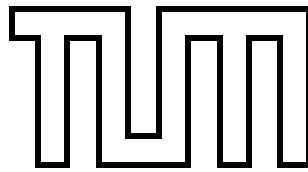
Software & Systems Engineering

Master's Thesis in Computer Science

Quality Analysis and Assessment of Source Code Comments

Daniela Steidl





FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

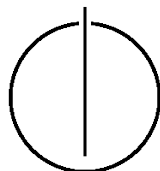
Lehrstuhl IV
Software & Systems Engineering

Master's Thesis in Computer Science

Quality Analysis and Assessment of Source Code Comments

Qualitätsanalyse und -bewertung von Code-Kommentaren

Author: Daniela Steidl
Supervisor: Prof. Dr. Dr. h. c. Manfred Broy
Advisor: Dr. Benjamin Hummel
Co-Advisor: Dr. Elmar Jürgens (CQSE GmbH)
Date: September 20, 2012



I assure the single handed composition of this master's thesis only supported by the declared resources.

München, September 20, 2012

Daniela Steidl

Abstract

A significant amount of source code in software systems consists of comments, *i. e.*, parts of the code which are ignored by the compiler. Comments in code represent a main source for system documentation. They help software developers both to understand code for maintenance or further development but also to use code as a library or framework in other projects. Although many software developers consider comments to be important, existing approaches for software quality analysis ignore system commenting or make only quantitative claims.

In this work, we present a detailed approach for quality analysis and assessment of code comments. The approach groups comments into the categories copyright, header, interface, inline, section, and task comments, as well as commented out code. Based on machine learning, we provide a successful classification algorithm for comment categorization both in Java as well as in C/C++ systems which enables a qualitative analysis of code comments. Furthermore, we give a precise model to define quality of comments in terms of consistency, completeness, coherence, and usefulness: comments should be consistent throughout the entire project, document the system completely, be coherent to the code, and useful for system understanding. The work shows how different quality aspects of the model can be assessed: For comment consistency, we implement a fully-automatic language detection and a copyright format check. To measure the coherence between comment and code, we evaluate three different heuristics with the help of a survey among 16 experienced software developers, and show that the heuristics can be successfully used in practice. Another survey with 20 participants validates two heuristics to judge the usefulness of comments. As a summary of our work, we provide a clearly arranged checklist for comment quality assessment in practice and use it for a comparison of the comment quality in two open source projects.

Contents

Abstract	v
1. Introduction	1
1.1. Problem Statement	1
1.2. Contribution	3
1.3. Outline	4
2. Context of Source Code Quality Analysis	5
2.1. Comment Code or Rewrite it?	5
2.2. Relationship between Code Comments and Code Quality	6
3. Related Work	7
3.1. Existing Research on Code Comments	7
3.2. Software Quality Analysis and Assessment	10
3.3. Source Code Recognition	11
4. Comment Categories	13
5. Quality Model	15
5.1. Activities	15
5.2. Facts and Impacts	16
5.3. Fact Assessment	22
5.4. Global vs. Local Facts	22
6. Comment Classification	25
6.1. Machine Learning Terminology	25
6.2. Training Data	28
6.3. Feature Extraction	29
6.4. Classification Algorithms	32
6.5. Evaluation on Java	33
6.6. Evaluation on C++	36
6.7. Threats to Validity	39
6.8. Summary	40
7. Comment Consistency	41
7.1. Language Recognition	41
7.2. Copyright Structure	45
8. Coherence between Code and Comments	49
8.1. Design of the Questionnaire	49

8.2. Coherence between Interface Comments and Method Names	50
8.3. Length of Inline Comments	57
8.4. Necessity of Comments	62
9. Usefulness of Comments	67
9.1. Design of the Questionnaire	67
9.2. Clarification through Comments	67
9.3. Helpfulness of Comments	69
10. Checklist for Comment Quality	75
10.1. Design of the Checklist	75
10.2. Case Study	76
10.3. Comment Quality in Continuous Quality Management	77
11. Conclusion	83
11.1. Results	83
11.2. Future Work	85
Appendix	89
A. Results of Copyright Consistency Check	89
B. Background Data of Survey Participants	95
Bibliography	101

1. Introduction

*//When I wrote this, only God and I understood what I was doing
//Now, God only knows*
— Karl Weierstrass, German mathematician

A significant amount of source code in software systems consists of comments, *i. e.*, parts of the code which are ignored by the compiler. Comments document source code and help both the original author and other developers to understand code for later modification or reuse; several researchers have conducted experiments showing that commented code is easier to understand than code without comments [1], [2]. The above statement, a famous programming quotation attributed to Karl Weierstrass¹, illustrates the urgent need to comment on the thought behind the code. Comments are the second most-used documentary artifact for code understanding, behind only the code itself [3].

In addition to aiding program comprehension, source code documentation is also necessary for maintenance and is an important part of the general documentation of a system. Relative to external documentation, documentation in source code can be a much more convenient tool for the developer to keep comments and code consistently up to date. Developers widely agree that poor general documentation often leads to misunderstandings [4], and several studies have shown that poor documentation significantly lowers the maintainability of software [5]. Especially for software maintenance, high quality comments are crucial: well commented source code makes it easier for software engineers to grasp general design decisions and understand implementation details in order to perform change requests or bug fixes. Although developers commonly agree on the importance of software documentation [3], commenting code is often neglected due to release deadlines and other time pressure during development.

Many previous approaches to analyzing software quality ignore comments completely, or make only quantitative claims [6], [7]. This is due to several factors hampering the analysis of code comments: in contrast to code, natural-language comments are not bound by a mandatory syntax. Additionally, comments serve a variety of purposes, such as documenting interfaces, referencing copyrights, commenting out code, or containing notes for the developer; each class demands its own treatment. This work can be considered a first approach to a detailed, quantitative and qualitative analysis and assessment of source code comments.

1.1. Problem Statement

Existing analyzing techniques for software quality do not consider quality of source code comments. There are three main reasons why comment quality has been neglected so far.

¹A German mathematician who made major contributions to the field of complex analysis

```
1 //increments x by one
2 x++;
```

Figure 1.1.: Example of a useless source code comment which explains the obvious

```
1 /**
2  * @return the name
3  */
4 public String getName() {
5     return name;
6 }
```

Figure 1.2.: Example of a comment intending to explain the interface but explaining the obvious

First, every comment has the same syntactic pattern, regardless of its purpose: In previous work, comments are only identified with a set of special characters that individual languages offer to mark the beginning and end of a comment. In Java, *e. g.*, “//” is used for a single line comment, and “/**...*/” is used for multiple lines. However, comments serve an array of intended purposes: They can be used to give an overview of a class, describe the functionality of a method or an attribute, convey details about a single implementation aspect, list remaining tasks, or declare copyright information. Developers also comment out code for debugging purposes or potential later use, and without deep analysis such code is indistinguishable from genuine comments. Many previous approaches make a quantitative claim about the comment ratio of the system, counting the number of lines containing a comment, divided by the total number of lines of code in the system [6], [7]. This metric, however, is too simple to be meaningful. It does not take into account that some comments, such as copyrights or commented out code, do not promote system understanding or enhance the system documentation quality.

Second, previous research has not made an attempt to give a complete and comprehensive model of comment quality. In general, comment quality has only been defined by the expectation of the software developers: That is, a comment is considered to be of high quality if it helps them to understand code. Developers expect comments to give them insights about the functionality of the code and about implementation details which are not obvious from the code itself. Intuitively, they consider both comments in Figures 1.1 and 1.2 to be useless, as they do not provide information beyond the code. While the first example comment is fabricated, the second was frequently seen during our analysis of open source code. However, beyond developers’ subjective expectations, a precise definition of comment quality does not exist. Coding conventions for various programming languages or guidelines on how to write good code may marginally touch on the topic of commenting code but mostly lack depth and precision. The Java Coding Conventions [8], *e. g.*, state that “discussion of nontrivial or non-obvious design decisions is appropriate” but also advise to “avoid duplicating information that is present in the code”. In practice, the granularity of this statement is too coarse for real applications.

Third, there are currently no automatic or semi-automatic methods for comment quality assessment. Even if an unambiguous standard of quality for code comments existed, developers would be free to ignore it as long as no assessment methods were available. Assessing comment quality is difficult as comments comprise natural language and have no mandatory format aside from syntactic delimiters for the compiler. Thus, the algorithmic solutions proposed by this work will be heuristic in nature.

1.2. Contribution

This work presents a first approach to a detailed analysis and assessment of comment quality in source code. We consider the assessment of comment quality as part of a software quality audit, where a group of quality engineers analyzes the quality of a software system.

First, we tackle the problem of differentiating between comments with different purposes: based on a machine learning approach, we perform comment categorization, comparing various different classification algorithms to achieve the best result. The classification model learned is successfully applied both for Java and C/C++ systems. With the help of comment categorization, we can make quantitative claims about how many comments potentially contribute to system understanding and how many serve other purposes. In addition, comment categorization creates the possibility to analyze different categories with different approaches, tailored to their individual purposes.

Second, we define a comprehensive quality model for code comments, with the main goal of capturing the relationship between the activities of the developers and the attributes of the comments. By revealing the direct impact of comment attributes on developers' activities, the model is descriptive and understandable. The model groups quality aspects based on four major quality criteria: consistency throughout the project, completeness of system documentation, coherence with source code, and usefulness to the reader. The model enforces different requirements for different comment categories, differentiating between fully-automatically, semi-automatically, and manually assessable facts.

Third, we provide algorithms and heuristics to enforce the model in practice. To assess comment consistency, we use language recognition to detect whether comments are written in the same language throughout the project, and we also check for copyright format consistency. For semi- and fully-automatically assessable facts regarding coherence and usefulness, we implement heuristic approaches and evaluate them with a survey among experienced software developers. The survey contains three independent parts with 16 to 20 participants each. For coherence, we measure whether comments describing a method are informative or missing content. For example, this heuristic will detect the comment in Figure 1.2 to be trivial because it only repeats the method name. We further reveal that the length of comments can be used as an indicator to detect comments that can be deleted or to find lines of code that should be extracted into a new method. The length also implies whether the comment contains local or global information. The survey shows that comments containing the word *nothing* are either unnecessary or necessary but not sufficiently coherent to the code. For an assessment of comment usefulness, our heuristics use the presence of question or exclamation marks to detect comments which are confusing or not helpful.

As a summary of our assessment methods, we provide a checklist for comment quality intended to be used for one-time audits but which may also aid continuous quality management. We conduct a case study to demonstrate the effective use of the checklist for comment quality assessment in practice, and compare the comment quality of two open source projects.

In contrast to previous work, our work is the first to approach the assessment of comment quality in a thorough manner, providing both an expressive model of comment quality as well as evaluated methods for the reinforcement of the model in practice.

1.3. Outline

The remainder of this work is organized as follows: Chapter 2 generally discusses an analysis of comments in the context of a quality analysis of source code. Chapter 3 presents related work. Different comment categories are described in Chapter 4. Afterwards, we define the quality model for code comments in Chapter 5 and present a machine learning approach for comment classification in Chapter 6. The remaining chapters present our approach for different aspects of the quality model: consistency (Chapter 7), coherence (Chapter 8), and usefulness (Chapter 9). Chapter 10 provides a checklist for comment quality, and shows its use in a case study with two open source projects. Chapter 11 summarizes the results of this work, and presents directions for future research.

2. Context of Source Code Quality Analysis

Before approaching quality analysis of code comments, a fundamental question is whether the analysis is seen in the context of a source code quality analysis or independently from it. This chapter debates the question by shedding light on an ongoing debate whether code should be commented at all (Section 2.1) and how source code and comment quality relate to each other (Section 2.2).

2.1. Comment Code or Rewrite it?

Don't comment bad code – rewrite it.
— Brian W. Kernighan and Phillip J. Plauger¹

The only truly good comment is the comment you found a way not to write.
— Robert C. Martin²

Among the community of software engineers, there are few people arguing that clean code does not require any comments because it is self-explanatory. Quotations like the two above are not uncommon. This point of view is motivated by the ideas of Extreme Programming (XP), emerging from the Agile Software Developing movement: Based on the DRY (Do not repeat yourself) paradigm, comments are considered to only repeat information that should be contained in good code already, *e. g.*, in expressive identifiers. Furthermore, comments are believed to be risky to get out-of-date or inconsistent. According to [10], comments are an inevitable sign of bad code. Hence, instead of commenting bad code, developers should spend the effort on rewriting the code. Interestingly, also the Java Coding Conventions [8] state: “The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.”

Although clean code simplifies the process of understanding code, comments are nevertheless indispensable. We believe that the usefulness of comments can not be judged without differentiating between various kinds of comments (see Chapter 4). Even though code might be of high quality, we would still expect developers to write a header comment of a class to give a short and precise overview of the class in natural language. It is less time consuming to glance at the header as opposed to reading through the entire source code of a class. Also, Ron Jeffries, one of the three XP-founders, states online: “In XP, we strongly recommend class comments, which give that overview that can be hard to pick up in browsing.”³ He makes a distinction between class comments, *e. g.*, comments preceding a class definition and other comments, and agrees that class comments can be a consolidated representation of information which is not explicitly given by code.

¹Citation from [9]

²Citation from [10]

³<http://c2.com/cgi/wiki?ToNeedComments>, last accessed on 06/05/2012

Even Martin, who claims that the only good comment is the one you did not write, admits that good comments exist when they explain the intent of a developer, give a warning of consequences, or “amplify the importance of something that may otherwise seem inconsequential”². Furthermore, Fluri et al. [11] invalidate the fear of inconsistent comments. In their work, the authors show that in 90% of the cases in their study, changes to source code simultaneously result in an adaptation of the corresponding comment (see Chapter 3 for more details). Hence, comments are usually kept up-to-date with evolving code in practice.

We believe that plenty of information which other developers depend on can not be expressed in pure source code, even when the code is clean. Such information includes but is not restricted to the purpose of a class within the entire system or pre- and postconditions of a method call. Non-trivial public methods need to be commented for users who do not have the source code at hand but who depend on information about purpose and intention of the method. Comments can also express non-obvious design decisions and, beyond that, the reasons causing these decisions.

2.2. Relationship between Code Comments and Code Quality

Commenting code is necessary even when the code is clean. However, we are also aware of some comments being strong signals for low code quality. From this point of view, we partially agree with the quote from the Java Coding Convention as above (“The frequency of comments sometimes reflects poor quality of code”). For example, if every line of code needs a comment to be understood, this might indicate that the code should better be cleaned up. Furthermore, low code quality is, for example, reflected by classes which are too long to be easily understood. When classes are getting too long as programmers add more and more functionality, developers use section comments to compensate the lack of structure. Section comments provide an overview of the next few methods and can be compared to a light version of a content table. Instead of grouping few methods into one section, these methods should be extracted and define a separate class.

For the purpose of comment quality analysis, it makes a fundamental difference if the quality of the underlying source code is given or potentially improvable. In other words, analyzing the comment quality in the context of an analysis of code quality will lead to a different result as opposed to analyzing the comment quality on its own: If the code quality has to be taken for granted, section comments nevertheless help to provide some structure for the class. Contrarily, if comments are analyzed together with the source code quality, section comments should be removed and class methods distributed over several smaller classes.

For this work, we make the underlying assumption that the given source code is not invariant but can be changed. Hence, we perform code comment analysis in the context of source code analysis. This work will use comments as an indicator for low source code quality to give refactoring recommendations.

3. Related Work

Quality analysis and assessment of code comments can be connected to many existing research areas. This section gives an overview of previous research about code comments (Section 3.1). It also provides insights about general quality analysis in software engineering relevant to this work (Section 3.2), and touches the topic of source code recognition (Section 3.3) with respect to our machine learning approach for comment classification.

3.1. Existing Research on Code Comments

Although no all-encompassing quality model of code comments has been developed so far, previous research has made attempts to investigate the role of comments in source code. Subsection 3.1.1 shows how quality of code comments has been evaluated previously, Subsection 3.1.2. deals in particular with task comments. Subsection 3.1.3 contains previous work about the coherence between comment and source code and Subsection 3.1.4 investigates whether code and comments evolve simultaneously.

3.1.1. Existing Quality Analysis of Code Comments

In [12], Khamis et al. provide an analysis tool called JavadocMiner for analyzing the quality of inline documentation. Thereby, the authors mainly focus on inline-documentation in the form of Javadoc comments. By using a set of heuristics, they aim to evaluate both the quality of the language and the consistency between source code and its comments. The quality of the language is measured with heuristics such as counting the number of tokens, nouns, and verbs, calculating the average number of words, or counting the number of abbreviations. With heuristics including the Fog index or the Flesch reading ease level, the authors target the readability of comments. For detecting inconsistencies between code and comments, this approach computes the ratio of identifiers with Javadoc comments to the total number of identifiers and checks whether all aspects of a method such as parameter or return type are documented. Furthermore, the SYNC heuristic finds return types, parameters, and thrown exceptions that are no longer up-to-date, *e. g.*, due to changes in the code. In a case study on different releases of ArgoUML and the IDE Eclipse, the authors observe that the modules with the highest quality code comments also have the least amount of reported defects. Also vice versa, the modules with the lowest quality documentation contain the most bugs.

This paper targets the same research questions as we do, namely how to measure the quality of code comments. However, this approach uses very basic heuristics. It neither differentiates between different comment types nor detects any inconsistencies between code and comments beyond structural requirements of Javadoc.

3.1.2. Analysis of Task Comments

Storey et al. [13] focus on an empirical study to explore which role embedded task annotations play in source code. They gather data from a survey among software developers, from code analysis of open source projects, and from personal interviews. They found out that task comments are frequently used in software development with the majority of comments containing @TODO tags. Todo comments mainly serve the purpose of documenting small tasks when opening a new bug report seems to be an overhead. However, there is the potential risk that developers never revisit task comments. Based on their analysis, the authors suggest several implications for tool designer, such as providing a filtering mechanism for task views, supporting meta data within task comments, or introducing ad hoc task clean-up wizards.

In addition to Storey et al., Ying et al. [14] also investigate the role of task comments. In a preliminary study they manually analyze task comments in AWB, an internal IBM code base. The authors interpret the intention of task comments, resulting in a detailed categorization of task comments such as bookmarks on past tasks, current tasks, future tasks, pointers to change requests, or tasks used for communication. However, there is no automatic or semi-automatic assessment of task comment quality.

In our work, task comments are part of our comment categorization. Contrarily, we do not focus on task comments in particular but provide a general assessment of comment quality which also includes other comment categories.

3.1.3. Coherence between Comments and Source Code

Tan et al. [15] explore the feasibility and benefits of an automatic analysis of comments to detect bugs in code and bad comments. According to the authors, a significant percentage of comments relates to hot topics, *e. g.*, memory allocation and synchronization. With a simple keyword search (“lock”, “alloc”, “signal”, “thread”), they detect hot comments in different Linux modules. A preliminary study analyzes synchronization-related comments in Linux based on a combination of natural language processing and topic-specific heuristics. With their technique, the authors are able to detect 12 bugs in Linux, two of them being confirmed by developers. This position paper convincingly reveals the need for an automated comment analysis. However, it is tailored to the specific topic of synchronization. In contrast, our approach analyzes comments independent from the context.

Lawrie et al. [16] use information retrieval techniques for function quality assessment under the guiding assumption that “if the code is high quality, then the comments give a good description of the code”. For each function and the corresponding comment the authors create a vector space model and use cosine similarity between comments and code to score functions. Functions will then be grouped according to their score. If the function quality assessment was successful, then functions within the same group will be of similar quality. The authors divide comments into two groups: Comments describing internals of a function for maintenance and comments providing information for external users. They only validate their approach with the first group of comments. The authors claim that the latter group will not have significant overlap with the code and hence result in low cosine similarity. Results are validated with an empirical case study, in which participants rate functions based on their comprehensiveness and overall quality. The authors conclude

that their tool measures relative function quality similar to human judgment. Similar as in this work, we also investigate the similarity relation between source code and comments. However, we compare the relation between comment and method name and hence focus on the group of comments that has been ignored by this work. We also deal with the question if comments can be used as an indicator for source code quality. We highly doubt though that the underlying assumption (high code quality implying descriptive comments) holds true in the general case. Contrarily, we believe that comments can rather be an indicator for low source code quality.

Furthermore, [17] and [18] also use information retrieval techniques to recover traceability links between code and documentation. Thereby, comments are considered to be part of general documentation together with requirement and design documents, user manuals etc. However, they do not play a major role in these approaches, which focus on the general relation model between code and free text documentation and not on the analysis of code comments in particular.

3.1.4. Evolution of Code and Comments

Jiang and Hassan [19] study the evolution of comments over time in the PostgreSQL project. They claim that developers commonly change code without updating its associated comments and that uncommented interfaces or interfaces with outdated comments are likely to cause bugs. For their study, the authors provide a coarse categorization of comments into header and non-header comments. Header comments are written prior to a function definition, whereas all other comments inside a function body or trailing the function are non-header comments. The authors monitor the percentage of functions with header comments, assuming that a drop over time indicates that developers are not updating the interface documentation. However, the study reveals a constant percentage of commented functions except for early fluctuation due to the commenting style of a particular active developer. The paper provides a first categorization of comments. However, our work will provide a more detailed comment categorization along with an automated classification algorithm.

Similar as Jiang and Hassan, Fluri et al. [11] expect that code and comments are not necessarily updated at the same time. Therefore, the authors investigate how code and comments evolve. They use a mapping between code and comments to observe their co-evolution over multiple versions of the system. The authors conduct a case study on three different Java projects, version-controlled with CVS. The case study, in contrast, reveals that among all comment changes triggered by source code changes, about 97% are done in the same revision as the source code change. Furthermore, approximately 70% of comments are mapped to one of the following seven types of source code entities: attributes, class and method declarations, control structures, loops, method calls, and variable declarations. The authors also evaluate the ratio between comments and source code over time to give a trend analysis whether developers increase or decrease their effort on code commenting. However, the results differed greatly among the three test cases such that no unique answer can be given. In particular, the authors did not differentiate between different types of comments, *e. g.*, lines of commented out code were also counted in the ratio between comments and source code. We highly suspect that this leads to a useful metric to assess the effort of code

commenting. Commented out code should be excluded in this metric because it does not provide any information gain for system understanding.

3.2. Software Quality Analysis and Assessment

Quality of software systems is one of the key research areas in software engineering. A variety of approaches has made proposals about models, metrics, and tools to analyze and assess the quality of a software system. In the following, we will present few approaches relevant for our work: Some of them had a direct influence as we transferred their ideas to our domain. Others reveal the present insufficient analysis and assessment of code comments and therefore show the necessity of our work.

3.2.1. Quality Models for Maintainability

Deissenböck et al. propose an activity based quality model to assess software maintainability [20]. This model has been the underlying basis of our model for comment quality, see Chapter 5. In contrast to other maintenance quality models such as [21], this model differentiates between maintenance activities of the developer and properties of the system. The model explicitly states the relation between properties and activities, resulting in a two-dimensional maintainability matrix. The authors represent system properties as facts, consisting both of an entity and an attribute. Facts can have positive or negative impacts on activities, which are explicitly described. Based on this separation of concerns, facts are the model elements that need to be assessed to measure the system's maintainability. However, the model differentiates between facts that can be assessed fully automatically, that require manual activities, or a mixture of both. We apply the design of this model to our quality model of code comments, also defining facts, impacts, and activities.

3.2.2. Code Comments in Assessment of Software Maintenance

In previous work, maintenance productivity has been studied extensively. However, the quality of code comments plays only a minor role in assessing a software system's maintainability. It is a commonly accepted fact that poor documentation constitutes a major problem affecting software maintainability [4]. As software is often maintained by people who did not develop it, poor documentation can cause a variety of effects - ranging from an increase in time to understand and maintain a software to a complete redesign and rebuilt of the system. In a worst case scenario, it is easier to rebuild a system completely than to understand and modify an existing one.

In [6], Garcia et al. analyze costs and benefits of maintainability. Thereby, maintainability is related to understandability, modifiability, and testability. In order to measure those concepts, the authors apply several metrics, among them lines of source code including comments, lines of comments, lines of easy modification, and lines of error detection. The number of lines of comments is thereby considered to be an understandability measure. On the one hand, this indicates the importance of code comments for software maintenance. On the other hand, the number of lines of comments (LC) can not be a sufficient metric to assess understandability: Commented out code or copyrights, for example, increase the

number of lines of comments without contributing to understandability. Hence, analyzing maintainability with metrics requires a more detailed analysis of code comment quality.

In a similar way, the authors of [7] use the number of lines of comments to calculate metrics assessing a software system's maintainability: For system commenting characteristics, they define the overall program commenting ratio as a 2-tuple, containing the percentage of comment lines in the whole program and the percentage of modules with header comments. For component commenting they measure intramodule commenting as the number of lines with comments divided by the total number of lines in the module, averaged over all modules. However, again, only measuring the number of comment lines can not be sufficient as these lines can contain any arbitrary content not contributing to understandability.

Both the work of [6] and [7] lack a differentiation between comments with different purposes. Our machine learning approach is able to classify different comment categories and hence helps to better identify those number of lines of comments that potentially promote understandability.

3.2.3. Coding Conventions

For almost every programming language, there are coding conventions to define general standards for writing source code. Either tool providers (such as Sun Microsystems [8]) or companies provide such conventions to explain the *dos and don'ts* for programmers in order to achieve high quality source code. However, these conventions are only recommendations for the developer instead of enforced requirements. Coding guidelines also suggest how to comment source code, mostly, however, in a very unspecific manner.

The Java Coding Conventions [8] differentiate between two types of comments, implementation comments and documentation comments: "Implementation comments are meant for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective, to be read by developers who might not necessarily have the source code at hand." This idea corresponds to our quality model in Chapter 5, where developers' activities are subdivided into developing code and using code. Furthermore, the guidelines state several purposes of code comments: Giving an overview of the code or providing additional, nontrivial, and non-obvious information about code, implementation, and design decisions. This categorization resembles our comment categorization in Chapter 4, which is, however, more precise. Although the Java Coding Conventions give recommendations on how to write a good comment, these recommendations are currently not assessable to be reinforced during development. Our approach, by contrast, provides a quality model combined with various algorithms and heuristics to assess comment quality fully or semi-automatically.

3.3. Source Code Recognition

For the purpose of this work, comments containing natural language and comments containing code need to be differentiated, for more details see Chapter 4 and 6. The following section presents previous research that has made attempts to recognize code snippets within text.

3.3.1. Machine Learning for Code Recognition

In [22], Tang et al. aim to clean email data with non-text filtering and text normalization methods. Non-text filtering thereby includes program code filtering besides header, signature, and quotation filtering. In order to recognize program code, the authors use support-vector machines (SVMs) to identify the start and end line of a piece of code. The support-vector machine is based on ten feature categories such as statement keywords (*e. g.*, like “i++”, “if”, “switch”, “while”), equation pattern features (“≤”, “ $a = b + c$ ”), function pattern, or function definition features. The machine learning approach leads to a precision of 92.97%, with a recall of 72.17%, and a F1-measure of 81.26%. The authors state that the precision of the model is high but the recall needs to be improved. They claim that “it is hard to find general patterns for the task” of program code detection. In our work, we apply similar machine learning techniques but achieve a higher recall while maintaining similar precision (see Chapter 6).

3.3.2. Extracting Source Code from Emails

The work in [23] focuses on analyzing email data to support program comprehension activities. According to Bacchelli et al., email data gathered from various developer email lists provides an alternative and complimentary view of a software system. Their approach classifies emails containing fragments of source code and extracts the source code pieces inside the email. In contrast to the work of [22], this algorithm is based on fast and lightweight techniques such as regular expressions and pattern matching which exploit syntactical characteristics of the Java programming language. To classify emails containing code fragments as well as to detect single lines consisting of code, the authors count the number of special characters (semicolon, curly brackets) and keywords (public, static) occurrences and empirically determine a threshold for the classification. As precision and recall were rather low with this approach, they improved the results by measuring the occurrence of lines characters (semicolon and braces) in combination with a regular expression matching on the method call pattern of Java. They achieve a precision of 94% and a recall of 85%. They claim to perform better than the more sophisticated machine learning approach of Tang et al. because Tang only achieved a recall of 72% with comparable precision. Precision and recall of this approach are approximately the same as for our approach. In contrast, we detect code snippets among comments and not among email data.

4. Comment Categories

For the purpose of this work, we differentiate between seven different types of comments: copyrights, headers, interface comments, inline comments, section comments, commented out code, and task comments. The following list gives a definition for each comment category and explains its purpose:

- **Copyright:** A copyright comment includes information about the copyright or the license of the source code file. Copyright comments are usually found at the beginning of each file.
- **Header:** A header gives an overview about the functionality of the class. In addition, it can provide information about the author of the class, the revision number, the peer review status etc. In Java, header comments are mostly found after the imports but before the class declaration.
- **Interface:** An interface comment describes the functionality of a method or a field. Interface comments are therefore located either before a method/field definition or in the same line as a field definition. They can provide information for the developer and be used for an API of the project.
- **Inline:** Developers use inline comments to comment on code within a method definition. Inline comments describe implementation decisions or other details.
- **Section:** A section comment summarizes a larger part of a class that covers one functional aspect. It usually addresses several methods (or fields) together which belong to the same functional aspect.
- **Code:** Commented out code is source code that developers want to be ignored by the compiler. Often code is temporarily commented out for debugging purposes or for potential later reuse.
- **Task:** A task comment is a note for the developer about an unfulfilled task. It either contains a remaining todo, a note about a bug that needs to be fixed, or a remark about an implementation hack.

Some literature also uses the term class comments. In our definition, class comments belong to the category of header comments.

Figure 4.1 shows an example code with comments illustrating each category.¹ Each comment is thereby annotated with a preceding note colored in yellow, stating the corresponding category.

¹Code taken from the book “Java in a Nutshell” by David Flanagan but modified and extended with more comments.

```

1  Copyright
2  // This example is from the book _Java in a Nutshell_ by David
3  // Flanagan. Written by David Flanagan. Copyright (c) 1996
4  // O'Reilly & Associates. For Demonstration purposes, source
5  // code was modified and comments were added by Daniela Steidl.
6  import java.applet.*;
7  ...
8  Header
9  /**
10 * This applet displays an animation. It doesn't handle errors
11 * while loading images.
12 */
13 public class Animator extends Applet implements Runnable {
14
15     Interface
16     /** the current image */
17     protected int current_image;
18
19     Section
20     /** *****
21      * Methods to start and stop the applet          *
22      * ***** */
23
24     Interface
25     // Read the basename and num_images parameters.
26     // Then read in the images, using the specified base name.
27     public void init() {...}
28
29     public void stop() {
30         Code
31         // animator_thread.sleep();
32         if ((animator_thread != null) && animator_thread.isAlive())
33             animator_thread.stop();
34         Inline
35         // We do this so the garbage collector can reclaim the Thread
36         // object. Otherwise it might sit around in the Web browser for
37         // a long time.
38         animator_thread = null;
39     }
40
41     public void run() {
42         Task
43         // @TODO: add functionality to launch animator in a separate
44         window
45         while(true) { ... }
46     }

```

5. Quality Model

In order to analyze and assess the quality of source code comments, we need a precise definition of comment quality. Previous research has not made an attempt to specify a quality model for code comments, only guidelines on how to write good code marginally give recommendations on how to write a good comment. However, they do not provide a model that can be evaluated by a clear set of rules. Consequently, these recommendations are not strictly reinforced during a development process.

In this work, we introduce a quality model that specifies in detail the purposes of a comment to help the developer understand code. Our quality model closely follows the guidelines for quality models in maintenance as proposed by Deissenböck et al. in [20]. The two-dimensional model differentiates between *activities* performed by the developer and *system properties*, also called *facts*. This separation of concerns reveals the influence of system properties on activities and explicitly shows how improving certain facts makes specific aspects of understanding source code easier. Thereby, both activities and facts are organized in a hierarchical tree structure with an increasing level of detail from the root node to the leaves: The root indicates the most general aspect, the leaves the most specific ones. In the following, Section 5.1 describes the actions involved in understanding source code. Section 5.2 defines facts of the system and gives an overview of impacts of facts on activities. Section 5.3 describes fact assessment and Section 5.4 shows that facts can have both global and local impacts.

5.1. Activities

Commenting source code aims at one main goal: Making it easier to understand the source code for someone who did not write it (see also [24], p. 23) or for the code author who wants to modify and reuse his own code later. Consequently, the main activity in our model is *source code understanding* and used as root node in the activity tree (Figure 5.1). Understanding source code has two distinct purposes: It is necessary to either further develop the code as a programmer (*Developing Code*) or to use the code within a different program, *e. g.*, as a library call (*Using Code*). In the latter, the source code is usually not available, so documentation is the only way to provide information about the system. Splitting the main activity of understanding code into these two subactivities corresponds to the categorization of comments by the Java Coding Conventions [8] into “Implementation” and “Documentation”. On the next level of detail, both activities can be further differentiated: In order to use code, a developer is particularly interested in calling public methods and accessing public fields, understanding the overall system design, and knowing about copyrights and authorships. In contrast, in order to develop code, understanding implementation and design details, as well as knowing tasks, hacks, and potential bugs play an important role. Furthermore, the developer wants the code to be easy to read in terms of appearance and structure.

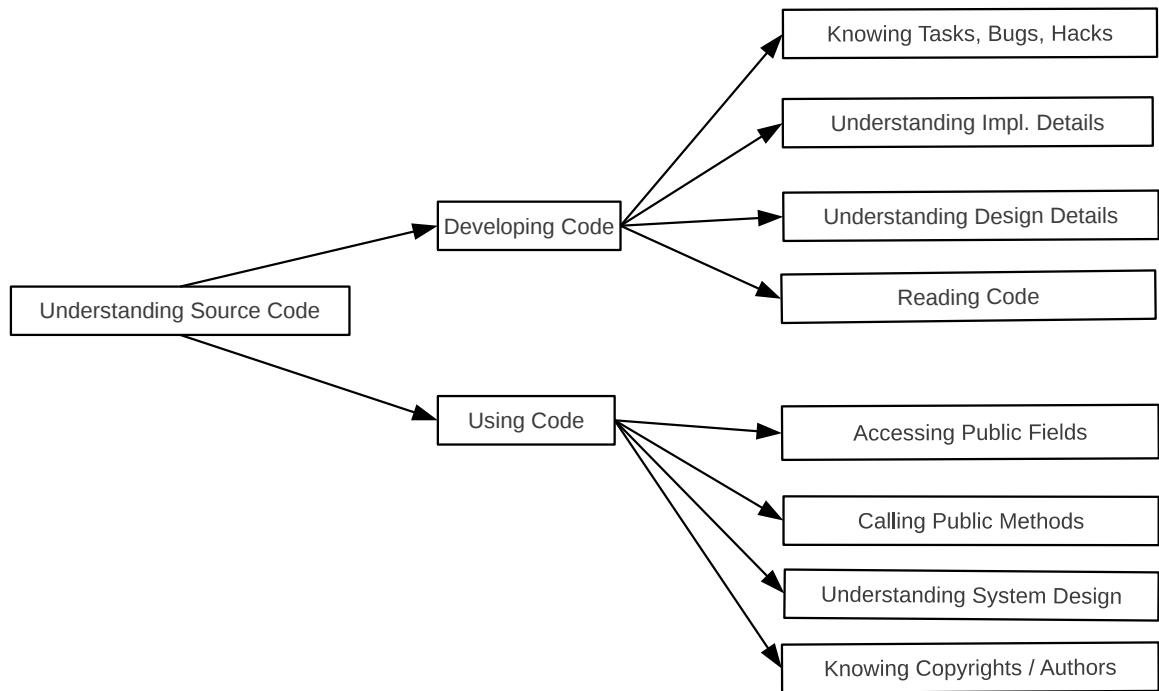


Figure 5.1.: Activities in the quality model

5.2. Facts and Impacts

Besides activities as one dimension, facts define the second dimension of the model. Thereby, facts consist of *entities* and *attributes*. We will first specify the entities and their general influence on activities. Later, we define specific attributes for each entity, which results in facts with either a positive or negative impact on the activities.

5.2.1. Entities

The entities of the model (Figure 5.2) correspond to the comment categorization as introduced in Chapter 4. The influence of entities on different activities is visualized in Figure 5.3 in the form of a two-dimensional matrix with entities as rows and activities as columns. An entry in one cell indicates that an entity has an *impact* on the corresponding activity, whereas impact can be both positive and negative. We will give an overview of the impacts in the following, and describe them in detail later. Obviously, copyright and header comments help the user to be aware of copyrights and authorships. In addition, headers mainly serve the purpose of giving an overview of the class functionality and therefore help understanding the system design. Inline comments describe details of the implementation and possibly promote readability of code by providing structure. Interface comments serve a larger variety of purposes: On the one hand, they help to understand the general system design as well as design details by providing information about the functionality of a method or a field. On the other hand, they describe in detail how to use the code, *e. g.*, how to call public methods or how to access public fields. Task comments inform the developer

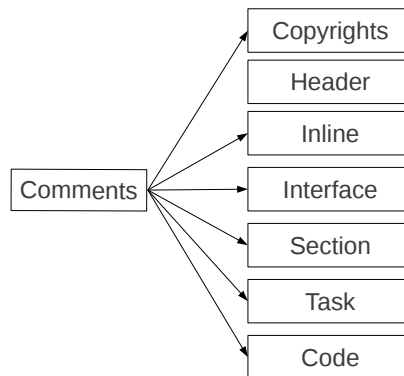


Figure 5.2.: Entities in the quality model

about remaining tasks, hacks, or potential bugs. As already briefly discussed in Section 2.2, section comments influence the readability of code and also the ability to understand design details. The question whether the impact of section comments is positive or negative will be addressed later. Commented out code hinders the ability to smoothly read through the code and also negatively influences the understandability of implementation details because it is not obvious whether commented out code is still needed. To sum it up, for understanding the functionality of the code, header, inline, and interface comments are the most beneficial.

5.2.2. Attributes

A practical model consists of comprehensible facts, associating each entity with one or more attributes. Creating a precise model inevitably leads to decisions that are debatable. Nevertheless, these decisions are to be made in one way or the other and we will justify our point of view as much in detail as possible.

We group entities and corresponding attributes according to four major quality criteria: *Consistency*, *completeness*, *coherence*, and *usefulness*. Consistency facts describe certain features of comments that ought to be consistent through the entire system. Also completeness covers global aspects of system commenting by reinforcing comments at certain positions, such as placing a copyright in each file. Coherence covers aspects of how comment and code relate to each other and hence deals with local aspects of a single comment. Usefulness describes properties that make a single comment necessary for system commenting.

The following list describes each fact, formatted as [Entity | ATTRIBUTE] and explains the impact on different activities. If an attribute applies to all entities, the entity is denoted with *All*. A visualization of the model is given in Figures 5.4 and 5.5. In order not to overload a single picture, the visualization is split up into two figures.

Consistency

- [All | CONSISTENT LANGUAGE] All comments should be written in the same language throughout a project. Switching between different languages while reading code requires needless additional attention. Therefore, we claim that the readability of code improves if all comments appear in the same language.

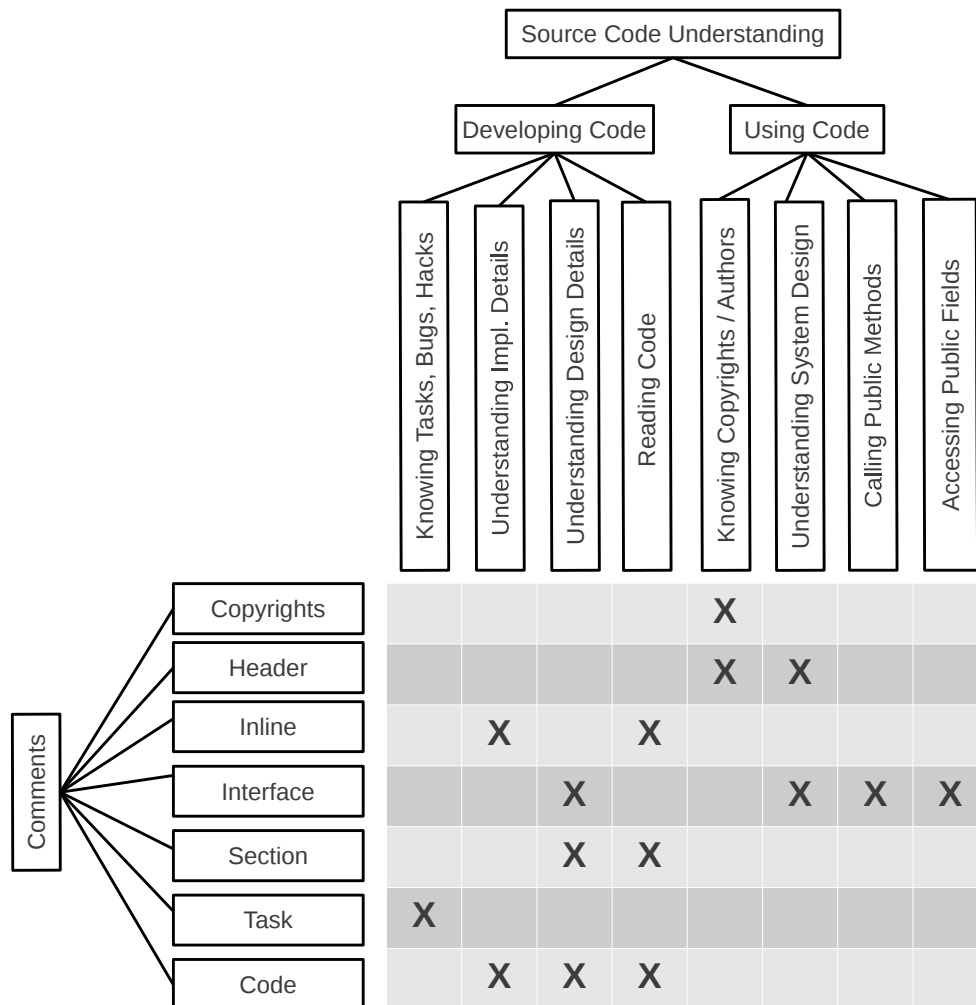


Figure 5.3.: Relation between entities and activities

- [Copyright, Header | CONSISTENT STRUCTURE] For one project, each file should be under the same copyright. In particular, copyrights are to be written in a consistent format. Additionally, every header should also be in a consistent format. Consistent copyrights and header have an positive impact on the activity of knowing copyrights and authors.

Completeness

- [Copyright, Header | EXISTENT IN EVERY FILE] Every file should have both a copyright and header file to provide complete information about license and authorships.
- [Interface | EXISTENT FOR EVERY PUBLIC DECLARATION] Every public declaration should be commented to provide sufficient information for a user who does not have the source code at hand. In such a situation, interface comments are the only source of information and should therefore cover the public interface of a class com-

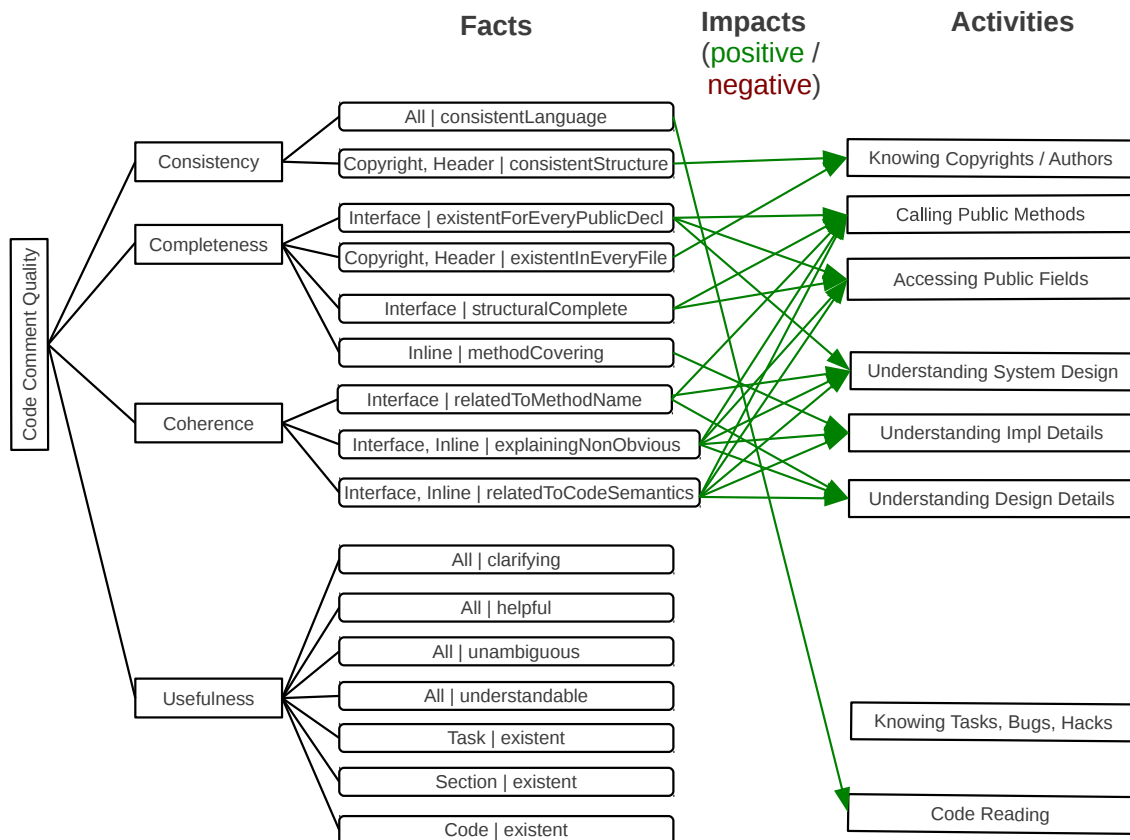


Figure 5.4.: Impacts between facts and activities (part one)

pletely. Complete interface documentation promotes calling public methods, accessing public fields, and understanding the system design as well as design details.

- [Interface | STRUCTURAL COMPLETE] Interface comments for public method declarations ought to be structural complete: All parameters, the return type, possible exceptions etc. should be stated and explained, making the use of public methods and fields easier.
- [Inline | METHOD COVERING] Depending on the quality of the code being granted or not (see the discussion in Section 2.2), the impact of inline comments varies. If the code quality can not be changed, it is useful to have comments in methods which are very long or have a high nesting depth. Hence, inline comments should cover the entire method in length and nesting depth to understand implementation details. In practice, it remains to be determined manually how many lines of code an inline comment should cover. As a useful threshold varies for different programming languages, we do not define a general threshold value. If the code quality is assessed at the same time as the comment quality, methods being too long should be split into several smaller methods and commented with an interface comment instead. In the case of an invariant source code, inline comments have a positive impact on understanding

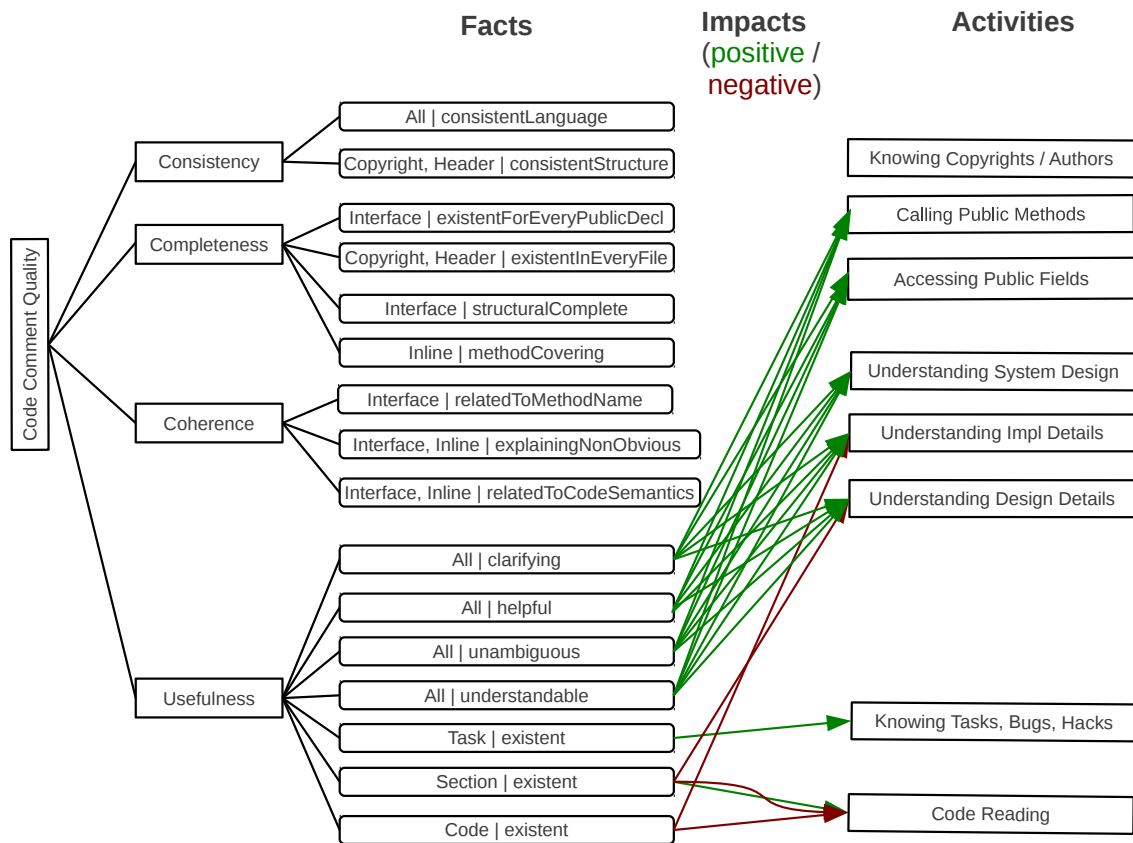


Figure 5.5.: Impacts between facts and activities (part two)

implementation details. If the source code can be adapted, they would be evaluated negatively because they indicate necessary refactoring activities.

Coherence

- [Interface | RELATED TO METHOD NAME] Interface comments should be related to the method name as this is a strong indicator that the comment is up to date and that a meaningful method identifier was chosen. This has a positive impact on calling public methods and understanding the system design as well as design details.
- [Interface, Inline | EXPLAINING NON-OBVIOUS] Developers expect interface and inline comments to explain the non-obvious and provide information that is not already contained in the code. They can give the reasons and the intention of certain decisions made during the development process which can not be expressed in pure source code. Interface comments in particular should provide more information than just repeating the method name. Comments explaining the non-obvious greatly, positively impact understanding the system design and implementation/design details as well as using public methods and fields.
- [Interface, Inline | RELATED TO CODE SEMANTICS] As a minimal requirement, interface and inline comments should be related to the code semantics: Their infor-

mation should correspond to the semantics of the code and not contradict it. As the previous fact, this fact helps for all understanding activities as well as for accessing methods and fields.

Usefulness

- [All | CLARIFYING] The comment should clarify the intent of the source code. It should make source code understanding easier and therefore not create additional confusion. Clear comments help to understand implementation and design details, the system design, and also to call and access public methods and fields.
- [All | HELPFUL] In general, readers of the code should perceive the comment as helpful. If source code understanding was not harder with the comment being deleted, the comment would not be helpful. Helpful comments make all activities about understanding and using code easier.
- [All | UNAMBIGUOUS] Comments should be unambiguous to provide additional precise information beyond the code. As before, unambiguous comments make all activities about understanding and using code easier.
- [All | UNDERSTANDABLE] The most fundamental requirement for a comment is to be easy to understand. If a developer can not understand a comment, he could have abstained from it anyway. Well-written comments help to understand implementation and design, and also to call or access public methods or fields.
- [Task | EXISTENT] If task comments exist, they help developers to be aware of further todos, unpleasant hacks, or potential bugs. For source code releases, however, they can also indicate that the code has not been completely cleaned up yet.
- [Section | EXISTENT] As discussed in Section 2.2, the assessment of section comments depends on the context of the quality analysis: The existence of section comments usually strongly indicates that the class is too long, hence, implying that the design of the class is hard to grasp. In addition, they hamper the code readability because the typical elaborate typographical display of section comments distracts from the code. Consequently, if the comment quality analysis is embedded into a code quality analysis, section comments should be removed and the class should be split up into several classes instead. On the other side, if the comment quality analysis is performed on its own and the code quality has to be taken for granted, section comments nevertheless provide some sort of structure for the class and hence improve the code readability: the code is hard to read anyway but section comments at least provide points of orientation.
- [Code | EXISTENT] Commenting out code should be avoided in general once the code was released. As previously noted, commented out code hinders the ability to smoothly read through the code if the developer has to scroll over large amounts of unused code. It further hinders understanding implementation details, as a new developer can not be sure if this code is just for debugging purposes, if it is still needed, or if it even contains important information.

5.3. Fact Assessment

The facts as described in Section 5.2 are obviously those elements of the model that need to be assessed to evaluate the quality of code comments. However, since some of them are semantic in nature and not automatically assessable, we differentiate between three types of facts:

1. *Automatically assessable facts*: These are facts which can be assessed algorithmically with a tool, such as [Interface | STRUCTURAL COMPLETE].
2. *Semi-automatically assessable facts*: Some facts can be partially assessed with a tool but require additional manual inspection, *e.g.*, [Interface, Inline | EXPLAINING NON-OBVIOUS].
3. *Only manually assessable facts*: These facts can not be evaluated with a tool at all but only with human knowledge, *e.g.*, [Interface, Inline | RELATED TO CODE SEMANTICS]. In his theorem, Rice [25] showed that there exists no automatic method to decide in general any non-trivial semantic property of a computer program. Hence, the judgment of the relation between comment and code semantics requires human knowledge.

Table 5.1 shows which facts we believe can be assessed automatically. We claim all completeness facts to be assessable fully automatically. However, we abstain from the assessment implementation as there is no challenging algorithmic or heuristic aspect involved. Usefulness facts with attribute EXISTENT can be assessed fully automatically with the help of the machine learning classifier for comment categorization (Chapter 6). Later chapters will show that the following facts can be assessed either fully- or semi-automatically:

- [All | CONSISTENT LANGUAGE]
- [Copyright | CONSISTENT STRUCTURE]
- [Interface | RELATED TO METHOD NAME]
- [Interface, Inline | EXPLAINING NON-OBVIOUS]
- [All | CLARIFYING]
- [All | HELPFUL]

Within the scope of this work, we do not address the ambiguity property of comments as well as their general understandability. We assume that ambiguity detection techniques such as [26] can also be applied to code comments so we rank this fact as semi-automatically.

5.4. Global vs. Local Facts

The facts as described previously cover two different aspects: Some of them are global in nature, the others deal with local properties. In order to improve quality of source code comments, you can either improve the quality of a single comment within its limited

Fact	F-A	S-A	O-M
[All CONSISTENT LANGUAGE]	X		
[Copyright, Header CONSISTENT STRUCTURE]	X		
[Interface EXISTENT FOR EVERY PUBLIC DECLARATION]	X		
[Copyright, Header EXISTENT IN EVERY FILE]	X		
[Interface STRUCTURAL COMPLETE]	X		
[Inline METHOD COVERING]	X		
[Interface RELATED TO METHOD NAME]	X		
[Interface, Inline EXPLAINING NON-OBVIOUS]		X	
[Interface, Inline RELATED TO CODE SEMANTICS]			X
[All CLARIFYING]		X	
[All HELPFUL]		X	
[All UNAMBIGUOUS]		X	
[All UNDERSTANDABLE]			X
[Task EXISTENT]	X		
[Section EXISTENT]	X		
[Code EXISTENT]	X		

Table 5.1.: Categorization of facts into fully automatically (F-A), semi-automatically (S-A), and only manually (O-M) assessable

scope of the following lines or you can work on global system commenting. Global system commenting comprises all aspects of comment consistency and also parts of completeness. Interface comments, copyrights, and headers existing in every file contribute to global system commenting. Other facts belonging to the categories coherence and usefulness, such as the relation between interface comments and method name or interface comments explaining the non-obvious are local in nature. They rather cover the quality of a single comment than system commenting.

6. Comment Classification

The previous chapter introduced the quality model for code comments, which is based on the categorization of comments described in Chapter 4. In this chapter, we successfully employ machine learning algorithms to automatically classify a comment. We use the existing machine learning library WEKA, which is available online.¹ WEKA is widely used within the machine learning community and provides implementations for most standard ML algorithms. Furthermore, it provides a possibility to save the training data and resulting classifier in a standardized format (.arff for the data², .model for the classifier).

The success of any machine learning algorithm depends strongly on the training data set, the extracted features to represent the data, and the type of classifier used. Section 6.1 gives an overview of basic machine learning terminology, Section 6.2 describes our training data set, Section 6.3 explains the feature extraction, and Section 6.4 introduces the classifier. We run the classifiers on both Java programs as well as programs written in C++, and compare the results in Section 6.5 and 6.6. Section 6.7 shows threats to validity, Section 6.8 summarizes this chapter.

6.1. Machine Learning Terminology

This section gives an overview of the basic machine learning terminology, from the fundamental problem of classification to evaluation techniques and metrics such as cross-validation, precision, and recall.

Classification Problem

In machine learning, *classification* is the problem of identifying to which class a new observation or object belongs to, whereupon the set of possible classes is known a priori. The algorithm performing the classification is called (machine) *learner* and is trained based on a given set of *instances* with a known classification, the *training data*. An instance represents the observation/object to be classified with a set of *features* and the *class label*. Features describe a potentially complex object with various simple attributes which are within the scope of this work either boolean or numeric (*i. e.*, integer or real-value). Class labels indicate the classification to be learned. After learning, the classifier is evaluated on a *test data* set of instances on which the algorithm has not been trained. Therefore, training and test data set should be disjoint.

Binary Classification Problem

If the observations to be classified can only belong to two different classes, the classification problem is *binary* as the classifier has to make a binary decision between the two class

¹<http://www.cs.waikato.ac.nz/ml/weka/>

²<http://www.cs.waikato.ac.nz/ml/weka/arff.html>

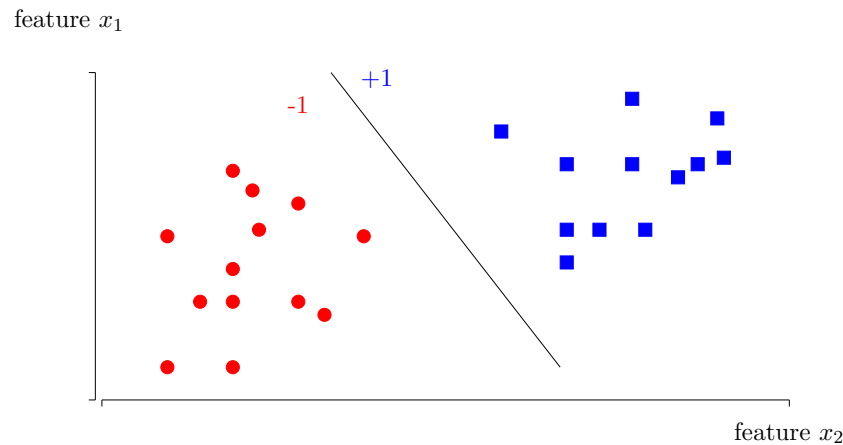


Figure 6.1.: Binary classification problem

labels. Figure 6.1 shows a binary classification problem with labels $+1$ and -1 , represented in a two-dimensional feature space. The drawn line constitutes one possible linear classifier for this data set. This classifier splits the two-dimensional feature space into two subspaces, assigning one label to each subspace. With this classifier, every new observation located to the left of the line will be classified with label -1 , all others with $+1$.

Multinomial Classification Problem

If instances of the data set have more than two distinct labels, the classification problem is *multinomial* and can be solved in different ways. One way of calculating a multinomial classification problem with n labels is solving n *1-versus-all* binary classification problems: For each label, a single classifier is trained to distinguish that class from all other classes. The final prediction is then performed by choosing the prediction with the highest confidence score. Another way is to compute $\binom{n}{2}$ binary classification problems for every pair of labels. The predicted probabilities can be coupled using Hastie and Tibshirani's pairwise coupling method [27] in order to determine the final classification.

Linearly Separable Data

A classifier can only learn to classify the entire training data correctly when the data is *linearly separable*, such as the example in Figure 6.1. A binary classification problem is called linearly separable if a hyperplane can be defined such that all instances of one class are on one side of the hyperplane and all instances of the other class fall on the other side. In case of non-linearly separable data, as shown in Figure 6.2, such a hyperplane does not exist.

Precision and Recall

The two commonly used metrics to evaluate a classifier in machine learning are *precision* and *recall*, combined together in the *F-score* metric. For simplicity, we only consider the binary $\{-1, +1\}$ -classification problem in the following. The test data set can be split in two different ways - according to the real labels of the data or according to the predicted

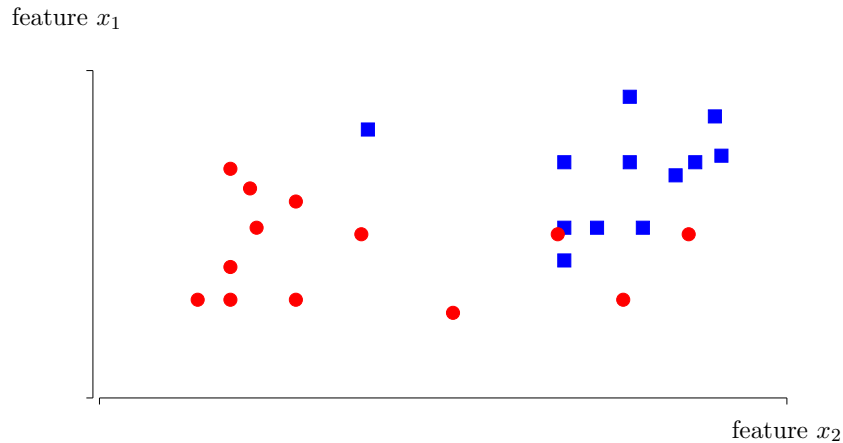


Figure 6.2.: Binary classification problem which is not linearly separable

	Predicted +1	Predicted -1
Label +1	TP	FN
Label -1	FP	TN

Table 6.1.: Classification of the test data set according to the label and the prediction of the classifier

labels of the classifier (see Table 6.1). If the predicted label matches the true label of an instance, this is called either a true positive classification, TP , if the Label was +1, or a true negative, TN , if the label was -1. In the other two cases, though, the classifier makes a mistake: If an instance was labeled with +1 but classified as -1, the classifier makes a false negative mistake, FN . If an instance had label -1 but was predicted to have label +1, the classifier makes a false positive mistake, FP .

The precision refers to the fraction of the true positives among all instances predicted to be +1, Equation 6.1. In other words, the precision represents the probability that the classifier makes a correct decision when predicting that the instance is labeled with +1.

$$\text{precision} = \frac{TP}{TP + FP} \quad (6.1)$$

Recall refers to the fraction of true positives among all instances labeled as +1, Equation 6.2. The recall denotes the probability that an instance with label +1 is detected by the classifier.

$$\text{recall} = \frac{TP}{TP + FN} \quad (6.2)$$

The F -score is calculated as the harmonic mean of recall and precision and can hence be interpreted as a weighted average of both, Equation 6.3. The F -score, also called F_1 -score or F -measure, reaches its best value at 1 and its worst score at 0:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \quad (6.3)$$

Cross Validation

Cross validation is a standard evaluation technique in machine learning to determine the quality of a classifier [28]. For cross validation, the training data set is split into k folds. The classifier is built k times with $k - 1$ out of k folds as training data. For each run, the remaining fold is used as test data and the classifier’s precision and recall are calculated on this fold. To get an overall evaluation, precision and recall are averaged over all k runs. Common values for k are 5 or 10.

6.2. Training Data

For comment classification, we use the categories as presented in Chapter 4 as class labels and hence deal with a multinomial classification problem. To obtain training data, we manually tagged comments and mapped them to one of the specified class labels. Thus, we created two separate training sets, one each for programs written in Java and C++.

Java Training Data

For the Java training data set, we used twelve open source projects from different domains and randomly sampled files from each of them. In total, the data set contains more than 2600 comments. Among them, we manually tagged about 800 comments for comment classification. Table 6.2 shows the number of comments tagged in each project as well as a short description of the project’s content. Table 6.3 presents the number of comments tagged in each category. JBoss, jMol, jEdit, pdfsam, and jabref are available for download at SourceForge.³ ConQAT⁴ and voTUM⁵ are also open source software. The source code of mylyn, EMF, and the eclipse.pde.core package is downloaded and extracted while installing the Indigo version of Eclipse.⁶

Project Name	Version	Content	# Comments
CSLessons	N.A.	simple code examples	39
EMF	2.7	modeling tool	24
Jung	2.0.1	framework for graph algorithms	32
ConQAT Engine	2011.9	quality assessment toolkit for SE	89
jBoss	6.0.0.	application server	16
voTUM	0.7.5	visualization for compiler optimizations	109
mylyn	3.6.0	monitoring tool	38
pdfsam	2.2.1	split and merge tool for pdfs	39
jMol	12.2	chemical visualization for 3D structures	389
jEdit	4.5	text editor	10
eclipse.pde.core	3.7	eclipse compiler	3
jabref	2.7.2	management tool for bibliographies	42

Table 6.2.: Java machine learning data set grouped by projects

³<http://sourceforge.net/>

⁴www.conqat.org

⁵<http://www2.in.tum.de/votum>

⁶<http://www.eclipse.org/downloads/>

Category	# Comments
copyright	61
header	48
inline	297
interface	271
section	65
code	73
task	15
Total	830

Table 6.3.: Java machine learning data set grouped by categories

C++ Training Data

As in the Java case, we manually tagged comments from C++ projects. Table 6.4 gives an overview of the projects, Table 6.5 indicates the number of C++ comments per category. The OGRE project, virtual dub, 7 zip, celestia, and boost are available for download at source forge. LLVM⁷ and the subversion tool tortoise⁸ are also open source. The commercial code is used from customers of the CQSE GmbH, with further information being under a non-disclosure agreement.

6.3. Feature Extraction

To apply any machine learning algorithm, the data set is represented by features. Table 6.6 shows the features that are calculated to determine the category of each comment and their meaning. In general, the features are of type boolean, integer, or double, as the WEKA library is able to handle different types of features. The features are designed such that they take different aspects of comments into account and vary in their computational complexity. Subsection 6.3.1 gives an overview of the simple features, whereas Subsections 6.3.2 and 6.3.3 show more details about complex features such as code recognition and context correlation.

6.3.1. Simple Features

Features such as the *copyright*, *header*, *javadoc*, *section*, or *task* feature search for specific keywords or characters within the comment. For example, the *copyright* feature is true if the comment contains “copyright” or “license”, *header* is true if the signal word “author” is found, and *task* becomes true if expressions like “hack”, “fixme”, or “todo” appear. Thereby, use of small or capital letters is not taken into account. The *section* feature looks for multiple consecutive usages of the same delimiter character such as “****”, “///”, or “###”. Other features such as *parenthesis*, *class distance*, *decl. distance*, *inside method*, *followed*, or *first* describe the position of the comment within the source code: *Parenthesis* is an integer feature indicating the hierarchical level of parentheses within the source code. It

⁷<http://llvm.org/releases/>

⁸<http://tortoisesvn.net/downloads.html>

Project Name	Version	Content	# Comments
commercial code	N.A.	N.A.	159
OGRE	1.8	graphics rendering engine	117
virtual dub	1.10.2	audio and video software	29
7 zip	9.22	zip program	10
celestia	1.6.1.	real-time 3D visualization of space	47
llvm	3.1	intermediate representation for compiler	71
tortoise	1.7.10	subversion tool	86
boost	1.49.0	c++ libraries	42

Table 6.4.: C++ machine learning data set grouped by projects

Category	# Comments
copyright	50
header	48
inline	144
interface	248
section	17
code	39
task	15
Total	561

Table 6.5.: C++ machine learning data set grouped by categories

counts how many braces are *open* at the comment position. At any token of the source code, a brace is considered to be open if the corresponding brace is closed after the current token. In Java, copyright and header comments should therefore have a *parenthesis* value of zero as they are supposed to be located before any class definition. Contrarily, the *parenthesis* value of interface comments should be one, as they appear within a class definition but outside of any method definition. For inner class definitions, we adapt the *parenthesis* value accordingly: Once the token “class” or “interface” is parsed within another class or interface definition, we decrement the *parenthesis* value. The *parenthesis* value is also adapted for header comments preceding an inner class definition.

Both the *class distance* feature as well as the *decl. distance* feature describe the relative position of a comment with respect to the next class definition or the next method/feature declaration. The relative position is measured by counting the number of lexical tokens between the comment and the next class definition or method/feature definition, respectively. The *inside method* feature indicates whether a comment is located within a method definition. This feature is calculated with the help of a shallow parser for both C++ and Java. The *followed* attribute states whether the comment is directly followed by another comment, and *first* denotes whether the comment is the first comment in the file. Finally, the *length* and *special characters* features refer to lexicographic properties of the comment. *Length* denotes the number of words in a comment; words are separated by white spaces. The *special characters* feature represents the percentage of characters such as “;”, “(”, “)”, “=” etc. among all characters of the comment.

Name	Type	Description
copyright	boolean	true if comment contains “copyright” or “license”
header	boolean	true if comment contains “author”
class distance	int	measures the distance in lexical tokens to the next “class” token
javadoc	boolean	true if comment contains a Javadoc tag such as @param, @return, @throws, @link, or @inheritDoc
parenthesis	int	indicates how many braces are open at the position of the comment
decl. distance	int	measures the distance in lexical tokens to the next method or attribute declaration
context	boolean	indicates whether the minimal Levenshtein distance between the words in the comment and the next declaration name is smaller than 2
section	boolean	true if comment contains a separator string multiple times (<i>e.g.</i> , “***”, “- -”, “///”)
length	int	counts the number of words, separated by white spaces, in the comment
task	boolean	true if comment contains “task”, “fixme”, or “hack”
followed	boolean	indicates whether the comment is directly followed by another comment
special characters	double	indicates the percentage of special characters in a comment such as “;”, “=”, “(”, “)”, “{”, “}”, “+”, “-”, “/”
code	boolean	true if comment contains commented out code
inside method	boolean	true if the comment is within a method definition
first	boolean	true if comment is the first comment of the file

Table 6.6.: Machine learning features for comment categorization

6.3.2. Code Recognition

The *code* feature indicates whether a comment contains commented out code. If the ratio of lines of code to all lines of the comment is higher than a threshold, the feature is true. A line is considered to contain code if one of the following four characteristics is fulfilled:

- it matches the regular expression⁹ for the Java method call pattern
`[a-zA-Z]+\.\.[a-zA-Z]+\(\. *\)`
- it matches the regular expression of an **if** or **while** statement:
`(if\s*\(\. *)|(while\s*\(\. *)`
- it ends with either “;” or “{”
- it contains “=”, “==”, “;”, or “void”

⁹We use the regular expression syntax as used in the `java.util.regex` package: <http://docs.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html>

After preliminary experiments, we set the threshold for code recognition to 0.1. Hence, if more than 10% of all lines fulfill one of the above criterion, we consider the comment to be commented out code.

The *code* feature was specifically designed for the Java programming language. However, we also use it for classifying C++ comments with reasonable success (see Section 6.6). Slight adaptations on the above characteristics such as the C++ method call pattern probably improve classification on C++ comments.

6.3.3. Context Correlation

The *context* feature provides information about the coherence between comment and a subsequent method name by calculating similarity between words in the method name and words in the comment. The method name is split into words based on a switch from lower to upper case or vice versa. For example, the method name “setStartPosition” would be split into “set”, “Start”, and “Position”. Based on the words of the method name, we calculate the powerset and compare it to the set of words (separated by white space) in the comment: If there are two words, one from the powerset of the method name and one from the comment, which have a Levenshtein distance smaller than two, the *context* feature is set to true. The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other, with the allowable edit operations being insertion, deletion, or substitution of a single character.¹⁰

6.4. Classification Algorithms

In this section, we introduce two different classification algorithms. Preliminary experiments showed that among other learning algorithms, such as k-Nearest-Neighbors or Bayesian learners, decision trees and support vector machines (SVMs) yield the best results. Therefore, we present the evaluation of two different decision trees and one SVM within the scope of this work, that promised good results.

6.4.1. Decision Trees

A decision tree is a tree with decision nodes as interior nodes and class labels as leaves. To classify an instance, the decision tree is handled as a set of **if-then-else** statements: Based on the value of a single feature, a decision is made at each interior node to further traverse the left or the right branch of the tree until a leaf is reached. The class label represented by the leaf is then assigned to the instance as its classification.

With the help of preliminary experiments, we selected two different implementations of the decision trees, the J48 algorithm [29] and REPTrees¹¹. The implementations differ in the way the tree is constructed and in the heuristics applied for pruning the decision tree. For more information, refer to the provided references as more details are not within the focus of this thesis.

¹⁰http://en.wikipedia.org/wiki/Levenshtein_distance

¹¹<http://weka.sourceforge.net/doc/weka/classifiers/trees/REPTree.html>

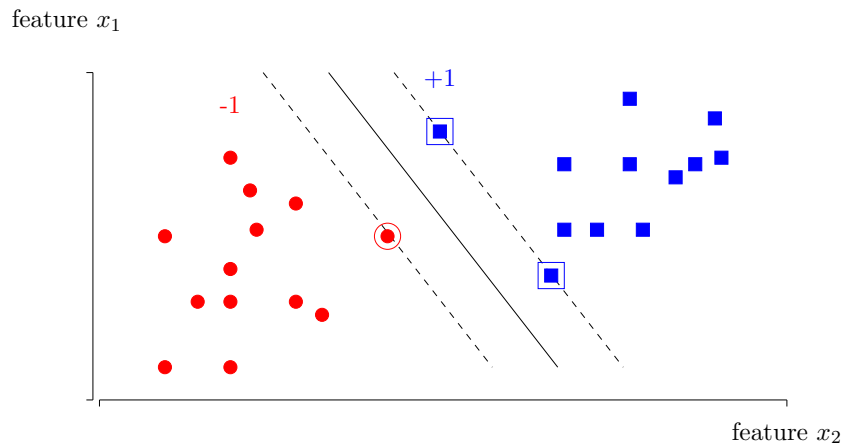


Figure 6.3.: Support vectors for a binary classification problem

6.4.2. Support Vector Machines

Support vector machines construct a set of hyperplanes in the high-dimensional feature space, defining subspaces for each class label. The best separation is achieved by hyperplanes that maintain the largest distance from the nearest training data points which are called support vectors. Figure 6.3 shows an example for a binary classification problem with one separating line for the given data. The two dashed lines represent the margin achieved by the separation, support vectors are surrounded with an additional circle or rectangle respectively. In addition to performing linear classification, support vector machines can also solve non-linear classification using so-called *kernel functions*. A kernel function transforms the non-linearly separable input data into a higher-dimensional feature space where the data becomes linearly separable.

The implementation of support vector machines in WEKA is based on Platt's algorithm [30]. Multinomial classification problems are solved by using coupling pairwise binary classification problems [27].

6.5. Evaluation on Java

In the following, we evaluate the training of decision trees (Subsection 6.5.1) and support vector machines (Subsection 6.5.2) on the Java data set. To evaluate the performance of a classifier, we use the standard five-fold-cross validation, and compare the precision, recall, and F -score of each classifier.

6.5.1. Decision Trees

Tables 6.7 and 6.8 show the results of five-fold-cross validation on the J48 and the REP decision tree. With a weighted average in precision and recall of 96% (J48) and 95% (REP), both classifiers learn the task of comment categorization very well. The most difficult concepts to learn are code and section comments. For code comments, the precision drops below 90% in both trees because code is often misclassified as inline, interface, or section

Class	TP Rate	FP Rate	Precision	Recall	F-Measure
code	0.945	0.012	0.885	0.945	0.914
copyright	1	0.001	0.984	1	0.992
header	0.958	0.001	0.979	0.958	0.968
inline	0.98	0.011	0.98	0.98	0.98
interface	0.967	0.018	0.963	0.967	0.965
section	0.831	0.007	0.915	0.831	0.871
task	0.933	0.001	0.933	0.933	0.933
Weighted Avg.	0.96	0.012	0.96	0.96	0.96

Table 6.7.: Results for J48 on Java

Class	TP Rate	FP Rate	Precision	Recall	F-Measure
code	0.959	0.015	0.864	0.959	0.909
copyright	0.984	0.003	0.968	0.984	0.976
header	0.958	0.005	0.92	0.958	0.939
inline	0.98	0.011	0.98	0.98	0.98
interface	0.934	0.021	0.955	0.934	0.944
section	0.785	0.013	0.836	0.785	0.81
task	0.933	0	1	0.933	0.966
Weighted Avg.	0.946	0.014	0.946	0.946	0.946

Table 6.8.: Results for REPTree on Java

comments and vice versa. Section comments are also occasionally difficult to distinguish from interface comments without human knowledge.

Figures 6.4 and 6.5 present a graphical representation of both decision trees. Interestingly, only a subset of the initial features is relevant for classification: we assume that this is partly due to pruning and partly due to some features having no information gain for classification. Some of the relevant features are strong indicators for one particular comment category, such as the *copyright* feature: If the comment contains “copyright” or “license”, it is almost certainly a copyright comment. Figure 6.4 shows that the J48 decision tree takes the *copyright* feature as root node, with the “copyright”-leaf as one child. The REP tree (Figure 6.5) uses the *copyright* feature only on the third decision level, nevertheless, it is the final decision between copyright and header. Other features can be significantly important for the classification without being a strong indicator for one single category: both decision trees use the *parenthesis* feature within the top three levels of the tree. Based on the *parenthesis* feature, the tree splits up into three branches for values 0, 1, or greater than 1. A *parenthesis* value of 0 indicates copyrights or headers, value 1 leads to interface, section, or code, and value 2 or larger signals inline, task, or code (Figure 6.5).

Classification Errors

The matrix in Table 6.9 shows the classification errors of the J48 algorithm. We do not show the classifications errors of the REP decision tree as they were similar. In the so-called confusion matrix, rows represent the true label of the comment, columns represent

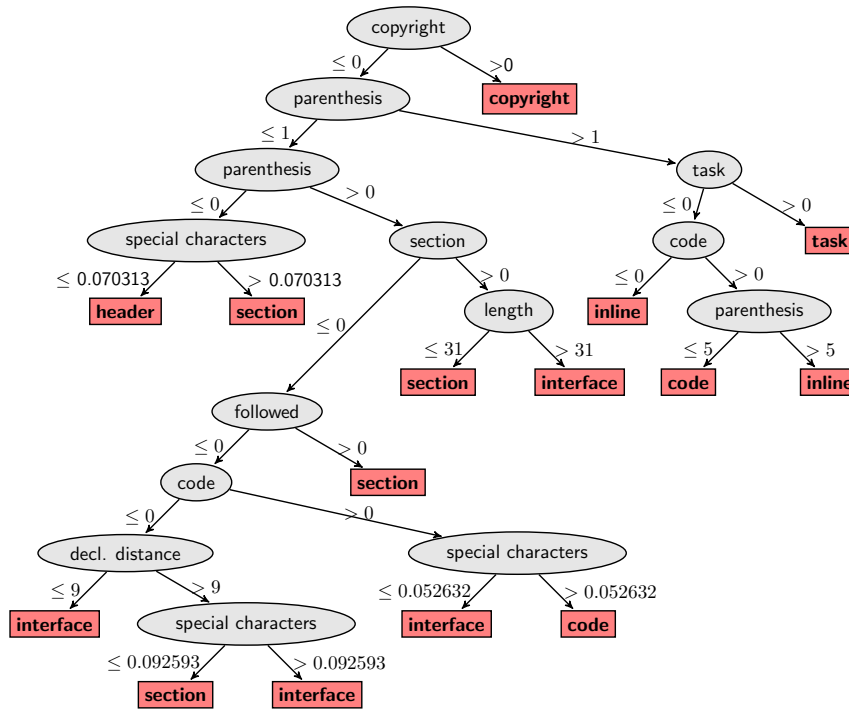


Figure 6.4.: J48 decision tree for comment classification in Java

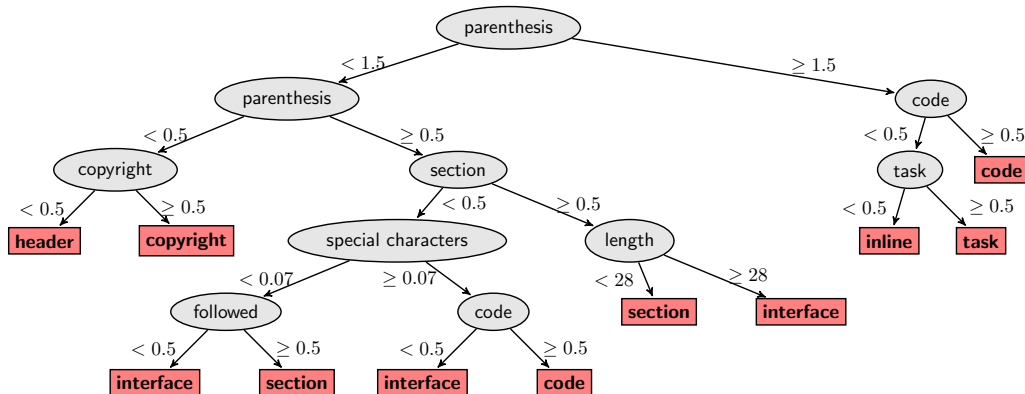


Figure 6.5.: REP decision tree for comment classification in Java

the label as determined by the classifier. An entry i in row x and column y hence indicates that i comments with true label x were classified with label y . Entries on the diagonal consequently state correct classifications. If the classifier didn't make any mistakes, then the matrix would only have non-zero entries on the diagonal.

In terms of code recognition, the matrix shows that four code examples were not recognized as code and 9 inline, interface, or section comments were misclassified as code. This is due to the low threshold we set for code recognition (see Subsection 6.3.2). Setting the threshold is a trade-off between precision and recall: The higher the threshold, the higher the precision but the lower the recall and vice versa.

code	copyright	header	inline	interface	section	task	← classified as
69	0	1	2	0	0	1	code
0	61	0	0	0	0	0	copyright
0	1	46	0	0	1	0	header
6	0	0	291	0	0	0	inline
1	0	0	4	262	4	0	interface
2	0	0	0	9	54	0	section
0	0	0	0	1	0	14	task

Table 6.9.: Confusion matrix of the J48 algorithm on Java

Class	TP Rate	FP Rate	Precision	Recall	F-Measure
code	0.918	0.018	0.827	0.918	0.87
copyright	1	0.001	0.984	1	0.992
header	0.896	0.001	0.977	0.896	0.935
inline	0.976	0.03	0.948	0.976	0.962
interface	0.937	0.021	0.955	0.937	0.946
section	0.754	0.008	0.891	0.754	0.817
task	1	0.001	0.938	1	0.968
Weighted Avg.	0.939	0.02	0.939	0.939	0.938

Table 6.10.: Results for SVM on Java

6.5.2. Support Vector Machines

The results of using support vector machines were slightly worse than the results of using decision trees but still good enough for comment classification in practice. The average precision and recall was around 94%, with similar problems in classifying code and section comments (see Table 6.10).

In contrast to decision trees, there is no simple visualization of support vector machines for multinomial classification because the high-dimensional feature space with its separating hyperplanes is hard to visualize. Nevertheless, Table 6.11 shows, *e. g.*, the resulting attribute weights for the hyperplane separating code and header comments with a binary classifier. As the threshold is close to zero (0.4008), the separating hyperplane is close to the origin of the feature space. Hence, negative weights indicate that the corresponding features are strong indicators for the code label, and features with positive weights favor the header label. the larger the absolute value of a weight, the more important the corresponding feature: *Code* and *inside method* are the two most indicative features for code, while *header* and *class distance* are strongly associated with headers. Both results match human intuition.

6.6. Evaluation on C++

After evaluating the classifiers on the Java training data set, we run the same experiments on the C++ data set and compare the results, again based on precision, recall, and *F*-score.

Weight (normalized)	Attribute
0.6543	header
0.6788	class distance
0.498	javadoc
-0.5334	parenthesis
0.3981	decl. distance
-1.237	code
-0.1877	special characters
0.0015	context
-1.1002	inside method
0.0098	length
0	first
0.4008	threshold

Table 6.11.: Weight vector of hyperplane for SVM binary classification between code and header in Java

6.6.1. Decision Trees

For classification on C++, the J48 and the REP decision tree perform approximately the same: They reveal the same (weighted) average recall, and the REP tree has a slightly higher precision. Tables 6.12 and 6.13 show precision and recall per class label. Compared to the Java results, precision and recall for code recognition are lower. This is due to the fact that the *code* feature was mainly tailored to suit the Java programming language and not specifically adapted to C++. In spite of this, precision and recall are still around 80%. We expect that an adaptation of this feature (*e. g.*, with the C++ method call pattern) increases the performance to a level comparable to that of the Java-based system.

Headers are harder to classify in C++ than in Java. In Java, there are only two categories with *parenthesis* value of 0, headers and copyrights. Consequently, if the comment has *parenthesis* value 0 and does not contain a copyright keyword, it is classified as a header, yielding to a high precision and recall. This pattern does not work on C++. Due to the structure of C++ programs, the *parenthesis* value does not have a significant influence for comment classification: For example, class and method definitions can be surrounded by additional curly brackets of a name space, which reduces the information gain of the *parenthesis* attribute. Figures 6.6 and 6.7 show that the *parenthesis* feature occurs near the bottom of the J48 tree, and not at all in the REP tree. Instead, both trees rely on the *inside method* feature. The *inside method* feature supplants the *parenthesis* feature by at least providing reliable information about whether a comment is within a method.

Classification Errors

Table 6.14 shows the confusion matrix of C++ data for the J48 algorithm. Comparing the confusion matrices on C++ and Java reveals that the interface column has more non-zero entries in the C++ matrix than in the Java matrix. Presumably, this is also due to the fact that the *parenthesis* feature reliably helps to identify Java interface comments, whereas this information is missing in C++.

Class	TP Rate	FP Rate	Precision	Recall	F-Measure
code	0.821	0.015	0.8	0.821	0.81
copyright	1	0	1	1	1
header	0.938	0.008	0.918	0.938	0.928
interface	0.972	0.032	0.96	0.972	0.966
section	0.765	0.007	0.765	0.765	0.765
task	0.867	0	1	0.867	0.929
Weighted Avg.	0.948	0.018	0.949	0.948	0.948

Table 6.12.: Results for J48 on C++

Class	TP Rate	FP Rate	Precision	Recall	F-Measure
code	0.846	0.019	0.767	0.846	0.805
copyright	1	0	1	1	1
header	0.958	0.008	0.92	0.958	0.939
inline	0.958	0.005	0.986	0.958	0.972
interface	0.96	0.019	0.975	0.96	0.967
section	0.882	0.013	0.682	0.882	0.769
task	0.8	0	1	0.8	0.889
Weighted Avg.	0.948	0.012	0.953	0.948	0.95

Table 6.13.: Results for REP on C++

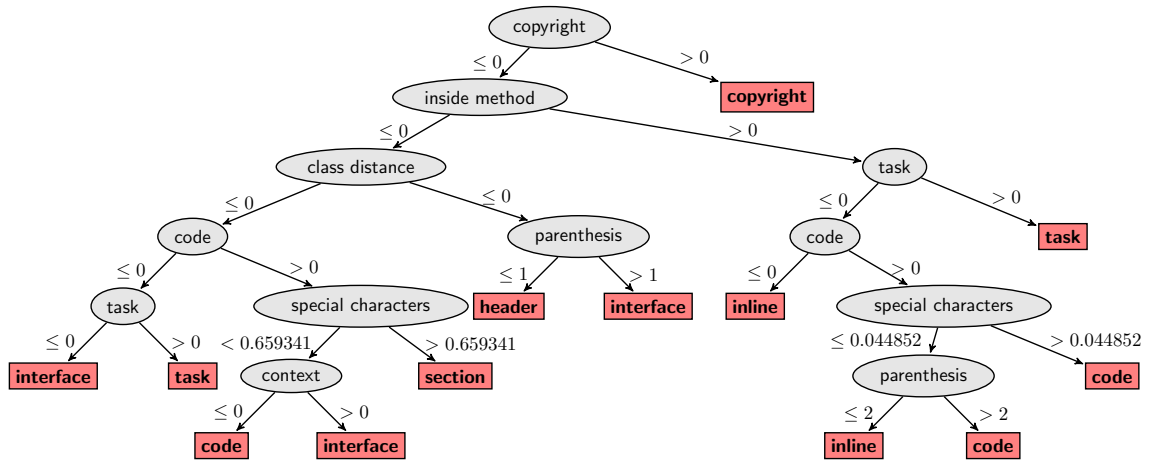


Figure 6.6.: J48 decision tree for comment classification in C++

6.6.2. Support Vector Machines

Support vector machines lead to better results than decision trees on C++ code. They have a higher recall on code recognition but a lower precision. For section comments, they achieve both a higher recall and a higher precision, see Table 6.15.

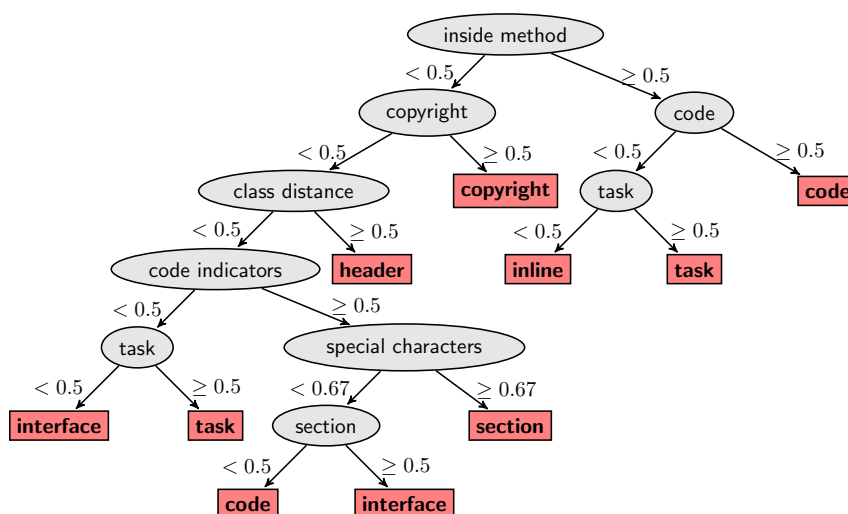


Figure 6.7.: REP decision tree for comment classification in C++

code	copyright	header	inline	interface	section	task	← classified as
32	0	0	3	2	2	0	code
0	50	0	0	0	0	0	copyright
0	0	45	0	3	0	0	header
6	0	0	138	0	0	0	inline
1	0	4	0	241	2	0	interface
1	0	0	0	3	13	0	section
0	0	0	0	2	0	13	task

Table 6.14.: Classification errors made by the J48 algorithm on C++

Class	TP Rate	FP Rate	Precision	Recall	F-Measure
code	0.897	0.021	0.761	0.897	0.824
copyright	1	0	1	1	1
header	0.938	0.008	0.918	0.938	0.928
inline	0.951	0	1	0.951	0.975
interface	0.972	0.016	0.98	0.972	0.976
section	0.882	0.004	0.882	0.882	0.882
task	1	0.002	0.938	1	0.968
Weighted Avg.	0.959	0.009	0.962	0.959	0.96

Table 6.15.: Results for SVM on C++

6.7. Threats to Validity

One could argue that the Java classifier was trained predominantly to classify comments in jMol, as almost half of the training data comments were tagged in this project (389 out of 830). To measure this effect, we also trained the classifier without the jMol data. Considering the decrease in the size of the training data set, the algorithm still performed

comparably well. In the remainder of this work, we used the classifier on other projects and experienced that classification works successfully independent from the underlying project. Hence, the classifier does not over-fit the training data.

6.8. Summary

In both the Java and the C++ environments, comment categories can be successfully classified with the help of machine learning algorithms as both decision trees and support vector machines perform well enough for comment categorization in practice. Our results are slightly better for the Java use case. This is due to several facts: First, the Java training set contains about 800 tagged comments, roughly 300 comments more than the C++ training set. Usually, more available training data leads to better results. Second, the code recognition algorithm is tailored to suit the Java programming language, and hence results in a higher precision and recall for Java. Third, the *parenthesis* attribute is a crucial decision feature for classification in Java. However, it does not contribute to classification in C++ due to the structural properties of C/C++.

7. Comment Consistency

The quality model (Chapter 5) groups facts according to four main quality criteria: consistency, completeness, coherence, and usefulness. In this chapter, we propose methods to analyze comment consistency. All facts belonging to this category can be assessed automatically. We implemented the language recognition (Section 7.1) to evaluate the fact [All | CONSISTENT LANGUAGE] as well as the consistency check of copyrights (Section 7.2) to assess [Copyright | CONSISTENT STRUCTURE]. Within the scope of this work, we have not implemented the consistency check for headers as proposed by the model.

7.1. Language Recognition

In order to easily read code, the comments' language should be consistent throughout the project. We use existing tool support for language recognition to detect if one or more languages were used for commenting code.

7.1.1. Implementation

Language recognition can be performed in different ways: First, we use the Apache Tika toolkit¹ which includes a language detection feature based on ngram analysis. Second, we experiment with GNU Aspell, an open source spell checker², and third, we use the ConQAT language decider based on language-specific probability distributions of pairs of letters.

For all three approaches, we normalize the comments before using them as an input for language detection: Normalization removes commented out code as detected by the machine learning classifier from Chapter 6 and also, for example, Javadoc tags, numbers, or special characters.

Tika

The Tika toolkit provides a generic profile for each supported language: It builds a language profile based on ngram analysis. A ngram is a sequence of n characters or words extracted from text and Tika includes precalculated ngrams for most common languages using a large reference text basis. If the reference examples are large enough, a ngram gives a representative characteristic for each language. To decide the language of a new string, the algorithm creates a set of ngrams for the input string and uses distance measurements to calculate similarity to the predefined ngram profiles. If the distance is smaller than a certain threshold, the algorithm chooses the corresponding language, otherwise it returns *unknown*.

¹<http://tika.apache.org/>

²<http://aspell.net/>

For maximal success, we manually restrict Tika to the English and German language to limit the number of choices. Besides that, Tika language recognition works very well on long input text. However, it is known that the language detection has less success with short chunks of text such as code comments.³

Aspell

For the second approach, we use the open source spell checker Aspell. This spell checker provides terminal commands to output its dictionaries both for the German and English language. The dictionaries are used as input for the language recognition in form of text files. If a word in the comment appears in one of both dictionaries, the word is considered to belong to the corresponding language. If neither list contains the word, we run the spell checker in the console for both languages. Words that are correctly spelled are then assigned to the corresponding language. Remaining words are not assigned to any language. For the complete comment, the total number of assigned words per language is counted and used to decide the comment's language by majority vote. If there are exactly as many English classified words as German classified words in the comment, then *unknown* is returned.

ConQAT

The open source toolkit for quality assessment, ConQAT, provides a language recognition based on probability distributions of consecutive pairs of letters. Consecutive pairs of letters of the word "thesis" are for example "th", "he", "es", "si", and "is". The probability distribution denotes the normalized relative frequency for each pair of letters as calculated based on example reference texts: The distribution for German is predicated on the three texts "Märchen für Kinder" (H. C. Andersen), "Effi Briest" (Theodor Fontane), and "Buch der Lieder" (Heinrich Heine)⁴. "The Outline of Science, Vol. 1" (J. Arthur Thomson), "Pride and Prejudice" (Jane Austen), and "Ulysses" (James Joyce)⁴ constitute the reference for the English distribution. Other languages are not available in the current implementation.

To decide the language for a given string, the text is split up in its words by white spaces. The algorithm decides the language for each word separately and makes the overall decision if more than 50% of the words were assigned to one language. Otherwise, the algorithm returns *unknown*. To decide the language for a given word, we enumerate the consecutive pairs of letters and calculate the product of their relative frequencies according to both the German and the English distribution. The algorithm decides for one language if its product of relative frequencies is significantly higher than the product corresponding to the other language. Significance is thereby defined by a threshold determined in preliminary experiments: the algorithm chooses one language if the ratio of both frequency products is higher than 2. Otherwise, the algorithm returns *unknown*.

7.1.2. Results

We evaluate the three implementations of language recognition on the same data set as used in training the Java comment classifier (see Section 6.2). In the training set, we manually tagged those comments written in German. All other comments were written in

³<https://issues.apache.org/jira/browse/TIKA-369>, last accessed on 06/18/2012

⁴downloaded from <http://www.gutenberg.org>

	Labeled English	Labeled German
Detected English	true positive (TP)	false positive (FP)
Detected German	false negative (FN)	true negative (TN)
not classified	classification failure English (CF-E)	classification failure German (CF-G)

Table 7.1.: Correct decisions, errors, and failures in language detection

English. After normalization, the training data set contains a total number of 2657 non-empty comments, 180 tagged as German and 2477 written in English. In the following, we show the results of each language decider. A language decider can make two kinds of mistakes, see Section 6.1: In the case of a false positive mistake, a comment written in German is classified as English. In the case of a false negative, a comment written in English is classified as German. Table 7.1 visualizes both mistakes. Further, we call cases in which the algorithm can not make a decision *classification failure*.

As in Section 6.1, we evaluate precision and recall of the classification. Precision denotes:

$$\text{precision} = \frac{TP}{TP + FP} \quad (7.1)$$

For comments where the algorithm makes a decision we evaluate the *conditional* recall with:

$$\text{recall}_{\text{cond}} = \frac{TP}{TP + FN} \quad (7.2)$$

Considering all comments, including those where the algorithm did not make a decision, we evaluate the *unconditional* recall with

$$\text{recall}_{\text{uncond}} = \frac{TP}{TP + FN + CF-E} \quad (7.3)$$

Tika Results

Table 7.2 shows that Tika fails to classify the language in 63 cases (2.4% of all comments). All these cases are written in English. In case of a decision, Tika makes 16 false positive mistakes and 279 false negative mistakes. This leads to a precision of 99.2% ($= \frac{2135}{2135+16}$) and a conditional recall of 88% ($= \frac{2135}{2135+279}$). The unconditional recall is 86% ($= \frac{2135}{2135+279+63}$).

	Labeled English	Labeled German
Detected English	2135	16
Detected German	279	164
not classified	63	0

Table 7.2.: Language recognition with Tika

Aspell Results

Table 7.3 presents the evaluation of the Aspell language decider. Aspell cannot make a decision in 399 cases (15%), among which 9 comments were written in German. If it does decide, then the precision is 99.7% with a conditional recall of 94% and an unconditional recall of 79%.

	Labeled English	Labeled German
Detected English	1963	5
Detected German	124	166
not classified	390	9

Table 7.3.: Language recognition with ASpell

ConQATLanguageDecider Results

Table 7.4 shows the results of the ConQATLanguageDecider. This approach returns a classification failure in 444 instances (17% of all comments). Among these, 37 comments were German. If the algorithm makes a decision, then it has a precision of 99.5% and a conditional recall of 91%. The unconditional recall is 76%.

	Labeled English	Labeled German
Detected English	1881	9
Detected German	189	134
not classified	407	37

Table 7.4.: Language recognition with ConQATLanguageDecider

As explained above, we set the frequency ratio threshold to 2. Preliminary experiments showed that determining the threshold is a trade off between classification failure and recall with a threshold of 2 leading to the best results. A higher threshold leads to more comments for which the language can not be decided but increases the recall. A lower threshold reduces the number of classification failures at the expense of the recall, which decreases.

Comparison

The comparison between Tika, ASpell, and ConQAT reveals that all three approaches have very similar precision (between 99 and 100%). Tika guesses the language more frequently: It only fails to classify in 2.4% of the comments, whereas Aspell can not make a decision for 15% and ConQAT for 17% of the comments. With 88%, Tika has the lowest conditional recall. Aspell has the highest conditional recall (94%), followed by ConQAT (91%). However, for language recognition in practice, we prefer to use the detection tool with the highest unconditional recall as this minimizes the classification failures. With 86%, Tika achieves the best uncondition recall, compared to 79% (Aspell) and 76% (ConQAT). Hence, we recommend Tika for usage in practice.

7.2. Copyright Structure

Besides consistent language, our quality model also demands consistent copyrights (Fact [Copyright | CONSISTENT STRUCTURE]) as for any kind of software system, information about the copyright of source code is essential. Files corresponding to one project should be under the same copyright with a consistent copyright format.

Definition

More precisely, we consider two copyrights to be consistent if and only if they have the same copyright holder and the same structure of the copyright. Thereby, we ignore the year of the copyright. Copyrights have the same structure if they consist of the same structural items such as file name, author, revision, disclaimer, copyright declaration etc. Thereby, we do not require complete correspondence between two copyrights of the same category but allow some deviation (see Subsection 7.2.1).

7.2.1. Implementation

The consistency check of copyrights consists of two phases: First, we create a copyright profile for each copyright comment and then apply a matching function on the profiles to determine similarity.

Copyright Profiles

To create the profile, the algorithm removes certain file-specific features to extract the general structure of the copyright. File-specific features include file name, file path, dates, times, years, authors, and revisions. Thereby, regular expressions function as pattern matching to retrieve these features. Table 7.5 shows an incomplete list of regular expressions, written in Java Syntax⁵, which replace file-specific features with a constant. Thereby, we do not take usage of small or capital letters into account. The table, *e. g.*, contains one regular expression to match dates of the format 1.1.2011 or 02-02-2012. Expressions for other date formats such as Aug-9-2000 are not listed, but implemented. Figure 7.1 shows the ConQAT Project copyright after removing file-specific features.

Furthermore, many copyrights are marked with a frame of delimiter symbols such as “*” “/” or “|”, see also Figure 7.1. The delimiter symbols are extracted and also stored in the copyright profile.

Matching function

The matching function on copyright profiles determines the similarity between different copyrights. It returns true if two profiles belong to the same profile category. In general, the matching function does not differentiate between use of small and capital letters. In the following, we will denote strings with small letters only. To determine the copyright holders, the line containing “copyright (c)” is extracted from each profile.

⁵<http://docs.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html>

```
/*-----+
|
| Copyright - The ConQAT Project
|
| Revision X
| you may not use this file except in compliance with the License.
| You may obtain a copy of the License at
|
| /XX/
|
| Unless required by applicable law or agreed to in writing, software
| distributed under the License is distributed on an "AS IS" BASIS,
| WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
| See the License for the specific language governing permissions and
| limitations under the License.
+-----*/
```

Figure 7.1.: Copyright of the ConQAT project

The line is extracted from a copyright after removing frame delimiters and all file specific features, such as the year of the copyright. In the case of Figure 7.1, the line

```
Copyright - The ConQAT Project
```

is returned. If both copyrights do not exhibit the same line, the matching function returns false. If both lines are equal, the algorithm resumes with the consistency check of the structure: For each copyright, it builds a set of words which includes the first word for each line, excluding frame delimiters. If both sets have an intersection of at least 50%, the structure is considered to be consistent. In preliminary experiments, the threshold was set to 50% to match our human intuition of how much structural deviation is allowed for copyrights to still be considered consistent.

In our current implementation, we ignore the year of the copyright. However, the algorithm can easily be adapted to the different notion that two copyrights should only be considered consistent if they are under the same copyright year: If the year of the copyright is not replaced, the equal test on the copyright line will fail and both profiles will be considered to be inconsistent.

7.2.2. Results

The algorithm ought to be both *sound* and *complete*. In this case, we define the algorithm to be complete if every copyright category of the code basis is enumerated, *i. e.*, any two inconsistent copyrights are represented by two different categories in the output. Furthermore, the algorithm is sound if any two categories in the output of the algorithm are indeed inconsistent.

Feature	Regular Expression	Substitution
Revision	<code>\\n.*(revision version).*\\n</code>	<code>\\n Revision X \\n</code>
Author	<code>\\n.*author.*\\n</code>	<code>\\n Author Xxx Xxx \\n</code>
Class Name	<code>\\n.*"+name+".*\\n</code> with <i>name</i> being a variable containing the file name	<code>\\n XXXX.java \\n</code>
Path	<code>\\n.*:/.*/*.*\\n</code>	<code>\\n /XX/ \\n</code>
Dates	<code>\\n.*[0-9]{1,4}\\s*(\\. - / \\s)\\s*([0-9] [a-zA-Z]{3,9})\\s*(\\. - / \\s)\\s*[0-9]{1,4}.*\\n</code>	<code>\\n Date XX.XX.XXXX \\n</code>
Year	<code>[1-2]{1,1}[0-9]{3,3}</code>	<code>""</code> (none)
Times	<code>[0-9]{2,2}:[0-9]{2,2}:[0-9]{2,2}</code>	<code>XX:XX:XX</code>

Table 7.5.: Preprocessing of a copyright comment for the consistency check

To prove completeness, we manually extract copyrights from the Java data set (Section 6.2) and compare the results. Both the algorithmic and the manual enumeration reveal the same categories which can be found in Appendix A. Categories are mostly characterized by different copyright holders as each open source Java project has its own copyright holder. Among each project the test files show identical copyrights besides of file specific features.

To prove soundness and test the threshold of 50% for structural similarity, we use code from customers of the CQSE GmbH written in C or C++. The code basis is too large for a human validation of completeness. However, for the proof of soundness, manual inspection of the enumerated categories reveals that they are indeed inconsistent: We consider the two copyrights in Figure 7.2, *e. g.*, to be inconsistent, showing the copyrights after replacing file-specific features with constants. Copyright and code details which are covered by the non-disclosure agreement of CQSE customers were marked manually in black. Both copyrights have the same copyright holder and some common features such as filename, revision, author and date. However, they are inconsistent because the copyright on the right contains significant information about the component, configuration, and model element which is missing in the left one.

```
// Revision X
// purpose: ████████████████████████████████
// author Xxx Xxx
// date started:
// Date XX.XX.XXXX
// last change by: ████████
//
// (c) Copyright ████████████████████
//
//-----
  XXXX.cpp
//-----
```

```
// Rhapsody : 7.2
// Component : ████████████████████
// Configuration : generic
// Model Element : ████████████████████
// XXXX.cpp
// Revision X
//
//
// Date XX.XX.XXXX
// Author Xxx Xxx
//
// (c) Copyright ████████████████████
//
```

Figure 7.2.: Two inconsistent copyrights

8. Coherence between Code and Comments

To successfully comment code, the coherence between code and comments is crucial. On the one hand, the comment obviously has to relate to the code because the main target of commenting is code understanding. On the other hand, programmer expect comments to provide additional information which is not contained in the code itself. Code understanding does not benefit from comments containing redundant information from the code.

In this chapter, we propose different approaches to assess facts belonging to the category coherence of the quality model (Chapter 5): We evaluate the relationship between interface comments and the corresponding method name in particular (Section 8.2) and also the coherence between interface or inline comments and their following lines of code in general (Section 8.3 and 8.4). All metrics are evaluated with the help of a survey whose general design is described in Section 8.1 and will be reused in later chapters (Section 9.1).

8.1. Design of the Questionnaire

The survey was designed in form of an online questionnaire, containing three independent parts with multiple evaluation tasks. As we did not expect every participant to fill out all parts, we displayed the links to the three parts in random order to favor a uniform distribution of participation. Each part consists of a general questionnaire to gather data about the participants' programming experience and one or more chapters with different evaluation tasks. For each evaluation task, the specific design and the hypothesis will be explained independently. In the remainder of this chapter, we refer to the evaluation of the relation between comment and method name as part one (Section 8.2), to the investigation of the coherence between inline comment and the following code as part two (Sections 8.3 and 8.4), and to the usefulness of comments as part three (Sections 9.2 and 9.3). In total, there were 16 programmers completing part one and two of the questionnaire, 20 filling out part three.

For each evaluation task, the participant was shown several comments together with their following or surrounding methods. Per comment, the participant answered one or two multiple-choice questions. The comments were selected from the same Java data set as in Chapter 6.2. Part one contained 30 comments, part two 36 comments, and part three 20 comments for evaluation. Throughout each evaluation task, the examples are numerated consecutively within this work. The enumeration, however, does not correspond to the ordering in which comments were displayed. Instead, they were displayed in random order.

We evaluate the participants' feedback based on majority vote and emphasize when the majority constitutes an absolute majority ($\geq 50\%$). Depending on the question, we visualize the given answers with traffic light coloring: Red indicates low comment quality, green represents high comment quality. Yellow and orange symbolize moderate quality.

Background of the participants

All test subjects had sufficient programming experience in Java, with mostly research background. The survey was sent to members of the Software and System Engineering Chair of the Technische Universität München, TUM,¹ the Software Engineering group of the University of Bremen², students of TUM, employees of the software quality consulting company CQSE GmbH and some of their clients.

Within the questionnaire part about their background, participants were asked amongst others:

- How many years of programming experience do you have?
- How many years of programming experience in Java do you have?
- How often are you using code from other projects, frameworks, or libraries?
- How many members did your largest team have?
- Have you ever worked on open source projects?

Within the scope of this work, we present detailed developer background data only for part one of the survey. The distribution among the participants in part two and three was not identical but similar enough to not make a significant difference. Every participant had at least 5 years, 62% more than ten years of programming experience in general. 94% had at least 5 years of programming experience in Java. 69% use code from other projects, frameworks, or libraries frequently, 75% have worked on commercial projects, 88% have worked on open source projects. Figure 8.1 shows more detailed results. Based on this data, we assume the participants to be sufficiently experienced to provide a useful evaluation. Participants indicate that they are experienced both in writing own code but also in using code of others. Hence, we assume that they are familiar with writing comments to document their own code but also reading comments in other projects. Further results of the background questions can be found in the Appendix B.

8.2. Coherence between Interface Comments and Method Names

The first part of the survey, with 16 programmers taking part, investigates the coherence between interface comments and the name of the following method. The same approach can be used for the relation between interface comments and the name of the following attribute but was not examined within the scope of this work. In the category coherence, the quality model of Chapter 5 defines two facts, [Interface | RELATED TO METHOD NAME] and [Interface | EXPLAINING NON-OBVIOUS]. Interface comments should relate to the method name to clearly express the connection between the comment and the functionality of the following method. However, the interface comment should provide more information than contained in the method name. For example, if the method is named “getId()” and the comment states “/** get id */”, then the comment is redundant and could as well be deleted without loss of information.

¹<http://www4.in.tum.de/>

²<http://www.informatik.uni-bremen.de/st/index.php>

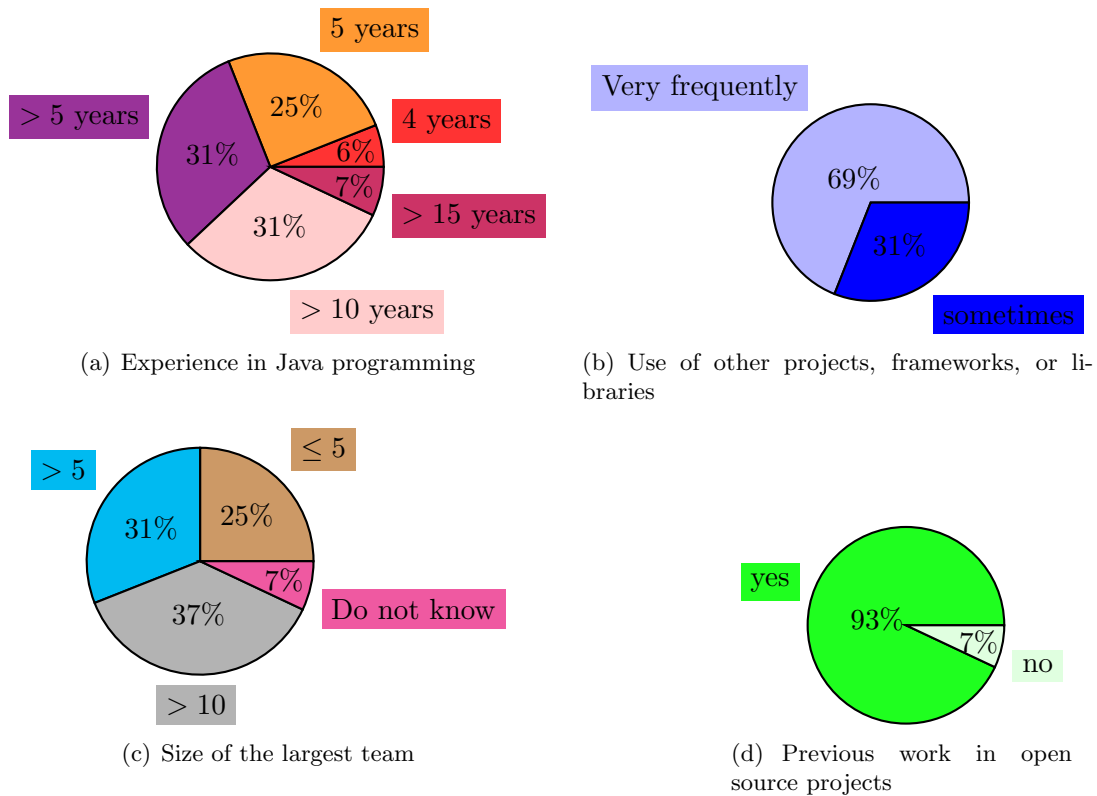


Figure 8.1.: Background of the participants of the survey

8.2.1. Metric

To measure the coherence between interface comment and method name we define a metric called *coherence coefficient*. Roughly speaking: To calculate the coherence coefficient, we extract words contained in the comment and compare it to the words contained in the method name. The comparison thereby counts how many words from one set correspond to an identical or at least similar word in the other set. This number of corresponding words divided by the total number of words in the comment denotes the coherence coefficient. The following enumeration provides a more detailed overview of the algorithm.

1. **Calculate list of words from normalized comment:** We enumerate words in the comment by splitting the comment string by white spaces. Beforehand, the comment is *normalized*: For normalization, we remove commented out code with the help of the machine learning feature as defined in Subsection 6.3.2. Further, comment normalization includes amongst others removing Javadoc tags, comment delimiters (“/**”, “//”, “*”, “**/” etc.), line breaks, and special characters. Words separated by white spaces are stemmed using the English language stemmer as implemented in ConQAT. To evaluate this metric, we only used code where both method name and comment were written in English.
2. **Calculate list of words from method name:** Java developers often follow the Sun Java Coding Conventions [8] to name methods: Thereby, the switch from a

lower case letter to an upper case letter indicates the start of a new word, using no other separating character. This habit is known as *camel-casing*. For example, `getPluginAuthor` represents the common method naming in Java. We use camel-casing as a word separator. After splitting the method name into its parts, words are stemmed in the same way as above.

- 3. Calculate coherence coefficient:** The coherence coefficient is based on the Levenshtein distance³ which defines the distance between two strings as the minimal number of edit operations to transform one string into the other with edits being insertion, deletion, or substitution of a single character. The algorithm counts how many words in the comment correspond to a word in the method name with a Levenshtein distance smaller than a certain threshold. Preliminary experiments showed that setting the threshold to 2 yields reasonable results.

This metric is very similar to the context feature used in machine learning for comment categorization as described in Section 6.3.3.

8.2.2. Research Question and Hypothesis

We expect that the coherence coefficient indicates whether the interface comment relates to the method name and also whether the interface comment provides additional information beyond the method name. Based on preliminary experiments with manual evaluation, we use the two thresholds of 0 and 0.5 to split interface comments into three groups: Comments with a coherence coefficient equal to 0, greater than 0 but less or equal to 0.5, and greater than 0.5. For the first and the third group, we propose one hypothesis:

Hypothesis 1

Comments with a coherence indicator equal to 0 indicate that the coherence between method name and comment is not sufficient: They should either have additional information to make the relation to the method name more obvious or they indicate that a better method name should have been chosen.

Hypothesis 2

Comments with a coherence indicator greater than 0.5 are trivial as they do not contain additional information.

For comments in the middle group with a coherence coefficient between 0 (excluding) and 0.5 (including) we do not formulate a hypothesis. We expect our thresholds to be sufficient to hold for Hypothesis 1 and 2. Ideally, with both thresholds being sharp, comments of the middle group should neither be trivial nor unrelated but contain additional information beyond the method name. In reality, we assume the middle group to be a gray area with most comments containing additional information but some comments being trivial or unrelated.

If Hypothesis 2 holds, comments with a coherence coefficient greater than 0.5 do not provide any gain for system commenting and do not help for system understanding. It remains debatable whether they should be removed or not. In particular, for an API

³http://en.wikipedia.org/wiki/Levenshtein_distance

```
1
2 /**
3  * Check for symmetry, then construct the eigenvalue
4  * decomposition
5  *
6  * @param A
7  *       Square matrix
8  */
9 public void calc(double [][] A) {
10     for (int i = 0; i < n; i++) {
11         for (int j = 0; j < n; j++) {
12             V[i][j] = A[i][j];
13         }
14     }
15
16     // Tridiagonalize.
17     tred2();
18
19     // Diagonalize.
20     tql2();
21     ...
22 }
```

Figure 8.2.: Example of an interface comment with a coherence coefficient of 0 where the method should be renamed

of the system based on Javadoc they might be required per default. Also, development environments such as Eclipse may produce warnings depending on user-settings when a method does not have an interface comment. For trivial methods such as getters or setters the method name possibly already contains all available information. Hence, the comment may not provide more information. However, we still expect the coherence coefficient to provide significant insights for analyzing comment quality. Even if there are reasons to not remove the comment, it is nevertheless valuable information how many comments are trivial. Combined with information about methods being one-line getters or setters this metric shows whether comments provide information beyond the method name.

We further expect comments with a coherence coefficient of 0 to be a strong indicator for bad comment or code quality, *i. e.*, a poor method identifier. Methods should be considered to be renamed if their interface comment contains some useful information but does not relate to the method name. Figure 8.2 shows an example of our data set where the comment contains useful information but the method name `calc` is not meaningful and should better be renamed to `calculateEigenvalueDecomposition`. Vice versa, if the method name is meaningful, the programmer should add more information to the comment to make the relation to the method obvious, see Figure 8.3 for an example.

```
1  /**
2   * Thanks to benoit.heinrich on forum.java.sun.com
3   * @param str
4   * @return
5   */
6  private String HTMLEncode(String str){
7      StringBuffer sbhtml = new StringBuffer();
8      for (int i = 0 ; i < str.length() ; i++){
9          char ch = str.charAt( i );
10         if ( ( ch >= '0' && ch <= '9' ) || ( ch >= 'A'
11             && ch <= 'Z' ) || (ch >= 'a' && ch <= 'z') ||
12             (ch == ' ') || (ch == '\n')){
13             sbhtml.append( ch );
14         }
15         else{
16             sbhtml.append( "&#" );
17             sbhtml.append( (int) ch );
18         }
19     }
20     return "<html><body>" + sbhtml.toString().replaceAll("\n", "<br>") + "</body></html>";
21 }
```

Figure 8.3.: Example of an interface comment with a coherence coefficient of 0 where the comment should provide more information

Question

To evaluate the coherence coefficient, the survey asks the participants to decide for each comment:

- It would not make a difference if the comment was not there because the comment is trivial. It does not provide additional information which is not already contained in the method name.
- The comment should have additional information because it is not obvious how comment and method name relate to each other.
- The method name could be more meaningful. The comment provides some useful information but a better method name could have been chosen.
- The comment provides additional information, which is not contained in the method name, and the method name is meaningful.

Additionally, the question provides the answer possibility “other” where the participant is able to enter a comment.

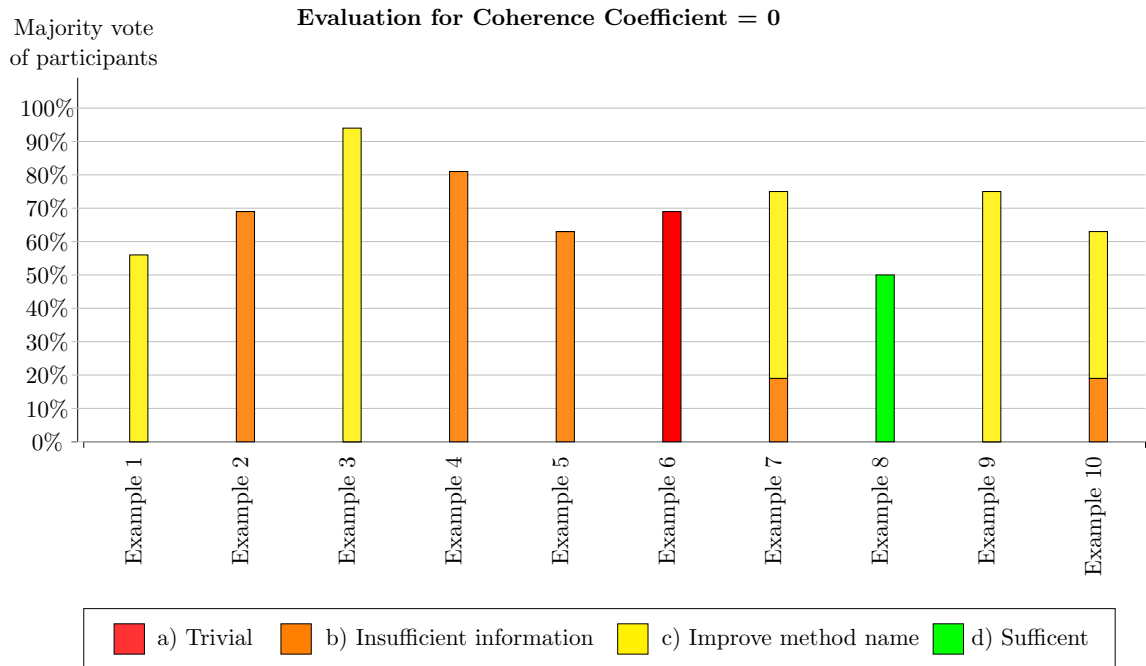


Figure 8.4.: Survey results for comments with a coherence coefficient of 0

8.2.3. Evaluation

To evaluate the hypotheses, we sample interface comments as identified by our machine learning classification according to their coherence coefficient: we group them into three categories based on the two thresholds 0 and 0.5 as explained above. From each group we randomly sample ten comments. In the questionnaire, we display all 30 comments in random order in order not to influence the opinion of the participants.

Figures 8.4, 8.5, and 8.6 show the results of the survey. The diagrams indicate the majority vote of the participants and visualize different answers with traffic light colors: Trivial comments are marked with red (answer *a*), comments with insufficient relation to the code with orange and yellow (answers *b* and *c*), and comments with additional information with green. For evaluation, answer *b* and *c* are unified as both indicate an insufficient coherence between comment and code. Hence, when the sum of votes for *b* and *c* constitutes the majority vote, we display both percentages in the same column: orange in the lower part, yellow in the upper part. A column being either completely orange or completely yellow indicates that the other response got 0% of the votes.

Figure 8.4 displays the answers for comments with a coherence coefficient equal to 0. Unifying answers *b* and *c*, the participants' majority vote constitutes an absolute majority for all ten examples. In five cases, the participants indicate that the comment is missing relevant information to make the relation to the method name more obvious. In three cases, they prefer to rename the method. Hence, in a total of 80% of the examples, developers vote for an insufficient relation between comment and method name, strongly supporting Hypothesis 1.

8. Coherence between Code and Comments

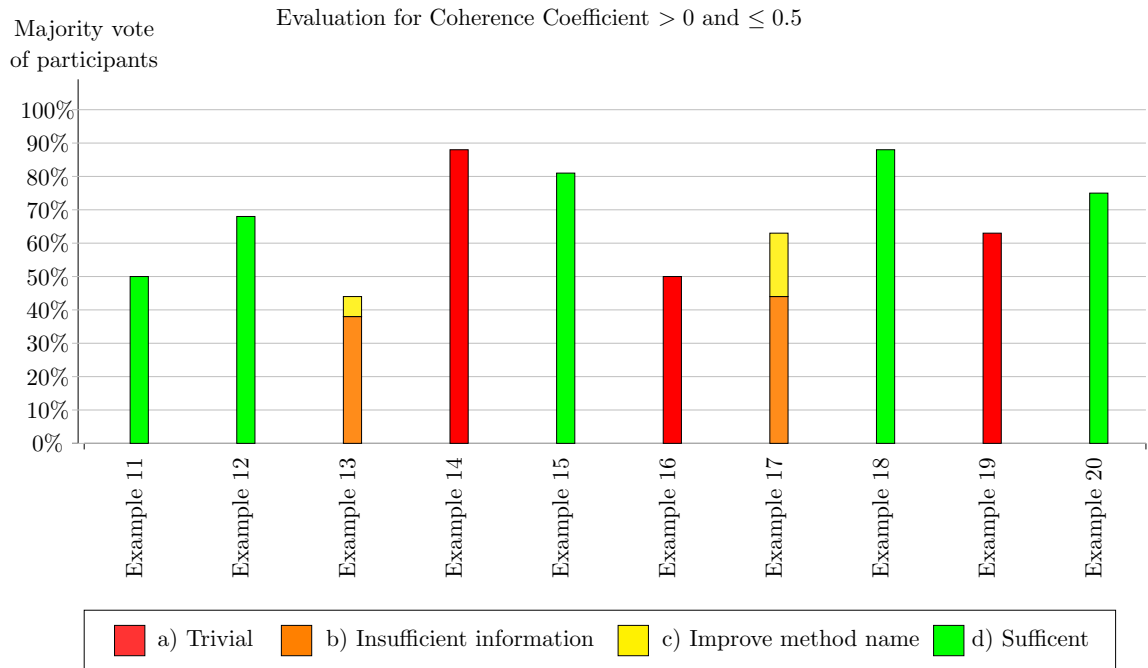


Figure 8.5.: Survey results for comments with a coherence coefficient > 0 and ≤ 0.5

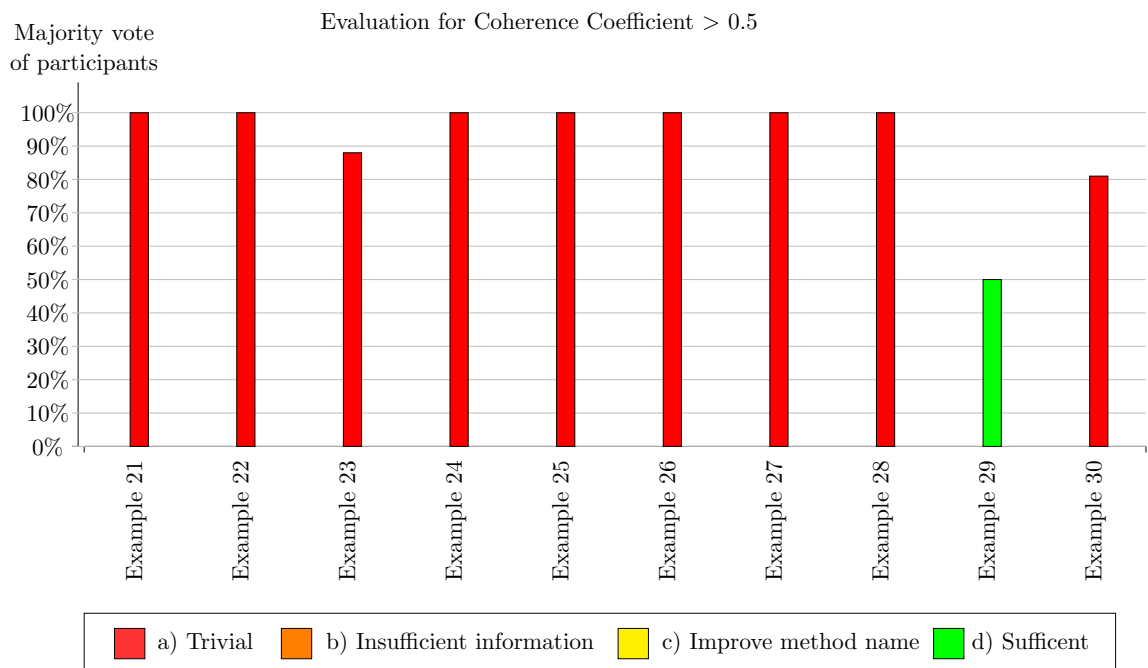


Figure 8.6.: Survey results for comments with a coherence coefficient > 0.5

The diagram in Figure 8.5 visualizes the answers given for comments with a coherence coefficient between 0 (excluding) and 0.5 (including). As assumed, the middle group is a

gray area: It reveals a tendency towards comments containing additional information but also contains trivial and unrelated comments: In five out of ten examples, the participants think that the comment contains additional information and are satisfied with the method name. In three cases, they consider the comment to be trivial, in two to provide insufficient information.

The rating of comments with a coherence coefficient greater than 0.5 can be found in Figure 8.6. In nine out of ten examples, the voters agree on the comments being trivial without information gain for system commenting. In all nine cases, the agreement is above 80%, hence very strong. Only in one example the participants consider the comment to contain additional information which is not expressed by the method name. This strongly confirms Hypothesis 2.

Conclusion

We conclude that the coherence coefficient metric is a useful metric to detect trivial comments and those with an insufficient relation to the method name in practice. Accepting a correctness of 90% for trivial comments, this analysis can be done fully-automatically. For comments with an insufficient coherence, the developer still needs to decide manually whether the method name should be improved or information should be added to the comment, which results in a semi-automatic analysis.

8.3. Length of Inline Comments

In the second part of the survey, we use the length of inline comments as an indicator of their coherence to the following lines of code. In total, there were 16 programmers participating in this part of the survey. In the model from Chapter 5, this corresponds to the fact [Inline | EXPLAINING NON-OBVIOUS]. Intuitively, shorter inline comments contain less information than longer inline comments but thresholds are not simply to define and also the role of very short or very long inline comments is not obvious.

8.3.1. Metric

The metric counts the number of words in a comment. Words are considered to be separated by white spaces. Beforehand, the comment is normalized in the same way as described in Subsection 8.2.1.

8.3.2. Research Question and Hypothesis

Based on preliminary experiments with manual evaluation, we set two thresholds, splitting all inline comments into three groups: Inline comments with at most two words, with more or equal to 30 words, and the remaining ones. We expect that the following hypotheses hold:

Hypothesis 1

Comments with at most two words should either be deleted or indicate that the following lines of code are better extracted into a new method with the comment

content expressed in method name. They contain only information that can be extracted from the following lines of code.

Hypothesis 2

Programmers want to keep comments with at least 30 words. They contain information that can not be extracted from the following line(s) of code.

Both hypotheses consist of two claims each which are to be evaluated independently. For comments with more than two but less than 30 words we expect that the length of the comment can not be used as a stand-alone indicator for either the delete/keep decision or a decision about the scope of information contained in the comment. Hence, we do not formulate a separate hypothesis for this group.

For each evaluated inline comment, we ask the participants to answer the following two (independent) questions:

Question 1

If you work on improving the quality of system commenting and you have the following choices, which one do you pick?

- a) Delete the comment because it is redundant and unnecessary. Without the comment the source code would not be harder to understand.
- b) Remove the comment by extracting a method. The lines of code following the comment can be extracted into a new method by using the content of the comment as a method name. After this refactoring, the comment would not exist anymore but be expressed in the method name.
- c) Keep the comment. The comment helps to understand the code and the system.

Question 2

Please decide between the following two options:

- d) The comment contains some information which can not be extracted from the following line(s) of code. The following describes an incomplete example list:
 - global information
 - explanations beyond the scope of the current method
 - general assumptions
 - design decisions
 - information describing the system's behaviour
 - information not obvious to express in code
 - potential risks or failures
- e) The comment contains only information which can be extracted from the following line(s) of code.

Both questions do not provide alternative answer choices.

8.3.3. Evaluation

To evaluate the hypotheses, we sample inline comments from the Java training data set (Section 6.2), excluding jMol. Beforehand, manual inspection revealed that inline comments from jMol have significantly less quality than inline comments in other projects, including but not limited to being out-of-date, at the wrong place, or just not understandable. From all other projects, we select randomly five inline comments. Out of this pool, we sample ten inline comments with at most two words, ten inline comments with ≥ 30 words, and ten comments with > 2 and < 30 words. Sampling from the preselected pool of comments ensures a uniform distribution over the quality of inline comments among different projects. We could have sampled ten comments per category from all inline comments of all projects. However, as some projects contain significantly more inline comments than others, the survey might have been biased towards the comment quality of a few individual projects instead of being representative for all projects.

First, Figures 8.7, 8.8, and 8.9 represent the results of Question 1. They visualize the vote distribution over all three possible answers, using traffic light coloring: Red indicates deleting the comment, yellow extracting a method by removing the comment, and green keeping the comment. For evaluation, we unify answer *a* and *b* because both represent a deletion of the comment. Hence, both answers are visualized in one column, using red color for the lower part, yellow for the upper part. We take the majority vote criterion to determine the final participants' answer per example.

For comments with at most two words, the majority of participants votes for deleting the comment in five cases. In two cases, 50% of the participants want to extract a new method and thereby remove the comment. Summing up, in 70% of all examples, the participants do not decide to keep the comment but to either remove it directly or remove it by extracting a new method. This supports the first part of Hypothesis 1. However, with an average majority vote of 64.3% in all ten decisions, the agreement among participants constitutes an absolute, but not very strong majority. For comments with at least 30 words, Figure 8.9 shows that in all ten cases developers want to keep the comment, with a very high agreement among each other of at least 88% in nine out of ten cases. This strongly validates part one of Hypothesis 2. As to be expected, Figure 8.8 reveals that programmers have no clear preference how to handle comments with more than two but less than 30 words: In five out of ten cases, they want to keep them, in three cases they would rather delete them completely, in one instance they would extract a method. One case results in a tie with 50% voting for keeping and 50% voting for deleting or extracting. The average agreement per decision is 60.2%.

The answers for Question 2 can be found in Figures 8.10, 8.11, and 8.12. Participants decide whether the comment contains some information that can not be extracted from the following line(s) of code which is considered to be global information and indicated with dark blue color. Information that can be extracted from the following lines of code is called local information and indicated with light blue color. Figure 8.10 reveals that eight out of ten comments with at most two words contain only local information, according to the survey participants, with an agreement of at least 75%. In contrast, comments containing at least 30 words contain global information in ten out of ten examples, with an agreement among voters of at least 92% in nine cases (Figure 8.12). Both facts support part 2 of Hypothesis 1 and Hypothesis 2. Comments between two and 30 words contain both local

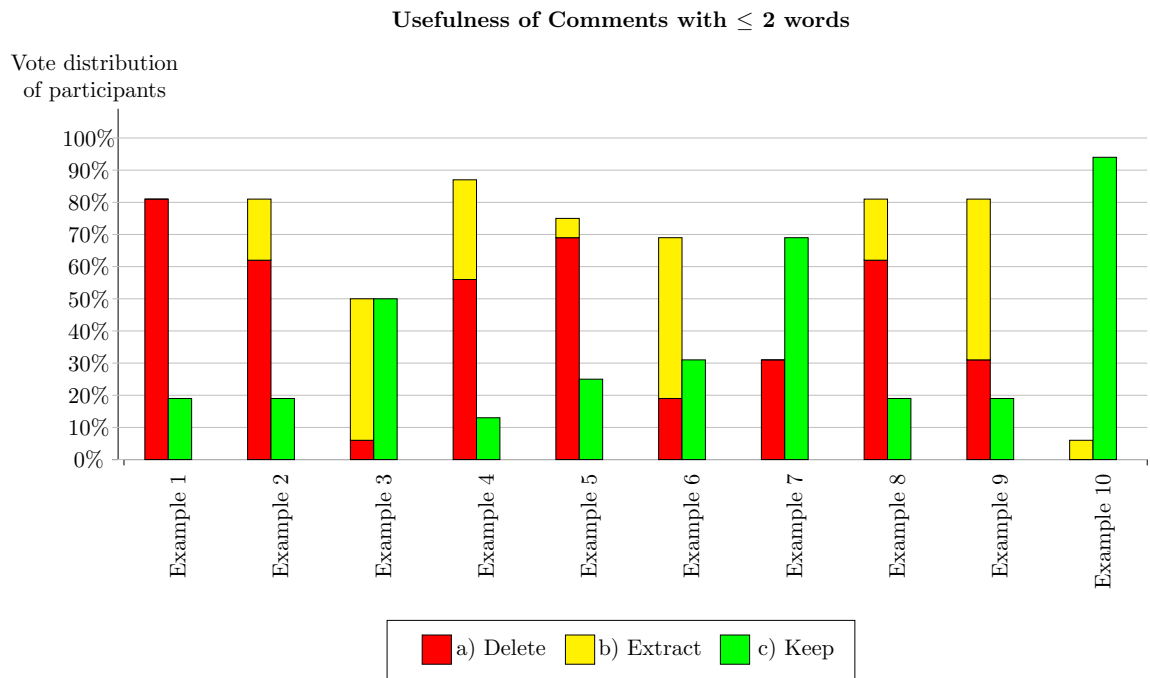


Figure 8.7.: Survey participants' decisions on how to deal with inline comments containing at most two words

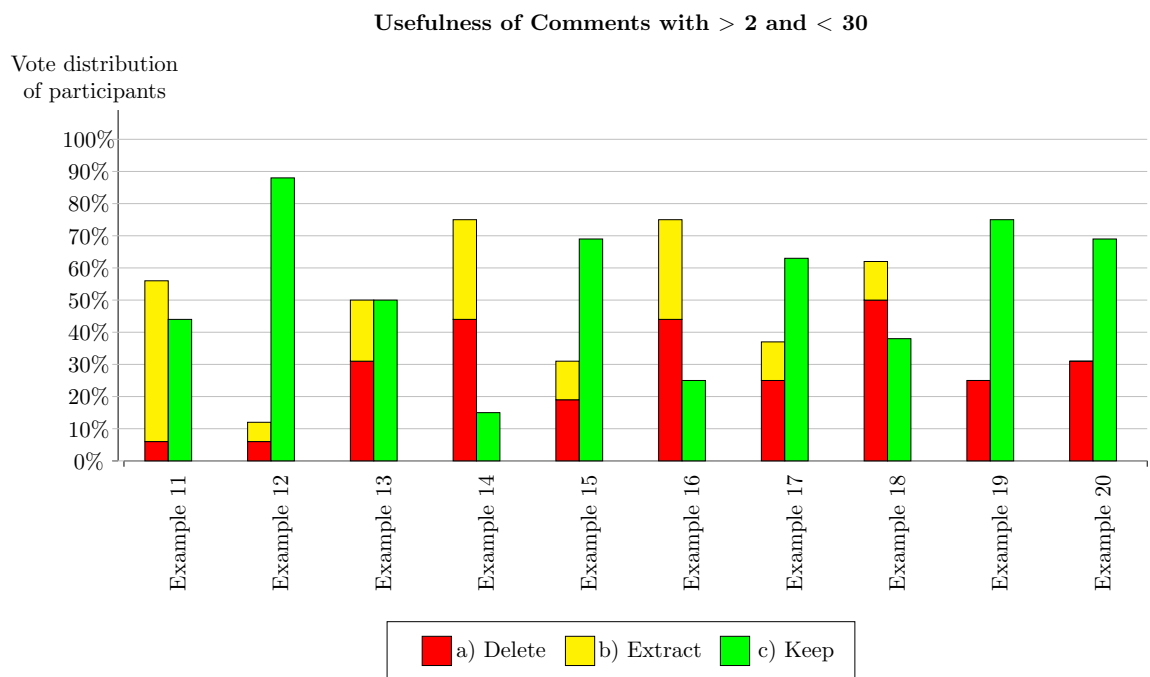


Figure 8.8.: Survey participants' decisions on how to deal with inline comments containing more than two and less than 30 words.

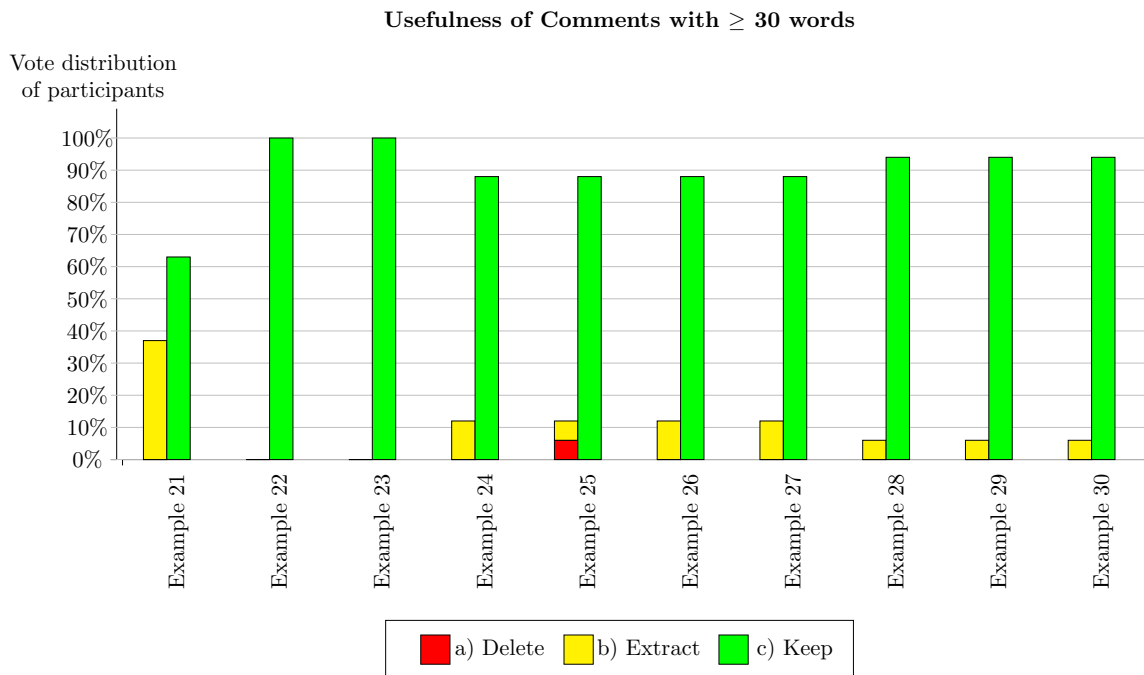


Figure 8.9.: Survey participants' decisions on how to deal with inline comments containing at least 30 words

and global information. Figure 8.11 shows that programmers consider six comments to be local, and four comments to be global, with a total average agreement of 73.4%.

Conclusion

Comments with one or two words strongly indicate local information which can be extracted from the following line(s) of code as well. Hence, these comments probably state the obvious, and survey participants tend to delete them or remove them by extracting a method. Consequently, these comments do not contribute to a better understanding of the code and can be detected fully-automatically using the length indicator. Interestingly, this metric can be used to suggest developers where lines of code should be extracted into a new method and hence give refactoring recommendations.

Contrarily, comments with more than 30 words contain significant global information which can not be extracted from the code and programmers want to keep them. The global information is non-obvious and promotes better understanding of the system. It would be inappropriate to conclude that inline comments should contain at least thirty words. In general, they should not. However, for a semi-automatic analysis and assessment of comment quality, the absence of such comments can potentially indicate that global aspects of the system are not commented. It remains to be verified if global aspects are documented elsewhere, such as in header comments or in external system documentation. Vice versa, too many inline comments being very long can also signal that too much global information needs to be documented in comments. This can potentially indicate that either

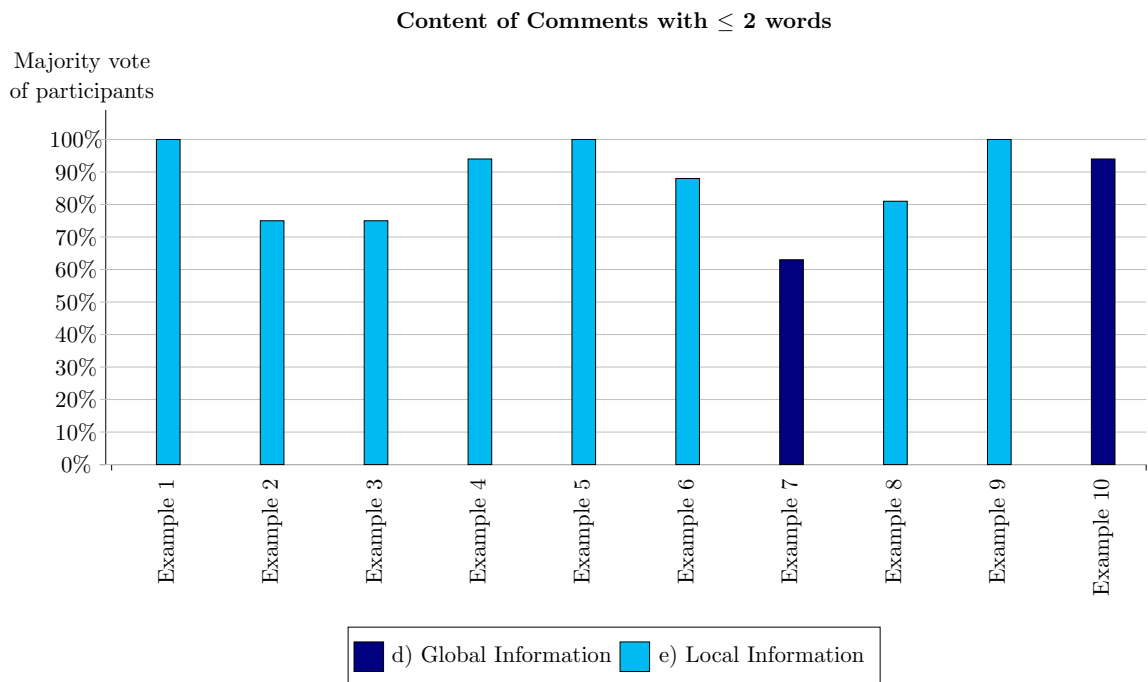


Figure 8.10.: Survey results with respect to global/local information content of inline comments with at most two words

architecture descriptions or framework documentation of the system are missing, domain-specific knowledge is not documented externally, or the system design is too general.

8.4. Necessity of Comments

In this section, we evaluate a third metric indicating the coherence between comment and code to assess the fact [Interface, Inline | EXPLAINING-NON-OBVIOUS]. 20 programmers evaluated this metric by filling out the questionnaire.

We perceive that comments containing the word *nothing* often do not have a meaningful content. A comment saying “nothing happens” or “does nothing” is usually also expressed by absence of the code and consequently explains the obvious. Hence, we use the following heuristic:

8.4.1. Heuristic

This heuristic indicates whether a comment contains the word *nothing*.

8.4.2. Research Question and Hypothesis

We assume that comments detected by the heuristic are either redundant and irrelevant, or they are relevant but lack important information: the comment “does nothing”, for example, might be relevant as it indicates a method not implemented on purpose. However, it also

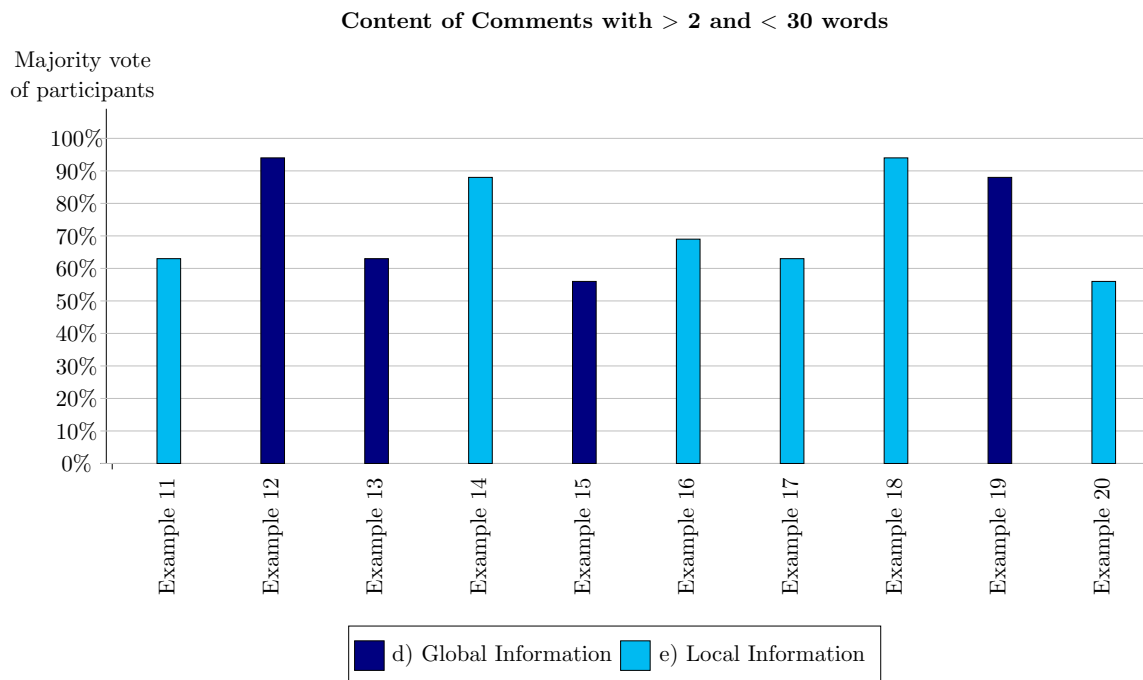


Figure 8.11.: Survey results with respect to global/local information content of inline comments with more than two and less than 30 words

automatically raises the question why this method does nothing. Such information should be documented within a comment. Hence, we propose the following hypothesis:

Hypothesis

Comments that contain the word *nothing* do not provide sufficient information. They are either completely unnecessary, or they are necessary but do not contain sufficient information.

For each comment under evaluation we ask the following question:

Question

Please rank the comment as

- a) Unnecessary: The comment is unnecessary because it does not provide additional information beyond the code.
- b) Necessary, but not enough: The existence of the comment contributes to a better understanding of the source code. However, the comment should provide more information.
- c) Necessary and enough: The existence of the comment contributes to a better understanding of the source code and provides sufficient information.

Additionally, the participant has the possibility to answer “Do not know” by entering a comment.

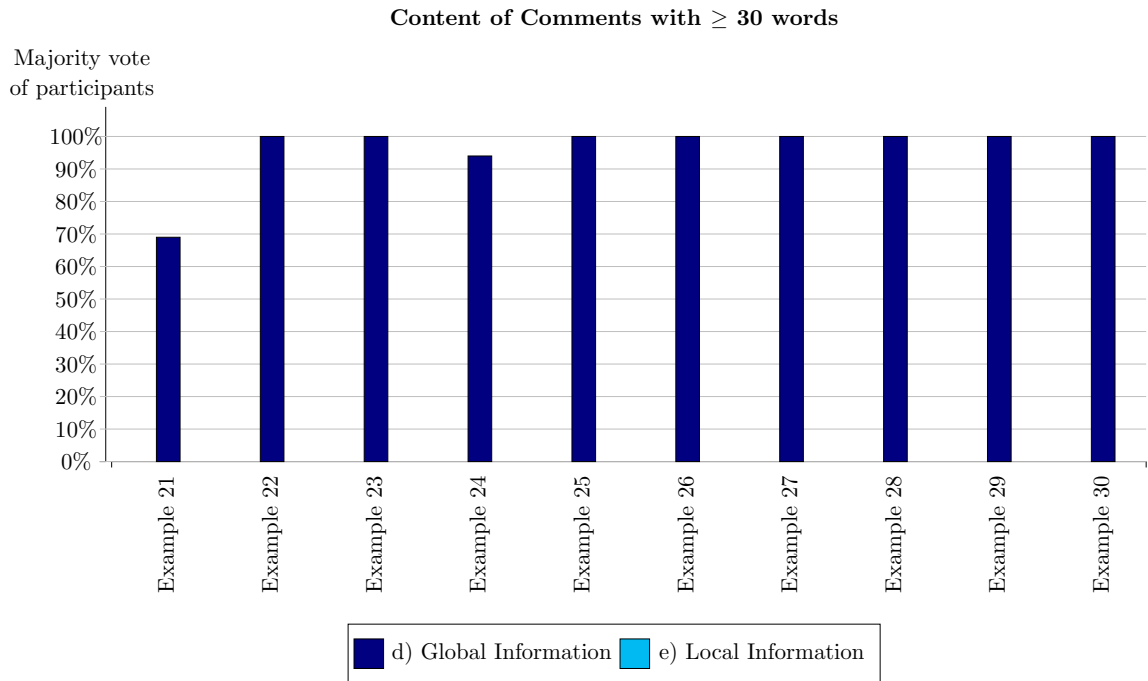


Figure 8.12.: Survey results with respect to global/local information content of inline comments with more than 30 words

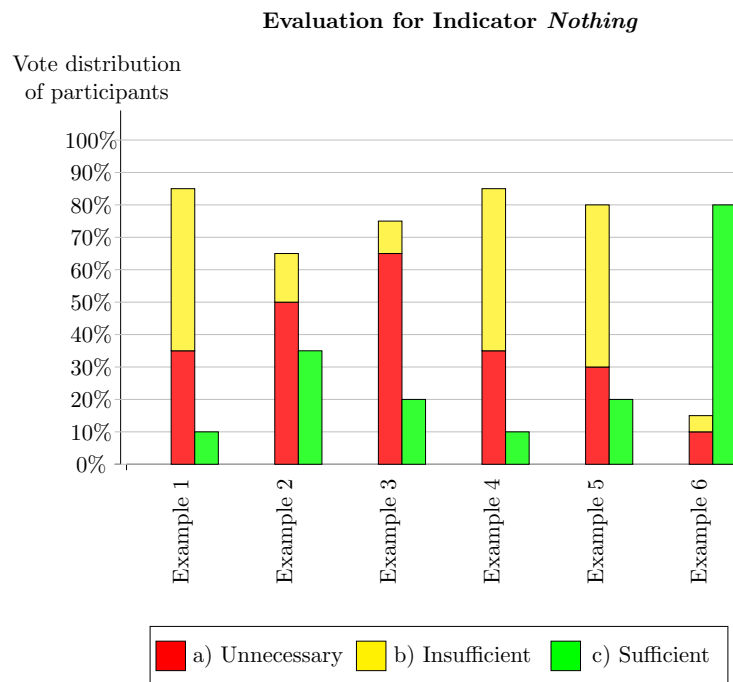


Figure 8.13.: Answers indicating the necessity of comments containing the word *nothing*

8.4.3. Evaluation

From the same test data set as in previous sections, we sample six comments containing the word *nothing*. Figure 8.13 shows the vote distribution among participants, using red color for answer “unnecessary” (answer *a*), yellow for “necessary but insufficient” (answer *b*), and green for “necessary and sufficient” (answer *c*). Thereby, answers *a* and *b* are unified for evaluation as both indicate that the participant does not rate the comment as sufficient. They are visualized in one column, using red in the lower and yellow in the upper part. Answers indicating “Do not know” are not visualized in the figure, hence the percentages of the three regular answers do not always sum up to 100%.

In two out of six cases, the majority of programmers perceives the comment to be unnecessary, in additional three cases to be necessary but insufficient. This means in five out of six cases the software engineer is not satisfied with the comment due to redundant or insufficient coherence to the code which strongly supports our hypothesis.

Conclusion

The heuristic detecting the word *nothing* retrieves comments that do not promote source code understanding as they are unnecessary or insufficient.

It remains an open research question if this heuristic combined with a length indicator can provide more information about the comment. Maybe, comments containing the word *nothing* are only unnecessary or insufficient when they contain a small number of words. Contrarily, more words might make up for the insufficient information provided by the word *nothing*.

9. Usefulness of Comments

Besides consistency, completeness, and coherence to code, usefulness makes up the fourth major criterion of the comment quality model (Chapter 5). It comprises facts such as [All | CLARIFYING] and [All | HELPFUL]. Section 9.2 describes an approach to detect comments that are confusing, Section 9.3 presents a metric to retrieve comments that are not helpful or indicate low source code quality. Both metrics are evaluated with the help of a questionnaire (Section 9.1).

9.1. Design of the Questionnaire

The questionnaire evaluating the metrics proposed in this chapter has the same design as described in Section 8.1. These metrics form the independent third part of the survey which was evaluated by 20 programmers. The data collected about their programming experience resembles the participants' data as gathered in Section 8.1.

9.2. Clarification through Comments

As expressed by the fact [All | CLARIFYING] in the quality model, comments serve the purpose to clarify source code. They are meant to help the reader to understand the source code and therefore they should not confuse while reading the code.

We perceive that a question mark used in a comment often creates confusion as the purpose of the question mark is not always obvious. Hence, in order to assess the clarification fact, we investigate if the presence of a question mark indicates that this comment confuses the reader.

9.2.1. Heuristic

The heuristic indicates whether the comment contains a question mark.

9.2.2. Research Question and Hypothesis

With the help of this heuristic, we make the following claim:

Hypothesis

Comments that contain a question mark do not help for clarification but confuse the reader.

For each comment under evaluation we ask the surveyees to answer the following question:

Question

Please indicate if the comment is

- a) Clarifying: The comment helps for clarification. It makes it easier to understand the source code.
- b) Confusing: After reading the comment, I am more confused than before. The comment creates uncertainty. It would have been better to leave the comment out.
- c) Neither clarifying nor confusing. The comment does not have any value.

Additionally, the question provides the answer possibility “Do not know”, providing the participant the opportunity to enter a comment.

9.2.3. Evaluation

For evaluation of our hypothesis, we randomly sample ten inline or interface comments containing a question mark from the Java training data set (Section 6.2). Figure 9.1 shows the participants’ vote distribution. Red indicates that programmers perceive the comment as confusing, green represents clarification, and yellow denotes that the comment is neither clarifying nor confusing. The figure does not represent answers voting for “Do not know”.

In nine out of ten cases, the programmers’ majority votes for confusion. Thereby, six decisions are based on an absolute majority. In two cases, where the answer is not based on an absolute majority, the vote for answer *c* (neither clarifying nor confusing) directly follows the vote for confusion: those who are not confused at least consider the comment to not have any value. In the remaining case, answers *b* and *c* receive the same vote. Hence, these comments do not help for clarification and do not promote understanding source code. This result strongly supports our hypothesis.

Discussion

Example 3 and 4 do not achieve an absolute majority among the voters. In both cases, the inline comment is written directly before an `if`-statement, describing the decision expressed by the `if`-statement, see Figure 9.2. We assume that the comment confuses some programmers because they understand the comment only after reading the following lines of code. Reading the `if`-statement makes it obvious that the question describes the `if`-structure and does not express a question raised by the author. Hence, we argue that the comment should be written without the question mark.

In contrast to the previous example, other comments containing a question mark directly express a question or doubts of the author, see Figure 9.3 (Example 6). The comment “can’t happen?” seems to be a question the author asked himself. Probably, the author could not provide a definite answer, so he commented on it. Consequently, an absolute majority of the developers is confused by this comment which hampers the quality of system commenting.

Conclusion

The heuristic detecting question marks retrieves comments that confuse the reader or do not promote system understanding.

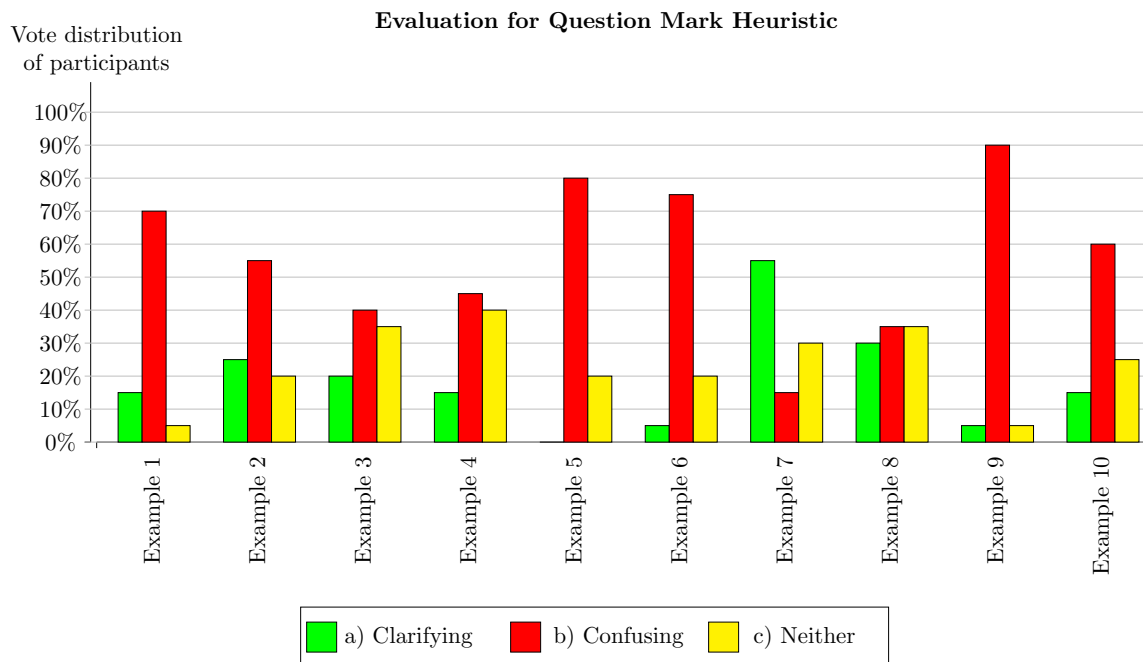


Figure 9.1.: Answers indicating clarification through comments with a question mark

9.3. Helpfulness of Comments

Programmers usually perceive comments that make source code understanding easier as helpful, expressed by the fact [All | HELPFUL] in the quality model. We examine whether an exclamation mark can be used as an indicator for comments that are not helpful as they do not promote source code understanding.

9.3.1. Heuristic

This heuristic detects comments that contain an exclamation mark.

9.3.2. Research Question and Hypothesis

Based on our experience, comments containing an exclamation mark make up a group of comments that is worth to be inspected manually during an assessment of comment quality. Exclamation marks seem to indicate special involvement of the author while writing the comment. We make the following claims:

Hypothesis 1

Comments with an exclamation mark indicate low quality of the source code.

Hypothesis 2

Comments that contain an exclamation mark are not helpful for source code understanding.

```
1
2 public PreprocessedCode getPreprocessedCode(IJavaElement javaClassElement)
3     throws ConQATException {
4
5     EcjCompilationResult ecjResult = JavaLibrary
6         .getEcjAST(javaClassElement);
7
8     if (ecjResult.getProblems().length > 0) {
9         throw new ConQATException("Could not compile "
10             + javaClassElement.getId() + ": "
11             + ecjResult.getProblems()[0]);
12     }
13
14     CompilationUnitDeclaration ast = ecjResult
15         .getCompilationUnitDeclaration();
16
17     if (ast.types == null || ast.types.length == 0) {
18         return null;
19     }
20
21     String code = javaClassElement.getTextContent();
22
23     // insertion of static modifiers?
24     List<Integer> positions = StaticAnalyzer.analyze(ast);
25     if (!positions.isEmpty()) {
26         code = insertStaticModifiers(code, positions);
27         ast = regenerateAST(javaClassElement, code);
28     }
29
30     return new PreprocessedCode(javaClassElement, ast, code);
31 }
```

Figure 9.2.: Inline comment with a question mark that confuses 45% of the participants (Example 4)

Both hypotheses are to be treated independently. For each comment under evaluation, the software engineers answer the following two (independent) questions:

Question 1

Please choose one of the following two options:

- a) This comment potentially indicates that the quality of the corresponding source code is rather low. While writing the comment, the author seemed to be aware of a potential bug or a hack. He decided to comment rather than to fix the code.
- b) This comment does not indicate any thoughts from the author about the quality of source code.

Question 2

Please choose one of the following two options:

- c) The comment is not helpful for source code understanding.
- d) The comment is helpful for source code understanding.

```

1
2 private void finishSaving(View view, String oldPath,
3 String oldSymlinkPath, String path,
4 boolean rename, boolean error)
5 {
6
7     //{{{ Set the buffer's path
8     // Caveat: won't work if save() called with a relative path.
9     // But I don't think anyone calls it like that anyway.
10    if(!error && !path.equals(oldPath))
11    {
12        Buffer buffer = jEdit.getBuffer(path);
13
14        if(rename)
15        {
16            ...
17        }
18        else
19        {
20            /* if we saved over an already open file using
21             * 'save a copy as', then reload the existing
22             * buffer */
23            if(buffer != null && /* can't happen? */
24                !buffer.getPath().equals(oldPath))
25            {
26                buffer.load(view, true);
27            }
28        }
29    } //}}}
30
31    ...
32 }

```

Figure 9.3.: Inline comment with a question mark that confuses an absolute majority of the participants (Example 6).

As both questions are Yes/No-questions, the participant does not have the option to answer “Do not know”.

9.3.3. Evaluation

We randomly sample ten interface or inline comments that contain an exclamation mark from the Java training data set. Figure 9.4 shows the survey’s results with respect to Question 1, Figure 9.5 with respect to Question 2. Answer *a* is shown with red, answer *b* with light green. The second figure represents answer *c* with orange and answer *d* with dark green. With respect to Question 1, programmers vote in four out of ten examples with absolute majority for the comment indicating low source code quality. Figure 9.6 shows Example 4. In the other six cases, at least 65% of the programmers judge the comment to not give any hints about source code quality. Hence, Hypothesis 1 is not strongly supported.

Regarding Question 2, the comment does not help the participant to understand the source code in seven out of ten cases. In one additional case, exactly 50% of the participants vote for each possible choice. In the remaining two examples, the participants perceive the comment as helpful. 70% of the examples not being helpful support Hypothesis 2.

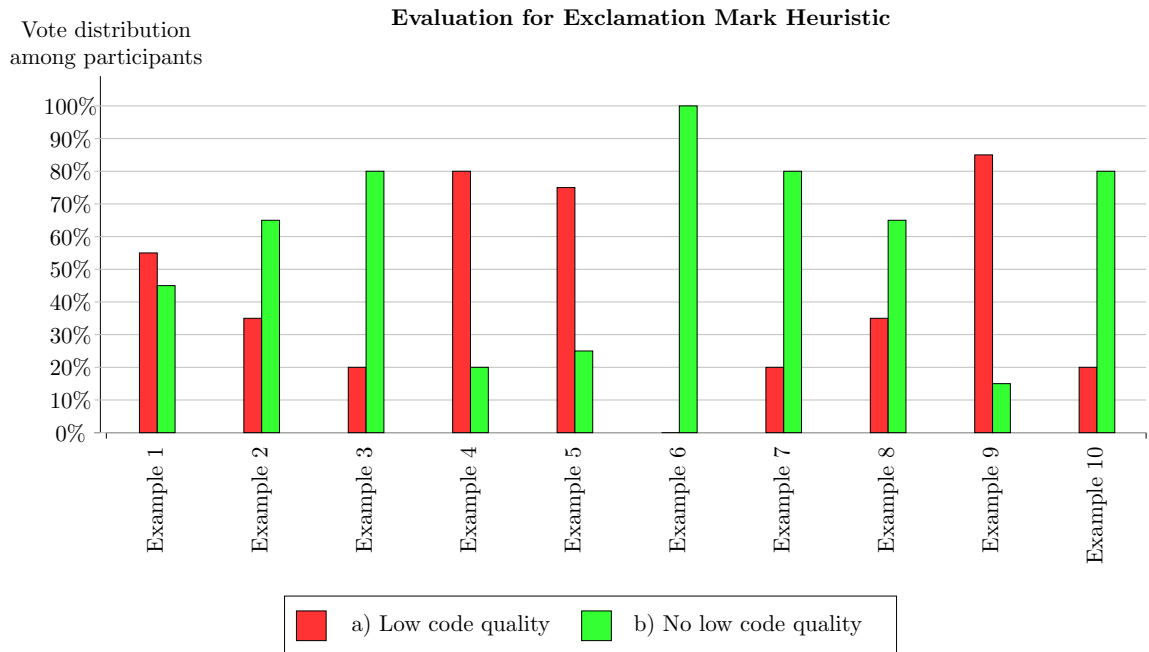


Figure 9.4.: Answers evaluating comments with an exclamation mark as an indicator for low source code quality

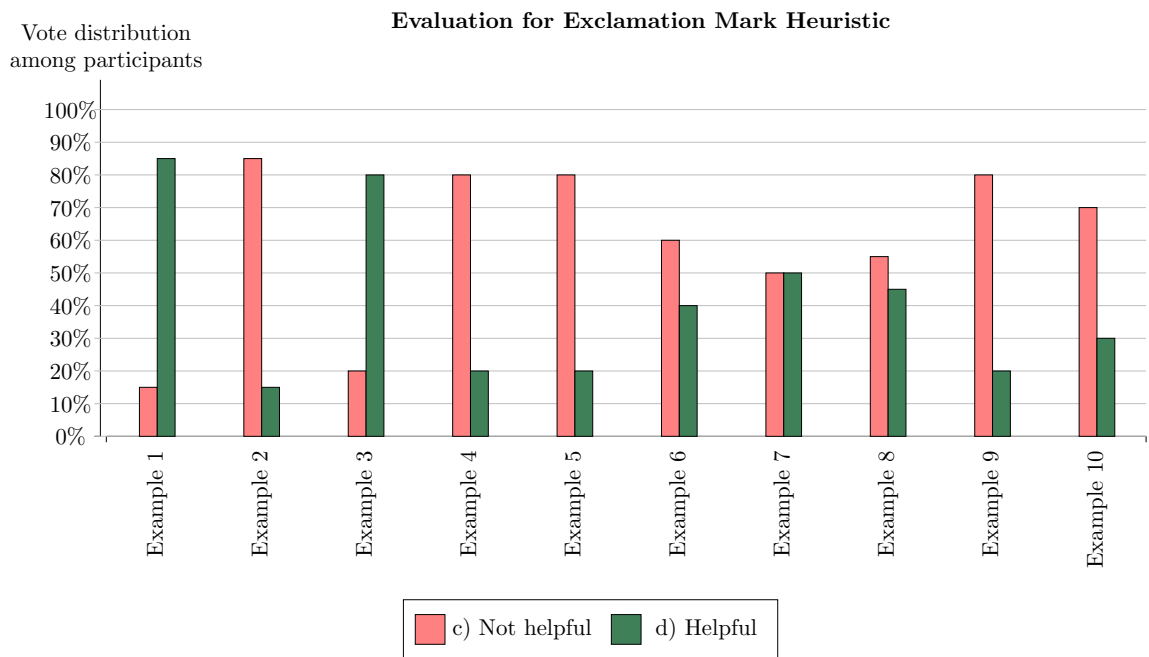


Figure 9.5.: Answers evaluating the helpfulness of comments containing an exclamation mark


```
1  /**
2   * CAUTION! returns a POINTER TO A TEMPORARY VARIABLE
3   * @param pointAngstroms
4   * @return POINTER TO point3fScreenTemp
5   */
6  synchronized Point3f transformPointNoClip(Point3f pointAngstroms) {
7      matrixTransform.transform(pointAngstroms, point3fScreenTemp);
8      adjustTemporaryScreenPoint();
9      return point3fScreenTemp;
10 }
```

Figure 9.6.: Inline comment indicating low source code quality (Example 4)

Conclusion

Based on the evaluation, we still consider the group of comments containing an exclamation mark to be relevant for manual inspection of system commenting. In 70% they are not helpful and hence do not promote source code understanding. As long as no better indicators are available, the exclamation mark heuristic is useful when looking for source code with low quality. Comments thereby can express weaknesses in the code that might not be detected by standard code quality metrics and hence are worth to be inspected manually.

10. Checklist for Comment Quality

To summarize our work and the associated implementation, this chapter provides a checklist for comment quality (Section 10.1), compares the comment quality of two open-source projects based on the checklist (Section 10.2), and discusses the use of the checklist for continuous quality management (Section 10.3).

10.1. Design of the Checklist

Figure 10.1 shows our checklist for an analysis and assessment of comment quality. Figure 10.2 thereby explains the use of symbols and colors in the checklist, as well as reference points for the usage of percentages. First, the comment ratio of a system is measured, which previous work employs as the standard and only metric. Whereas most previous work (*e. g.*, [6], [7]) measure the comment ratio in lines of code that consist of comments divided by the total number of lines of code, we denote the comment ratio based on character count and calculate the number of characters belonging to a comment divided by the total number of characters in the program. In a second step, we analyze the character count distribution over copyright, header, interface, inline, task, and section comments, as well as commented out code. Thereby, only header, interface, and inline comments can potentially contribute to system understanding. Section comments are to be treated as a hint for class extraction if source code improvement is possible. Commented out code should be cleaned up.

After this, we assess quality facts regarding completeness, consistency, coherence, and usefulness. Completeness is evaluated based on the percentage of files that contain copyrights and headers as well as the percentage of (public) methods and fields that contain interface comments. It remains to be decided for the specific use case whether the interface documentation should only be complete for public methods and fields or for all, including private and protected ones. Consistency of system commenting is assessed with the help of language recognition to determine if all comments are written in the same language and with a consistency check for copyrights. In terms of coherence, we evaluate how many interface comments are trivial or not sufficiently related to the method name. Trivial interface comments do not contribute to system understanding and their percentage should be subtracted from the overall percentage of comments potentially promoting system understanding. For unrelated interface comments, it remains to be decided whether the method should be renamed to create coherence to the comment or information should be added to the comment. For inline comments, we use the length indicator to find very short comments (≤ 2 words) and long comments (≥ 30 words). Short comments should either be removed because they explain the obvious or the following line(s) of code should be extracted into a new method with the comment content expressed in the method name. Their percentage number should also be subtracted from the overall comment percentage contributing to system understanding. If there are numerous very long comments, it remains to be manually inspected if the global view of the system should be better documented in an

architecture description. Contrarily, if very few or no long comments are available, it needs to be checked if global aspects of the system are documented elsewhere, such as in header or interface comments or external documentation. The coherence analysis also comprises comments that contain the word “nothing” as they were shown to be either unnecessary or necessary but insufficient. For the fourth major quality criterion, usefulness, we use the question mark and exclamation mark heuristic to find comments that are confusing or not helpful. To draw a conclusion, fact assessment for the criteria coherence and usefulness reveals comments that should be improved as they do not sufficiently contribute to system understanding.

10.2. Case Study

We conduct a preliminary case study to apply the comment checklist in practice and compare the comment quality of ConQAT and jMol (for references see Section 6.2). We evaluate the source code of the ConQAT Engine, excluding the incubator package which contains code still under development. Figure 10.3 and 10.4 show the results of both projects in a direct comparison.

The ConQAT Engine has an overall comment ratio of 40% among which 50% are copyrights. Headers, interface, and inline comments sum up to 47.2% of all comments. This reduces the overall comment ratio of the system which contributes to system understanding to maximal approximately 19%. The small percentages of section and task comments as well as commented out code can be neglected. In contrast, the overall comment ratio in jMol is only 25%, with 28% copyrights, 5% headers, 28% interface, and 21% inline comments. The jMol project has a large number of section comments (6%) and does not seem to be cleaned up yet as the percentage of commented out code denotes 11%. These numbers imply that the total comment ratio contributing to system understanding is maximal about 14%.

In terms of completeness, the ConQAT Engine achieves high scores, with 99% files having a copyright and 98% files having a header. In addition, 98% of the methods are attached with an interface comment. The ConQAT project settings require every method to have an interface comment, otherwise the compiler produces warnings. We assume this enhances the high percentage of commented methods. In jMol, 97% of the files contain a copyright, too, but only 37% of the files contain a header. This confirms the previous result of only 5% of the comments being header comments, indicating that classes are not sufficiently commented in jMol. The interface documentation for methods is even worse with only 16% of the methods having a comment.

Regarding consistency, the ConQAT Engine is furthermore well documented with respect to our assessment: the copyright is consistent and English is the only language used. The Tika language classifier detects 99.2% of the comments to be written in English. We interpret the missing 0.8% as classification errors. JMol, however, does not have a consistent copyright as our algorithm reveals 16 different copyright categories. The Tika language detector classifies 92% of the comments to be English, 3% to be German, and can not make a decision in 5%. Manual inspection reveals that the 3% German comments are classification errors which are to be neglected.

With respect to coherence between comments and code, 7% of all ConQAT interface comments are trivial, 10% are not sufficiently coherent to the method name. Among jMol interface comments, only 3% are trivial and 39% are not sufficiently coherent to the code. On the one hand, the project settings of ConQAT lead to a complete documentation of methods at a first glance. On the other hand, however, they might also facilitate developers writing quick and trivial interface comments. This reduces the comment ratio contributing to system understanding in ConQAT to maximal 18%. JMol developers barely write interface comments in general as only 16% of the methods are commented. Hence, it is not surprising that they abstain from writing trivial interface comments. However, 39% of the comments being insufficient indicates rather low interface documentation quality. Among inline comments in ConQAT, 20% have at most two words, 3% are longer than 30 words, compared to 44% too short and 4.5% too long comments in jMol. As the total number of inline comments in ConQAT (3%) is very small, the 20% of inline comments being too short don't have much influence on the general system commenting. Contrarily, the system commenting in jMol consists of 21% inline comments, with almost half of them being too short. Hence, this is another indicator for low comment quality in jMol. The same argument applies to inline comments with more than 30 words: 4.5% out of 21% of all comments being inline and too long results in a rather large number. It remains to be manually checked if an external architecture description is missing in jMol. For both projects, the relative percentage of inline comments containing the word *nothing* is with 0.8% and 0.2% very small. In jMol, about 4% of all inline comments have a question mark or an exclamation mark. These comments should be improved and inspected manually to reveal low source code quality.

Based on this discussion, we conclude that the ConQAT Engine is overall better commented than jMol. However, this case study is only meant to reveal preliminary results to get a first insight about how an assessment of comment quality can be done in practice. Future work is required to conduct a larger case study and compare the checklist results to manual quality judgment in a general manner.

10.3. Comment Quality in Continuous Quality Management

The initial, underlying purpose of comment quality analysis in this work has been a one-time audit of a software system. However, parts of the checklist in Figure 10.1 can be used for continuous quality management. We believe that a continuous assessment of the distribution over different comment categories combined with continuous information about completeness, consistency, and coherence can reveal general quality trends of software systems. In particular, the coherence analysis of trivial or unrelated interface comments and of inline comment length should be monitored on a regular basis. Contrarily, the nothing-, question mark-, and exclamation-mark heuristics are too simple to be used for continuous assessment: Once these comments are revealed by an audit, they can be adapted very easily such that they won't be detected by the heuristic again. For example, removing the exclamation mark will remove the comment from the findings but not necessarily improve the quality of the comment. Hence, these heuristics are more appropriate for an one-time audit.

Checklist for Code Comments

1 Comment Ratio

Comments %

2 Classification

Copyright	<input type="text" value="_____"/> %	
Header	<input type="text" value="_____"/> %	✓
Interface	<input type="text" value="_____"/> %	✓ Potentially contributing to system understanding
Inline	<input type="text" value="_____"/> %	✓
Section	<input type="text" value="_____"/> %	✎ Check for class extraction
Task	<input type="text" value="_____"/> %	
Code	<input type="text" value="_____"/> %	✎ Delete

3 Completeness

Percentage of files with copyright %

with header %

Percentage of (public) methods with interface comment %

4 Consistency

Are the copyrights consistent? Yes No Different copyrights #

Is English the only language used? Yes No English comments %

German comments %

5 Coherence

Interface comments which are trivial % ✎ Improve or delete

which are unrelated % ✎ Check for method renaming vs. adding information to the comment

Inline comments with ≤ 2 words % ✎ Check for method extraction or ✎ remove

≥ 30 words % ✎ If too many: Is an architecture view missing?

✎ If none: Is global information documented elsewhere?

Inline/Interface comments which are unnecessary or insufficient ("nothing"-heuristic) % ✎ Improve or delete

6 Usefulness

Inline/Interface comments which are confusing (?-heuristic) % ✎ Improve or delete

which are not helpful (!-heuristic) % ✎ Improve or delete

✎ Check src code quality

Figure 10.1.: Checklist for comment quality



<p>Explanation</p> <p>Use of Symbols</p> <p> Instruction to improve comment quality</p> <p> Instruction to improve source code quality</p> <p>Use of Color</p> <ul style="list-style-type: none">• Use of Green: Comments meeting a quality fact that positively influences system understanding• Use of Red: Comments indicating that either comment or source code quality can be improved <p>Reference for percent numbers</p> <ol style="list-style-type: none">1. Comment Ratio: Percentage of all characters in the source code which belong to comments2. Classification: Percentage of all characters in comments which belong to the given category3. Completeness: Percentage based on number of files and number of (public) methods.4. Consistency: Percentage of all characters in comments which are written in English (German)5. Coherence: Percentage of all interface (inline) comments that fulfill the given property, measured in number of comments.6. Usefulness: Percentage of all interface (inline) comments that fulfill the given property, measured in number of comments.

Figure 10.2.: Appendix of comment checklist

Comparison between ConQAT Engine and jMol			
1 Comment Ratio			
	ConQAT	jMol	
Comments	<u>39.8</u> %	<u>25.3</u> %	
2 Classification			
	ConQAT	jMol	
Copyright	<u>50.4</u> %	<u>27.8</u> %	
Header	<u>15.6</u> %	<u>5.4</u> %	
Interface	<u>28.6</u> %	<u>28.0</u> %	
Inline	<u>3.0</u> %	<u>21.4</u> %	
Section	<u>1.9</u> %	<u>6.1</u> %	
Task	<u>0.3</u> %	<u>0.3</u> %	
Code	<u>0.2</u> %	<u>11.0</u> %	
3 Completeness			
		ConQAT	jMol
Percentage of files	with copyright	<u>99.3</u> %	<u>96.7</u> %
	with header	<u>98.5</u> %	<u>36.6</u> %
		ConQAT	jMol
Percentage of all methods	with interface comment	<u>98.1</u> %	<u>16.1</u> %
4 Consistency			
	ConQAT	jMol	
Are the copyrights consistent?	Yes	No: # 16 categories	
Is English the only language used?	English comments	<u>99.2</u> %	English comments <u>91.5</u> %
	German comments	<u>0.6</u> %	German comments <u>2.7</u> %

Figure 10.3.: Quality of comments in ConQAT Engine compared to jMol (part one)

5 Coherence		ConQAT	jMol
Interface comments	which are trivial	<u>7.2</u> %	<u>3.3</u> %
	which are unrelated	<u>10.0</u> %	<u>39.4</u> %
Inline comments	with ≤ 2 words	<u>21.5</u> %	<u>44.4</u> %
	≥ 30 words	<u>3.3</u> %	<u>4.5</u> %
		ConQAT	jMol
Inline/Interface comments	which are unnecessary or		
	insufficient (“nothing”-heuristic)	<u>0.8</u> %	<u>0.2</u> %
6 Usefulness		ConQAT	jMol
Inline/Interface comments	which are confusing (?-heuristic)	<u>0.4</u> %	<u>1.7</u> %
	which are not helpful (!-heuristic)	<u>0.3</u> %	<u>1.6</u> %

Figure 10.4.: Quality of comments in ConQAT Engine compared to jMol (part two)

11. Conclusion

To conclude this work, we present a summary of our results (Section 11.1) and give an overview of future work (Section 11.2).

11.1. Results

This work constitutes the first detailed approach to a complex analysis and assessment of comment quality. As a fundamental contribution, we provided a machine learning approach to classify comments as copyright, header, interface, inline, section, task, or commented out code. The algorithm successfully learned comment classification both on Java and C/C++ systems, with slightly better results for Java due to a larger size of the training data and a feature representation specifically tailored for the Java programming language. The classifier trained on Java achieved an average precision and recall of 96%, using the J48 implementation of decision trees.

Furthermore, we proposed a quality model for code comments with a concrete definition of quality, differentiating between the four major quality criteria consistency throughout the project, completeness of the documentation, coherence between comment and code, and usefulness to the reader. By specifying different facts, the model distinguished between properties of the comments and activities of the developers. Therefore, we explicitly denoted the influence of certain system properties on developers' activities which made the model comprehensive.

We implemented several approaches to assess different facts of the quality model which ensure the reinforcement of the model in practice: In terms of comment consistency, we compared three language detection approaches for the English and German language which determine whether all comments of a system are written in the same language. The Tika language detection produced the best results with a precision of 99% and a recall of 86%, indicating high usability in practice. For copyright comments, we implemented a consistency check to decide whether a project has a unique copyright holder and a consistent copyright format. We showed that the algorithm is both sound and complete.

To assess facts from the quality criteria coherence and usefulness, we implemented different heuristic approaches, which were evaluated with the help of a survey among experienced software developers. The survey consisted of three independent parts with 16 to 20 participants each and revealed a variety of interesting results: To calculate the coherence between interface comment and method name, we proposed a metric based on the Levenshtein distance. Survey participants agreed that the metric can be used to filter two kinds of interface comments: trivial interface comments which do not provide additional information beyond the method name and interface comments with an insufficient coherence to the method name where either information has to be added to the comment or where the method name should be improved. Based on a precision of 90%, trivial comments can be detected fully-automatically. For comments with an insufficient coherence, developers manually need to

make the decision between method renaming and comment improvement. Furthermore, we found out that the length of a comment, counting the number of words, is a useful indicator for the coherence between comment and code: Inline comments with at most two words generally contain only local information. In 80% of the examples, developers wanted to either delete the comment completely or remove it by extracting a method as the comment does not provide additional information beyond the code. The decision between deletion and method extraction remains to be done manually. Interestingly, programmers considered comments with at least 30 words to contain important global information and wanted to keep them in ten out of ten cases. Comments with at least 30 words can give two hints about the system: If almost none of them are available, it remains to be manually checked if global aspects of the system are documented elsewhere. Contrarily, if many comments contain more than 30 words, maybe an architectural or global view of the system is missing. In addition, a majority of developers confirmed in the survey that comments containing the word “nothing” are either unnecessary or necessary but not sufficiently coherent to the code such that the comments do not contribute to system understanding.

For comment usefulness, we used heuristics detecting question and exclamation marks: In nine out of ten examples, participants considered comments containing a question mark to be confusing. In 70% of the test cases, comments with an exclamation mark were perceived to be not helpful. We also tested the hypothesis that comments with exclamation marks indicate low source code quality. Developers only confirmed this in four out of ten examples. However, as long as no better indicators are available, the exclamation mark heuristic combined with manual inspection is useful to detect source code with low quality.

Some of the proposed fact assessment methods suggest code refactoring activities (such as method renaming, method extraction, code quality improvement) as we considered an audit of comment quality as part of a general assessment of source code quality. We found out that comments thereby can express weaknesses in code that might not be detected by other software quality metrics or heuristics.

Finally, we provided a checklist for comment quality as a condensed version of our approach to be used for a one-time audit but partly also for continuous quality management. We applied the checklist to two open source projects, ConQAT and jMol, and compared the results to get a first intuition of assessing comment quality in practice. The primitive analysis computes a total comment ratio of 39.8% for ConQAT and 25.3% for jMol. However, comment categorization and coherence analysis reveal that among all comments at most 18% can contribute to system understanding in ConQAT, 14% in jMol. Further assessment shows that ConQAT is better documented in terms of consistency and completeness, whereas jMol has deficits in class and method documentation. Although this case study was only a preliminary experiment, it already showed that the numbers provided by the checklist are strong indicators to evaluate the commenting quality of a system.

To draw a conclusion, this thesis is the first approach to a detailed, quantitative and qualitative analysis and assessment of source code comments. We provided both an expressive model of comment quality as well as ways to reinforce the model in practice; we also evaluated these methods empirically and showed their successful acceptance among developers.

11.2. Future Work

We do not claim that the checklist in Chapter 10 contains an all-encompassing assessment of comment quality but that it builds an important foundation for further analysis. In a preliminary case study we used the checklist on two open source projects. An extension of this case study to more open source projects will show how the assessment reveals different comment quality benchmarks. So far, in our work, we have mainly analyzed code comments in Java. It will be interesting to see how our assessment can be transferred to different programming languages, maybe reevaluated with a larger survey.

Ideas for future work include consistency checks for grammar and spelling, ambiguity detection, or the analysis of readability of comments: In preliminary experiments, we tried to transfer the idea of ambiguity detection of [26] from requirement engineering to the domain of code comments by using similar keywords to detect ambiguities on the lexical level. For future work, such keywords need to be evaluated with the help of another survey to determine their usability in practice. Additionally, we tried to confirm the hypothesis of [12] that indices such as the Fog index or the Flesch reading ease level can help to assess the readability of code comments. However, first attempts have shown that evaluating readability metrics is a difficult task, which requires more work in the future.

On a completely different note, it might be interesting to investigate if an ontology representation of the system which is based on its comments can give information about the quality of system commenting. Ratiu and Deissenböck [31] propose to build a representation of the system based on identifiers and relations between them and compare it to real-world ontologies. Maybe comparing both the program representation as well as the real-world ontology to concepts extracted from comments can reveal aspects of the system which are not sufficiently commented. However, this idea goes beyond the scope of this thesis.

Appendix

A. Results of Copyright Consistency Check

The following listing shows copyright categories of the Java test set. Linebreaks were occasionally manually changed for layout within this document.

Category 1

```
/*
 * Copyright (c) , Tasktop Technologies and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * /XX/
 *
 * Contributors:
 *   Tasktop Technologies - initial API and implementation
 */
```

Category 2

```
/*
 * Copyright (C) R. Nagel
```

All programs in this directory and subdirectories are published under the GNU General Public License as described below.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by Revision X your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 0-USA

A. Results of Copyright Consistency Check

Further information about the GNU GPL is available at:
/XX/

*/

Category 3

```
/*
 * This file is part of VoTUM
 * Copyright (C) - Bogdan Mihaila
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * Revision X
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA 0-, USA.
 */
```

Category 4

```
/*
XXXX.java
 * :tabSize=8:indentSize=8:noTabs=false:
 * :folding=explicit:collapseFolds=1:
 *
 * Copyright (C) Matthieu Casanova
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * Revision X
 * of the License, or any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
```

```
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place - Suite 330, Boston,
* MA 0-, USA.
*/
```

Category 5

```
/*
XXXX.java
* :tabSize=8:indentSize=8:noTabs=false:
* :folding=explicit:collapseFolds=1:
*
* Copyright (C) , Slava Pestov
* Portions copyright (C) mike dillon
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
Revision X
* of the License, or any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place - Suite 330, Boston,
* MA 0-, USA.
*/
```

Category 6

```
/*
Date XX.XX.XXXX
* Copyright (C) by Andrea Vacondio.
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published
* by the Free Software Foundation;
Revision X
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
* See the GNU General Public License for more details.
* You should have received a copy of the GNU General Public License
* along with this program;
```

A. Results of Copyright Consistency Check

```
* if not, write to the Free Software Foundation, Inc.,
* 59 Temple Place, Suite 330, Boston, MA 0- USA
*/
```

Category 7

```
/*
XXXX.java
* :tabSize=8:indentSize=8:noTabs=false:
* :folding=explicit:collapseFolds=1:
*
* Copyright (C) , Slava Pestov
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
Revision X
* of the License, or any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place - Suite 330, Boston,
* MA 0-, USA.
*/
```

Category 8

```
/**
* <copyright>
*
* Copyright (c) - IBM Corporation and others.
* All rights reserved. This program and the accompanying materials
* are made available under the terms of the Eclipse Public License v1.0
* which accompanies this distribution, and is available at
/XX/
*
* Contributors:
* IBM - Initial API and implementation
*
* </copyright>
*
Date XX.XX.XXXX
*/
```

Category 9

```
/**
 * Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) </p>
 * <p>Company: </p>
 *
author Xxx Xxx
Revision X
*/
```

Category 10

```
/*
 * JBoss, Home of Professional Open Source.
 * Copyright , Red Hat Middleware LLC, and individual contributors
author Xxx Xxx
 * distribution for a full listing of individual contributors.
 *
 * This is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as
Revision X
 * the License, or (at your option) any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this software; if not, write to the Free
 * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
/XX/
*/
```

Category 11

```
/*
Date XXX XX, XXXX n *
 * Copyright (c) , the JUNG Project and the Regents of the University
 * of California
 * All rights reserved.
 *
 * This software is open-source under the BSD license; see either
 * "license.txt" or
/XX/
*/
```

Category 12

```
/*-----+
|
| Copyright - The ConQAT Project
|
| Revision X
| you may not use this file except in compliance with the License.
| You may obtain a copy of the License at
|
| /XX/
|
| Unless required by applicable law or agreed to in writing, software
| distributed under the License is distributed on an "AS IS" BASIS,
| WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
| See the License for the specific language governing permissions and
| limitations under the License.
+-----*/
```

Category 13

```
/*-----+
| ConQAT
|
| Date XX.XX.XXXX
|
| Copyright (c) - Technische Universitaet Muenchen
|
| Technische Universitaet Muenchen          #####
| Institut fuer Informatik - Lehrstuhl IV   ## ## ## ## ##
| Prof. Dr. Manfred Broy                   ## ## ## ## ##
| Boltzmannstr. 3                           ## ## ## ## ##
| 85748 Garching bei Muenchen               ## ## ## ## ##
| Germany                                   ## ##### ## ##
+-----*/
```

B. Background Data of Survey Participants

Figure B.1 shows further results of the survey about the participants' background regarding the following questions:

- How many years of programming experience do you have?
- What was the largest project you have ever been working on? (Measured in Lines of Code)
- Have you ever worked on commercial projects?

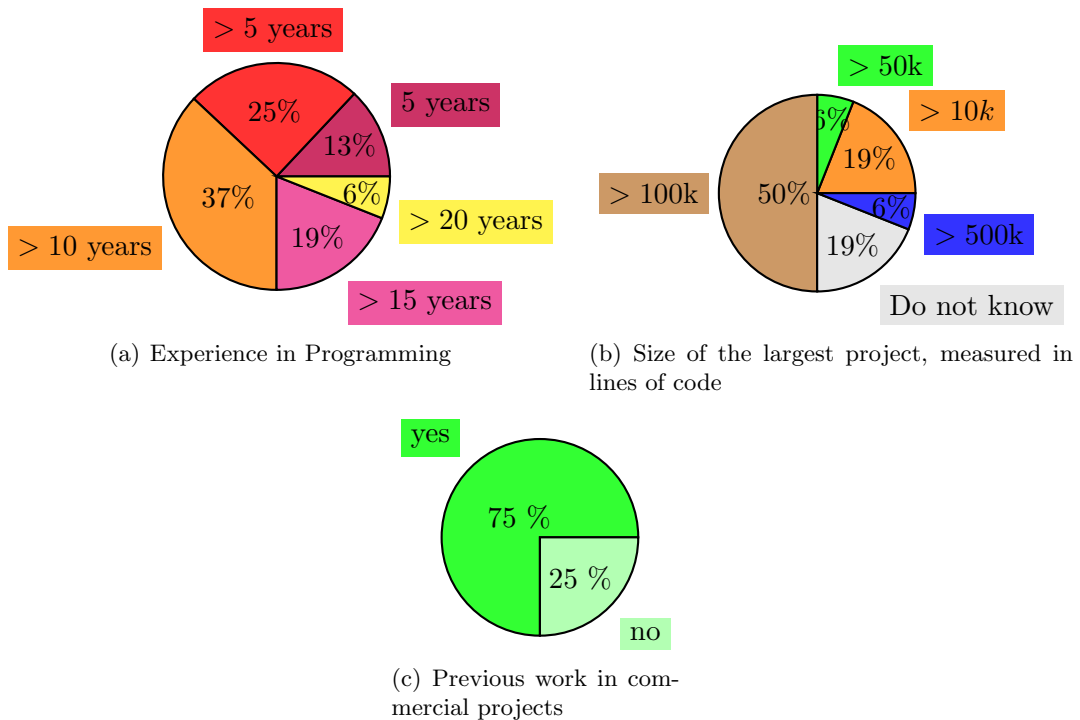


Figure B.1.: Background of the survey's participants

List of Figures

1.1.	Example of a useless source code comment which explains the obvious . . .	2
1.2.	Example of a comment intending to explain the interface but explaining the obvious	2
4.1.	Examples for each comment category in source code	14
5.1.	Activities in the quality model	16
5.2.	Entities in the quality model	17
5.3.	Relation between entities and activities	18
5.4.	Impacts between facts and activities (part one)	19
5.5.	Impacts between facts and activities (part two)	20
6.1.	Binary classification problem	26
6.2.	Binary classification problem which is not linearly separable	27
6.3.	Support vectors for a binary classification problem	33
6.4.	J48 decision tree for comment classification in Java	35
6.5.	REP decision tree for comment classification in Java	35
6.6.	J48 decision tree for comment classification in C++	38
6.7.	REP decision tree for comment classification in C++	39
7.1.	Copyright of the ConQAT project	46
7.2.	Two inconsistent copyrights	48
8.1.	Background of the participants of the survey	51
8.2.	Example of an interface comment with a coherence coefficient of 0 where the method should be renamed	53
8.3.	Example of an interface comment with a coherence coefficient of 0 where the comment should provide more information	54
8.4.	Survey results for comments with a coherence coefficient of 0	55
8.5.	Survey results for comments with a coherence coefficient > 0 and ≤ 0.5 . .	56
8.6.	Survey results for comments with a coherence coefficient > 0.5	56
8.7.	Survey participants' decisions on how to deal with inline comments containing at most two words	60
8.8.	Survey participants' decisions on how to deal with inline comments containing more than two and less than 30 words.	60
8.9.	Survey participants' decisions on how to deal with inline comments containing at least 30 words	61
8.10.	Survey results with respect to global/local information content of inline comments with at most two words	62

8.11. Survey results with respect to global/local information content of inline comments with more than two and less than 30 words	63
8.12. Survey results with respect to global/local information content of inline comments with more than 30 words	64
8.13. Answers indicating the necessity of comments containing the word <i>nothing</i>	64
9.1. Answers indicating clarification through comments with a question mark . .	69
9.2. Inline comment with a question mark that confuses 45% of the participants (Example 4)	70
9.3. Inline comment with a question mark that confuses an absolute majority of the participants (Example 6).	71
9.4. Answers evaluating comments with an exclamation mark as an indicator for low source code quality	72
9.5. Answers evaluating the helpfulness of comments containing an exclamation mark	72
9.6. Inline comment indicating low source code quality (Example 4)	73
10.1. Checklist for comment quality	78
10.2. Appendix of comment checklist	79
10.3. Quality of comments in ConQAT Engine compared to jMol (part one) . . .	80
10.4. Quality of comments in ConQAT Engine compared to jMol (part two) . . .	81
B.1. Background of the survey's participants	95

List of Tables

5.1. Categorization of facts into fully automatically (F-A), semi-automatically (S-A), and only manually (O-M) assessable	23
6.1. Classification of the test data set according to the label and the prediction of the classifier	27
6.2. Java machine learning data set grouped by projects	28
6.3. Java machine learning data set grouped by categories	29
6.4. C++ machine learning data set grouped by projects	30
6.5. C++ machine learning data set grouped by categories	30
6.6. Machine learning features for comment categorization	31
6.7. Results for J48 on Java	34
6.8. Results for REPTree on Java	34
6.9. Confusion matrix of the J48 algorithm on Java	36
6.10. Results for SVM on Java	36
6.11. Weight vector of hyperplane for SVM binary classification between code and header in Java	37
6.12. Results for J48 on C++	38
6.13. Results for REP on C++	38
6.14. Classification errors made by the J48 algorithm on C++	39
6.15. Results for SVM on C++	39
7.1. Correct decisions, errors, and failures in language detection	43
7.2. Language recognition with Tika	43
7.3. Language recognition with ASpell	44
7.4. Language recognition with ConQATLanguageDecider	44
7.5. Preprocessing of a copyright comment for the consistency check	47

Bibliography

- [1] Ted Tenny. Program Readability: Procedures Versus Comments. *IEEE Trans. Softw. Eng.*, 14(9):1271–1279, 1988.
- [2] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 215–223. IEEE Press, 1981.
- [3] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A Study of the Documentation Essential to Software Maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting and Designing for Pervasive Information*, SIGDOC '05, pages 68–75. ACM, 2005.
- [4] Carl S. Hartzman and Charles F. Austin. Maintenance productivity: Observations based on an experience in a large system environment. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 138–170, 1993.
- [5] Bennet P. Lientz. Issues in Software Maintenance. *ACM Computing Surveys*, 15(3):271–278, 1983.
- [6] Manuel J. Barranco García and Juan Carlos Granja Alvarez. Maintainability as a Key Factor in Maintenance Productivity: A Case Study. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 87–93, 1996.
- [7] Paul Oman and Jack Hagemester. Metrics for Assessing a Software System's Maintainability. In *Proceedings of the 1992 Conference on Software Maintenance*, pages 337–344. IEEE, 1992.
- [8] Inc. Sun Microsystems. *Code Conventions for the Java Programming Language*, 1997.
- [9] Brian W. Kernighan and Phillip J. Plauger. *The Elements of Programming Style*. McGraw-Hill, 1978.
- [10] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2008.
- [11] Beat Fluri, Michael Wursch, and Harald C. Gall. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 70–79, 2007.
- [12] Ninus Khamis, René Witte, and Jürgen Rilling. Automatic Quality Assessment of Source Code Comments: the JavadocMiner. In *Proceedings of the Natural Language*

- Processing and Information Systems, and 15th International Conference on Applications of Natural Language to Information Systems, NLDB '10*, pages 68–79. Springer-Verlag, 2010.
- [13] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. TODO or To Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 251–260. ACM, 2008.
- [14] Annie T. T. Ying, James L. Wright, and Steven Abrams. Source code that talks: an exploration of Eclipse task comments and their implication to repository mining. In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR '05*, pages 1–5. ACM, 2005.
- [15] Lin Tan, Ding Yuan, and Yuanyuan Zhou. HotComments: How to Make Program Comments More Useful? In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems, HOTOS '07*, pages 19:1–19:6. USENIX Association, 2007.
- [16] Dawn J. Lawrie, Henry Feild, and David Binkley. Leveraged Quality Assessment using Information Retrieval Techniques. In *Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC '06*, pages 149–158. IEEE Computer Society, 2006.
- [17] Andrian Marcus and Jonathan I. Maletic. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 125–135. IEEE Computer Society, 2003.
- [18] G. Antoniol, G. Canfora, A. de Lucia, and G. Casazza. Information Retrieval Models for Recovering Traceability Links between Code and Documentation. In *Proceedings of the International Conference on Software Maintenance, ICSM '00*, page 40. IEEE Computer Society, 2000.
- [19] Zhen Ming Jiang and Ahmed E. Hassan. Examining the Evolution of Code Comments in PostgreSQL. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06*, pages 179–180. ACM, 2006.
- [20] F. Deissenböck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard. An Activity-Based Quality Model for Maintainability. In *Proceedings of the 23rd International Conference on Software Maintenance, ICSM '07*, pages 184–193, 2007.
- [21] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit. *Characteristics of Software Quality*. North-Holland, 1978.
- [22] Jie Tang, Hang Li, Yunbo Cao, and Zhaohui Tang. Email data cleaning. In *Proceedings of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD '05*, pages 489–498. ACM, 2005.
- [23] Alberto Bacchelli, Marco D’Ambros, and Michele Lanza. Extracting Source Code from E-Mails. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, ICPC '10*, pages 24–33. IEEE Computer Society, 2010.

- [24] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.
- [25] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.
- [26] Benedikt Gleich, Oliver Creighton, and Leonid Kof. Ambiguity Detection: Towards a Tool Explaining Ambiguity Sources. In *Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 218–232, 2010.
- [27] Trevor Hastie and Robert Tibshirani. *Classification by pairwise coupling*, 1998.
- [28] Ron Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI '95*, pages 1137–1143. Morgan Kaufmann, 1995.
- [29] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
- [30] John C. Platt. Advances in Kernel Methods. chapter Fast training of support vector machines using sequential minimal optimization, pages 185–208. MIT Press, 1999.
- [31] Daniel Ratiu and Florian Deissenböck. Programs are Knowledge Bases. In *Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC '06*, pages 79–83. IEEE, 2006.