MASTER'S THESIS

A search for the Ten Commentments: An exploratory study on automated quality assessment of comments in Java source code

Lung, C.

Award date:
2022

A search for the Ten Commentments: An exploratory study on automated quality assessment of comments in Java source code

Chun Fei Lung

29 April 2022

A search for the Ten Commentments: An exploratory study on automated quality assessment of comments in Java source code

Chun Fei Lung

29 April 2022

Submitted to the Open University of the Netherlands
in partial fulfilment of the requirements for the degree of
Master of Science in Software Engineering

Student number:

Course: IM9906 – Software Engineering Graduation Assignment

Institution: Open University of the Netherlands
Faculty: Faculty of Science
Programme: Master's Programme in Software Engineering

Graduation committee
Primary supervisor: dr. Ebrahim Rahimi
Secondary supervisor: prof. dr. Erik Barendsen

Abstract

Context It is not always clear to developers how they should write good comments, nor are there many tools that help developers assess the quality of their comments.

Objective Our goal is to gain better insight into the features that are associated with good source code comments, by developing a predictive model that can automatically assess the quality of comments in a software project.

Method First we derive features that may affect comment quality from scientific literature and compare them with those found in open-source Java projects. We then conduct an online survey among software developers to gather quality ratings for a diverse set of comments, which we use to construct a predictive model for comment quality.

Results Our results suggest that a wide array of features exist for comments, but not all may be equally discernable in open-source projects. Our survey shows that a lot of disagreement exists between different developers about which comments are high-quality, presumably due to factors which cannot be derived from the source code itself. Consequently, our predictive models are only able to partially explain the variance in ratings given by developers.

Conclusion Disagreement among developers about what constitutes a high-quality comment poses challenges for the construction of automated predictive models for comment quality.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In the mid 1970s Brooks [5] suggested that the 'total lifetime cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it'. This number was expected to rise as software played an increasingly important role in society [2]. Indeed, it is estimated that in some cases up to 90 percent of a system's lifetime cost can be attributed to maintenance [66].

A large part of that maintenance effort is not spent on actual development of code, but rather on reading and understanding code [18, 60]; an activity that is typically referred to as program *comprehension*, program *understanding*, or program *readability*. Here we use the term program comprehension. Koenemann and Robertson [40] define it as follows:

Definition 1 (Program comprehension)**.** The process of understanding program code unfamiliar to the programmer.

Program comprehension can be supported in several ways, but documentation is arguably the most important way to augment maintainers' understanding of the implementation of a software system. Good documentation – together with good software design and coding practices – can improve the maintainability of a software system, thus reducing the effort required to maintain it [15]. Maintainability is one of the characteristics in the ISO/IEC 25010 software product quality model, which provides the following definition [35]:

Definition 2 (Maintainability)**.** Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers.

Although many different types of documentation exist, a survey by de Souza et al. [15] shows that software maintainers primarily rely on source code and *comments* in their program comprehension process. Comments in source code can convey information that cannot be derived clearly from the source code alone, and as such can play an considerable role in supporting the maintenance of software. ISO/IEC/IEEE [36] define comments as follows:

Definition 3 (Comment)**.** Information embedded within a computer program, job control statements, or a set of data that provides clarification to human readers but does not affect machine interpretation.

Comments come in many shapes and forms. We list a few important types for the Java programming language here, as described by Steidl et al. [69]. For an

Listing 1.1: This header comment provides a brief description of a class in the RDF4J project and lists its two authors. It has been slightly edited to fit within the confines of this page.

```
/**
 * An iterating implementation of the {@link GraphQueryResult} interface.
 *
 * @author Arjohn Kampman
 * @author Jeen Broekstra
 */
public class IteratingGraphQueryResult extends IterationWrapper<Statement,
        QueryEvaluationException> implements GraphQueryResult {
```

exhaustive overview of different comment types and their purposes, we refer to Pascarella and Bacchelli [57].

Header comments (alternatively: *class comments*) appear near the top of a class or file. They explain the purpose or meaning of the class itself and may include additional metadata. Listing 1.1 shows an example of a header comment.

Member comments (alternatively: *method* or *field comments*) are placed above fields or methods (Listing 1.2). These comments tell those who wish to call the method what it does and how it should be used.

Javadoc comments[1] are a special type of comment that also contains semi-structured descriptions of entities like method parameters and can be used to automatically generate API documentation. Similar concepts exist in other languages, e.g. godoc for Go and pydoc for Python.

Inline comments typically appear within method bodies and are used to convey information about their implementation (Listing 1.2). They are primarily written for maintainers.

Code comments contain code which has been commented out (Listing B.7). This can be done for different reasons: it might have been intended for debugging purposes and accidentally left in, or be meant for future reuse.

Task comments tell developers that some task still needs to be done (Listing 1.4). Todo comments are a common type of task comment.

Not all comments are equally valuable. For instance, conventional wisdom suggests that comments are only useful if they are correct, up-to-date, and provide information that cannot be easily – or even feasibly – extracted from the code itself [30]. An example of a comment that is superfluous can be seen in Listing 1.5: it does not convey any additional information beyond what can be deduced from the class and method name.

Normative sets of guidelines can often be found in professional handbooks [68]. Some notable examples include *Clean Code* [45], *Code Complete* [70], and

[1] *Javadoc* – https://docs.oracle.com/en/java/javase/14/javadoc/javadoc.html

Listing 1.2: A member comment and inline comment in Java

```
/**
 * Reverses the order of the elements in the specified list.<p>
 *
 * This method runs in linear time.
 *
 * @param list the list whose elements are to be reversed.
 */
public static void reverse(List<?> list) {
    int size = list.size();
    if (size < REVERSE_THRESHOLD || list instanceof RandomAccess) {
        for (int i=0, mid=size>>1, j=size-1; i<mid; i++, j--)
        swap(list, i, j);
    } else {
        // instead of using a raw type here, it's possible to capture
        // the wildcard but it will require a call to a supplementary
        // private method
        ListIterator fwd = list.listIterator();
        ListIterator rev = list.listIterator(size);
        for (int i=0, mid=list.size()>>1; i<mid; i++) {
            Object tmp = fwd.next();
            fwd.set(rev.previous());
            rev.set(tmp);
        }
    }
}
```

Listing 1.3: A method in which a single line of code has been commented out presumably to deactivate it.

```
public void start(BundleContext context) throws Exception {
    logger.debug("Starting org.protege.common bundle");
    context.registerService(
    javax.xml.parsers.SAXParserFactory.class.getName(),
    javax.xml.parsers.SAXParserFactory.newInstance(), null);
    //  CommonProtegeProperties.getDataDirectory().mkdir();
    if (logger.isDebugEnabled()) {
        startDebug(context);
    }
}
```

Listing 1.4: This comment was written to remind its author or other maintainers of an edge case that is yet to be handled.

```
public final void setTagLabel(String tagLabel) {
    // TODO what if tagLabel is null or empty ?
    this.tagLabel = tagLabel;
}
```

Listing 1.5: Example of a comment that does not convey any information beyond what can be deduced from the code

```
public class WorkspaceNameValue implements SkyValue {
    /**
     * Returns the name of the workspace.
     */
    public String getName() {
        return workspaceName;
    }
```

The Pragmatic Programmer [34]. However, it is often unclear if the advice in these books is based on conjecture and anecdotal evidence rather than empirical observations.

The importance of empirical evidence for such guidelines is highlighted by the existence of studies that have found counter-intuitive effects of comments. For instance, Börstler and Paech [3] discovered that source code *with* comments can be perceived as more readable than source code *without* comments – even if those comments are 'bad' and should not affect comprehension.

This can make it hard for developers to determine how they should go about writing comments. While numerous tools exist to help developers assess the quality of their code, tools that help developers assess the quality of their *comments* are few and far between.

1.1 Objective

The lack of tooling and clear, empirical evidence for guidelines on what makes a good comment makes it hard for developers to understand when and how they should write comments.

While there are studies on the effect of comments on program comprehension, most are more than 20 years old [3]. This suggests that our understanding of comments on program comprehension in modern contexts is limited at best [52]. Two recent examples include studies by Khamis et al. [37] and Steidl et al. [69]. In both studies a model for comment quality was designed. However, the approach by Khamis et al. is entirely based on heuristics and has not been validated by developers. In contrast, the approach by Steidl et al. *is* partially based on surveys with developers, but does not appear to be fully automated. Moreover, in neither

case is it entirely clear why certain features were considered for their models and which were left out.

Our **research objective** is therefore to gain better insight into the features that are associated with good comments. We do this by developing a predictive model that can automatically assess the quality of comments in a software project.

To clarify what we mean with the quality of comments, we provide the following definition of comment quality, which is partially based on the ISO/IEC/IEEE [36] vocabulary definition of 'quality' and the definition of overall code quality by Stegeman et al. [68] as we could not find any succinct definitions in literature:

Definition 4 (Comment quality)**.** The degree to which a comment clarifies a piece of source code to its reader, determined by just looking at the comment and the source code it describes, i.e. without any form of testing or checking against specification.

We limit the scope of our study to English-language comments in open-source Java projects, largely for practical reasons:

- Most open-source software (OSS) projects are developed internationally and use English as their *lingua franca*. Limiting our scope to English means that the population from which we can gather input for our model is as large as possible, whilst avoiding potential issues with translation.
- Automated analysis of the textual content of comments requires the use of computational linguistics tools and techniques. These are much more mature for global languages like English than for regional languages, like Dutch or German.
- Source code from open-source projects can be easily discovered, accessed and reused, unlike source code from closed-source projects which may require lengthy, manual approval processes and non-disclosure agreements.
- Java is currently one of the most popular programming languages, especially in empirical research. Most contemporary studies base their findings on Java. By focussing on Java, we can reuse existing knowledge and tools from those studies.

1.2 Contribution

The theoretical contribution of this study is twofold. Firstly, we provide an overview of scientific research on features of comments. Secondly, we develop a predictive model that helps us gain better insight into the role of comments in the program comprehension process. Our model should also be usable as part of more comprehensive models of software maintainability.

For practitioners, we aim to gather insights from literature and our own findings that can be converted into empirically validated guidelines, are easy to follow and ideally can be assessed automatically as part of developers' daily development and continuous integration workflows. This may help developers write comments for their source code that are more helpful to those who will maintain it, which improves maintainability of the software system and may eventually lead to a decrease in maintenance costs.

1.3 Outline

The remainder of this thesis is structured as follows: In chapter 2 we present a brief overview of research on automated assessment of code and comments in source code. This provides groundwork for our study, which is needed for the subsequent parts of the thesis. Chapter 3 decomposes our overall research objective into three research questions that are more adequately scoped. It also discusses the four-phase approach that we will use to answer those research questions. We present the results for each of the four phases of our study in chapter 4. These are then discussed in chapter 5, along with the limitations of our study and opportunities for future work. Finally, we conclude this thesis by summarising the main findings of our study in chapter 6.

Chapter 2

Background

The objective of our study is to gain better insight into features of good source code comments by developing a predictive model for comment quality. As briefly mentioned in section 1.1, there have already been studies on the effect of comments on program comprehension and proposals for models for comment quality. This chapter summarises that existing work.

2.1 Static analysis of source code

Maintainability is an internal quality of software. This means that although it affects the ease of its development and maintenance, it cannot be observed directly. It is however possible to estimate how maintainable a software project is by analysing its code using a metric or model.

Historically, many metrics and models have been proposed to assess the maintainability of software projects. Some metrics are relatively simple and only produce simple values.

The best known example are lines of code (LOC) metrics, which count the number of lines of code in a software project. These assume that projects with more lines of code are harder to maintain [25, 77]. Simple variants that count the number of physical lines are easy to comprehend and implement. However, they are also sensitive to variables like coding styles and language verbosity, which can greatly affect how many lines of code are needed to implement certain functionality. More sophisticated approaches count the number of logical lines, where each line consists of a single statement that may or may not be spread over multiple physical lines [51]. They might also exclude comment lines, which are supposed to make code *easier* to maintain.

The general idea of LOC metrics is that larger volumes of code are harder to understand, but in reality it is more likely that understandability – and thus maintainability – is affected by the number of possible outcomes that can be generated by a unit of code, e.g. due to the presence of conditional statements such as `if` and `while`. McCabe's cyclomatic complexity is a metric that can be used to compute the number of linearly independent execution paths through a unit of code [46]. Newer metrics, like cognitive complexity [48], improve upon this idea by focussing on how developers actually reason about these paths.

There are many more examples of metrics that are designed to measure how maintainable a software system is, either on its own or compared to others.

For instance, Halstead complexity measures count the number of distinct operators among other things, and use these to derive higher-level metrics like program level and programming effort [26]. Although the theoretical underpinnings of the measures are often criticised, a number of empirical studies have provided evidence of the predictive value of Halstead complexity measures [77].

The maintainability index is a regression-based metric that combines several smaller metrics. This includes volume-based metrics from Halstead complexity measures. The metric interestingly also includes the proportion of LOCs that are dedicated to comments; the assumption being that code with more comment lines is more maintainable (see section 2.2). The output variable of this metric is a number, the index, that represents a system's maintainability 'score' [11]. The scores are meaningless on their own, but computing scores for multiple systems makes it possible to compare their relative maintainability. The more recently proposed SIG maintainability model improves upon the maintainability index in multiple ways. First, by making use of straightforward sub-metrics that are easy to implement and compute. This makes the model easier to understand, especially to non-technical stakeholders. Secondly, the SIG maintainability model generates multiple scores, where each score covers a particular aspect of maintainability. This makes the model more transparent and helps its users conduct root cause analyses for each sub-score [31].

Our research focusses on quality analysis of comments rather than maintainability of code. Nevertheless, the qualities of these maintainability models can also apply to a predictive model for comment quality: a good model should make it possible to compare the relative quality of two (sets of) comments, should be easily understandable, and help its users write better comments.

2.2 Assessment of comment quality

We have discussed several examples of maintainability models for source code, some of which took comments into account. Comments are part of the source code of software systems, but have characteristics that are very different from 'actual' code:

- Their presence has no effect on the run-time behaviour of a software system. Nevertheless, some comments do not only affect the behaviour of human readers. For instance, the aforementioned Javadoc comments and annotations may be used by developer tools like integrated development environments (IDEs) and linting tools;
- Code makes software more complex and has a detrimental effect on analysability, and thus also on maintainability. Conversely, comments can be used to make code *more* analysable and maintainable [15].
- Whereas code must strictly adhere to syntax, grammars and logic to be of any use, no such restrictions apply to the contents of comments: developers are technically free to write whatever they want in a comment, in whatever format they desire.

Comments can contain virtually any text and can be used for any purpose. This makes automated assessment of comment quality extremely difficult; even with the help of state-of-the-art natural language processing (NLP) tools, which are generally more suited towards general texts as those found in newspapers [73]. Nevertheless, metrics have been devised for comment quality. Some metrics take a heuristic approach, while others use NLP techniques. In the remainder of this section we discuss a few common classes of metrics and models for several aspects of comment quality.

2.2.1 Lines of comments

Studies show that the presence of comments makes source code more readable [74]. One might therefore say that a project with many lines of comments is more maintainable than a project that contains fewer lines of comments.

While this makes sense to some extent, it does not take into account the fact that not all comments are useful. We have already seen an example of a superfluous comment in Listing 1.5, but there are also other types of comments for which their contribution to maintainability is debatable at best. Examples include comments that consist solely of code (Listing B.7) and copyright notices.

Moreover, an exploratory study by He [29] showed that comment density varies significantly between programming languages. Their analysis of popular projects showed that Java and Python projects have higher comment densities than projects in other languages. This makes interpretation of line-based metrics more difficult.

2.2.2 Code-comment consistency

A common problem with software documentation is that it is outdated. This is especially problematic for types of documentation that are closely related to the implementation, like comments [42]. Many approaches therefore focus on assessing to what extent member comments appear to be in sync with their methods.

An early example of such an approach was by Tan et al. [73], who used a combination of NLP, machine learning, statistics, and static analysis to detect inconsistencies between comments and code. An inconsistency could mean that a comment is outdated or the code contains a bug. An evaluation with four large codebases showed that these inconsistencies can be used to discover bugs in software systems.

Several other studies further explored the detection of similarity between comments and code based on the textual contents of comments. Steidl et al. [69] determined the coherence between member comments and method names by measuring the similarity between the words that are used in the comments and the code. Such an approach is fairly limited, because in order to be seen as similar, the words have to be virtually exactly the same. Liu et al. [43] conducted a study on the role of synonyms and polysemy[1] in detection of code-comment consistency. They found that accuracy can be improved by using WordNet[2], a lexical database for English, to include words in the detection process that have similar meanings, but look very different. A similar discovery was made by Corazza et al. [12], who manually assessed the coherence of 3,636 methods in three OSS applications. They conclude

[1] The opposite of synonyms, i.e. when a single word has multiple meanings.
[2] *WordNet | A Lexical Database for English* – https://wordnet.princeton.edu/

that lexical similarity alone does not suffice in practice. However, a more sophisticated vector space model with a tf-idf schema that takes the regularity of words into account, produces promising results. These two studies show that analysis of comments possibly requires the use of natural language processing techniques and datasets.

2.2.3 Readability

To our knowledge no comment quality metric exists that is solely based on its (free-format) content. This should not come as a surprise, as comments are intended to supplement code. Nevertheless, we feel that a brief discussion of general text readability is warranted given that comments virtually always consist of natural language text, e.g. Dutch or English.

Like source code, a text in English can be easy to read *without* being understandable. Comprehension of text is hard to measure automatically, even more so than of code. Automated assessment of readability is very feasible in comparison. While this is not the same as assessing understandability, it is nonetheless valuable because readability is a *prerequisite* for understandability [76].

Readability metrics are typically based on features like the average lengths of words and sentences, variety, and likely familiarity of the reader with certain words. The accuracy of these metrics has been studied extensively for the English language, but less for others [13].

2.2.4 Models for comment quality

All comment quality metrics discussed up to this point only focus on a single feature. To our knowledge there are only a few models for comment quality that aim to determine the helpfulness of comments from multiple perspectives, although none are actually available to practitioners.

Schreck et al. [65] developed the Quasoledo tool. This tool checks whether comments are written for every method and parameter declaration, that comments have an appropriate length (i.e. not so short that they provide insufficient information, but also not so verbose that developers do not read them), and are easy to read for their target audience. Their approach notably does not take code-comment consistency into account yet.

Based on those early insights, Khamis et al. [37] developed a comment quality assessment tool and model, the JavadocMiner. It makes use of NLP techniques to analyse the textual contents of comments. In addition to the features originally considered by Quasoledo, JavadocMiner also takes into account code-comment similarity, can check whether comments use the 'correct' writing style (as suggested by Javadoc style guides), and verifies that comments add value beyond what can be deduced from the method name. An evaluation of comments analysed by the tool confirmed earlier findings by Tan et al. [73] that good comments are likely correlated with a lower number of bugs, although it is not entirely clear to what the role is of the various parts of their model.

A third model for comment quality was proposed by Steidl et al. [69], who described a method to assess certain quality aspects of comments in Java and C++ code, following a conceptual model that is based on entities (comment categories), activities (the developer's intention), and four quality criteria which a comment must meet:

Coherence Comments must be clearly related to the code, but must not exactly repeat the code.

Usefulness Comments must help readers gain a better understanding of the code that is being described.

Completeness Comments must be present throughout the source code in the places where you expect them, e.g. in the header of a file, and above fields and methods.

Consistency Comments must be consistent throughout the source code, e.g. written in the same language and following the same format.

An evaluation of their model on five OSS projects shows that it provides more insight than metrics based on simple comment ratios. The model can also reveal where comments need to be refactored. It was later extended by Sun et al. [71], who also included assessment of header comments and proposed a method for generating suggestions that may help developers improve their comments. The insights that this model provides are somewhat similar to those of the SIG maintainability model that we discussed earlier in section 2.1: both look at quality from various perspectives and aim to produce actionable analysis results. These can be seen as requirements for future predictive models for comment quality.

Chapter 3

Study design

Recall that our objective is to develop a predictive model that helps us gain better insight into features that are associated with good comments in source code. We therefore formulate our main research question as follows:

> How well do features of source code comments predict comment quality in Java code?

Our main research question cannot be answered immediately. We therefore decompose it into three sub-questions.

RQ1 What features of source code comments can be derived from literature?

We start our study by cataloguing features of comments in source code that may affect comprehension, and thus comment quality. Not only are we interested in which features exist, but also how we can recognise them, and why and how they may affect comprehension. The answer to this first question provides the theoretical background for our model.

RQ2 What do features of source code comments look like in open-source Java projects?

After we have learned which features of source code comments have been studied by others, we quantify how common each feature is in real-world Java projects and what they look like in practice. This helps us empirically validate our findings from scientific literature.

RQ3 How do comment features affect developers' perceived comprehension of Java code?

Finally, our third research question helps us bridge the gap between features and comment quality. We do this by conducting an online survey in which we elicit information about the helpfulness of comments, which we then relate to features using machine learning algorithms. This allows us to study the relationships between each feature and the perceived quality of comments. Our overall research strategy consists of four phases:

1. We start with a systematic literature review on features of source code comments. A literature review gives us a good overview of existing perspectives and findings about the efficacy and usage of comments in source code. This answers **RQ1** and yields a list of candidate features that we can implement in the next phase.

2. The candidate list of features shows what aspects of comments are described in literature, but not what comments look like in practice. We therefore analyse comments in a small sample of representative Java projects. This answers **RQ2**. The analysis also tells us which features cannot be computed reliably and should be excluded from the definitive list, as inaccurate values would negatively affect the predictions of our model.

3. The quality of a comment cannot be measured directly, but is *perceived* by developers who read it during software maintenance activities. We therefore conduct an online survey in which we ask developers to rate the helpfulness of comments in Java source code.

4. The *subjective* ratings made by survey respondents can be combined with the list of 'objective' features to construct predictive models that automatically assess the quality of individual comments in Java source code. This answers **RQ3** and achieves the main objective of our study.

We now describe the method of each phase in more detail.

3.1 Systematic literature review

We use a systematic literature review process to answer our first research question. A systematic literature review process is much like a regular literature review process, except that it follows a clear protocol that is determined *a priori* and describes the precise steps, keywords, and methods used to obtain the results. Such a process offers numerous benefits over an *ad hoc* search process: important publications are less likely to be overlooked and since each step in the process is clearly described, the results can be reproduced more easily by others [39, 44]. Systematic literature reviews are also commonly used within software engineering. An example of such a study was conducted by da Silva et al. [14]. Our study uses a similar method.

3.1.1 Search strategy

Our search strategy consists of the following components: the searched databases, the used keywords, inclusion criteria, and the protocol that we follow to conduct the search.

Databases

We search for publications in four digital libraries that are likely to contain publications we are looking for: the ACM Digital Library, IEEE Digital Library, Google Scholar, and the dblp computer science bibliography.

The ACM and IEEE libraries provide first-party access to important papers within the domain of program comprehension. Google Scholar and dblp may return additional papers that are not covered directly by ACM or IEEE publications.

Search query

Our goal is to find publications related to *comments* and program *comprehension*. This makes *comment* AND *comprehension* [52] a good starting point for a search query.

We further refine this query to tailor it for our use case. First, studies do not always use the term *comprehension*. Some use *understanding* (e.g. [53, 64]), while others use *readability* (e.g. [7, 58, 74, 80]). We therefore include **synonyms** of the term comprehension in our query. We also add keywords that often appear in **titles of papers** about comments: *maintainability*, *quality*, and *good*. Finally, we wish to make sure at an early stage that retrieved publications discuss comments **within the context of *source code* (or *program code*)**, as filtering them later is much more time-consuming.

The final version of our search query is as follows: *(source* OR *program)* AND *code* AND *comment* AND *(comprehension* OR *understandability* OR *readability* OR *maintainability* OR *good* OR *quality)*.

Inclusion criteria

We define several inclusion criteria that each publication must meet in order to be considered for further analysis. A publication must be 1) written in English and describe results that apply to comments in English, 2) published in a peer-reviewed journal or peer-reviewed conference proceedings, 3) about comments in source code, e.g. not comments in pull requests, and 4) have been cited more than 20 times according to Google Scholar.

Search protocol

Our protocol is a modified version of the protocol originally used by da Silva et al. [14] for their systematic literature review on systematic literature reviews in software engineering and consists of six steps:

1. Perform a search in all databases using our query. We download the first 200 papers that we find.
2. Remove duplicates using automatic and manual methods. We do not count follow-up papers as duplicates, because they may provide additional insights.
3. Filter papers by reading the title and abstract, and estimating whether the paper meets our inclusion criteria. Sometimes it is not immediately clear whether a paper meets all criteria. When this happens, we also search for all instances of 'comment' and read the surrounding text so we can make a more informed decision.
4. Sort articles in descending order of citations as shown in Google Scholar's search results, approximate impact of the journal or proceedings using its h5-index[1], and year of publication. This allows us to achieve the best coverage of features in a brief amount of time, as highly cited papers are more likely to contain contributions that others within the scientific community have found valuable.

[1] https://scholar.google.com/citations?view_op=top_venues – *Google Scholar Metrics*

5. Apply backward snowballing to discover other articles that may be relevant for our review.

3.1.2 Data analysis

We used ATLAS.ti[2] to annotate papers. Our annotation process followed a combined deductive-inductive approach to assign the following codes to the identified features:

Name The names that are used for a feature by researchers and practitioners. Ideally these are explicit mentions, but indirect references are also included.

When multiple names exist, we attempt to choose a name that most closely reflects what is used in literature. If there is more than one viable alternative, we also mention these alternative names. To simplify our analysis, we normalise the name of each feature so that its purported effect would be non-negative, e.g. 'is correct' rather than 'is wrong'.

Cataloguing the names that are used for features makes it easier for us to compare our results with existing work.

Rationale Reason(s) why the authors believe that the feature is worth taking into account, e.g. based on experiments or interviews with developers. For us, these can also be used to justify their inclusion in a model.

The data for *rationale* cannot be easily aggregated onto a nominal or ordinal scale. We therefore attempt to summarise different descriptions while keeping track of aspects that authors do not universally agree on.

Effect Whether the feature is thought to help or hinder the program comprehension process, based on argumentation or empirical findings. A feature that has a clear effect on comprehension is more likely to be useful for a model for comment quality than a feature for which the literature suggests that no effect exists.

Since our feature names are normalised, each feature should have a non-negative effect. Specifically, we distinguish between three types of effects: clear (positive) effects, mixed effects, and no *known* effects. We consider a feature to have mixed effects when the outcome of a single study is inconclusive or if two or more studies disagree with each other.

Validation Whether the feature was empirically validated, e.g. using a comparative study, case study, or survey. This information helps us understand to some extent how certain we can be that a feature actually contributes to comprehension.

Ideally, each feature in our review is *validated* empirically, as this proves that a feature contributes to the helpfulness of a comment. Some features are only *validated indirectly*, e.g. a feature might have been validated together with other features, but not on its own. This makes it impossible to attribute effects to any single feature. Other features might *not* have been *validated* at all. We label a feature as (partially) validated if at least one such study exists.

[2]https://atlasti.com/ – *ATLAS.ti: The Qualitative Data Analysis & Research Software*

Measurement How the feature can be measured, e.g. using an equation, a specific requirement that a good comment should meet, or more general descriptions of desirable characteristics.

When there are multiple ways to measure a feature, we try to summarise them all, while paying special attention to different perspectives and potential implementation pitfalls.

Quality criterion Features can contribute to quality along multiple dimensions and as such can be categorised in different ways. We choose to take a deductive approach here by assigning each feature to one of the four quality criteria in the quality model by Steidl et al. [69] as described in section 2.2 (coherence, usefulness, completeness, consistency). Each criterion can be seen as very short summary of why a comment with a particular feature is high-quality. We therefore assign features based on their described rationale.

Aside from 'Quality criterion', all codes are generated using an inductive approach. Annotation is done in three steps: first, we annotate only the type (e.g. 'Name'). Then, we revisit each paper to assign feature-specific codes to the initial set of type annotations. Finally, we revisit our list of feature-specific codes to merge codes that are very similar to each other.

3.2 Mining repositories for comment features

To learn more about what comments look like in practice, we developed Coalaty; a tool for automated **co**mment **a**nal**y**sis and **a**ssessment of understandabili**t**y. Its source code is available via https://figshare.com/s/d10cf6030ec7de1ef258. In this section we describe how Coalaty works and which features it can extract from Java projects.

3.2.1 Coalaty architecture

Coalaty is implemented using the Java programming language as a command-line application. The application requires two input arguments: a directory that contains Java source code and a semicolon-delimited csv[3] file that will contain the computed features for each analysed comment. Figure 3.1 shows Coalaty's analysis pipeline. The process for analysis of a single project consists of five steps:

1. First, Coalaty identifies all `.java` source code files within a directory. Files that contain `test` or `example` anywhere in their relative path name are ignored, as test and example classes are not representative of 'normal' Java code.

2. The open-source JavaParser library[4] is one of the most popular parsers for the Java language. Our tool uses JavaParser to parse each individual Java source file into an abstract syntax tree, a representation that can be easily traversed using Java code. JavaParser works using static analysis, i.e. Java files are not compiled or executed in any way. We assume that any `.java` file contains valid, compilable Java code that does not make use of experimental

[3] comma-separated values
[4] https://javaparser.org/

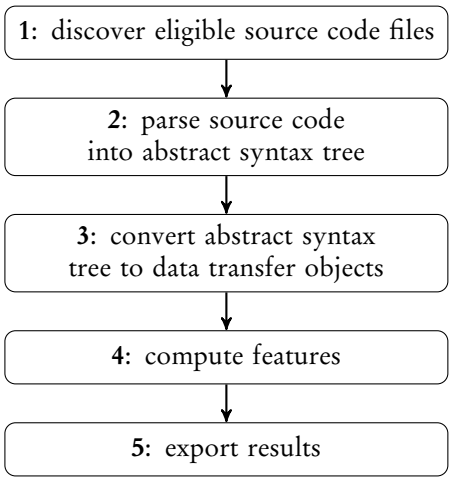


Figure 3.1: A schematic overview of Coalaty's analysis pipeline

language features. Any file that cannot be parsed by JavaParser due to invalid syntax is automatically skipped by the tool, but will generate a stack trace.

The specifics of our usage of JavaParser and its limitations are described in appendices B.1–B.2. Most importantly, we only consider comments that are placed above entities, like classes, interfaces, and methods. Inline comments are harder to match unambiguously to a clearly defined region of code, which is likely a requirement for many important comment features and are thus excluded from our analysis. Moreover, an implementation of symbol resolution is beyond the scope of this project. Variable types and identifiers are therefore not resolved to fully qualified names, but treated as simple strings. This means that in some rare cases, our tool might not be able to differentiate between two interfaces or methods in different classes that share the same name. We expect that this will have limited impact on the output of the tool.

3. To streamline the remainder of our analysis, we convert JavaParser's abstract syntax tree objects into minimalistic data transfer objects (DTOs), which are smaller data structures that are more suitable to our purpose, i.e. focussed on the analysis of comments rather than source code in general.

   For performance reasons we also precompute a few low-level text-related metrics that form the basis for multiple features that are based on the textual contents of source code and comments:

   - Natural language text is divided into tokens (which correspond roughly with words) using the open-source Stanford CoreNLP library[5] and sanitised in order to remove HTML and other meaningless tokens.
   - Tokens are stemmed (i.e. reduced to their base form) using the open-source Apache OpenNLP library[6], which makes it easier to match similar tokens with each other.

[5] https://stanfordnlp.github.io/CoreNLP/
[6] https://opennlp.apache.org/

- The number of syllables in each token is computed using the open-source `syllablecounter` library[7] .
- We use languagetool.org's Java API[8] to verify that comments are written in English, as only those are within the scope of our study.

A more thorough explanation of these concepts can be found in appendix B.1. After the conversion of JavaParser's abstract syntax tree objects to DTOs, our tool has all the information that it needs for its comment analysis.

4. In the fourth step, the tool computes comment features using the information that is stored within the data transfer objects.

   The previous phase of our study (section 3.1) yields a list of features that are described in scientific literature, along with instructions or guidelines on how to measure them. As we discussed earlier in chapter 2, there may be more than one way to measure one feature, e.g. the length of a unit of code can be determined by computing the number of logical or physical lines. On the other hand, there could also be features for which measurement is infeasible. Each feature that we identify in our literature review will therefore be mapped to zero or more distinct implementations of that feature.

   When implementing features, we try to make as much use of existing libraries as possible. Not only does this save time, it also makes it more likely that the resulting implementation is correct, as libraries are generally thoroughly tested. In some cases reuse is not possible or unnecessary. In such cases, we provide our own implementation, which we verify using unit tests or by comparing computed results with manual inspections or results from existing studies (see section 3.2.2).

5. Finally, the analysis results are exported to a comma-separated values (CSV) file. Such files can be easily viewed and processed using open source tools, like Python, R and SQLite, or commercially available off-the-shelf software, like Microsoft Excel or IBM SPSS Statistics.

   Listing 3.1 shows a pretty-printed example of what Coalaty's output looks like. The output file starts with a header that describes each field in the CSV file. Each subsequent line contains a reference (`location`) to a relative path to a Java source file and the physical lines of code which contain a single comment, followed by the `project` name that is deduced from the directory name, the entity above which the comment is placed, and the computed features.

3.2.2 Software repositories

To learn more about how features are distributed for real-world Java code would be to mine OSS Java software repositories. It would be infeasible to mine every Java repository in existence. Even a selection of the 100 most popular repositories would be computationally prohibitive. Selecting most popular repositories also does not guarantee that sample is representative for Java code in general.

[7] https://github.com/m09/syllable-counter
[8] https://dev.languagetool.org/java-api

Listing 3.1: Pretty-printed example of Coalaty's csv output format

```
location              ;project;type   ;feature1;feature2;feature3;featureN
path/to/Foo.java:12-14;Example;Class  ;       1; 0.34216;       0;       2
path/to/Bar.java:45-52;Example;Method ;       0; 0.12345;      43;       1
```

Rather than trying to select a representative sample ourselves, we reuse two representative samples from existing studies. This makes it more likely that our results are generalisable and also makes it easier to validate the correctness of our tool, as its computed features should be very similar to those reported in the original studies.

The first sample consists of six heterogeneous Java libraries and frameworks from different ecosystems: Apache Spark, Eclipse CDT, Google Guava, Apache Hadoop, Google Guice, and Vaadin. The second sample consists of five heterogeneous oss Android projects that are developed using Java: AFWall+, Amaze File Manager, AntennaPod, ownCloud, and WordPress. The source code for these projects can be obtained via Pascarella and Bacchelli [57] and Pascarella [56] respectively.

3.3 Survey

Coalaty provides us with a wealth of information about the features of all comments contained in the aforementioned eleven Java projects.

The values that are computed for each comment give us an idea of what each comment looks like, but are otherwise meaningless and free of judgment. In order to bridge the gap between our 'meaningless' values and a quality rating we need data that helps us understand the relation between features and perception of the quality of a comment.

We gather this data using an online survey, which we distribute and promote among people with programming experience, e.g. professional and hobbyist software developers, computer science and software engineering staff and students, and data scientists. In this survey we ask respondents to rate the helpfulness for a small set of comments that we analysed earlier using Coalaty, as they have characteristics that are known a priori.

3.3.1 Overview of survey design

The survey consists of three parts:

- In accordance with ethical guidelines for social research [6], we ask respondents to provide explicit consent for their participation prior to the start of the actual survey. Respondents can withdraw from the survey at any time as long as they have not submitted their responses. Incomplete responses are discarded in order to conform with regulations set by the research ethics committee (cETO) of the Open University of the Netherlands.
- The main part of the survey consists of a sequence of code snippets from the real-world software projects that we used in the previous phase (sec-

tion 3.2.2). Each code snippet includes at least one Java comment, which we ask respondents to rate the helpfulness of using a five-point Likert scale (Figure 3.2). This approach – with comment features used as *independent* variables and quality ratings as *dependent* variables – has been used in many studies on code quality. Examples of such studies include [3], [7], [27], and [64]. We discuss the selection process for suitable snippets in more detail below.

In addition to a statement about the perceived helpfulness of a comment, we sometimes also show one of the following secondary statements (Table C.1) about a particular aspect of that comment that might stand out due to a particular feature:

- I think this comment is easy to understand.
- I think this comment provides the right amount of information.
- I think this code would be just as easy to understand without the comment.
- I think this comment meets my expectations of what a comment should look like.

Snippets are presented individually with syntax highlighting, as this is what the majority of developers will be used to. We take care to present snippets in an aesthetically neutral way to avoid halo effects [28], which could inadvertently affect how respondents perceive the quality of the comment in the snippet.

Moreover, we show snippets in a randomised order in order to avoid order effects, which would cause answers to questions to be *consistently* influenced by questions that respondents have previously seen [6].

- We conclude the survey with a few questions about the demographic background of the respondent, specifically their level of education, English proficiency, and programming experience [22]. This helps us interpret the results and makes it possible to correct for non-response of certain parts of the population [19] and control for other factors that may affect how the respondent perceives code snippets and comments therein [80].

We conducted two small-scale pilot studies [19] with acquaintances to ensure that the questions in the survey are clear and completing the survey takes an appropriate amount of time. Participants in the pilot studies were asked to refrain from taking the finalised version of the survey, as their participation in the pilot could potentially influence their responses.

A more comprehensive description of the survey and its supplementary materials can be found in appendix C.

3.3.2 Snippet selection

The second phase of our study (see section 3.2) yields a very large number of real-world snippets from representative Java projects whose features are known, but their quality (i.e. perceived helpfulness) is not. Since respondents only have a finite amount of time, we can only include a very small, heterogeneous sample of snippets in the survey. We create this sample as follows:

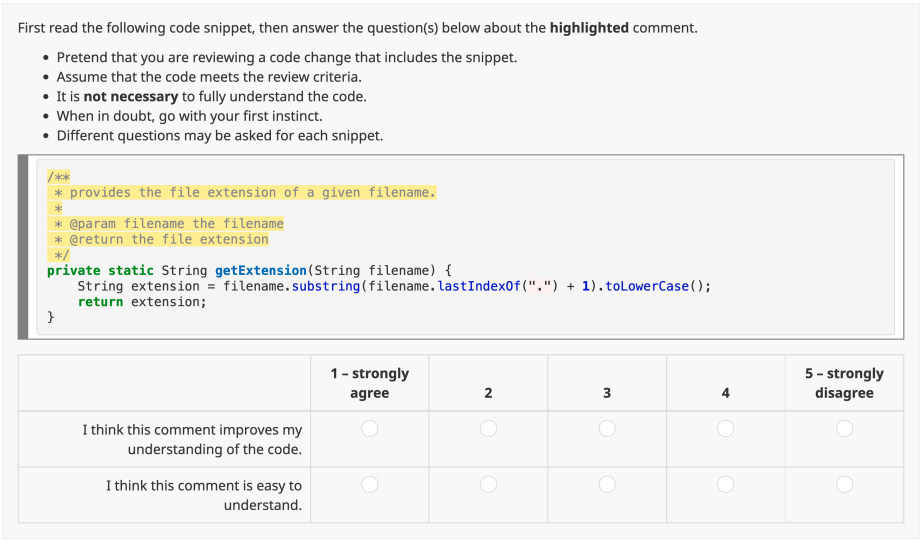


Figure 3.2: One of the snippets included in the survey

1. We start with the complete set of snippets that was extracted from the eleven Java projects that we discussed earlier in section 3.2.2. Features are extracted for each of the snippets and mapped to numerical values.

2. Snippets are grouped based on the values computed for each of their features. We label snippets with very high or low values (but not extreme outliers) as eligible for inclusion as this makes it easier to attribute any possible effects on their perceived helpfulness to the feature that is associated with the abnormally high/low value.

3. We exclude snippets that are unsuitable for inclusion in a survey. This includes snippets that are completely impossible to understand without the right context [3] or that consist of an extreme high number of lines [7]. To keep things simple, we also choose to limit our scope to member (method) comments for now.

4. We randomly select one or two snippets from each group for our sample. Ideally we would include all remaining snippets in our sample, but this is not practical. Only few respondents would be able to complete very long surveys and due to resource limitations we can only distribute one version of the survey. Including a high number of snippets would therefore result in less – not more – data [19].

A complete list of included snippets can be found in Appendix C.4.

3.3.3 Distribution

The survey is created and hosted using LimeSurvey Professional[9], an online survey tool that allows us to collect responses in a GDPR-compliant way. We primarily re-

[9]https://www.limesurvey.org/

cruit respondents by advertising the survey on our website[10], in private Slack and Discord communities, in public communities like Reddit and Dev.to, on social media platforms that are popular among developers, like Twitter and LinkedIn, and our personal networks at the Open University of the Netherlands and the Dutch Broadcasting Foundation (*Nederlandse Omroep Stichting*). Respondents who complete the survey are also encouraged to share it within their own networks, as this may help us reach potential respondents whom would have been unreachable otherwise.

Two types of monetary incentives were offered to respondents for completing and sharing the survey: we raffled three $50 Amazon gift cards among respondents who completed the survey and pledged to donate €1 to the Dutch Cancer Society (*KWF Kankerbestrijding*) on behalf of each respondent.

There is evidence that offering monetary incentives increases response rates – especially among respondents from demographic groups who would normally hesitate to participate in potentially time-consuming surveys [17, 67]. This also helps us to reduce the potential for non-response errors [19].

3.4 Model construction

In section 3.2 we explained how we compute features for all comments in a representative sample of Java projects. This yields a dataset that tells us what each individual comment looks like, e.g. whether it is lengthy, placed above a hard-to-understand method, or contains any information that cannot be derived directly from the code. Section 3.3 then described how we conduct a survey in which we basically ask respondents to rate the 'quality' comments in several dozen snippets from that dataset.

While ratings for the comments in those snippets already provide valuable insights on their own, we are not really interested in these *specific* snippets: what we actually wish to understand is the relationship between comment features and perceived comment quality. The relationship between comment features and perceived comment quality can be explored using various machine learning methods, many of which are readily available via popular Python libraries like scikit-learn[11].

3.4.1 Machine learning methods

Although there is a wide array of different machine learning methods that can be used to predict comment quality, there are only a few widespread methods that are also explainable. This is an important criterion for our model: a model that is not explainable cannot be understood by its users, which in turn makes it hard to act on reports of comments that are reportedly low-quality.

Linear regression

Linear regression (alternatively: multiple linear regression) is commonly used in studies that try to relate specific features of comments to program comprehension [7, 58, 61, 69, 73].

[10] https://chuniversiteit.nl/

[11] *scikit-learn: machine learning in Python* – https://scikit-learn.org/stable/

A linear regression model uses the form $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$. Here, y is the outcome that we wish to predict (with a certain degree of certainty), which is the quality of a comment expressed as a numeric value. Given that a five-point Likert scale can be seen as an approximation of an interval scale from 1 to 5 [81], the outcome variable would be a value between 1 and 5. Predictor variables (features), which affect the outcome, are denoted using x. Finally, the coefficient (weight) of each predictor variable is denoted using β [23]. We can determine these coefficients by 'fitting' a regression model to a dataset that contains the predictor variables.

As the name already suggests, a linear regression model assumes that a linear relationship exists between each feature and the outcome that one wants to predict, and may generate unexpected results when this assumption is violated. Only features for which a linear relationship exists with the outcome variable can therefore be used for linear regression [23] (see section 3.4.2).

Logistic regression

Logistic regression is somewhat similar to normal regression, except that it can be used for classification, i.e. the prediction of a categorical variable from a set of categorical and continuous variables [23]. For example, a comment might be classified as either 'good' or 'bad' with a certain probability [62]. To allow for better comparisons between different models, we will assume that comments can be classified in one of five ways: very helpful, somewhat helpful, neutral, somewhat unhelpful, and very unhelpful.

Decision trees

A decision tree is a machine learning model that makes predictions ('decisions') by navigating through a tree from the root until it reaches a leaf node that contains the predicted outcome variable, which can be either a discrete or continuous value [62]. In each internal node of the tree, it decides which branch should be followed to reach the appropriate decision based on a test performed on one of the predictor variables.

Decision trees can be used for both regression and classification, which makes them quite versatile. Another benefit of decision trees is that their simple, intuitive format can make them very understandable for laymen.

On the other hand, decision trees can quickly become unwieldy for certain types of problems, for instance when the overall outcome depends on the value of a very large number of predictor variables or when the model includes many real-valued attributes. Another potential downside of decision trees is that they can be sensitive to changes in training data: even small differences or additions in the training data may result in trees that look vastly different [62].

Support vector machines

Support vector machines can be used for both regression and classification. They were especially popular in the early 2000s for supervised learning problems, without the need for specialised prior knowledge about a domain. Nevertheless, support vector machines remain very useful due to their ability to generalise, flexibility to represent complex functions and resistance to overfitting [62], i.e. when a model performs very well on the data on which it was trained, but badly when it is used to make predictions based on data which it has not seen yet.

3.4.2 Data preparation

Earlier, we assigned features to each of the comments in our dataset and gathered ratings for a small selection of those comments using a survey. By relating each snippet to the features that were originally computed for the snippet, we can match features with helpfulness ratings. Once combined with the demographic background information provided by respondents, we have a first complete version of our dataset.

This dataset may already work for some machine learning methods, but not all. For instance, for regression analysis all values must be numerical and present (i.e. no null values). Moreover, the data should not include extreme outliers as this might result in models that make sub-optimal predictions.

No action is required for the first two requirements:

1. All values in Table 4.5 can be expressed as floats or integers, and all survey responses can be coded as ordinal values;

2. Our repository mining tool does not generate null output values and all questions in the survey were required.

As discussed in section 3.3.2, we have tried to choose our snippets such that they are at least somewhat representative. However, as the dataset includes 50 variables, outliers may still occur somewhere in the data. Such outliers may cause our model to over-fit ('overly specialise') for the comments in our specific dataset and lead to worse performance on comments in general.

Moreover, many variables express ratios or percentages, where an increase from 2% to 4% is substantial, but an equally large increase from 40% to 42% is negligible [24]. Our regression models must be able to distinguish between these two situations.

Both issues can be resolved by applying logarithmic transformations like these to some of our variables [47]:

- a log transformation[12] $log(x + 1)$ to variables that either represent natural numbers without an upper bound (e.g. the number of lines) or natural numbers that cover two or more orders of magnitude (e.g. many of our readability formulas). This assigns more importance to differences in lower values, which reduces the impact of outlier values;

- a logit (alternatively: log-odds) transformation $log(\frac{x}{1-x})$ to variables that represent proportions as numbers between 0 and 1, which helps us improve the interpretation of proportional differences for restricted-range variables [24]. Because the result for 0 and 1 are undefined we remap all data proportionally to a range between 0.025 and 0.975.

Variables that express simple ranges within the same order of magnitude (like booleans and values derived from Likert scales) do not require any transformations. It is unclear what the effect of log transformations on such variables would be [24].

[12] $x + 1$ is used, because $log(0)$ is undefined.

Feature selection

While it does not hurt to have an abundance of data available, we do not want to include every single variable in our model. There are several reasons for this:

- Predictor variables with zero variance do not contribute to a model in any way [23].
- For regression-based machine learning methods, the regression coefficient of each predictor variable must be significantly different from zero. Linear regression specifically also requires that all predictor variables have a linear relationship with the outcome variable [23].
- Some machine learning methods are sensitive to the curse of dimensionality. Their performance deteriorates when used with a large number of predictor variables [63, 78], e.g. a model might make 'random' predictions that are largely guided by noise rather than features that are actually meaningful [75]. Having too many variables also makes models harder to understand. By creating a scree plot of the eigenvalues of principal components for our dataset we can determine what number of variables is sufficient for our model [23].
- Two or more predictor variables might be highly correlated with each other, a phenomenon that is known as multicollinearity. We discuss the issue of multicollinearity in more detail below.

Multicollinearity

Some of our variables may be correlated with each other; this is a phenomenon known as collinearity. Low levels of collinearity are acceptable, but high levels can give rise to problems [23]:

- High levels of collinearity also increase the standard errors of the coefficients in our linear regression model. This results in a model that may work well in our sample of snippets, but not on comments in general.
- Each variable that makes it into the model affects the outcome of its predictions in some way. When two variables are highly collinear, they affect the outcome in the *same* way. Intuitively, this means that the outcome is affected by only one of the variables, while the second contributes nothing.
- Finally, when two variables are highly collinear it makes it hard to determine which of the two is actually important – knowing which variables are important is crucial for gaining a better understanding of how and why the model works.

We therefore test for multicollinearity after constructing our model by computing variance inflation factor (VIF) scores for each of our independent variables. High VIF scores are a sign that a variable has a strong relationship with other variables, while a VIF score of 1 means that no correlation with other variables exists.

There are no hard rules about what VIF values are cause for concern [23]. Nevertheless, there are several rules of thumb:

- Some sources suggest that variables with a VIF score higher than 10 are undesirable [49], while other sources suggest 5 [24] or even 3 [1] as a threshold value. Yet others argue that high VIF scores are not necessarily problematic on their own [54];
- An average VIF score greater than 1 implies that a regression model suffers from multicollinearity [4].

For our study, we will use a threshold of 5, as this appears to be one of the most commonly chosen thresholds. When we compute the VIF scores for our model and find that at least one variable with a score above the threshold exists, we first determine whether it is likely that two variables really measure roughly the same thing [54]. If so, we drop the variable that is harder to compute accurately as this will likely lead to an increase in data quality, and subsequently, predictions. We then recompute VIF scores until all variables have acceptably low VIF scores.

3.4.3 Model validation

Like all software, machine learning models should be tested after construction in order to verify that they work properly, but the method of verification differs somewhat. Correctness for most software is rather black and white: either it works correctly or it does not. Machine learning models on the other hand make predictions, which are unlikely to be correct 100% of the time. Moreover, the quality and result of predictions strongly depends on the data that is used as input. Model testing therefore is typically based on assessing the extent (for regression) or likelihood (for classification) that a model produces right answers for a specific set of data [23].

Up to this point, we assumed that there is only one set of data, a training set, which is used as input for a model. In reality this set cannot simply be used to evaluate our models: if we were to reuse the training set, a model would likely receive a biased score that is too high as it is already 'familiar' with the samples in the training set [62]. Machine learning models are therefore ordinarily trained using only part of the available data. The dataset is typically split into a training set and a test set. A model is trained using the training set and evaluated using the latter. While simple, a naive split of the dataset into two parts may have adverse effects — especially if the dataset has a limited size. Dividing the dataset into two parts reduces the number of individual samples that are used to train the model. Furthermore, it introduces the risk that examples in the training or test set (or both!) are no longer entirely representative of the set as a whole. This may result in models that perform worse than when they would have been trained on the entire dataset.

A method called k-fold cross validation allows us to mitigate these issues by making better use of the training data [62]. It works by splitting the data into k equally sized subsets, which then take part in k rounds of learning. In each round, one of the subsets is used as a validation set, while the remaining subsets are used for training [41]. Each subset is therefore used $k - 1$ times for training, and once for testing. A commonly used value for k is 10 [9, 12, 57, 61, 75].

Chapter 4

Results

This chapter describes the results of our study on what makes good comments in Java source code. The study consists of five phases. Each phase is described in its own section.

First, section 4.1 describes the results of our systematic literature review of features of comments in source code, and answers **RQ1**, 'What features of source code comments can be derived from literature?'.

In section 4.2 we discuss the results of our analysis of comments in our sample of Java projects and how they relate to our findings from the literature review. This answers **RQ2**, 'What do features of source code comments look like in open-source Java projects?'.

In the third phase, we conducted an online survey on the effect of comments on perceived code understandability. This yields the data for the fourth phase. Section 4.3 provides high-level statistics about the respondents and responses of our survey.

In section 4.4 we take a deeper dive in the survey results and construct a model that can predict the effect of individual Java method comments on the perceived understandability of their corresponding methods. This answers **RQ3**, 'How do comment features affect developers' perceived comprehension of Java code?'.

4.1 Systematic literature review

Our search yielded 650 publications in total: we found 200 possibly relevant publications using Google Scholar and the ACM and IEEE databases, while our query in dblp yielded only 50 search results.

While filtering this initial dataset we discarded 53 duplicate results that appeared in multiple databases. We also discarded 371 publications that were not peer-reviewed or did not describe comments in source code. After all filtering, we had 45 distinct publications that are eligible for review. An export of the raw annotations can be downloaded from https://figshare.com/s/16a9dd4bf6b01c2accb5.

Before we discuss the results of our analysis, we present an overview of the overall characteristics of this dataset in order to make it easier to contextualise and interpret our findings.

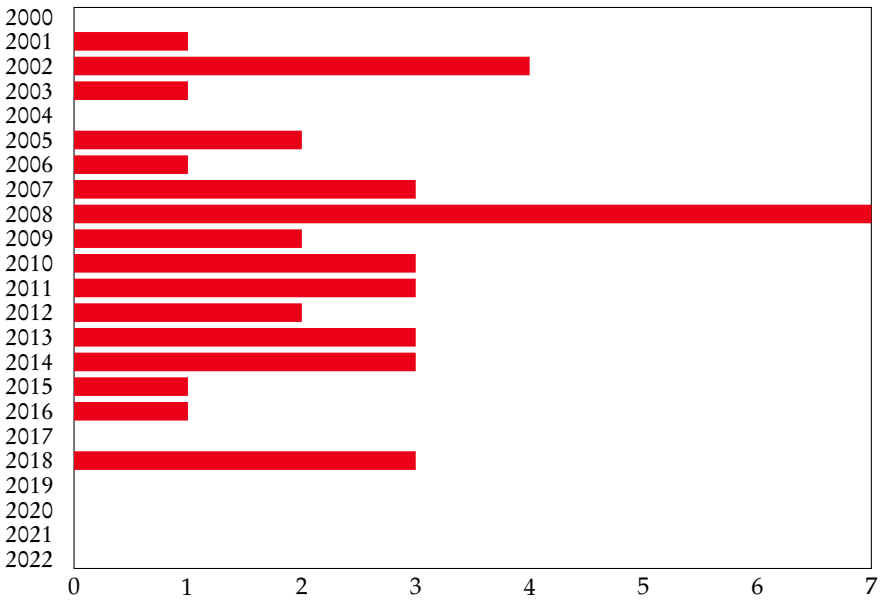


Figure 4.1: Overview of reviewed papers per year since 2000

Table 4.1: Overview of reviewed papers per venue

Venue	n	%
Intl. Conf. on Program Comprehension (ICPC)	11	24.4
IEEE Trans. on Software Engineering	5	11.1
Intl. Conf. on Software Engineering (ICSE)	5	11.1
Intl. Conf. on Automated Software Engineering (ASE)	2	4.4
Other (n=1)	22	48.9

4.1.1 Years of publication

Most of the reviewed papers were published in the 2000s and later (Figure 4.1), with a few exceptions from as early as 1981. The dataset does not include papers published after 2018. This is presumably because our inclusion criteria require that papers have been cited more than 20 times according to Google Scholar, which might not have happened yet during our search.

4.1.2 Venues

Table 4.1 shows that our annotated papers come from a diverse range of venues. Most of the reviewed papers were published in the proceedings of the *International Conference on Program Comprehension* (ICPC). Other popular venues include the *IEEE Transactions on Software Engineering* and the proceedings of the *International Conference on Software Engineering* (ICSE).

Table 4.2: Overview of reviewed papers per subject

Subject	n	%
Automated generation of comments	9	20.0
Vocabularies and natural language	6	13.3
Assessment of code quality	5	11.1
Up-to-date comments	5	11.1
Usage of comments and documentation	4	8.9
Effect on comprehension	4	8.9
Assessment of comment quality	3	6.7
Improving program comprehension	2	4.4
Task annotations	2	4.4
Automating program comprehension	1	2.2
Code reviews	1	2.2
Comments in practice	1	2.2
Recommendations for comments	1	2.2
Static analysis	1	2.2

4.1.3 Subjects

Based on titles and abstracts many papers seem to cover source code comments from similar perspectives. As can be seen in Table 4.2, one out of five papers discusses ways to automatically generate comments from code using templates or machine learning techniques. Other popular subjects include the linguistic content of comments, causes and consequences of out-of-date comments, and methods to determine the overall code quality of software systems.

4.1.4 Overview of features

Table 4.3 lists the 22 features that we have identified in our literature review. The third (n) and fourth columns (%) respectively list the absolute and relative *distinct* number of papers in which they were mentioned, i.e. a feature that is mentioned in two different papers will be counted twice (even if both papers are written by the same authors), but a feature that is mentioned twice in a single paper is only counted once.

The most frequently mentioned feature is *Describes code* (CODE), which appears in 27 papers. This means that it is mentioned in more than half of the papers in our dataset. Conversely, the least frequently mentioned feature *Does not use abbreviations* (NOABBR) only appears in 4 papers.

There are two other features that stand out in this table. The first is *Is present* (PRESENT), which stems from the (somewhat controversial) notion that *a* comment is always better than *no* comment. *Is Javadoc* (JAVADOC) is mentioned in an equally large number of papers. Its inclusion is somewhat surprising, given that Javadoc is specific to Java, which is only one of many programming languages. A possible explanation for this is Java's popularity in educational contexts [21], which makes it more likely that readers and experimental subjects are familiar with its syntax and concepts.

Table 4.4 summarises the findings for each of the features in our literature review. The 'Effect' column lists the purported effect of each feature. The 'Validated'

Table 4.3: Features and papers in which they are mentioned, in descending order of mentions

Feature	Description	n	%	Papers
CODE	Describes code	27	60.0	[R3, R5, R8, R9, R11–R13, R15–R19, R21–R26, R28, R29, R31, R35–R38, R42, R44]
PURPOSE	Describes purpose	21	46.7	[R1, R2, R5, R11, R13, R15, R16, R21, R23–R27, R29, R31, R33, R35, R37, R38, R41, R44]
UPTODATE	Is up-to-date	17	37.8	[R8, R9, R12, R13, R15, R18, R21, R24, R26, R29–R32, R35–R37, R40]
UNDERST	Is understandable	17	37.8	[R2, R5, R7, R12, R13, R15, R16, R19, R24, R28–R31, R33, R36, R37, R41]
SUCCINCT	Is succinct	17	37.8	[R3, R5, R9, R11, R12, R15, R17, R23, R24, R28–R31, R35, R42–R44]
PRESENT	Is present	16	35.6	[R2, R5, R9, R11, R12, R15, R16, R19, R24, R26, R27, R31, R34, R35, R38, R44]
JAVADOC	Is Javadoc	16	35.6	[R3, R8, R11, R13, R15, R17, R23, R26, R27, R29–R31, R33, R39, R41, R43]
COMPLETE	Is complete	14	31.1	[R9, R13–R15, R23–R25, R27, R29, R31, R33, R35, R39, R45]
CORRECT	Is correct	14	31.1	[R6, R9, R11–R14, R18, R23, R32, R37, R39, R41–R43]
IDENTIFIERS	Uses meaningful identifiers	13	28.9	[R6, R9–R11, R19, R20, R22, R24, R29, R30, R32, R33, R41]
HIGHLVL	Is high-level	12	26.7	[R5, R15, R19, R23–R25, R27, R28, R32, R33, R35, R45]
SUBLANG	Is written in sublanguage	12	26.7	[R4, R5, R11, R12, R20, R30–R32, R37, R39, R42, R43]
CONSIST	Is consistent	11	24.4	[R4, R5, R11, R12, R14, R15, R17, R22, R35, R36, R41]
HARD	Is for complex code	10	22.2	[R1, R2, R4, R13, R15, R19, R31, R42–R44]
TASKS	Mentions task	10	22.2	[R5, R6, R9, R11, R13, R26, R35–R37, R45]
DECISION	Describes design decision	9	20.0	[R7, R13, R17, R18, R21, R26, R31, R35, R37]
EXTRA	Provides extra information	8	17.8	[R5, R11, R19, R24, R35, R37, R42, R43]
CONTEXT	Describes context	8	17.8	[R15, R23–R26, R33, R39, R42]
ENTITY	Is near entity type	6	13.3	[R8, R9, R11, R15, R28, R39]
USAGE	Describes usage	6	13.3	[R13, R19, R23, R25, R26, R33]
TRACE	Enables traceability	5	11.1	[R9, R14, R17, R20, R36]
NOABBR	Does not use abbreviations	4	8.9	[R10, R15, R20, R33]
	Overall	44	100.0	

n = number of papers in which a feature is mentioned
% = percentage of papers in which a feature is mentioned

Table 4.4: High-level summary of features

Feature	Effect	Validated	Coherence	Usefulness	Completeness	Consistency
CODE	±	Y	•			
PURPOSE	+	Y		•		
UPTODATE	+	Y	•			
UNDERST	+	Y*		•		
SUCCINCT	±	Y	•			
PRESENT	+	Y			•	
JAVADOC	?	Y				•
COMPLETE	+	Y			•	
CORRECT	+	Y	•			
IDENTIFIERS	+	Y	•			
HIGHLVL	+	Y	•			
SUBLANG	+	Y				•
CONSIST	+	N				•
HARD	+	Y		•		
TASKS	±	N		•		
DECISION	+	N		•		
EXTRA	+	N		•		
CONTEXT	+	N		•		
ENTITY	+	Y	•			
USAGE	+	N		•		
TRACE	±	N		•		
NOABBR	+	Y		•		

+ = positive effect, ± = mixed effect, ? = unknown effect
Y= validated, Y* = validated indirectly, N= not validated

column shows whether the effect of the feature has been validated in at least one study. Finally, the last four columns show to which of the four major quality criteria proposed by Steidl et al. [69] the feature can be assigned.

Most features are thought to have a clear effect (+) on program comprehension. Particularly noteworthy is the finding that Javadoc on its own does not have any explicitly mentioned effects (?) on *comprehension*. This is likely because many of the benefits of Javadoc are realised via other features or only indirectly related to program comprehension, e.g. via the generation of external documentation.

Three features are possibly controversial (±). For example, while most studies imply that including references to external resources (*Enables traceability*; TRACE) is beneficial for program comprehension, we also found a study which suggested that such references can be seen as 'cluttering the code' [R36]. While this does not discount such features entirely, it does suggest that more research may be needed about the effect of such features.

Just over two thirds of the features have been validated in some way using experiments, surveys or case studies. The effect of one feature, *Is understandable* (UNDERST), was only validated together with other features. This makes it difficult to determine to what extent the understandability of a comment contributes to program comprehension on its own.

Finally, Table 4.4 shows that most features are related to the 'usefulness' criterion, followed by 'coherence', 'consistency', and 'completeness'. The remainder of this section discusses the names, rationale and measurement of each feature for these four criteria in more detail.

4.1.5 Features related to coherence

The coherence criterion essentially states that comments must be related to the code that they are describing. There are seven features in this category: *Describes code* (CODE), *Is up-to-date* (UPTODATE), *Is succinct* (SUCCINCT), *Is correct* (CORRECT), *Uses meaningful identifiers* (IDENTIFIERS), *Is high-level* (HIGHLVL), and *Is near entity type* (ENTITY). Aside from *Is correct* (CORRECT), all features in this category can be easily computed from software artefacts, like the source code and its change history.

Describes code (CODE)

A comment should provide a **summary of the code** by describing its most important parts, which may include both data and code structures [R13, R23, R37].

Rationale: This feature is mentioned in most of the papers in our review, but is possibly also one of the most controversial features.

Many developers believe that comments are not meant to explain the *how* [R13]: code should ideally be written in a self-documenting way by using small, easily comprehensible methods in conjunction with meaningful method names, as it is then no longer necessary to describe the code in a comment [R3].

Others have a more optimistic view of such summaries. Because comments are written in natural language and can explain the code at an appropriate level of detail [R33], they can be easier to understand (UNDERST) than code. Furthermore, explanations of internals can also be helpful for future maintainers of the code [R19]. Finally, documenting some higher-level aspects of code, like design patterns, may also be beneficial for comprehension [R28].

Some studies have demonstrated that describing the code in comments has a positive effect on comprehension [R36], although it is possible that such comments only affect the performance of novice programmers [R29].

Measurement: If a comment contains a summary of the code, one would typically expect that it describes the same concepts using similar words as those found in the code [R8].

Words in the comment text are generally easy to detect, but those in the code are hidden in the names of classes, methods, and variables – often in a snake_cased or CamelCased format – and need to be pre-processed first to extract the individual words from such identifiers [R35]. Moreover, stop words (e.g. 'the', 'is') that occur often in natural language and are relatively meaningless may need to be removed [R42].

To determine whether the words in comments are sufficiently similar to those in the code, one first needs to determine which words from the two sets correspond to each other. Then, the overall comment similarity can be computed by dividing the number of corresponding words by the total number of words in the comment [R35] or by using a cosine similarity measure [R19].

Is up-to-date (UPTODATE)

Comments should be kept **up-to-date** with their code and vice versa.

Rationale: Normally, whenever a change is made to source code the corresponding comments are updated as well [R8]. This is not guaranteed however, as developers may lack the time or motivation or simply forget to do so [R37]. When this happens, comments can become outdated. Comments that are incorrect (CORRECT) may mislead developers and eventually lead to bugs [R9, R21, R37].

Measurement: The most accurate methods to detect whether a comment is up-to-date work by using the change history to determine which changes to code and their corresponding comments were (not) made. We refer to Fluri et al. [R9] and Ibrahim et al. [R13] for a more thorough explanation of this process.

Is succinct (SUCCINCT)

Comments should include all the necessary information, but still be **succinct** (also: **short** or **concise**).

Rationale: Comment length is a double-edged sword. On one hand, longer comments allow developers to include or repeat more information, which may aid comprehension [R28, R35]. On the other hand, long comments take more time and concentration to read [R28]. They are also disliked by many developers, who prefer to obtain their knowledge from the code [R17, R30].

Measurement: The ideal comment length is partially subjective and subject to individual circumstances, like the complexity of the associated code [R31] (also see *Is for complex code*; HARD). Nevertheless we did identify several heuristics in our review, all of which assume that comments should be succinct:

- comments should be at most 30 words long [R12, R35];
- comments with at most two words are too short to contain information that is not already in the code [R35];

- empty comment blocks should not be treated as comments, as IDEs often generate them automatically whenever a developer adds a new class, field, or method [R9];
- HTML tags in Javadoc should not count towards the length of comments [R31];
- inline comments commonly consist of one or at most three sentences [R5].

It is important to note that these guidelines apply primarily to the English language. Other languages (e.g. Dutch or Chinese) may have different characteristics and thus require other thresholds.

Is correct (CORRECT)

A good comment is factually **correct** (also: **accurate**); a developer should be able to trust its contents.

Rationale: Developers who wish to reuse or modify source code use comments to improve their understanding of it. Misleading comments that provide developers with incorrect information can cause them to introduce bugs [R37]. Some therefore believe that 'wrong comments are worse than none at all' [R13].

Moreover, one paper [R13] mentions a study which shows that most developers who encounter an incorrect comment also lose confidence in the reliability of other comments in the same codebase and thus ignore them. This could imply that incorrect comments have a disproportionately large negative impact on comprehension.

Measurement: Unlike code, comments cannot be tested to see if they are (still) valid [R37]. It is therefore not possible to measure whether a comment is correct. All is not lost however, as it is still possible to verify that a comment is up-to-date (UPTODATE), which might be seen as a *prerequisite* for correctness.

Uses meaningful identifiers (IDENTIFIERS)

Code should **use meaningful identifiers** for its class, method, attribute and variable names, and so should comments.

Rationale: It is generally accepted that use of good meaningful identifiers in code that reflect that domain and behaviour of the code make it easier to comprehend [R19, R22].

In some cases meaningful identifiers can even make code self-documenting, removing the need for redundant comments [R19] that do not provide any extra information (EXTRA) on top of what can be inferred from the code.

It is important to include meaningful identifiers and other domain terms from the source code in the comments that remain. Such terms can help comprehension by:

- creating a shared lexicon or vocabulary that facilitates communication among developers and other stakeholders [R6];
- making it easier to find related software artefacts, such as documentation and requirements (TRACE) [R20, R30];
- reinforcing the cohesion of a class [R22], so that readers can more easily understand it as a single unit;

- implicitly making clear that the comment is up-to-date with its comment when it mentions the same identifiers (UPTODATE) [R35].

Measurement: Measurement of this feature can be done in a way that is somewhat similar to that of *Describes code* (CODE), as in both cases there should be some overlap between terms used in comments and the source code.

Is high-level (HIGHLVL)

Comments should use natural language or some other notation to provide **information on a higher abstraction level** than that of the code it describes.

Rationale: The code already is a rich source of low-level information. Comments that are written on the same low level are thus too specific to be of much use if one wants to understand the entire class or method [R27, R28]. This is also why class- and method-level comments have a greater impact on comprehension than comments within methods [R9].

Measurement: It is not entirely clear how this feature can be measured. One paper [R32] suggests that comments that are more high-level can be written using hypernyms (e.g. 'fruit' rather than 'pineapple') and holonyms (e.g. 'computer' rather than 'cpu'). This may have consequences for measurement of other features, like *Describes code* (CODE), *Is up-to-date* (UPTODATE) and *Uses meaningful identifiers* (IDENTIFIERS).

Is near entity type (ENTITY)

A good comment is not only well-written, but also well-placed. We have already discussed that comments are especially valuable when placed near code that is hard to understand (HARD). But some **source code entities**, like (public) methods and variable declarations, are also more likely to warrant a comment than entities like import declarations and variable assignments.

Rationale: It can be useful to document interfaces, public classes, methods, and attributes as these are intended to be reused by other developers [R11]. Conversely, comments on private members and declarations are invisible to external users and thus primarily helpful for maintainers [R9].

In either case, Fluri et al. [R9] note that commenting high-level (HIGHLVL) scopes has a greater impact on comprehension than lower-level scopes or simple statements (HARD).

Measurement: Based on the papers in our review, the list of source code entities that may affect the helpfulness of a comment at the very least includes the following entities: 1. attribute declaration [R8, R9], 2. class declarations [R8, R9, R11], 3. class constructors [R11], 4. method declarations [R8, R9, R11], 5. control structures [R8, R9], 6. loop structures [R8], 7. method calls [R8, R39], 8. variable assignments [R11], 9. variable declarations [R8], 10. package declarations [R8, R11], 11. import declarations [R8, R11] and 12. try-catch blocks [R8, R11].

Comments can be matched to source code entities using a few heuristics. Most importantly, comments are virtually always placed in direct proximity to a source code entity. Multi-line comments typically – but not necessarily – precede their corresponding source code entity, while single-line comments can sometimes also be placed on the same line as the entity [R8, R9]. Note that single-line comments on consecutive lines can sometimes be considered as a single comment. Moreover,

when comments describe code (CODE) one would expect that there are words that appear both in the comment and the corresponding source code [R9, R39].

4.1.6 Features related to usefulness

Of the four criteria included in the quality model by Steidl et al. [69], the usefulness criterion is associated with the largest number of features in our review: *Describes purpose* (PURPOSE), *Is understandable* (UNDERST), *Is for complex code* (HARD), *Mentions task* (TASKS), *Describes design decision* (DECISION), *Provides extra information* (EXTRA), *Describes context* (CONTEXT), *Describes usage* (USAGE), *Enables traceability* (TRACE), and *Does not use abbreviations* (NOABBR). However, these features are likely also some of the hardest to measure, as usefulness may depend on the current activity of a developer. Moreover, 'usefulness' is perceived by developers and thus partially subjective.

Describes purpose (PURPOSE)

A comment should **explain the purpose** of code, i.e. the **rationale** or **why** it exists from a functional perspective.

Rationale: Comments should explain the *why* of code as opposed to the *how* or the *what*, as the *why* cannot be inferred from the code (CODE) [R13, R16]. Because the rationale behind code is easily forgotten, omitting such information from comments might mean that developers need to invest a large amount of time in rediscovering it [R31].

Measurement: Even though this feature is mentioned by many papers in our review, none has proposed a method to detect this feature automatically.

Is understandable (UNDERST)

Code can be hard to understand. Comments allow developers to provide explanations about their code in a more easily understandable way. For this to be possible, first and foremost the comment itself must be **understandable**.

Rationale: Comments are written in natural language, which can be 'much more direct, descriptive, and easy to understand than source code', usually by being less rigorous [R37]. Moreover, the text should also be easy to read on its own, e.g. not be verbose, confusing or contain many spelling and grammatical mistakes [R5, R28, R37].

Measurement: For a comment to be more easily understandable than code, it needs to meet several criteria:

- It needs to be written in a natural language (e.g. English) at a level that is appropriate for its target audience, i.e. not too high, but also not too low.

  Readability heuristics such as the Flesch-Kincaid Grade Level, the Gunning-Fog Index, the SMOG Index, the Automated Readability Index, or other heuristics that count the number of tokens, nouns and verbs, can be used to automatically assess how hard a text is to read. [R2, R15];

- It needs to be free from linguistic mistakes [R5, R16].

Is for complex code (HARD)

A comment should be present for **code that is hard to understand** [R2, R13] or unusual in some other way, e.g. for mathematically intensive code [R4, R15]. The converse is also true: comments are not needed for trivial code that is easy to understand [R31, R42, R43].

Rationale: Comments can be used to enhance the understandability of code, which makes them especially useful when paired with code that is hard to understand [R2]. This is not the only way to make code easier to understand however: refactoring and use of meaningful identifiers (IDENTIFIERS) can improve understandability of code such that it can be seen as 'self-documenting', making comments unnecessary [R19, R31].

Measurement: Hard-to-understand code can be detected using code quality models that are based on code features like the number of identifiers, number of operators, method length, and cyclomatic complexity [R2], or newer methods like cognitive complexity [8, 48].

Two special cases were mentioned in the papers: mathematical calculations [R15], which are harder to understand than normal code, and code related to graphical user interfaces, which may be *easier* to understand [R31]. It is unclear how well these special cases are covered by code quality models.

Mentions task (TASKS)

Comments can be used by developers to **record code-related tasks** that they or future maintainers still need to perform. Comments that are made specifically to record tasks are often called **task comments**, **task annotations**, or **todo comments**.

Rationale: Task comments are primarily a way for developers to communicate with others or their future selves [R6, R13] and therefore do not contribute directly to program comprehension.

Having said that, there may be some indirect effects. For instance, a developer can use a task comment to document a known issue that they have not resolved yet [R35, R36]. This can be seen as a way to provide extra information (EXTRA) or to record (the deferral of) a design decision (DECISION). On the other hand, out-of-date (UPTODATE) task comments can also be hard to understand or obscure meaningful information, and thus negatively impact comprehension [R36].

Measurement: Task comments typically follow a fixed format and usually start with a keyword like `TODO`, `HACK`, `XXX`, `FIXME`, or `REVISIT` [R13, R36]. This makes it easy to find and list them using search tools or IDEs [R13].

Describes design decision (DECISION)

Comments should **document design decisions** and closely related technical information, like requirements, assumptions or informal constraints like pre- and post-conditions.

Rationale: Inferring design decisions and related information from source code can be difficult. One study in particular notes that developers often find such questions hard to answer without external help, which might not always be available [R18]. Including such information explicitly in the comments thus makes it easier for maintainers, external developers and testers to understand what some code should and should not do [R17, R37].

Measurement: It is not clear if and how the presence of information related to design decisions in a comment can be determined algorithmically.

Provides extra information (EXTRA)

Comments should **provide extra information** that cannot be deduced from the source code.

This feature can be interpreted in different ways. Some [R24, R42] appear to believe that a comment would be *more* helpful if it provides extra information, possibly in addition to a summary of the code (CODE). However, others even go as far as to argue that comments are *only* helpful if they provide extra information and should otherwise be removed [R11, R35, R43].

Rationale: Reading source code can provide insights into *what* a program does, but not *why* [R5], as the rationale (PURPOSE) and other types of information, like design considerations (DECISION), cannot be expressed using code. By recording such information in comments, developers can help readers understand implementation and design details [R11, R35].

Measurement: While this feature can in some ways be considered to be the polar opposite of *Describes code* (CODE), it can actually be measured in a very similar way. Steidl et al. [R35] validated the following hypotheses in their study:

- If more than half of the words in a comment are very similar to those used in the method name, it does not provide any extra information;
- Comments with at most two words do not provide any extra information;
- Comments with at least 30 words likely *do* contain information that cannot be deduced from the code.

Describes context (CONTEXT)

A good comment should tell readers more about its *surrounding* **context** beyond the current unit of code, e.g. related classes [R15, R24, R25], and control or data flows [R26].

Rationale: Descriptions of the surrounding context can provide information on a level higher than that of the commented code. This helps developers understand the role of a unit of code within a set of classes, or within a control or data flow. Padioleau et al. [R26] suggest that contextual information may also help developers check for bugs in code or navigate to related code.

Measurement: The presence of contextual information can be confirmed in several ways. One can check whether (methods from) other classes are mentioned in a comment [R15, R24, R39].

Moreover, comments for methods that override or overload another may refer to their parent implementations. Such comments likely include the words 'override' or 'overload' [R39].

Finally, we note that features like *Describes purpose* (PURPOSE), *Uses meaningful identifiers* (IDENTIFIERS), and *Enables traceability* (TRACE) can also be seen as ways to provide contextual information to a unit of code.

Describes usage (USAGE)

Comments for a class or method should **explain how** a developer is supposed **to use it**.

Rationale: Public classes and methods are made for external users. Consequently their comments should accommodate the needs of those who wish to use them [R19].

Measurement: None of the papers offers concrete guidance on how to detect this feature, although it is suggested that comments for methods should describe their input parameters [R23], return value(s) [R25], and pre- or post-conditions [R13] when possible.

In the case of Java programs, this requirement can be easily met by writing class and method comments that follow the Javadoc conventions (JAVADOC), as Javadoc already requires the writer to explain input parameters and return values [R33].

Enables traceability (TRACE)

Comments should **enable traceability** by referring the reader to specific information in external sources.

Rationale: Comments are not the only form of documentation. Other sources, like specifications, design documents, and issue trackers may also contain extra information (EXTRA) that cannot be easily deduced from the source code or comments. Including references can make it easier to find such documentation and relate changes in a system to them [R14]. However, some developers feel that such references clutter the code [R36].

Measurement A comment can support traceability in multiple ways. Traceability links are explicit references to an external source, in the form of hyperlinks [R17] or identifiers, e.g. bug numbers [R14]. Traceability can also be made possible implicitly however, by ensuring that comments use meaningful identifiers (IDENTIFIERS) that appear in those external sources [R20].

Does not use abbreviations (NOABBR)

Text within comments should **not use abbreviations** in lieu of full forms to make comments more succinct (SUCCINCT).

Rationale: Not only does the official Java documentation recommend that developers avoid the use of abbreviations in comments [R15], most empirical studies have also demonstrated that comments with full word identifiers are easier to comprehend than comments with abbreviations [R10, R20]. One study even found that abbreviations are strongly correlated with the number of bug defects in a system [R15].

Measurement: Detection of abbreviations appears to be rarely discussed. Most papers only provide a few examples of abbreviations (e.g. 'aka' for 'also known as'). However, one paper [R33] notes that abbreviations can also be part of identifier names (e.g. `butSelectAll` when referring to a `Button`), which can be mentioned in comments. Techniques based on repository mining can be used to identify such hidden abbreviations.

4.1.7 Features related to completeness

The main idea behind the 'Completeness' criterion is that everything that *can* be commented *must* be commented. This category includes two features about the existence of comments, *Is present* (PRESENT) and *Is complete* (COMPLETE), where the second might be seen as a stronger or more opinionated version of the first.

Is present (PRESENT)

A good comment **is present** (alternatively: is **not missing**). The presence of comments in a codebase can enhance the understandability of code that is being covered by the comments.

Rationale: The notion that adding comments to code can make it more understandable has been around for a long time, and indeed many studies have shown this to be true [R1, R9, R19, R26] – with one major caveat, which is that those studies only considered the total quantity or volume of comments within an entire codebase, rather than the presence of individual comments [R35].

Several other features in our review show that adding comments to code may only make sense in certain situations, for instance when the code is hard to understand (HARD) or when there is extra information that is not already clearly expressed by the code (EXTRA).

It is also interesting to note that changes and additions in well-commented code are more likely to be accompanied with updated and new comments [R13, R26], somewhat analogously to the broken windows theory which suggests that visible signs of (dis)order encourage further (dis)order.

Measurement: The extent to which a codebase is documented can be determined by computing the ratio between all units of code (e.g. methods) that have a comment and all units of code that theoretically could have a comment [R31].

Is complete (COMPLETE)

The public part of a codebase should be (almost) **completely commented** and each individual comment should also be **exhaustive**, i.e. describe all its inputs and outputs [R15], partial or exceptional behaviour [R23, R27], and other information that is necessary to properly understand the code [R39].

Rationale: Completely documenting classes and methods makes it easier for other developers to understand how the application programming interface (API) works and can be used [R35]. Unsurprisingly, this is also what is recommended by the Java documentation guidelines [R33].

This requirement only seems to apply to classes and methods that are part of a program's API as these have a specification, whereas private classes and methods are internal implementation details. Note that protected classes, methods, and fields can also be considered to be part of a program's API [R31].

Measurement: There are two types of completeness. Each comes with its own method of measurement [R31].

First, every public and protected class, method, and field declaration should be accompanied by a comment. The ratio of commented declarations can provide an indication of the completeness on a repository level.

Secondly, comments for all public and protected class, method, and field declarations should contain descriptions for their input parameters, return value, and ex-

ceptions that may be thrown. For Java, these correspond with the `@param`, `@return`, and `@throws` Javadoc tags respectively.

4.1.8 Features related to consistency

Three features are clearly related to the consistency of comments throughout a codebase: *Is Javadoc* (JAVADOC), *Is written in sublanguage* (SUBLANG), and *Is consistent* (CONSIST).

Is Javadoc (JAVADOC)

Javadoc is both a tool and standardised format for certain types of comments in the Java programming language [R15, R23]. Its scope is therefore technically limited to Java, although many other programming languages have similar tools and standardised formats, e.g. pydoc for Python, Godoc for Go, and PHPDoc for PHP.

Rationale: The Javadoc tool is primarily designed to process specially formatted comments in source code to generate API documentation for external consumption. It therefore does not contribute directly to comprehension.

However, it can be argued that the Javadoc format and associated guidelines also encourage developers to document all aspects of their code (COMPLETE) [R26, R31] in a consistent way (CONSIST) [R17], which *can* have an impact on comprehension.

Measurement: Javadoc comments only (need to) appear above public and protected classes, methods, and variables. They can be easily detected as they are delimited using `/** ...*/`, which are generally not used for regular comments [R35, R43]. Javadoc comments often contain structured metadata in the form of tags, which developers can use to document specific aspects of a class, method, or variable [R23, R31].

Is written in sublanguage (SUBLANG)

Comments should be written using a **specialised natural language** (i.e. a **sublanguage**) that is free from linguistic mistakes [R13, R15, R16], and tailored towards developers and the program's problem domain [R5, R37].

Rationale: The natural language in comments often adheres to certain conventions. Some of these conventions are derived from programming standards [R12], but most seem to have evolved and gained acceptance through the community.

Examples include the verbatim inclusion of source code identifiers [R37], a 'telegraphic' writing style that omits articles and pronouns [R5], and use of terms that have very specific meanings within software development or the problem domain, e.g. 'buffer' and 'memory' [R37].

Following conventions makes comments more consistent (CONSIST), which can make them easier to understand [R17, R31].

Measurement: Sublanguages follow rules that are slightly different from those of 'normal' natural languages. These differences can make the use of regular readability tools and spell checkers impractical. We refer to [R5] for a more comprehensive treatise on characteristics of sublanguages. Here, we only list some of the most frequently mentioned differences:

- comments often are not full sentences [R5, R15, R43];

- the full stop symbol is often omitted [R4, R43];
- identifiers in text can resemble spelling errors [R37, R39];
- mathematical expressions [R5].

Is consistent (CONSIST)

A comment should be **consistent** with other comments in terms of language [R5, R35], format [R41], and writing style (SUBLANG) [R12, R14]. Its use of terminology should also be consistent with the code and other types of documentation [R22, R36]. Finally, depending on the conventions of the development team comments might also have to be consistently placed above certain source code entities (ENTITY) [R11].

Rationale: As should be clear from the description above, 'consistent' can refer to various tangentially related concepts. In most cases however these have the same goal. Consistency among comments in the source code of the same or other programs can make them easier to read [R14], which in turn makes it easier to comprehend the comment and the code.

Measurement: While none of the papers explicitly mentions a method to measure consistency of comments, it would make sense that all comments within a codebase should be written in the same language (SUBLANG) and follow Javadoc conventions (JAVADOC). Furthermore, if (virtually) all source code entities of a particular type have corresponding comments it is likely that the intention is that each entity of that type *must* have comments [R35]. Similar arguments can be made for the inclusion of license and copyright information in headers [R35].

4.1.9 Conclusion

In this first phase we set out to answer the question 'What features of source code comments can be derived from literature?'.

The most important features include the extent that a comment describes what the code does, describes what the code is for, is kept up-to-date with its code, is understandable by its readers, and has a reasonable length (i.e. is somewhat succinct).

Comments that have these features are more likely to be helpful to programmers than comments that do not have these features or only have part of these features. The effect of more than half of the features that we have identified in our review has been empirically validated. Nevertheless, in some cases the effect is unclear – despite the existence of empirical studies.

Most features can be measured using static analysis of source code. A few features require information from additional sources such as version control systems. Unfortunately, there are also a few features for which it is still unclear how they should be measured.

4.2 Mining repositories for comment features

In order to learn more about what comments look like in practice, we used our repository analysis tool Coalaty (see section 3.2) on a representative sample of open-source Java projects. As you may recall from chapter 3, this also gives us an

Table 4.5: Overview of implemented features

Feature	Metric	Description
CODE	overlap_percentage	ratio of tokens in the comment that also appear in code
CODE	cosine_similarity	cosine similarity between the comment text and code
UNDERST	flesch_ease	comment text readability, according to Flesch
UNDERST	flesch_kincaid	comment text readability, according to Flesch-Kincaid
UNDERST	gunning_fog	comment text readability, according to Gunning fog
UNDERST	smog_index	comment text readability, according to the SMOG index
UNDERST	automated_readability	comment text readability, according to the ARI
SUCCINCT	tokens	number of tokens in comment text
SUCCINCT	sentences	number of sentences in comment text
JAVADOC	is_javadoc	true if the comment is enclosed by `/**` and `*/`
JAVADOC	is_block_comment	true if the comment is enclosed by `/*` and `*/`
JAVADOC	is_line_comment	true if the comment is preceded by `//`
COMPLETE	coverage	ratio of documented and applicable Javadoc tags
SUBLANG	omitted_full_stops	ratio of omitted full stops in the comment text
SUBLANG	local_identifiers	number of mentioned identifiers within scope
SUBLANG	math_symbols	distinct arithmetic symbols in comment text
CONSIST	is_english	ratio of comment text in proper English
HARD	identifiers	number of identifiers in the accompanying code
HARD	operators	number of operators in the accompanying code
HARD	method_length	method length in number of lines of code
HARD	cyclomatic_complexity	cyclomatic complexity for a method
TASKS	tasks	number of tasks in the comment text
EXTRA	method_name_similarity	true if > 50% of comment text repeats the method name
EXTRA	extra_info_score	ratio of ideal comment length (≥ 30 characters)
CONTEXT	global_identifiers	number of project identifiersin comment text
CONTEXT	mentions_parents	true if the comment text refers to a parent or override
ENTITY	is_for_attribute	true if the comment is above an attribute declaration
ENTITY	is_for_class	true if the comment is above a class declaration
ENTITY	is_for_constructor	true if the comment is above a class constructor
ENTITY	is_for_method	true if the comment is above a method declaration
ENTITY	is_for_package	true if the comment is above a package declaration
ENTITY	is_for_enum	true if the comment is above an enum declaration
ENTITY	is_for_annotation	true if the comment is above an annotation declaration
ENTITY	is_for_interface	true if the comment is above an interface declaration
ENTITY	control_structures	number of control structures within comment scope
ENTITY	loop_structures	number of loop structures within the comment scope
ENTITY	method_calls	number of method calls within the comment scope
ENTITY	variable_assignments	number of variable assignments within scope
ENTITY	variable_declaration	number of variable declarations within scope
ENTITY	try_catch_blocks	number of try-catch blocks within the comment scope
USAGE	describes_inputs	ratio of inputs documented using Javadoc
USAGE	describes_output	ratio of outputs documented using Javadoc
TRACE	hyperlinks	true if the comment text includes at least one hyperlink
TRACE	issue_numbers	number of possible issue numbers in the comment text
NOABBR	abbreviations	true if comment text includes at least one abbreviation

Table 4.6: Project size in lines of code (LOC), and number of files and comments

Project	LOC	Files	Parsed files	Comments	Parsed comments
AFWall+	28,841	645	114 (18%)	1,399	411 (29%)
Amaze File Manager	40,987	905	205 (23%)	2,072	555 (27%)
AntennaPod	60,963	1,477	316 (21%)	1,900	1,021 (54%)
Apache Hadoop	194,770	1,471	514 (35%)	7,439	3,065 (41%)
Apache Spark	6,475	110	33 (30%)	243	103 (42%)
Eclipse CDT	173,992	1,700	776 (46%)	12,491	4,068 (33%)
Google Guava	36,530	317	73 (23%)	1,040	528 (51%)
Google Guice	10,943	125	42 (34%)	227	151 (67%)
ownCloud	62,903	1,174	204 (17%)	3,520	1,248 (35%)
Vaadin	55,018	783	149 (19%)	1,912	964 (50%)
WordPress	140,847	2,329	642 (28%)	5,766	1,457 (25%)

idea of which features cannot be computed reliably and thus are not suitable for our predictive model.

Table 4.5 lists the implemented features. The first column, *Feature*, lists the features from our review (Table 4.3) for which we have provided one or more implementation based on the descriptions of how they can be measured in sections 4.1.5–4.1.8. The second column, *Metric*, shows our name of the implementation. Finally, the third column, *Description*, briefly describes its meaning. We refer the reader to appendix B.3 for a more detailed description of these metrics.

Some notable examples of features for which no implementation exists include *Is correct* (CORRECT) and *Is up-to-date* (UPTODATE). These cannot be computed using the source code as the sole artefact.

4.2.1 Sampled projects

Our sample includes the source code for 11 projects of varying sizes[1]. Table 4.6 provides a summary of various size-related metrics for these projects.

With only 6,475 physical lines of code (LOCs), Apache Spark is clearly the smallest project in our sample. WordPress is the project with the highest file count, while Apache Hadoop is the project with the most lines of code. Neither of these metrics are very useful for us however: most (but especially Android) projects include files that are not source code, e.g. configuration files, stubs, graphical and audiovisual assets, and documentation. The LOC metric is a bit more useful, but also includes test and example files which are excluded from our analysis. After excluding these test and example Java files, Eclipse CDT is clearly the largest project, with 776 Java files that includes approximately[2] 12,491 distinct comments.

The last column in Table 4.6 shows the actual number of comments that have been analysed by our tool. This number is considerably lower than the total number of comments in the Java code, because many comments cannot be clearly

[1]The source code can be obtained via [56] and [57].

[2]Determining the precise number of comments is non-trivial. See appendix B.1 for a more detailed explanation.

matched to some part of the code. Appendix B.2 discusses such comments in more detail.

4.2.2 A first look at comment features in the wild

Table 4.7 provides a high-level summary of the computed metrics for the software projects in our sample[3]. Definitions of each metric are provided in appendix B.3. Note that each metric is independent and may have different units of measurement. In most cases the values between two metrics can therefore not be compared with each other directly. We describe some of the major observations below.

Most comments are placed above methods (65%), followed by attributes (19%), classes (10%), constructors (3%), interfaces (1.7%), enums (0.1%), annotations, (0.1%), and packages (0.01%). Note that this does not necessarily mean that developers primarily choose to document methods: a more likely explanation for this finding is that methods simply occur more often in software projects. Without a comparison with the actual number of entities (methods, attributes, classes, et cetera) it is not possible to determine which entities are commented relatively often.

The majority of comments are Javadoc comments (80%) that can be used to generate API documentation, followed by line comments (11%) and block comments (9%).

Javadoc comments can contain structured metadata, like `@param` and `@return`, which should be used whenever an entity accept arguments or returns values. Our findings suggest that this only happens 42% of the time, which means that most Javadoc comments are technically incomplete.

Although some comments reach lengths of more than 1,400 tokens and as much as 57 sentences, most comments are fairly short. The average comment contains just under 20 tokens and is 2 sentences long. The distributions of both features are heavily skewed however: at the 75th percentile the number of tokens in a comment is only 24, while more than half of all comments only contain one sentence.

The findings for the entities that have been documented with a comment are somewhat similar to those for the comments themselves: the characteristics of commented entities varies wildly, with most entities being relatively simple and thus easy to understand and a smaller number of outliers with extremely high cyclomatic complexity, operators, variable assignments, declarations, or outgoing method calls.

4.2.3 Similarities and differences between projects

Until now, we have only discussed the 'average' comment and assumed that our findings apply to all software projects equally. This is true to some extent, as the mean values for 13 of the 45 metrics show very little differences (< 5%) between all 11 projects and are thus very similar to the mean values in Table 4.7:

- `overlap_percentage` (B.3.1),
- `cosine_similarity` (B.3.2),
- `cyclomatic_complexity` (B.3.21),
- `extra_info_score` (B.3.24),

[3]The complete dataset can be downloaded from https://figshare.com/s/d10cf6030ec7de1ef258.

Table 4.7: Descriptive statistics for computed metrics

Metric	mean	std	min	25%	50%	75%	max
overlap_percentage	0.314360	0.259611	0.000	0.1000	0.286	0.5000	1.000
cosine_similarity	0.236707	0.185578	0.000	0.0910	0.224	0.3540	1.000
flesch_ease	39.433770	105.543308	-1908.175	37.5030	65.642	81.5965	122.158
flesch_kincaid	11.170730	14.447508	-3.400	5.3200	8.350	11.5725	279.800
gunning_fog	11.944296	7.979842	0.000	8.0015	11.314	14.6050	52.607
smog_index	9.395972	3.574248	0.000	8.8420	9.607	11.2080	33.358
automated_readability	16.229663	18.885221	-11.510	8.8640	12.419	16.7500	459.490
tokens	19.704007	27.304685	0.000	6.0000	12.000	24.0000	1424.000
sentences	2.090030	1.893671	0.000	1.0000	1.000	3.0000	57.000
is_javadoc	0.792783	0.405327	0.000	1.0000	1.000	1.0000	1.000
is_block_comment	0.094532	0.292578	0.000	0.0000	0.000	0.0000	1.000
is_line_comment	0.112685	0.316220	0.000	0.0000	0.000	0.0000	1.000
coverage	0.413921	0.428375	0.000	0.0000	0.333	1.0000	1.000
omitted_full_stops	0.563724	0.440106	0.000	0.0000	0.667	1.0000	1.000
local_identifiers	1.584016	10.841801	0.000	0.0000	0.000	1.0000	1161.000
math_symbols	0.086783	0.440223	0.000	0.0000	0.000	0.0000	5.000
is_english	0.953817	0.143156	0.000	1.0000	1.000	1.0000	1.000
identifiers	21.668216	54.475709	0.000	2.0000	7.000	20.0000	2212.000
operators	1.941111	5.490300	0.000	0.0000	0.000	2.0000	140.000
method_length	6.454653	13.693136	0.000	0.0000	2.000	7.0000	353.000
cyclomatic_complexity	2.044794	3.843659	0.000	0.0000	1.000	2.0000	122.000
tasks	0.008413	0.102745	0.000	0.0000	0.000	0.0000	4.000
method_name_similarity	0.363589	0.481050	0.000	0.0000	0.000	1.0000	1.000
extra_info_score	2.153845	3.535514	0.000	0.0000	1.267	2.8670	82.600
global_identifiers	5.216737	5.431169	0.000	2.0000	4.000	7.0000	73.000
mentions_parents	0.000590	0.024291	0.000	0.0000	0.000	0.0000	1.000
is_for_attribute	0.185817	0.388973	0.000	0.0000	0.000	0.0000	1.000
is_for_class	0.105601	0.307338	0.000	0.0000	0.000	0.0000	1.000
is_for_constructor	0.038669	0.192811	0.000	0.0000	0.000	0.0000	1.000
is_for_method	0.649399	0.477176	0.000	0.0000	1.000	1.0000	1.000
is_for_package	0.000959	0.030959	0.000	0.0000	0.000	0.0000	1.000
is_for_enum	0.001476	0.038391	0.000	0.0000	0.000	0.0000	1.000
is_for_annotation	0.001033	0.032127	0.000	0.0000	0.000	0.0000	1.000
is_for_interface	0.017047	0.129450	0.000	0.0000	0.000	0.0000	1.000
control_structures	1.312966	3.083077	0.000	0.0000	0.000	1.0000	113.000
loop_structures	0.108258	0.414158	0.000	0.0000	0.000	0.0000	7.000
method_calls	3.453767	8.461595	0.000	0.0000	0.000	3.0000	324.000
variable_assignments	0.643347	2.090380	0.000	0.0000	0.000	1.0000	63.000
variable_declaration	0.833075	2.148275	0.000	0.0000	0.000	1.0000	40.000
try_catch_blocks	0.071877	0.323038	0.000	0.0000	0.000	0.0000	8.000
describes_inputs	0.416067	0.489406	0.000	0.0000	0.000	1.0000	1.000
describes_output	0.420670	0.490278	0.000	0.0000	0.000	1.0000	1.000
hyperlinks	0.007970	0.102060	0.000	0.0000	0.000	0.0000	3.000
issue_numbers	0.000000	0.000000	0.000	0.0000	0.000	0.0000	0.000
abbreviations	0.213195	0.694580	0.000	0.0000	0.000	0.0000	15.000

- `global_identifiers` (B.3.25),
- `method_name_similarity` (B.3.23),
- `is_for_attribute` (B.3.27),
- `is_for_method` (B.3.30),
- `method_calls` (B.3.37),
- `variable_assignments` (B.3.38),
- `try_catch_blocks` (B.3.40),
- `describes_inputs` (B.3.41), and
- `describes_output` (B.3.42).

There does not appear to be an obvious reason why the average computed values for these metrics are so similar across projects. It is possible that there are widely accepted software development practices that developers try to follow. For instance, the values for `overlap_percentage` (B.3.1), `cosine_similarity` (B.3.2), `extra_info_score` (B.3.24), and `method_name_similarity` (B.3.23) all are within the range that was determined to be ideal by Steidl et al. [69]. Moreover, the `cyclomatic_complexity` (B.3.21) and some of its related metrics, like `variable_assignments` (B.3.38) and `try_catch_blocks` (B.3.40), have a low value almost everywhere, which suggests that most code in methods is kept simple. On the other hand, `describes_inputs` (B.3.41) and `describes_output` (B.3.42) have lower values than one would expect, given the supposed importance of completeness for Javadoc comments.

To determine whether and which differences exist between projects, we computed the same descriptive statistics per project. Tables D.1–D.11 in appendix D list the metrics whose mean values differ most noticeably from the overall mean across all projects, i.e. they show a difference of at least 50% relative to the overall mean.

At first glance it is already clear that there are differences between projects. Many of the low values in Table 4.7 are apparently due to the complete absence of certain features in many of the projects. For instance, comments are not placed above packages in 6 out of 10 projects. The distribution of values also differs between projects, which possibly suggests that different projects prioritise different aspects of source code. For example, based on the computed metrics it appears that the source code of the Google Guice project is relatively tidy, which possibly removes some of the need to describe all inputs and outputs (Table D.8). On the other hand, the source code for Amaze File Manager appears to have longer methods that contain more logic, which is associated with a higher number of task comments (TODOs) and references to documentation (Table D.2).

4.2.4 Correlations between comment metrics

As discussed earlier in section 3.4.2, some of the metrics may be correlated with each other. High levels of multicollinearity are to be avoided.

A simple and intuitive way to detect multicollinearity[4] is by plotting correlations between each pair of features in a correlation matrix. Figure 4.2 visualises correlations between the metrics that we computed for the comments across all projects in our dataset[5].

Pairs of metrics that are not or very weakly correlated with each other are visualised using light, lowly-saturated colours, whereas strongly correlated pairs are visualised using bright, highly-saturated cells. The diagonal red line in the centre shows self-correlations, which are meaningless and thus best ignored. The line divides the matrix into two halves, which are mirrored versions of each other. Note that the row and column for `issue_numbers` (B.3.44) is blank due to the lack of any variance for this metric in our sample of software projects.

From this visualisation it is possible to extract several factors. Factors are groups of metrics that are highly collinear, which suggests that each of the metrics essentially aims to measure the same 'thing'. This has implications for our model construction in a later phase, as metrics that are highly collinear can be safely discarded in order to keep models as simple as possible (see section 3.4.2). Collinear metrics can be found by examining clusters of at least two adjacent bright, highly-saturated cells that appear below the diagonal line and relating them to the corresponding metrics.

We can identify the following factors, which neatly map to some of the features that we saw earlier in literature and from which the underlying implementations were derived:

Code-comment similarity `overlap_percentage` (B.3.1) and `cosine_similarity` (B.3.2) are two specific implementations of the *Describes code* (CODE) feature, which is about the extent that the comment and the code look alike.

Comment readability For the *Is understandable* (UNDERST) feature we implemented five different commonly used readability formulas, `flesch_ease` (B.3.3), `flesch_kincaid` (B.3.4), `gunning_fog` (B.3.5), `smog_index` (B.3.6), and `automated_readability` (B.3.7). These metrics are clearly visible as the second cluster in Figure 4.2. Note that the correlation between `flesch_ease` (B.3.3) and the other readability formulas in negative. This is because its direction is reversed, i.e. a higher score means that a text is *less* readable rather than more readable.

Method difficulty This factor is actually spread across two clusters, both of which include metrics that aim to capture how difficult it is to understand a unit of code: `operators` (B.3.19), `method_length` (B.3.20), `cyclomatic_complexity` (B.3.21), `control_structures` (B.3.35), `loop_structures` (B.3.36), `method_calls` (B.3.37), `variable_assignments` (B.3.38), `variable_declaration` (B.3.39), and `try_catch_blocks` (B.3.40). In our literature review, these were related to the *Is for complex code* (HARD) and *Is near entity type* (ENTITY) features, which are about the difficulty and properties of code respectively. Many of these metrics are also part of general code complexity measures, which we described in chapter 2.

Comment length The length of a comment can be measured in various ways, e.g. by counting the number of `tokens` (B.3.8) or `sentences` (B.3.9). A longer

[4]Section 4.4 discusses a more formal approach to detection of multicollinearity.

[5]Note that the lower left diagonal half is a mirrored version of the upper right half.

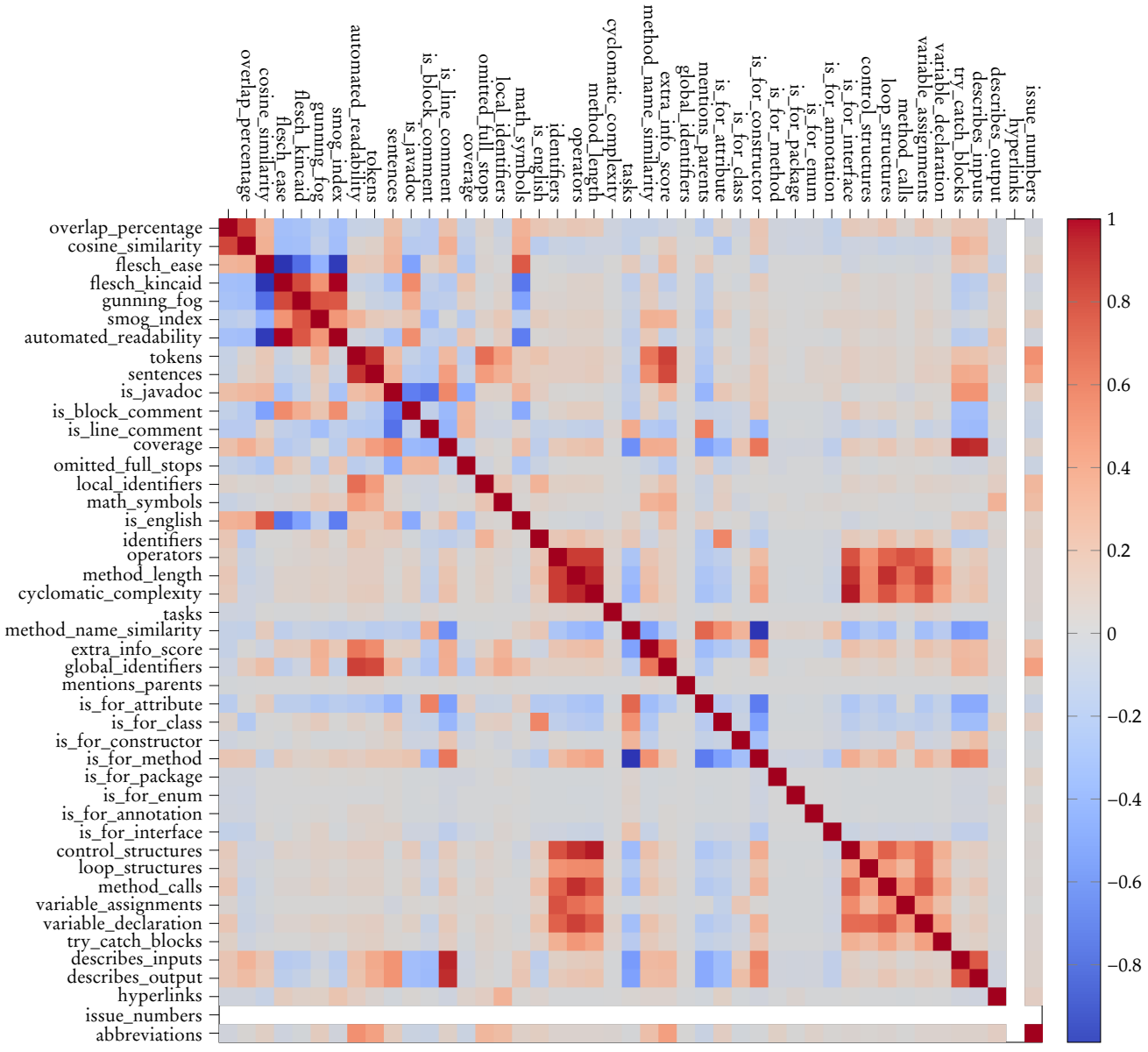


Figure 4.2: Correlation matrix for comment metrics

comment is more likely to contain extra information that cannot be deduced from the code, which results in a higher `extra_info_score` (B.3.24). Interestingly enough, there appears to be an even stronger relationship with `global_identifiers` (B.3.25). This suggests that longer comments are often used to describe the context in which a unit of code is used. These metrics are derived from three different features: *Is succinct* (SUCCINCT), *Provides extra information* (EXTRA), and *Describes context* (CONTEXT). The first is related to the 'Coherence' quality criterion, while the other two metrics are related to 'Usefulness' (Table 4.4). This could suggest that different categorisations are possible.

Comment completeness A Javadoc comment `describes_inputs` (B.3.41) and `describes_output` (B.3.42). These two metrics are highly correlated with `coverage` (B.3.13), which combines the two earlier metrics into one single value. Note that despite the clear relationship between these two sets of metrics, they are derived from two seemingly unrelated features in our review: *Describes usage* (USAGE) and *Is complete* (COMPLETE); the purpose of `coverage` (B.3.13) is simply to determine whether the usage has been documented completely.

4.2.5 An ideal number of principal components?

Principal component analysis (PCA) is another correlation-based method that allows us to look at our data in a different way [23]. PCA works by converting our metrics into principal components, each of which consisting of a combination of linearly uncorrelated metrics [63]. It is therefore not directly applicable for our model, which must be easily explainable. However, it can still be useful to estimate how many variables we should keep when we construct a predictive model in the next phase.

Figure 4.3 shows a scree plot, in which the variance explained by each principal component is plotted against the nth principal component that we retain. This plot shows that the first (and 'best') principal component only manages to explain about 16% of the variance in the data, with each subsequent principal component contributing less to the overall ability to explain the variance.

Of particular interest in this plot is the so-called point of inflexion [23], which one might informally describe as the 'elbow' in the plotted line. It appears that this point of inflexion lies around the 7^{th} component, which means that the ideal number of components would be 6. Unfortunately these 6 components would only be able to explain 54.3% of the variance in the data, which would likely not suffice for a well-performing predictive model.

4.2.6 Conclusion

After learning what features of source code comments are described in scientific literature (section 4.1), we set out to answer our second research question, 'What do features of source code comments look like in open-source Java projects?', in order to empirically validate our findings from theory.

We constructed a repository mining tool that computes many of the features from our literature review, based on the descriptions of how they can be measured that we have found in the reviewed papers. This resulted in 45 distinct implementations of features (Table 4.5). We used this tool to analyse comments in the source

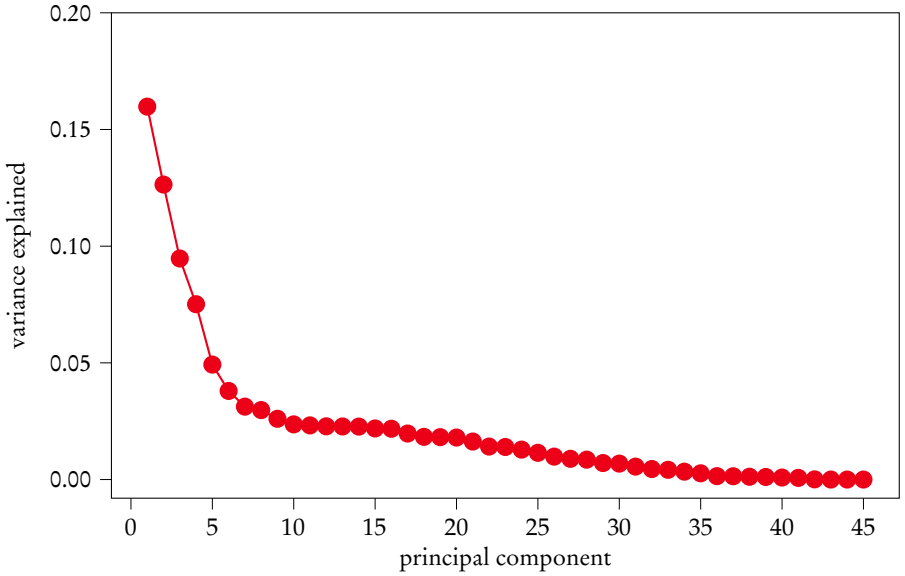


Figure 4.3: Scree plot of principal components for our comment metrics

code of a representative sample of 11 open-source Java projects (Table 4.6).

Most of the comments that we found are placed above methods. When we looked at the overall characteristics of comments across projects, we saw that 13 out of 45 features have values that are very similar in all of the projects in our sample. Many of these 13 features have values that correspond with what has been suggested in literature as good values. At the same time, there are also many features for which the values differ greatly between projects, with varying degrees of adherence to the best practices that we identified in scientific literature. Even without a predictive model for comment quality we can therefore already see that differences in comment quality exist between projects.

4.3 Measuring the perceived helpfulness of comments

We conducted a survey to learn more about what developers actually consider to be features of good comments. The survey was active from 18 June 2021 until 25 November 2021. During this period we recorded 220 survey starts, which resulted in 64 complete responses that were *potentially* usable for further analysis[6].

4.3.1 Detection and exclusion of fraudulent responses

Monetary incentives were offered to respondents for completing the survey (see section 3.3.3) in the form of a gift card raffle and a charitable donation. We suspected that these incentives might attract some respondents who cheat by completing the survey as quickly and with as little effort as possible and/or by taking the survey multiple times. It is possible to detect such fraudulent responses by incorpor-

[6]An anonymised file containing all survey responses can be downloaded from https://figshare.com/s/d10cf6030ec7de1ef258.

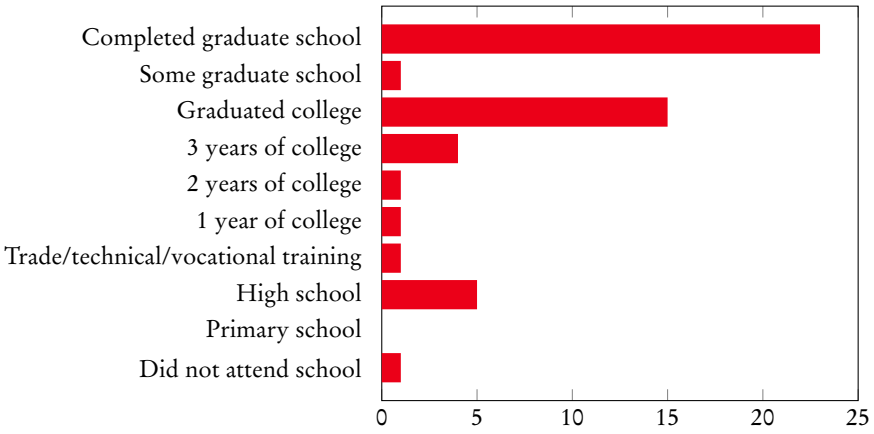


Figure 4.4: Self-reported education levels

ating check questions into surveys [10]. However, we opted to not include such questions in our survey in order to keep it as simple as possible for respondents.

We therefore conducted an informal fraud analysis for survey responses that were submitted by those that opted to participate in the raffle. Although more sophisticated methods of fraud detection based on behavioural monitoring [32] and detection of suspicious IP addresses [79] exist, these could not be used due to privacy concerns.

First, we identified responses where each answer was answered in the same way (e.g. 'strongly agree') as it is extremely unlikely that someone would provide the same answers for each of the snippets in the survey. Then, we looked for responses that were made within the same period of time and were extremely similar to each other, despite large differences in demographic background.

We identified 12 responses that were likely fraudulent. After excluding these from our dataset, 52 valid responses remain for further analysis.

4.3.2 Demographic background of respondents

We concluded the survey with three sets of questions about the demographic background of respondents.

Level of education

Figure 4.4 shows that our 52 respondents are highly educated. Close to half of all respondents has completed graduate school and holds at least a master's or doctoral degree, while about a third is currently enrolled in a bachelor's or graduate programme. One respondent reported that they 'Did not attend school'. It is not clear whether this is due to a misinterpretation.

When we compare the level of education in our sample with those found in the Stack Overflow Developer Survey of 2021 [55], it appears that highly educated respondents are over-represented in our sample. Given that master's degrees are relatively common among software developers in our personal and professional networks, this was to be expected.

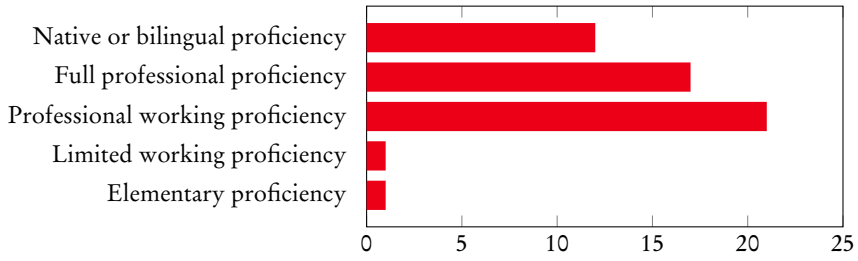


Figure 4.5: Self-reported English proficiency level

English proficiency

We also asked respondents to estimate their English proficiency. A lower level may affect how well they are able to understand comments and instructions provided by the survey, as both are written in English. Figure 4.5 shows these self-reported English proficiency levels.

Unsurprisingly, virtually all respondents report having at least professional working proficiency, while a large number also claim to have full professional or bilingual proficiency. Two respondents reported having only limited or elementary level of proficiency. Coincidentally, these respondents had only completed high school at the time of their survey submission.

Programming experience

We asked respondents about their experience with computer programming using four questions that were proposed by Feigenspan et al. [22]. Figure 4.6 visualises the responses for these four questions.

When asked to estimate their programming experience on a scale of 1 to 10, most respondents answered with a 7 or higher (Figure 4.6a). While our survey shows snippets with Java code, it appears that our respondents' programming language of choice is not always Java (Figure 4.6b).

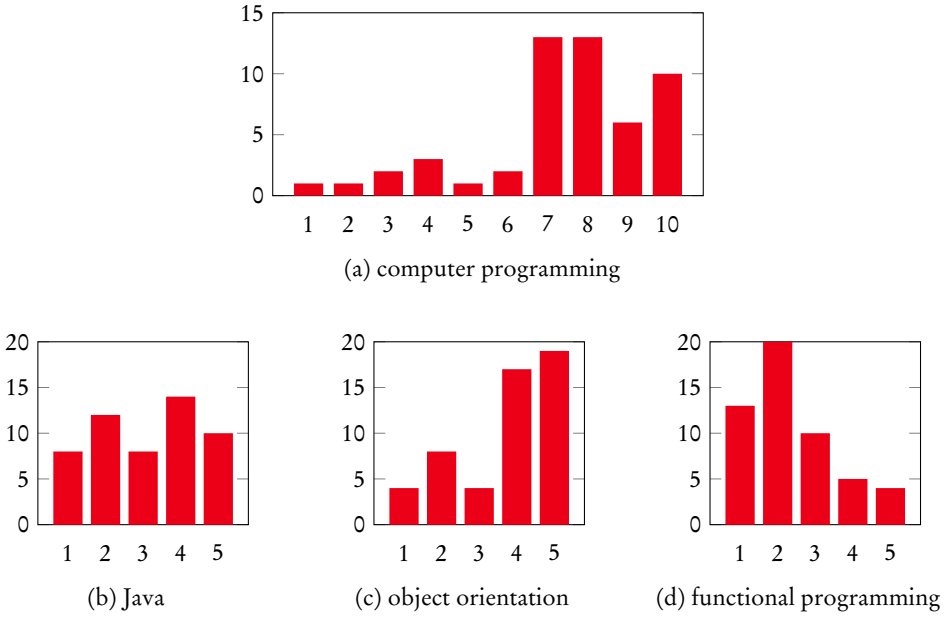
Most respondents do seem to use other, possibly similar object-oriented languages as they report high levels of experience with object-oriented programming (Figure 4.6c).

Finally, we asked respondents about their experience with functional programming using less mainstream languages like Haskell, Erlang, or Elixir. Experience with such languages can be a sign that a respondent's programming skills are above average, which could affect how they perceive code and comments. Figure 4.6d suggests that this is not the case for most of our respondents.

4.3.3 Helpfulness ratings for snippets

The main part of the survey consisted of code snippets that included a Java comment, and the statement 'I think this comment improves my understanding of the code' (Figure 3.2). Respondents were asked to indicate their agreement with this statement using a five-point Likert scale.

Figure 4.7 provides a summary of the responses for each snippet in the form of a stacked bar chart. Agreement is visualised using blue bars on the left, while



(a) computer programming

(b) Java (c) object orientation (d) functional programming

Figure 4.6: Self-reported programming experience (higher is better)

disagreement is visualised using red bars on the right and neutral responses are shown in light grey. A vertical, dotted line in the centre of the visualisation divides it into two equally-sized halves. If we treat the Likert scale as an interval scale[7], one might say that this divider line shows the mode for each snippet.

Overall observations

In the visualisation there are more blue bars on the right side of the dotted divider line than red bars on the left. This suggests that respondents were overall at least somewhat positive about the helpfulness of comments. When we code 'strongly agree' as 5 and 'strongly disagree' as 1, the mean value for all responses is 3.168, which one might interpret as neutral or at best very weak agreement.

What the visualisation also clearly shows is that there is no such thing as universal agreement about the helpfulness of comments: for each snippet there were many respondents who thought it improved their understanding of the code, but simultaneously also many respondents who did *not* believe that the comment was helpful.

Clear examples of good and bad comments

While the helpfulness of most comments in our survey might be seen as controversial there are a few exceptions.

For instance, the majority of respondents believed that the comments in snippets C.3, C.9, C.21, C.29, and C.31 were not helpful. At first glance, it appears that

[7] A sizeable amount of disagreement exists within the scientific community about the 'right' way to analyse Likert scales – especially those with a low number of answer options, like ours. Some argue that they are ordinal scales [81], which normally means that they cannot be used with arithmetic operations.

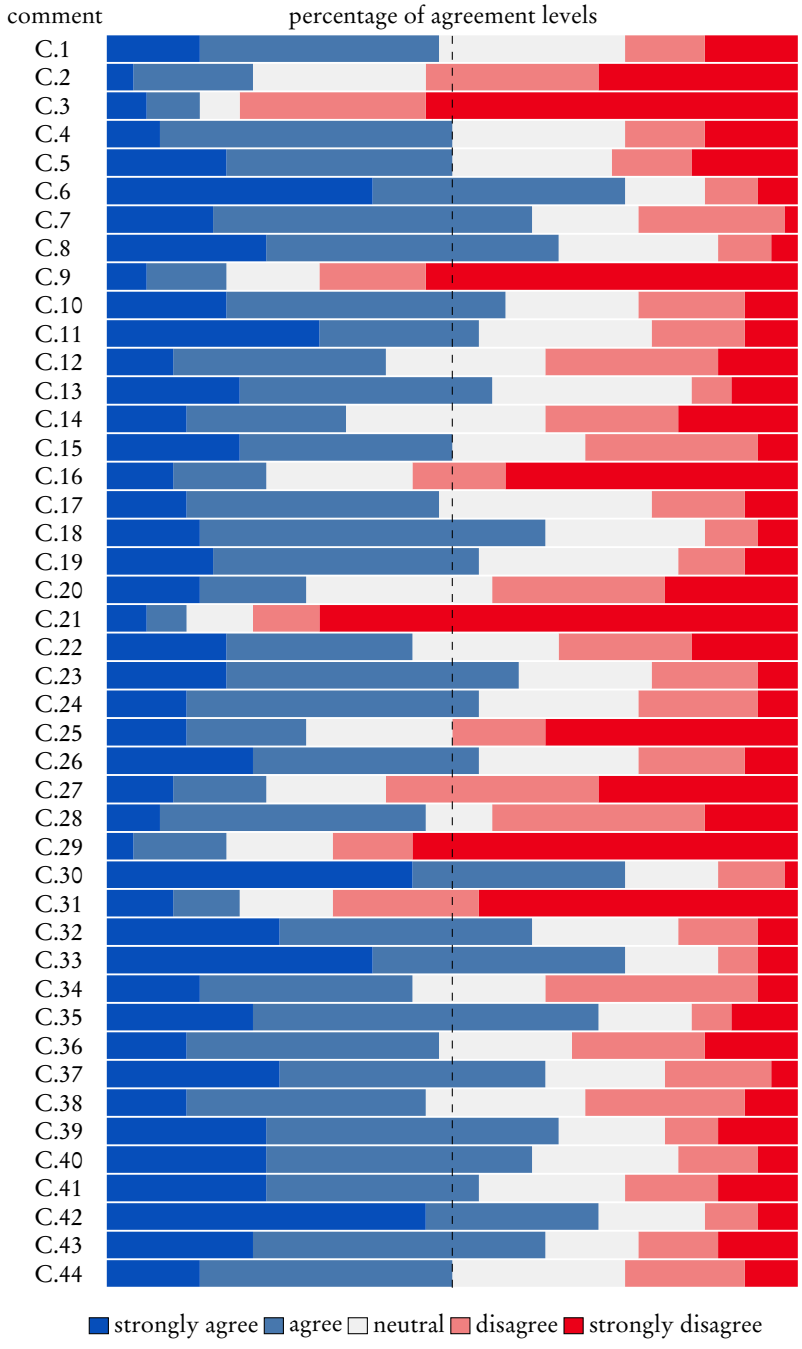


Figure 4.7: Levels of agreement with statements that a comment was helpful. Comments C.1–C.44 can be found in Appendix C.4 by looking up the identifiers next to the chart. They are also clickable in the digital version of this document.

these comments are all relatively short (SUCCINCT) and do not contain information that can be deduced from the code (CODE).

Conversely, the comments in snippets C.6, C.8, C.30, C.33, C.35, and C.42 were almost universally thought to be helpful. It is possible that some relationship exists between these positive ratings and features like *Is high-level* (HIGHLVL), *Provides extra information* (EXTRA), and *Describes usage* (USAGE).

We discuss possible relationships between features and the perceived helpfulness of comments in more detail in section 4.4.

Potential factors

Our literature review showed that there may be several factors that contribute to the perceived helpfulness of comments. A large number of snippets were accompanied by a secondary statement about one of four factors (Table C.1), which respondents also rated on a five-point Likert scale: whether a comment is easy to understand, looks like a proper Javadoc comment, contains the right amount of information, or includes too little information to be of use to a reader.

Figure 4.8 visualises the answers that respondents have provided for each of these secondary statements. Each dot represents the overall answer of a single respondent. Their position on the horizontal axis is determined by transforming the responses to these statements to a nominal scale and computing the mean value for responses across all snippets where the statement was shown. The position on the vertical axis is determined in a similar way for the primary statement about the perceived helpfulness of the comment in the snippet for that same set of snippets.

As expected, a clear positive relationship exist for three of these factors, i.e. a comment is more likely to be perceived as helpful if it:

- is easy to understand (Figure 4.8a);
- follow conventions for Javadoc comments (Figure 4.8b);
- does not contain too little or too much information (Figure 4.8c).

Figure 4.8d shows that the relationship is a lot less clear for the fourth factor, which relates to the question whether the code would have been just as easy to understand without the comment. One would expect that a comment that is thought to be unnecessary is also seen as less helpful, but this does not appear to be the case for everyone. This may suggest that the helpfulness ratings are mostly unaffected by the comment's perceived 'right to exist'.

Effect of control variables

Since programming experience is an important confounding factor in controlled experiments that involve program comprehension [22], we suspected that programming experience could also affect the perceived helpfulness of comments. After all, comments are used to provide explanations about code and different readers may require different amounts of information. Moreover, a respondent's level of education and mastery of English may also affect how they perceive comments.

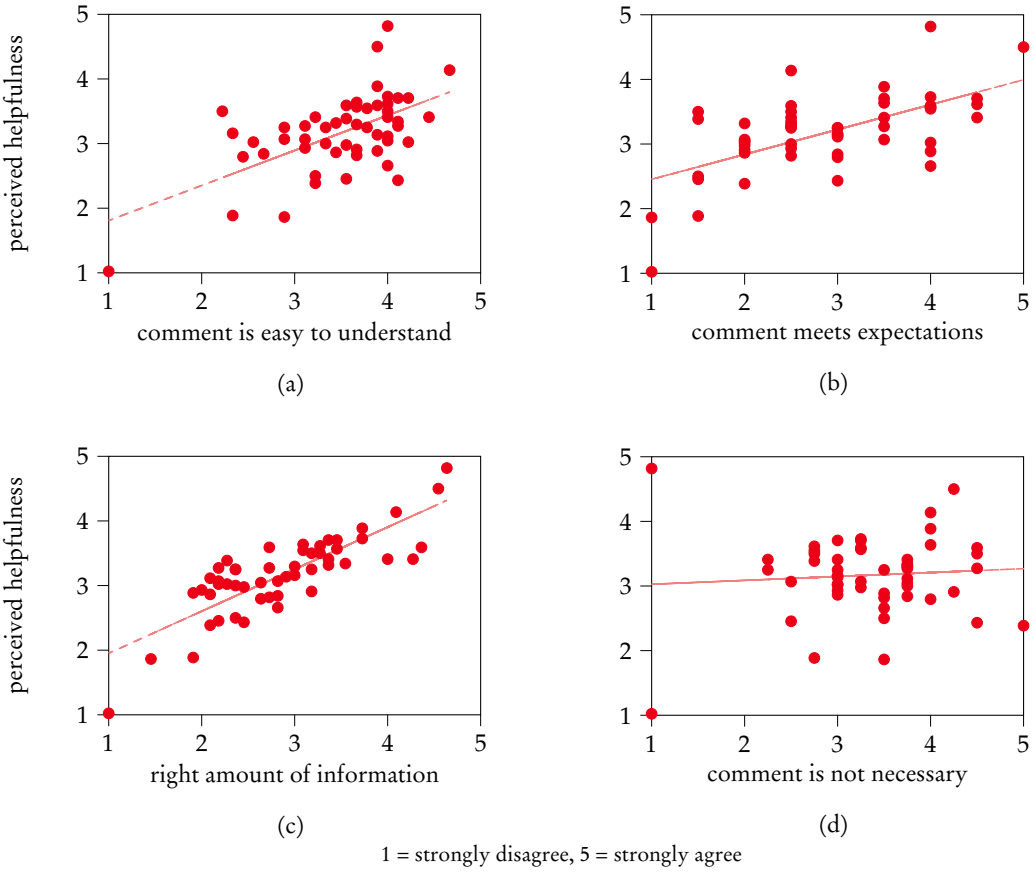


Figure 4.8: Relationships between secondary statements and perceived helpfulness

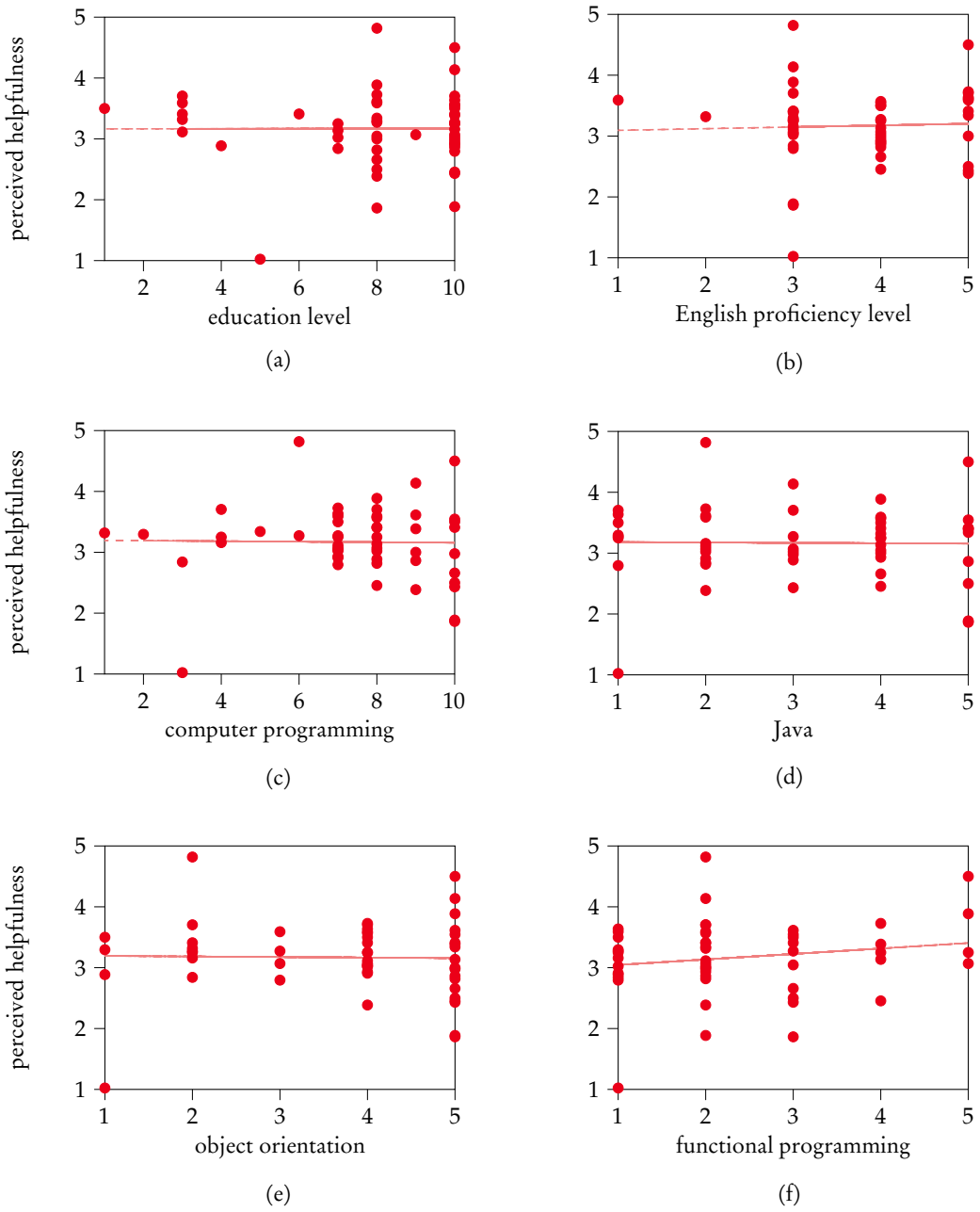


Figure 4.9: Relationships between control variables and perceived helpfulness

Figure 4.9 shows scatter plots for each of the six control variables that we measured using our survey. The horizontal axis shows the self-reported levels of education or proficiency, while the vertical axis represents the average helpfulness rating given by a respondent across all snippets.

The correlations with perceived helpfulness appear to be negligible for virtually all control variables, as is demonstrated by the nearly horizontal best-fit line. The only exception is the amount of *experience with functional programming* (Figure 4.9f), which seems to have a slight positive relationship with perceived helpfulness.

Overall, the lack of correlations suggests that programming experience does not have a discernable effect on the perception of comments. Assuming that the self-reported experience levels are accurate, this is a somewhat surprising result.

4.3.4 Conclusion

Our survey we asked respondents to rate the perceived helpfulness of 44 snippets, whose features we computed earlier (section 4.2). It was completed by 52 respondents, of which most are highly educated. Results show that in general a lot of disagreement exists among respondents: there only a few snippets for which a clear majority of respondents agrees that they are either helpful or unhelpful. This disagreement makes for noisy training data, which can make it difficult to train a well-performing machine learning model for comment quality.

4.4 The relationship between comment features and perceived comment quality

We constructed and evaluated several machine learning models which we trained on our dataset of input variables (the features computed using our repository mining tool, listed in Table 4.5) and the output variables (the perceived helpfulness of comments, as rated by respondents in our survey) using the popular scikit-learn library for Python.

4.4.1 Data cleaning

As discussed earlier in section 3.4.2, linear regression models require that a linear relationship exists between each input feature and the output variable. Visualisations make it easy to detect the type of relationship (linear or otherwise) between two variables [72].

Figure 4.10 shows these relationships using scatter plots. Each dot represents one of the comments in our survey. The computed value of the feature determines its position on the horizontal axis, while the averaged perceived helpfulness ratings for that comment determines its position on the vertical axis. The plots clearly show that the relationship is not always linear. More specifically, we can distinguish between three types of features:

1. Features for which a linear relationship may or may not exist, e.g. the `control_structures` (B.3.35) and `variable_declaration` (B.3.39) features;

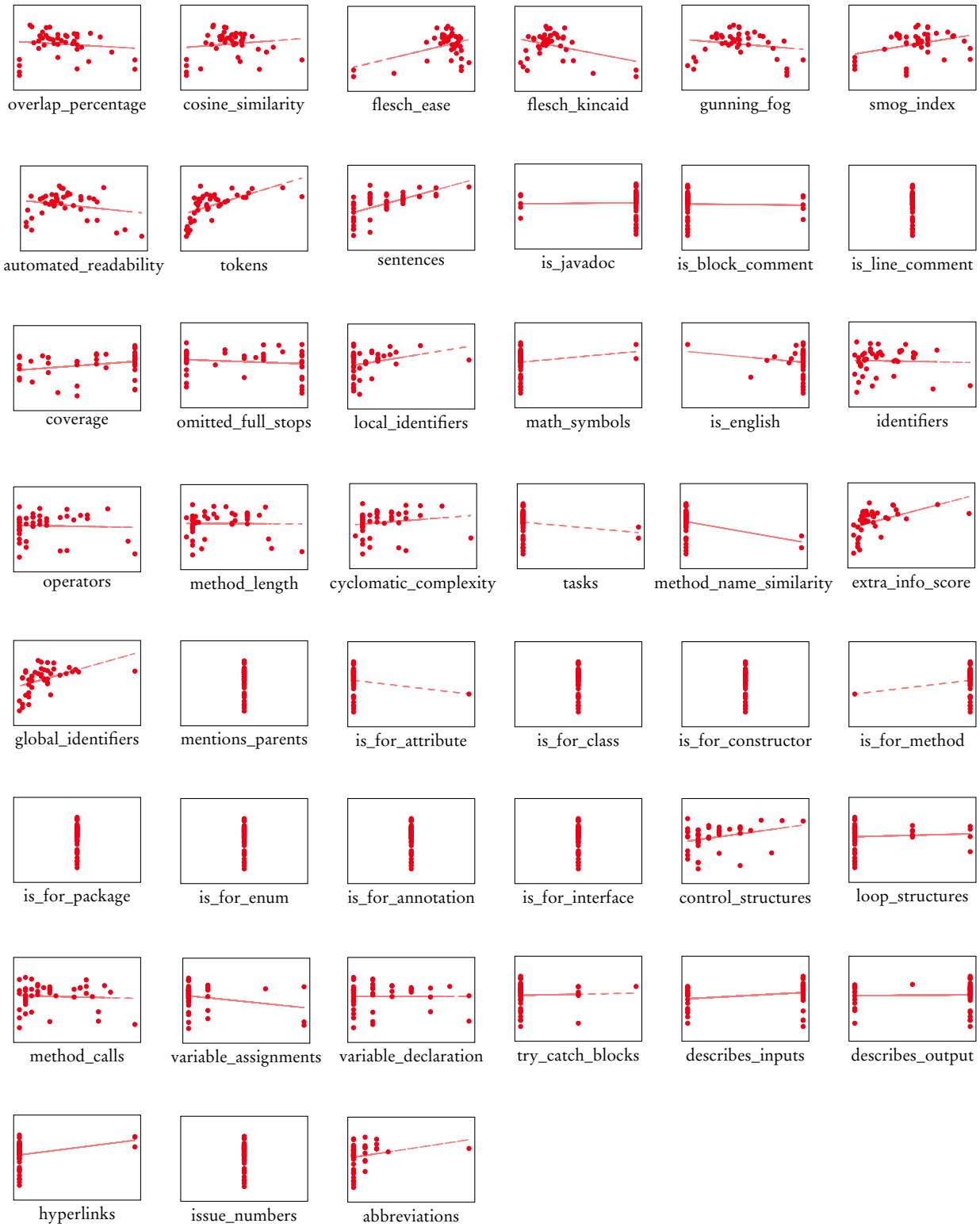


Figure 4.10: Relationships between comment feature values (horizontal) and perceived helpfulness ratings (vertical)

2. Features for which a *non-linear* relationship appears to exist. Examples include `tokens` (B.3.8), which likely has a logarithmic relationship with helpfulness, and `gunning_fog` (B.3.5), which shows a quadratic relationship with helpfulness.

   Note that both results are expected: The precise number of `tokens` (B.3.8) matters more when the number is low – with larger numbers, the number of tokens is more likely to be perceived as simply 'large' (see section 3.4.2). With regard to the readability-based `gunning_fog` (B.3.5) feature, we previously discussed in section 4.1.6 that comment text should not be too difficult or simplistic.

3. Features which have zero variance in our sample and thus no relationship at all, e.g. `mentions_parents` (B.3.26) and `is_for_annotation` (B.3.33);

Features in the first category can be included in a regression model as is, while features in the second category can be included after applying a transformation so that the relationship becomes linear: for features with a logarithmic relationship we apply a log transformation, while features for which a quadratic relationship exists we compute the inverse distance to the value that yields the highest helpfulness rating.

The remaining features are essentially meaningless for our purpose and are thus discarded. We also discard features that are highly multicollinear, i.e. those that have a variance inflation factor greater than 5. Table 4.8 provides an overview of which features are transformed or discarded before or during model construction.

4.4.2 Comparison of model performance

We used 10-fold cross-validation to train and evaluate six different predictive models. Three models are regressors (linear, decision tree, and support vector machine) and three are classifiers (logistic 'regressor', decision tree, and support vector machine).

Table 4.9 lists the three regressors, along with three metrics that are commonly used for regressor models. The goal of all three metrics is to estimate the distance between predictions of the model and actual (expected) values, but this is done in slightly different ways:

Mean absolute error (MAE) This is the simplest of the three metrics. It looks at the absolute difference (i.e. error) between the predicted and expected value for each example in the data. The final score is obtained by computing the mean of all these errors. Lower values are better.

Root-mean-square error (RMSE) The second metric is conceptually similar to MAE, but differs in two ways: 1) the error between each predicted and expected value is squared, and 2) the overall score is the square root of the averaged errors. This has the effect of assigning greater weight to larger errors. Again, lower values are better.

Adjusted R^2 R^2 is a metric that tells us how much of the variance the regression model is capable of explaining. The Adjusted R^2 is a modified version of that metric that also takes the effect of including larger numbers of variables into account. Values are generally between 0 and 1, higher values are better.

Table 4.8: Overview of features for machine learning models

Feature	Included	Reason for exclusion	Applied transformation
overlap_percentage	no	high VIF (> 5)	distance to ideal value
cosine_similarity	yes		distance to ideal value
flesch_ease	no	high VIF (> 10)	
flesch_kincaid	yes		distance to ideal value
gunning_fog	no	high VIF (> 10)	distance to ideal value
smog_index	no	high VIF (> 5)	distance to ideal value
automated_readability	no	high VIF (> 10)	distance to ideal value
tokens	no	high VIF (> 10)	log
sentences	yes		log
is_javadoc	no	high VIF (∞)	logit
is_block_comment	no	high VIF (∞)	logit
is_line_comment	no	low variance	
coverage	no	high VIF (> 10)	logit
omitted_full_stops	no	high VIF (> 10)	logit
local_identifiers	yes		log
math_symbols	yes		
is_english	no	high VIF (∞)	logit
identifiers	no	high VIF (> 10)	
operators	no	high VIF (> 10)	distance to ideal value
method_length	no	high VIF (> 10)	log
cyclomatic_complexity	no	high VIF (> 5)	
tasks	yes		
method_name_similarity	no	high VIF (> 10)	logit
extra_info_score	yes		log
global_identifiers	no	high VIF (> 5)	log
mentions_parents	no	low variance	
is_for_attribute	no	high VIF (∞)	logit
is_for_class	no	low variance	logit
is_for_constructor	no	low variance	
is_for_method	no	high VIF (∞)	
is_for_package	no	low variance	
is_for_enum	no	low variance	
is_for_annotation	no	low variance	
is_for_interface	no	low variance	
control_structures	yes		
loop_structures	yes		
method_calls	no	high VIF (> 5)	
variable_assignments	yes		
variable_declaration	yes		
try_catch_blocks	yes		
describes_inputs	no	high VIF (> 10)	logit
describes_output	no	high VIF (> 10)	logit
hyperlinks	yes		
issue_numbers	no	low variance	
abbreviations	yes		
_control_education	no	high VIF (∞)	
_control_english	no	high VIF (∞)	
_control_prog_exp	no	high VIF (∞)	
_control_oo_exp	no	high VIF (∞)	
_control_java_exp	no	high VIF (∞)	
_control_func_exp	no	high VIF (∞)	

VIF = variance inflation factor

Table 4.9: Performance comparison for regression models

Model	MAE	RMSE	Adjusted R^2
Linear regressor	0.31704	0.36646	-0.74188
Decision tree regressor	0.31466	0.39379	-1.52904
Support vector machine regressor	0.30496	0.36906	-0.23653

RMSE = root-mean-square error
MAE = mean absolute error

All three models have very similar a MAE, ranging from 0.30496 (best) to 0.31704 (worst). The difference in RMSE between the three models is also negligible, as it ranges from 0.36646 (best) to 0.39379 (worst). Given that the values that could be predicted range from 1 to 5, these do not seem like bad results at first. However, the Adjusted R^2 paints a different picture: the values are negative for all three models. This suggests that none of these models are a good fit for the data and thus really capable of predicting comment quality. Another factor that may have contributed to these results is the large number of features relative to the number of distinct training examples.

Classifiers are assessed using different methods and were therefore evaluated separately. Table 4.10 shows the results for the classification models, along with results for two popular metrics that are appropriate for models that distinguish between multiple different classes:

Accuracy The accuracy is defined as the proportion of examples for which the predicted class is equal to the actual class of the example. Values are between 0 and 1, higher is better.

Balanced accuracy Accuracy scores can be inaccurate when datasets are very imbalanced, i.e. when some classes appear more often than others. The balanced accuracy score takes any imbalances that may be present in the data into account. The resulting values and interpretation are the same as those for the 'normal' accuracy metric.

Not only do the logistic regressor and decision tree classifier have a similar level of accuracy, the difference between the accuracy and *balanced* accuracy scores is also negligibly small. The support vector machine classifier has an accuracy score of 0.64 and a balanced accuracy score of 0.625, which is slightly higher than those of the other two classifiers. Overall, these results suggest that from these six different models, the support vector machine classifier provides the best results.

4.4.3 Comparing the comment quality predictions of different models

While these accuracy scores seem acceptable at first sight, further analysis shows that these numbers may be misleading. We used each of the six models to predict the comment quality for the 11 software projects that we mined earlier (Table 4.6).

Table 4.10: Performance comparison for classification models

Model	Accuracy	Balanced accuracy
Logistic regressor (classifier)	0.540	0.54167
Decision tree classifier	0.535	0.55833
Support vector machine classifier	0.640	0.62500

Table 4.11: Summary of predictions of comment quality

Model	Min	Max	μ	σ
Linear regressor	2	5	4.1228	0.5249
Decision tree regressor	2	4	3.8026	0.4146
Support vector machine regressor	2	4	3.0568	0.3071
Logistic regressor (classifier)	2	4	3.2817	0.5102
Decision tree classifier	2	4	3.7645	0.4302
Support vector machine classifier	2	4	2.9589	0.5702
All models	2.3	4.2	3.5032	0.4595

A high level summary of the predictions that are made by the models can be seen in Table 4.11. Predictions from regression models have been rounded to the nearest integer.

Even though predictions can theoretically be in a range from 1 (worst) to 5 (best), most models make predictions that are in a much narrower range, from 2 to 4. Consequently, the standard deviation of predicted values (σ) is also very low. The support vector machine classifier, which is the best performing model, makes predictions with a standard deviation of 0.5702; the largest of all models. At the same time, it is also the most pessimistic model, as its average prediction value (μ) is only 2.9589, whereas all other models on average predict values that are above 3. Conversely, the linear regressor makes predictions that are relatively optimistic. It is also the only model that is capable of predicting the maximum value, 5. The six models do not always produce the same or even similar predictions. For instance, out of 8,800 comments there are 1,041 for which the linear regressor model predicted a 5, the highest possible rating, and the support vector machine classifier predicted a 2, the lowest rating produced by any of our models.

We can also compute the mean for comment quality predictions in a different way. Table 4.12 shows the distribution of the overall quality ratings when we combine the outputs of the six models by computing the mean value of their predictions. In some cases this results in quality ratings that are very low or very high, although most comments end up receiving a comment quality rating of 3.5 on a scale from 1 to 5. This table also shows that the standard deviation is highest for scores in the middle of the range (2.8–3.7), which suggests that these scores are at least partially the result of 'disagreement' among the models. The only exception within this range is formed by comments with a mean comment quality rating

Table 4.12: Aggregated predictions of comment quality, from best to worst

Mean quality rating	Count	σ
4.2	29	0.4082
4.0	382	0.1788
3.8	641	0.4572
3.7	1655	0.6655
3.5	3855	0.5996
3.3	559	0.7203
3.2	1112	0.7117
3.0	367	0.2367
2.8	39	0.7677
2.7	79	0.5696
2.5	68	0.5520
2.3	14	0.5164
Total	8800	0.5894

of 3.0, although it is not clear why.

4.4.4 What predictions look like in practice

Another intuitive way to informally evaluate the quality of these predictions is by examining some comments that are allegedly very high- or low-quality.

As shown in Table 4.12, our dataset includes 29 comments that have received the highest possible mean average rating (4.2). Assuming that the models are trustworthy, this would imply that these comments are indisputably high-quality. A manual inspection of these comments suggests that this is indeed the case. What many of the comments seem to have in common is that they have non-trivial descriptions in the comment text, describe the input parameters, return values, and possibly other attributes that may be relevant for the reader.

Listings 4.1 and 4.2 show two examples of such high-quality comments which appear to meet all four quality criteria defined by Steidl et al. [69], i.e. coherence, usefulness, completeness, and consistency: the comments clearly describe the code beyond what can be deduced directly from the code itself and document all aspects of their methods, all in English whilst following Javadoc conventions.

When we look for comments that have received the lowest quality ratings, we see that there are 14 comments with an average rating of 2.3 across all models. Note that these ratings are obtained by four models that predicted a 2, while two models predicted a value of 3. This could suggest that these comments are not necessarily very bad comments, even though they received the lowest ratings in our dataset. However, when we inspect the comments with these low ratings, it is clear that they are all comments that do not contain any meaningful information. Two examples of such comments are shown in Listings 4.3 and 4.4.

The accuracy of predictions is less clear between these two extremes. For instance, the comments in Listings 4.5 and 4.6 both received an average quality rating of 3.5, even though one is clearly higher-quality than the other.

Listing 4.1: A high-quality rating that is likely correct

```
/**
 * Encapsulate the acquisition of the JWT token from HTTP cookies within
     ↳ the
 * request.
 *
 * @param req servlet request to get the JWT token from
 * @return serialized JWT token
 */
protected String getJWTFromCookie(HttpServletRequest req) {
  String serializedJWT = null;
  Cookie[] cookies = req.getCookies();
  if (cookies != null) {
      for (Cookie cookie : cookies) {
          if (cookieName.equals(cookie.getName())) {
              LOG.info(cookieName
              + " cookie has been found and is being processed");
              serializedJWT = cookie.getValue();
              break;
          }
      }
  }
  return serializedJWT;
}
```

Listing 4.2: A high-quality rating that is likely correct

```
/**
 * Gets the next page that should appear in the wizard.  This takes
     ↳ into account the selected
 * project type, project nature, and toolchains.
 *
 * @param currentPageID - The unique ID of the page the wizard is
     ↳ currently displaying.
 * @return The next page that should be displayed in the wizard, or
     ↳ null if at the end of the wizard.
 * @since 3.0
 */
public static IWizardPage getNextPage(String currentPageID)
{
    // find the current page in the set of pages
    MBSCustomPageData pageData = getPageData(currentPageID);
    Iterator iterator = pageSet.iterator();
    while (iterator.hasNext())
    {
        if (pageData.equals(iterator.next())) {
            IWizardPage nextPage = null;
            while (iterator.hasNext() && nextPage == null)
            {
                MBSCustomPageData potentialPage = (MBSCustomPageData)
                    ↳ iterator.next();
                if (isPageVisible(potentialPage.getID()))
                nextPage = potentialPage.getWizardPage();
            }
            return nextPage;
        }
    }
    return null;
}
```

Listing 4.3: A low-quality rating that is likely correct

```
/**
 * @param string
 */
private void log(String string) {
    if (verbose)
    System.err.println(string);
}
```

Listing 4.4: A low-quality rating that is likely correct

```
/**
 * @param operation
 * @return
 */
private boolean isDetach(String operation) {
    return (operation.startsWith("det")  && "detach".indexOf(operation)
        ↳  != -1); //$NON-NLS-1$ //$NON-NLS-2$
}
```

Listing 4.5: A high-quality comment that received an acceptable rating

```
/**
 * Assigns sort modes
 * A value from 0 to 3 defines sort mode as name/last modified/size/
    ↳ type in ascending order
 * Values from 4 to 7 defines sort mode as name/last modified/size/type
    ↳  in descending order
 * <p>
 * Final value of {@link #sortby} varies from 0 to 3
 */
public void getSortModes() {
    int t = Integer.parseInt(sharedPref.getString("sortby", "0"));
    if (t <= 3) {
        sortby = t;
        asc = 1;
    } else if (t > 3) {
        asc = -1;
        sortby = t - 4;
    }

    dsort = Integer.parseInt(sharedPref.getString(PreferencesConstants.
        ↳ PREFERENCE_DIRECTORY_SORT_MODE, "0"));
}
```

Listing 4.6: A comment that repeats the code yet still received a high rating

```
/**
 * Sets the name.
 *
 * @param name
 *            <code>IASTName</code>
 */
public void setName(IASTName name);
```

4.4.5 Effect of features on comment quality

The models that we constructed can use a small set of features to predict the quality of individual comments, to a certain extent. Many contemporary types of models are essentially black boxes: their performance can be evaluated, but their inner workings are unclear. But for some of the models that we included in our comparison it is relatively easy to examine what the fitted model looks like on the inside.

Table 4.13 shows the coefficients for a linear regressor model that has been trained on the complete dataset. Given the lacklustre performance metrics for our linear regressor model these coefficients should clearly be taken with a grain of salt. However, they still provide some information about the relative importance of each feature. There are several things that stand out in this table:

- `cosine_similarity` (B.3.2) has the largest absolute coefficient value, which is almost thrice as much as that of the next coefficient. This is not surprising, especially considering the importance of coherence for comments [69] and the fact that code-comment similarity can be computed for all comments.
- The next three features in the list, `hyperlinks` (B.3.43), `tasks` (B.3.22) and `abbreviations` (B.3.45), on the other hand have in common that they can only be found in a small proportion of comments. This naturally limits the overall effect that they can have on the quality of a comment.
- Many of the remaining features are either related to characteristics of the code that is described by the comment or the amount of information presented in the comment itself. Their effect on comment quality is relatively small, possibly because these do not make or break a comment, but merely make it slightly more or less helpful.

4.4.6 Conclusion

After computing features for a large number of comments that we extracted from a representative sample of open-source Java projects (section 4.2) and eliciting perceived helpfulness ratings from a small subset of those comments (section 4.3), we obtained the necessary ingredients to construct a predictive model for comment quality.

In this section we have tried to answer our third research question, 'How do comment features affect developers' perceived comprehension of Java code?', and main research question, 'How well do features of source code comments predict comment quality in Java code?'. Figure 4.10 shows the relationships between individual comment features (as computed using our repository mining tool in section 4.2) and perceived helpfulness (as rated by our respondents). We have used this data to train six machine learning models, of which three are regressors and three are classifiers. Although the models appear to be capable of correctly classifying some examples of comments that are clearly good or bad, there is still much room for improvement when it comes to accuracy and range of predicted values.

Table 4.13: Coefficients for the linear regressor model, ordered by absolute effect size

Feature	Coefficient
cosine_similarity	2.00765
hyperlinks	0.70491
tasks	-0.28479
abbreviations	0.11193
sentences	0.10017
loop_structures	0.09770
math_symbols	-0.08359
flesch_kincaid	0.05315
try_catch_blocks	0.05153
extra_info_score	-0.05042
variable_assignments	0.04321
local_identifiers	0.02337
control_structures	0.01771
variable_declaration	-0.01287

Chapter 5

Discussion

In this study we have explored the various ways and places in which comments are written and contribute to program comprehension. As discussed earlier in chapter 2, some parts of our research are entirely novel, while others have already been studied in the past. In the latter case, our contributions primarily serve as a partial replication of those earlier studies.

5.1 Literature review

We started our study with a literature review, in order to answer our first research question, 'What features of source code comments can be derived from literature?'. We identified 22 features, which can be found in Table 4.3 and are described in more detail in section 4.1.

Many of the identified features were originally described for different purposes and from different perspectives. Consequently, sometimes different authors refer to separate concepts using a very similar or even the exact same name, or several authors have slightly different interpretations of certain features. We have tried to reconcile such differences, although we should note that this is partially a subjective process: reconciliation requires careful interpretation of feature descriptions within the context of each study, as they cannot always be taken at face value.

We have seen that in some cases opinions differ about the effect of certain features (Table 4.4). However, in some cases different features may also conflict or interact with each other. For instance, it is not possible to both document *all* public methods (*Is complete*; COMPLETE), while at the same time *only* documenting methods that are hard to understand (HARD).

An important reason why these conflicts seem to exist between features is that comments can be read as part of many different software development activities. For instance, a library maintainer might be more interested in information about the implementation of a method, while a user of that same library likely benefits more from a description of what the method does and how it should be used. Our review does not take these developer activities into account yet.

Another possible explanation is that some of the arguments given in favour of certain features are subjective, and may depend on experience with computer programming, a particular language, its culture and ecosystem, a specific project, or personal preferences.

Altogether, based on the potential conflicts between features and the role of subjectiveness it is already likely that a predictive model for comment quality will either need to take all these conflicts and nuances into account or gloss over them. Even when the goal is not to construct a model, but merely analyse specific aspects of comments in a software project, one would have to understand comments from different perspectives.

5.1.1 Comparison with features for existing models

Previous studies about models for the assessment of comment quality in a software project, like Khamis et al. [37] and Steidl et al. [69], showed which features may contribute to comment quality and to what extent. However, they do not explain why these specific features were chosen, which makes it hard to determine how complete these models for comment quality are. Further complicating the matter is the lack of a complete overview of all possible comment features. While some work is currently being done on this subject [59], no results have been published to date.

Our review therefore provides a first insight into the wide array of features that exist for comments, and what portion of those features have been included in existing models for comment quality.

The features that we have identified in our review can be used to design new or extend existing models for comment quality, which implicitly reconcile some of these contradictions and interaction effects.

Schreck et al. [65] presented the Quasoledo tool, which measures the quality of comments in a Java project by computing various metrics related to completeness (*Is complete*; COMPLETE), quantity (*Is succinct*; SUCCINCT), and readability (*Is understandable*; UNDERST). Our overview can therefore be seen as a superset to the set of features included in their tool.

Compared to the set of features that was implemented by Khamis et al. [37], our overview includes a larger number of distinct features. They consider two types of feature categories: internal features and 'code vs. comment' features. Internal features are roughly similar to *Is understandable* (UNDERST), although Khamis et al. focus more on general readability and writing style. 'Code vs. comment' can be seen as a combination of our *Is complete* (COMPLETE), *Is up-to-date* (UPTODATE), and *Provides extra information* (EXTRA) features.

Finally, we have already briefly discussed the quality model used by Steidl et al. [69], which is based on entities, activities, and criteria. Entities are concepts whose quality are being evaluated, i.e. the comments. Activities refer to the intention of the developer, e.g. wanting to understand implementation details while developing code, or being aware of the copyright license when reusing code. The four criteria (coherence, usefulness, completeness, and consistency) apply to entities during specific developer activities. Our review covers all types of comments rather than focussing on specific types of entities, like class comments or inline comments. On the other hand, our review only focusses on features of comments and does not take developer activities into account, as those are beyond the scope of this study.

5.2 Repository mining

Following our literature review, we implemented a repository mining tool that can help us answer our second research question, 'What do features of source code comments look like in open-source Java projects?'. Our literature review had already shown that features can often be measured in different ways. For instance, if we want to know if a comment is placed above code that is hard to understand, we could compute its cyclomatic complexity or its length in number of lines. Our tool therefore provides multiple implementations for most features. Table 4.5 lists all the features for which we have provided implementations in the form of metrics. More detailed explanations of these metrics can be found in appendix B.3.

At the same time, our tool does not implement all of the features that we identified in our literature review. Some features cannot be measured automatically, most notably *Describes purpose* (PURPOSE), *Is up-to-date* (UPTODATE), *Is correct* (CORRECT), and *Describes design decision* (DECISION). The results therefore paint an incomplete picture. However, because the tool still covers most of the features, the results should still provide a decent overview of the characteristics of comments and the code that they document in our sample of projects.

5.2.1 Overall statistics

When it comes to the interpretation and implementation of features, we already discussed many of the possible considerations in section 4.1. Our review showed what features look like in theory. Comments in practice often conform well to guidelines that are described in scientific literature, but this is not always the case.

Coherence

A significant point of contention is what the main purpose of comments is: should they describe what the code does, should they also provide information about the context of the code, or should they *only* provide information about the context?

Several metrics measure the similarity between the comment and the commented code. On average, comments are at least somewhat similar to their commented entities. When computed as an `overlap_percentage` (B.3.1) the average similarity is 31%, whereas the average `cosine_similarity` (B.3.2) is lower, at an equally acceptable 23%. At 36% the level of similarity is higher when we only look at similarities between the comment text and the entity *name* (`method_name_similarity` (B.3.23)), but still well below the maximum threshold of 50% as determined by Steidl et al. [69]. The computed values for the `extra_info_score` (B.3.24) feature suggest that most comments tend to be long enough to have the ability to provide additional information beyond what can be deduced from entity name.

Overall, based on the length of comments and the amount of difference between the comments and the code it describes, it appears that comments in practice generally do not simply repeat identifier names, but also include extra information beyond what can be deduced directly from the source code.

Comment texts

The relative succinctness of comments clearly affects their readability scores. Most readability formulas (Flesch-Kincaid, Gunning fog index, SMOG index, and the

automated readability index) estimate that the texts in comments are understandable for high school students. Given that most software developers have received a bachelor's degree or higher [55] this may suggest that on average comments are written such that they are easy to understand.

The only readability formula that gives a different result is the Flesch reading ease score, as it suggests that comment texts are primarily suitable for college graduates. This is somewhat unexpected, as are the minimum and maximum values for some the readability scores. These aberrant results are likely due to the fact that comment texts are not entirely representative of most English texts. Readability formulas are designed and optimised for regular texts, like newspaper articles. Moreover, some formulas are intended to be computed on texts with specific (minimum) lengths [76]. Be that as it may, when taken as a whole the computed readability scores appear to reflect the actual readability of comment texts.

Finally, we found that comment texts rarely include math symbols, hyperlinks, and tasks (i.e. todo comments) in our sampled projects. Sentences are however terminated with full stops in only half of all comments. While the absence of full stops at the end of sentences is only one of many signals that comment texts use a sublanguage, it is a pretty clear one. Issue numbers do not seem to appear in our sample at all. On the other hand it appears that abbreviations are fairly common in comments.

5.2.2 Differences between projects

In section 4.2.3 we already showed that differences exist between projects. There are several aspects that stand out: the placement of comments, indications of the code quality of a project, and the readability of comment texts.

Placement of comments

One of the most noticeable differences is placement. We saw earlier that comments are primarily placed above methods, classes and attributes, but rarely above constructors, packages, enumerations, annotations, and interfaces.

The project-specific tables (Table D.1–D.11) show that not only are these latter entities less commented on – in many projects they are not commented on *at all*. This is especially the case for AFWall+, Amaze File Manager, AntennaPod, ownCloud, and WordPress, which all have in common that they contain source code for end-user Android applications. Conversely, in the library and framework projects in our sample (particularly Apache Spark, Google Guava, and Google Guice) those same entities tend to be commented more thoroughly, presumably to facilitate reuse by developers that incorporate the library or framework in their own application. However, exceptions exist. For instance, enumerations are heavily commented in Amaze File Manager and WordPress, while attributes are commented often in AFWall+.

Code quality

Although our study focusses on comment quality rather than code quality, some comment features are related to code quality: code that is hard to understand may be more likely to warrant a (good) comment (*Is for complex code*; HARD).

Although it is not possible to test that assumption from this data alone, we do see signs that code quality differs between projects. The cleanest project in our sample is likely Google Guice (Table D.8), which contains a below-average number of try-catch blocks, operators, variable assignments, variable declarations and method calls, and has shorter methods. We might consider Amaze File Manager (Table D.2) to be its polar opposite in our sample, as its methods tend to be longer and are more complex. It is possible that software that is explicitly intended for reuse by other developers, like libraries and (to a lesser extent) frameworks, must have higher-quality code than client applications whose source code is only seen by their maintainers.

Readability

Until now we have only focussed on features that appear in the project-specific tables, but there are also features that are largely conspicuous in their absence.

Given that each projects has different types of requirements and thus may also attract certain types of developers, one might expect that comments in some projects are more readable than in others. However, aside from the Flesch reading ease feature no other features based on readability formulas appear in any of the project-specific tables.

This does not mean that differences do not exist, merely that they are not large enough to be included in them. This is logical, as Table 4.7 has shown that virtually all readability scores have fairly small standard deviations. Moreover, readability scores are supposed to work on a fixed scale, where a 50% difference in any direction is massive.

Considering that most readability formulas do not show up in any of these tables and the minimum and maximum values of `flesch_ease` (B.3.3) seem somewhat spurious, we might conclude that perhaps our implementation of the Flesch reading ease score is unsuitable for comments.

5.3 Perceived comment quality

After having computed the features for the most important comments in a sample of Java projects, we distributed a survey among people with programming experience, in which we elicited helpfulness or 'quality' ratings for some of those comments. These ratings, combined with the features that were originally computed for the comments, can be fed into a machine learning model for comment quality.

Machine learning models are generally trained using a large number of distinct training examples. Examples can be mined from historical data or created from scratch by having annotators make predictions manually. In order to capture as much of the diversity in features that occurs in real-life examples, the number of distinct examples to be annotated are typically very large. To reduce the impact of subjectivity, each example is often annotated by several annotators. The overall expected prediction for each example can be obtained by averaging annotations or using a voting system, where the most commonly annotated prediction 'wins'.

For our study, we took a slightly different approach due to practical limitations. We used a rather limited number of examples, 44, which were annotated by 52 respondents. As we will see later in section 5.4, such an approach is far from ideal for training predictive models.

However, our approach does provide additional insights that could not have been obtained using a more traditional approach to gathering annotations. Figure 4.7 showed that there is almost universal disagreement about which comments should be seen as helpful.

As discussed earlier, a possible explanation for this could be that in our survey we did not take the developer's personal preferences and current software development activity into account, i.e. the context in which the comment is used. Taking all of these variables into account would likely require a higher number of respondents and the use of multiple surveys that are tailored for specific activities, with snippets that are appropriate within the context of the activity.

5.3.1 Resolving disagreements

The large amount of disagreement in our survey responses makes for a somewhat noisy ground truth, which in turn leads to models that have to deal with a large degree of ambiguity.

There are two commonly used solutions to resolve such disagreements. Both work by combining or summarising annotations in order to get rid of noise. However, neither solution is ideal, as both solutions also discard valuable information about the extent of disagreement about the perceived helpfulness of comments.

Averaging annotations is problematic, primarily because Likert scales are not strictly interval scales, which means that arithmetic operations (like averaging) could be seen as methodically unsound. Secondly, since virtually all comments have their proponents and opponents, the average annotation value rarely corresponds with 'strongly agree' or 'strongly disagree', i.e. clear signals of comment quality. Using some kind of voting mechanism makes it more likely that the combined annotation value is a clear signal (e.g. for C.33) that can be used to train decisive models.

To stay on the safe side, we ultimately chose to combine annotations by coding the Likert-scale responses as a scale from 1 to 5 and computing the mean value. We acknowledge that this method has its drawbacks, as it causes our machine learning models to make predictions that gravitate towards the mean (section 4.4.3). However, we also need to point out that its drawbacks would not have been known, had we only obtained a very low number of annotations per code snippet. Moreover, many studies have shown that even though Likert scales are not strictly interval scales, treating them as such still results in meaningful findings [81].

5.4 A model for comment quality

Once we obtained the necessary ingredients for our training data – a set of comment features and corresponding helpfulness ratings – we constructed and evaluated six predictive models for comment quality, of which three were based on regression and three on classification.

Neither is entirely ideal for our use case. Regression is primarily intended for predictions on an interval scales, which five-point Likert scales technically are not. Classification models work well for predictions of categorical variables which have no real relationship to each other, even though one clearly exists, e.g. 'strongly agree' and 'agree' are more related to each other than 'strongly agree' and 'strongly disagree'. Nevertheless, such models can still produce useful results.

5.4.1 Model performance

Our predictive models for comment quality do not perform as well as we had hoped, which makes it harder to answer our third research question, 'How do comment features affect developers' perceived comprehension of Java code?'.

Table 4.13 lists the features and their coefficients for the linear regression model that we constructed. Although this table provides some high-level insight into what features are likely important, the low accuracy of the model means that we cannot clearly deduce how each feature exactly affects how well a comment helps developers understand code.

This was to be expected to some degree. Section 5.3 already discussed the lack of universal agreement about the helpfulness of comments, which makes it harder to produce well-performing models.

What also does not work in our favour, is the low number of training examples. Because we computed the mean of all annotations per snippet, this means we train our models with only 44 examples. Each example comes with a list of features that – while much shorter than the full list of computed features (Table 4.8) – is quite long relative to the limited number of unique training examples. Consequently, trained models are more prone to overfit to the specific examples in our dataset and thus become less capable of making accurate predictions for comments not within our sample.

5.4.2 Improving the quality of predictions

There are several ways to improve the quality of predictions. We previously already discussed the inclusion of features, like *Is correct* (CORRECT), that were currently left out. Another very obvious solution is to gather more annotations for a larger set of snippets, as this increases the size of the training dataset and thus the diversity and representativeness of training examples. A combination of technical, time, and budgetary constraints made this an unviable solution for our study: The LimeSurvey tool in our study works well for traditional surveys, but is cumbersome to use when large numbers of code snippets need to be formatted. Moreover, our current method of survey distribution is costly and labour-intensive, especially since we cannot promote the survey via traditional channels, e.g. survey panels or by sharing direct links to the online survey.

Additionally, there are other machine learning techniques that may achieve better results. A well-known example is deep learning, where learning takes place in a manner not entirely dissimilar from how networks of neurons function in the human brain [62]. While deep learning does not appear to have been used yet for prediction of comment quality, the technique has been applied within the context of source code comments. For instance, Hu et al. [33] designed a model that can predict natural language descriptions for code snippets that lack a comment.

Existing machine learning models can also be combined. We already provided one method of doing so in section 4.4.3, but there are many others, which are all known under the term 'ensemble learning' [63]. Ensemble learning works in a manner akin to wisdom of the crowds; a single model can be flawed, e.g. be prone to biases. However, when the outputs of several independent predictive models are combined, each model can compensate for mistakes and biases from other models and thus achieve better results than any of the individual models would have on its own.

5.5 Limitations

In this section we discuss possible limitations of our study. We also describe what measures we have taken to mitigate them – or in cases where that is not possible, how they may impact our conclusions.

5.5.1 Literature review on comments in source code

Our systematic literature review covers a wide array of publications about comments in source code from a long period that ranges from 1981 to 2018. Nevertheless, there are several limitations.

Coverage of relevant publications

The keywords in our search query were chosen based on a preliminary, informally conducted literature review. It is possible that a different set of keyword (e.g. some studies refer to comments as 'summaries') would have yielded a result that provides different insights into what makes good comments. We believe that our use of snowballing has sufficiently mitigated this threat, as it has helped us discover papers that were not covered by the original search query. The large number of discarded and still to be annotated papers suggests that our query may even have been a bit too broad.

Given the large amount of results that our search query yielded in the four databases, it was necessary to choose a subset that could be feasible annotated within the time allocated for our review. Since we chose to annotate papers with a high number of citations first, we believe our review covers the most important and impactful findings in the field.

Generalisability

One possible issue that remains is that most of the papers – and hence findings – in our review apply to C and C-like programming languages (e.g. C++ and Java). Our findings could therefore be less applicable to comments in low-level assembly languages, declarative languages (e.g. SQL) or functional languages (e.g. Haskell, Idris, and Clean). However, since most software is nowadays written using C-like programming languages this should not pose any problems in practice. According to the TIOBE index[1], many of the most popular programming languages are C-like. This suggests that our findings apply to most code that is in use and maintained today.

Annotator fatigue

Finally, our literature review was conducted using only a single annotator. This may have inadvertently introduced biases in determining what does and what does not qualify as a feature. Moreover, the number of annotated papers per annotator is relatively large. This could have resulted in annotator fatigue, and consequently lower-quality annotations [38]. We have attempted to mitigate these effects by only annotating a limited number of papers per day and annotating papers using multiple passes. This lets us partially emulate the benefits of annotating with a fresh pair of eyes.

[1] https://www.tiobe.com/tiobe-index/

5.5.2 Comment features in practice

In the second phase of our study, we computed the features for a large number of comments from several real-world software projects in order to learn more about comment features in general.

Choice of projects

Mining features is computationally expensive. We have therefore chosen to limit the scope of our study to comments in a sample of open-source software (OSS) projects. Of course, OSS projects may have characteristics that differ from those of closed-source, commercial projects. In practice the line between traditional OSS and commercially-developed software is somewhat vague, as a large share of contributions to OSS is actually made by employees at commercial organisations [82]. Instead, it is more important to ensure that comments are sampled from a diverse range of projects [50]. While we acknowledge that we could have analysed a larger set of projects, we believe that the projects in our sample are sufficiently heterogeneous and thus representative for most real-world projects.

Choice of features

Aside from our choice of projects, one might also question our choice of computed comment features. Our literature review yielded 22 features, of which we have only implemented a subset due to feasibility issues and practical considerations. For instance, the correctness of a natural-language comment cannot be automatically measured in a reliable way. Our analysis therefore paints an incomplete picture of what comments look like in practice. Regardless, what remains is still a fairly comprehensive set of features that provides a wealth of information about the state of comments in a software project.

Choice of interpretations

Moreover, in some cases we have implemented different interpretations of certain features, e.g. the readability of a comment text. However, there are also features where many different interpretations exist and for which we have provided only one implementation, e.g. merely counting the number of *physical* lines of code. It is infeasible to implement all possible interpretations of features, nor do we believe it is necessary to do so: while certain features can be interpreted using different methods, each of those methods aims to measure the same thing. Any differences that arise from the choice of a specific interpretation are therefore minor and only noticeable when different interpretations are compared directly between each other.

5.5.3 Measuring the perceived helpfulness of comments

In the third phase of our study, we showed a selection of comments from real-world projects to respondents and asked them to indicate to what extent they believed that the comments improved their understanding of the code.

Chosen snippets

The set of comments that we included in our survey is very small relative to the complete set of mined comments, let alone the set of *all* comments that have been written for software projects. Given the large variety in comments, it is logical that many types, shapes, and forms of comments did not appear in our survey in any way. We have tried to select comments that are representative of the set as a whole and have features that appear relatively often. Our results therefore still apply to types, shapes, and forms of comments that appear frequently in software projects.

Measurement of perceived helpfulness

Our survey is essentially designed to measure the *perceived* effect of comments on comprehension. Some studies [3] show that perceived comprehensibility might not always coincide with actual comprehensibility. We could have administered tests consisting of 'fill in the blanks' cloze tests or multiple choice questions to measure the *actual* comprehensibility, but such an approach is not without its problems: Firstly, badly worded questions may strongly affect comprehensibility measurements. Secondly, such an approach is much more time-consuming. Börstler and Paech [3] for example only manage to include six snippets in their survey. This did not suffice for our study, as we needed data for a larger number of comments.

Another possible issue that may occur with surveys is social desirability bias, the tendency for respondents to answer questions in a way that they believe would put them in a more positive light with the person surveying them [6]. For example, if a respondent is asked to rate a particular code snippet, they might rate it more positively in an attempt to appear more intelligent or to please an interviewer. We attempted to mitigate this bias by constantly reminding respondents that the purpose of the survey is to measure the quality of *comments* rather than the ability of the respondents to understand comments and that responses are anonymous. This should have encouraged respondents to provide more honest answers, but we have no way to confirm whether our reminder has had the intended effect.

Non-response bias

Participation in the survey was voluntary. This may have caused participation bias (alternatively: *non-response bias*): respondents that are more inclined to complete surveys might not be representative of the developer population as a whole. We have tried to mitigate this risk by actively promoting the survey among members of demographic groups that would not have participated otherwise. Nevertheless, the average level of education of our respondents is relatively high, particularly compared to benchmarks like Stack Overflow's annual developer surveys [55]. This suggests that some degree of non-response bias is present.

5.5.4 Construction of models for comment quality

We combined our computed comment features with responses from our survey to create a set of training data, which we then used to train several machine learning models.

The largest limitation of our study is the limited size of the training dataset, which contains a relatively large number of features and a modest number of com-

ments that models can learn from. Ideally, we would have been able to collect data about a larger variety of comments using our survey. Each comment would then only require a few ratings per snippet, as is common in studies that are based on machine learning techniques. However, due to ethical considerations we were severely limited in our choice of tools and platforms via which we could distribute our survey, which made such an approach infeasible. Moreover, distribution of a second survey with these limitations in place would have been prohibitively expensive.

5.6 Future work

In this study we have identified features of comments in source code that have been described in scientific literature. We used machine learning to study the effect of some of these features, with limited success.

One of the main limitations of our study is the relatively modest size of our training dataset. A large-scale follow-up study is needed to further explore the relation between each feature, and possible interaction effects between features on the perceived quality of a comment.

Another limitation of our study is that not all features could be feasibly incorporated into our comment feature mining tool. Since some of these features are mentioned often in scientific literature on comments in source code, their exclusion from models for comment quality likely has a detrimental effect on their predictive power. Further research is therefore needed to determine how these features can be measured reliably, possibly by developing heuristics.

Despite its current limitations, our study has yielded several insights into comment quality. For instance, our repository mining revealed that quality differences exist between projects. A future study could explore the relationship between code quality and comment quality, and comment quality and certain project characteristics, like project type (framework, library or end-user application), by mining a larger and even more diverse sample of software projects.

Moreover, our findings about the features, and their reported and confirmed effects can be easily reworded into recommendations for writing comments in source code. It would be interesting to see how these recommendations relate to those from the professional handbooks listed in chapter 1, as many software developers base their practices on the guidelines that are described in professional literature.

Chapter 6

Conclusion

Although guidelines exist for writing good comments, they are not always validated empirically. Moreover, software developers do not have access to automated tooling that allows them to evaluate the quality of comments in the source code of their software project. The objective of our study therefore was to gain better insight into the features that are associated with good comments.

We started our study by conducting a systematic literature review in which we derived features of source code comments from scientific literature. Our review has yielded a comprehensive overview of 22 features (Table 4.3), along with their definitions, number of mentions, purported effect on code comprehension, rationale, and information about the extent at which they have been validated (section 4.1).

These findings from our literature review were confirmed using repository mining. More specifically, we analysed the presence and values of features of source code comments (Table 4.5) that occur in a representative sample of open-source Java projects (Table 4.6). Our results show that comments are placed primarily above methods and tend to be fairly succinct, while still being long enough to provide information beyond what can be deduced from the name of commented entities (section 4.2). Comments are not the same in every project however: libraries and frameworks appear to be better commented and have higher-quality code than end-user applications (section 4.2.3).

To study the effect that each of these comment features has on the perceived comprehension of Java code, we conducted a survey among software developers. In this survey, respondents were shown a randomised sequence of 44 snippets with a comment and asked to what extent that comment improved their understanding of the accompanying code. Our survey received 220 responses in total, of which 52 were usable for further analysis (section 4.3). The results suggest that for most comments a large amount of disagreement exists about whether they are thought to have a beneficial effect on comprehension (Figure 4.7). Moreover, it appears that respondents' demographic backgrounds have little effect on their perception of the helpfulness of comments in source code.

The insights that we derived from our literature review, repository mining, and survey on comments in source code finally led to the development of several predictive models that attempt to assess the quality of comments in a software project. Although the models are capable of predicting comment quality to some extent, further research is needed to improve their performance (section 4.4).

Bibliography

[1] Paul D Allison. *Multiple regression: A primer*. Pine Forge Press, 1999.

[2] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25 (12):1226–1241, 1976. ISSN 0018-9340. doi: 10.1109/TC.1976.1674590. URL https://doi.org/10.1109/TC.1976.1674590.

[3] Jürgen Börstler and Barbara Paech. The role of method chains and comments in software readability and comprehension – an experiment. *IEEE Transactions on Software Engineering*, 42(9):886–898, 2016. doi: 10.1109/TSE.2016.2527791. URL https://doi.org/10.1109/TSE.2016.2527791.

[4] Bruce L Bowerman and Richard T O'Connell. *Linear statistical models: An applied approach*. Brooks/Cole, 1990.

[5] Frederick P. Brooks. *The mythical man-month – Essays on software engineering, 2nd edition*. Addison-Wesley, 1995. ISBN 978-0-201-83595-3.

[6] Alan Bryman. *Social research methods*. Oxford University Press, 4rd ed. edition, 2012.

[7] Raymond P. L. Buse and Westley Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010. doi: 10.1109/TSE.2009.70. URL https://doi.org/10.1109/TSE.2009.70.

[8] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58, 2018.

[9] Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. Automatically detecting the scopes of source code comments. *Journal of Systems and Software*, 153:45–63, 2019. doi: 10.1016/j.jss.2019.03.010. URL https://doi.org/10.1016/j.jss.2019.03.010.

[10] Scott Clifford and Jennifer Jerit. Cheating on political knowledge questions in online surveys: An assessment of the problem and solutions. *Public Opinion Quarterly*, 80(4):858–887, 2016.

[11] Don M. Coleman, Dan Ash, Bruce Lowther, and Paul W. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994. doi: 10.1109/2.303623. URL https://doi.org/10.1109/2.303623.

[12] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. Coherence of comments and method implementations: a dataset and an empirical investigation. *Software Quality Journal*, 26(2):751–777, 2018. doi: 10.1007/s11219-016-9347-1. URL https://doi.org/10.1007/s11219-016-9347-1.

[13] Scott A Crossley, David B Allen, and Danielle S McNamara. Text readability and intuitive simplification: A comparison of readability formulas. *Reading in a Foreign Language*, 23(1):84–101, 2011.

[14] Fabio Q. B. da Silva, Andre Luís de Medeiros Santos, Sérgio Soares, A. César C. França, Cleviton V. F. Monteiro, and Felipe Farias Maciel. Six years of systematic literature reviews in software engineering: An updated tertiary study. *Information and Software Technology*, 53(9):899–913, 2011. doi: 10.1016/j.infsof.2011.04.004. URL https://doi.org/10.1016/j.infsof.2011.04.004.

[15] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia Marçal de Oliveira. Which documentation for software maintenance? *Journal of the Brazilian Computer Society*, 12(3):31–44, 2006. doi: 10.1590/S0104-65002006000400004. URL https://doi.org/10.1590/S0104-65002006000400004.

[16] Michael L. Van de Vanter. The documentary structure of source code. *Inf. Softw. Technol.*, 44(13):767–782, 2002. doi: 10.1016/S0950-5849(02)00103-9. URL https://doi.org/10.1016/S0950-5849(02)00103-9.

[17] Whitney DeCamp and Matthew J Manierre. "money will solve the problem": Testing the effectiveness of conditional incentives for online surveys. *Survey Practice*, 9(1):2823, 2016.

[18] Lionel E. Deimel. The uses of program reading. *SIGCSE Bulletin*, 17(2):5–14, 1985. ISSN 0097-8418. doi: 10.1145/382204.382524. URL https://doi.org/10.1145/382204.382524.

[19] Don A Dillman, Jolene D Smyth, and Leah Melani Christian. *Internet, phone, mail, and mixed-mode surveys: the tailored design method*. John Wiley & Sons, 2014.

[20] Letha H. Etzkorn, Lisa L. Bowen, and Carl G. Davis. An approach to program understanding by natural language understanding. *Nat. Lang. Eng.*, 5(3):219–236, 1999. URL http://journals.cambridge.org/action/displayAbstract?aid=48489.

[21] Onyeka Ezenwoye. What language? - the choice of an introductory programming language. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–8, 2018. doi: 10.1109/FIE.2018.8658592.

[22] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring programming experience. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 73–82, 2012. doi: 10.1109/ICPC.2012.6240511.

[23] Andy Field, Jeremy Miles, and Zoë Field. *Discovering statistics using R*. SAGE publications, 2012.

[24] John Fox and Sanford Weisberg. *An R companion to applied regression*. SAGE publications, 2018.

[25] Robert B. Grady. Successfully applying software metrics. *Computer*, 27(9): 18–25, 1994.

[26] Maurice Howard Halstead. *Elements of software science*, volume 7. Elsevier New York, 1977.

[27] Dorsaf Haouari, Houari A. Sahraoui, and Philippe Langlais. How good is your comment? a study of comments in Java programs. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, Banff, AB, Canada, September 22–23, 2011*, pages 137–146. IEEE Computer Society, 2011. doi: 10.1109/ESEM.2011.22. URL https://doi.org/10.1109/ESEM.2011.22.

[28] Jan Hartmann, Alistair G. Sutcliffe, and Antonella De Angeli. Towards a theory of user judgment of aesthetics and user interface quality. *ACM Transactions on Computer-Human Interaction*, 15(4):15:1–15:30, 2008. doi: 10.1145/1460355.1460357. URL https://doi.org/10.1145/1460355.1460357.

[29] Hao He. Understanding source code comments at large-scale. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 1217–1219. ACM, 2019. doi: 10.1145/3338906.3342494. URL https://doi.org/10.1145/3338906.3342494.

[30] Andrew Head, Caitlin Sadowski, Emerson R. Murphy-Hill, and Andrea Knight. When not to comment: questions and tradeoffs with API documentation for C++ projects. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 – June 3, 2018*, pages 643–653. ACM, 2018. doi: 10.1145/3180155.3180176. URL https://doi.org/10.1145/3180155.3180176.

[31] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In Ricardo Jorge Machado, Fernando Brito e Abreu, and Paulo Rupino da Cunha, editors, *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12–14, 2007, Proceedings*, pages 30–39. IEEE Computer Society, 2007. doi: 10.1109/QUATIC.2007.8. URL https://doi.org/10.1109/QUATIC.2007.8.

[32] Arto Hellas, Albina Zavgorodniaia, and Juha Sorva. Crowdsourcing in computing education research: Case amazon mturk. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, Koli Calling '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450389211. doi: 10.1145/3428029.3428062. URL https://doi.org/10.1145/3428029.3428062.

[33] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In Foutse Khomh, Chanchal K. Roy, and Janet Siegmund, editors, *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27–28, 2018*, pages 200–210. ACM, 2018. doi: 10.1145/3196321.3196334. URL https://doi.org/10.1145/3196321.3196334.

[34] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., USA, 2000. ISBN 020161622X.

[35] ISO/IEC. Systems and software engineering – Systems and software quality requirements and evaluation (square) – System and software quality models. Technical report, 2011.

[36] ISO/IEC/IEEE. International standard – Systems and software engineering – Vocabulary. Standard, 2017.

[37] Ninus Khamis, Juergen Rilling, and René Witte. Assessing the quality factors found in in-line documentation written in natural language: The JavadocMiner. *Data & Knowledge Engineering*, 87:19–40, 2013. doi: 10.1016/j.datak.2013.02.001. URL https://doi.org/10.1016/j.datak.2013.02.001.

[38] Levi King and Markus Dickinson. Annotating picture description task responses for content analysis. In *Proceedings of the thirteenth workshop on innovative use of NLP for building educational applications*, pages 101–109. Association for Computational Linguistics, 2018.

[39] Barbara A. Kitchenham, Tore Dybå, and Magne Jørgensen. Evidence-based software engineering. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *26th International Conference on Software Engineering (ICSE 2004), 23–28 May 2004, Edinburgh, United Kingdom*, pages 273–281. IEEE Computer Society, 2004. doi: 10.1109/ICSE.2004.1317449. URL https://doi.org/10.1109/ICSE.2004.1317449.

[40] Jürgen Koenemann and Scott P. Robertson. Expert problem solving strategies for program comprehension. In Scott P. Robertson, Gary M. Olson, and Judith S. Olson, editors, *Conference on Human Factors in Computing Systems, CHI 1991, New Orleans, LA, USA, April 27 - May 2, 1991, Proceedings*, pages 125–130. ACM, 1991. doi: 10.1145/108844.108863. URL https://doi.org/10.1145/108844.108863.

[41] 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 – Volume 2*, IJCAI'95, pages 1137—1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1558603638.

[42] Timothy Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6): 35–39, 2003. doi: 10.1109/MS.2003.1241364. URL https://doi.org/10.1109/MS.2003.1241364.

[43] Yangchao Liu, Xiaobing Sun, and Yucong Duan. Analyzing program readability based on wordnet. In Jian Lv, He Jason Zhang, and Muhammad Ali Babar, editors, *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, EASE 2015, Nanjing, China, April 27–29, 2015*, pages 27:1–27:2. ACM, 2015. doi: 10.1145/2745802.2745837. URL https://doi.org/10.1145/2745802.2745837.

[44] Craig Lockwood and Eui Geum Oh. Systematic reviews: Guidelines, tools and checklists for authors. *Nursing & Health Sciences*, 19(3):273–277, 2017.

[45] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[46] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837. URL https://doi.org/10.1109/TSE.1976.233837.

[47] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: a case study of the Qt, VTK, and ITK projects. In Premkumar T. Devanbu, Sung Kim, and Martin Pinzger, editors, *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 – June 1, 2014, Hyderabad, India*, pages 192–201. ACM, 2014. doi: 10.1145/2597073.2597076. URL https://doi.org/10.1145/2597073.2597076.

[48] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2020.

[49] Raymond H Myers and Raymond H Myers. *Classical and modern regression with applications*, volume 2. Duxbury press Belmont, CA, 1990.

[50] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18–26, 2013*, pages 466–476. ACM, 2013. doi: 10.1145/2491411.2491415. URL https://doi.org/10.1145/2491411.2491415.

[51] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A SLOC counting standard. Technical report, COCOMO II Forum 2007, 2007.

[52] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. Commenting source code: is it worth it for small programming tasks? *Empirical Software Engineering*, 24(3):1418–1457, 2019. doi: 10.1007/s10664-018-9664-z. URL https://doi.org/10.1007/s10664-018-9664-z.

[53] Eriko Nurvitadhi, Wing Wah Leung, and Curtis Cook. Do class comments aid Java program understanding? In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 1, pages T3C–13–T3C–17. IEEE, 2003.

[54] Robert M O'brien. A caution regarding rules of thumb for variance inflation factors. *Quality & quantity*, 41(5):673–690, 2007.

[55] Stack Overflow. Stack Overflow Developer Survey 2021, 2021. URL https://insights.stackoverflow.com/survey/2021#developer-profile-education. Accessed: 2021-10-03.

[56] Luca Pascarella. Classifying code comments in Java mobile applications. In Christine Julien, Grace A. Lewis, and Itai Segall, editors, *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27–28, 2018*, pages 39–40. ACM, 2018. doi: 10.1145/3197231.3198444. URL https://doi.org/10.1145/3197231.3198444.

[57] Luca Pascarella and Alberto Bacchelli. Classifying code comments in Java open-source software systems. In Jesús M. González-Barahona, Abram Hindle, and Lin Tan, editors, *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20–28, 2017*, pages 227–237. IEEE Computer Society, 2017. doi: 10.1109/MSR.2017.63. URL https://doi.org/10.1109/MSR.2017.63.

[58] Daryl Posnett, Abram Hindle, and Premkumar T. Devanbu. A simpler model of software readability. In Arie van Deursen, Tao Xie, and Thomas Zimmermann, editors, *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21–28, 2011*, pages 73–82. ACM, 2011. doi: 10.1145/1985441.1985454. URL https://doi.org/10.1145/1985441.1985454.

[59] Pooja Rani. Speculative analysis for quality assessment of code comments. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 299–303. IEEE, 2021.

[60] Darrell R. Raymond. Reading source code. In *Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '91, pages 3–16. IBM Press, 1991.

[61] Paige Rodeghero and Collin McMillan. Detecting important terms in source code for program comprehension. In Tung Bui, editor, *52nd Hawaii International Conference on System Sciences, HICSS 2019, Grand Wailea, Maui, Hawaii, USA, January 8–11, 2019*, pages 1–10. ScholarSpace / AIS Electronic Library (AISeL), 2019. URL http://hdl.handle.net/10125/60186.

[62] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, USA, 2020. ISBN 978-0134610993.

[63] Omer Sagi and Lior Rokach. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(4):e1249, 2018.

[64] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: how far are we? In Grigore Rosu, Massimiliano Di

Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 – November 3, 2017*, pages 417–427. IEEE Computer Society, 2017. doi: 10.1109/ASE.2017.8115654. URL https://doi.org/10.1109/ASE.2017.8115654.

[65] Daniel Schreck, Valentin Dallmeier, and Thomas Zimmermann. How documentation evolves over time. In Massimiliano Di Penta and Michele Lanza, editors, *9th International Workshop on Principles of Software Evolution (IWPSE 2007), in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia, September 3–4, 2007*, pages 4–10. ACM, 2007. doi: 10.1145/1294948.1294952. URL https://doi.org/10.1145/1294948.1294952.

[66] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003. ISBN 0321118847.

[67] Eleanor Singer and Cong Ye. The use and effects of incentives in surveys. *The ANNALS of the American Academy of Political and Social Science*, 645 (1):112–141, 2013. doi: 10.1177/0002716212458082. URL https://doi.org/10.1177/0002716212458082.

[68] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. Towards an empirically validated model for assessment of code quality. In Simon and Päivi Kinnunen, editors, *Proceedings of the 14th Koli Calling International Conference on Computing Education Research, Koli, Finland, November 20–23, 2014*, pages 99–108. ACM, 2014. doi: 10.1145/2674683.2674702. URL https://doi.org/10.1145/2674683.2674702.

[69] Daniela Steidl, Benjamin Hummel, and Elmar Jürgens. Quality analysis of source code comments. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20–21 May, 2013*, pages 83–92. IEEE Computer Society, 2013. doi: 10.1109/ICPC.2013.6613836. URL https://doi.org/10.1109/ICPC.2013.6613836.

[70] Steve McConnell Steve. *Code Complete, Second Edition*. Microsoft Press, USA, 2004. ISBN 0735619670.

[71] Xiaobing Sun, Qiang Geng, David Lo, Yucong Duan, Xiangyue Liu, and Bin Li. Code comment quality analysis and improvement recommendation: An automated approach. *International Journal of Software Engineering and Knowledge Engineering*, 26(6):981–1000, 2016. doi: 10.1142/S0218194016500339. URL https://doi.org/10.1142/S0218194016500339.

[72] Danielle Albers Szafir. The good, the bad, and the biased: Five ways visualizations can mislead (and how to fix them). *Interactions*, 25(4):26–33, 2018.

[73] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: bugs or bad comments? */. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14–17, 2007*, pages 145–158. ACM, 2007. doi: 10.1145/1294261.1294276. URL https://doi.org/10.1145/1294261.1294276.

[74] Ted Tenny. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, 1988. doi: 10.1109/32.6171. URL https://doi.org/10.1109/32.6171.

[75] Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. "automatically assessing code understandability" reanalyzed: combined metrics matter. In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28–29, 2018*, pages 314–318. ACM, 2018. doi: 10.1145/3196398.3196441. URL https://doi.org/10.1145/3196398.3196441.

[76] Philip van Oosten, Dries Tanghe, and Véronique Hoste. Towards an improved methodology for automated readability prediction. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of the International Conference on Language Resources and Evaluation, LREC 2010, 17–23 May 2010, Valletta, Malta*. European Language Resources Association, 2010.

[77] Hans Van Vliet. *Software Engineering: Principles and Practice*. Wiley Publishing, 3rd edition, 2008. ISBN 0470031468.

[78] Michel Verleysen and Damien François. The curse of dimensionality in data mining and time series prediction. In *International Work-Conference on Artificial Neural Networks*, pages 758–770. Springer, 2005.

[79] Philip D Waggoner, Ryan Kennedy, and Scott Clifford. Detecting fraud in online surveys by tracing, scoring, and visualizing ip addresses. *Journal of Open Source Software*, 4(37):1285, 2019.

[80] Eliane Stampfer Wiese, Anna N. Rafferty, and Armando Fox. Linking code readability, structure, and comprehension among novices: it's complicated. In Sarah Beecham and Daniela E. Damian, editors, *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2019, Montreal, QC, Canada, May 25–31, 2019*, pages 84–94. IEEE/ACM, 2019. doi: 10.1109/ICSE-SEET.2019.00017. URL https://doi.org/10.1109/ICSE-SEET.2019.00017.

[81] Huiping Wu and Shing-On Leung. Can Likert scales be treated as interval scales? — A simulation study. *Journal of Social Service Research*, 43(4): 527–532, 2017.

[82] Y. Zhang, M. Zhou, A. Mockus, and Z. Jin. Companies' participation in OSS development – An empirical study of OpenStack. *IEEE Transactions on Software Engineering*, 2019. ISSN 2326-3881. doi: 10.1109/TSE.2019.2946156. Early access.

Appendix A

Systematic literature review

Table 4.3 listed all features that were identified in our literature review, along with the absolute and relative number of distinct references across papers, and references to the papers in which they were mentioned. The corresponding bibliography can be found below.

A.1 List of reviewed papers

[R1] Raymond P. L. Buse and Westley Weimer. A metric for software readability. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 121–130. ACM, 2008. doi: 10.1145/1390630.1390647. URL https://doi.org/10.1145/1390630.1390647.

[R2] Raymond P. L. Buse and Westley Weimer. Learning a metric for code readability. *IEEE Trans. Software Eng.*, 36(4):546–558, 2010. doi: 10.1109/TSE.2009.70. URL https://doi.org/10.1109/TSE.2009.70.

[R3] Michael L. Van de Vanter. The documentary structure of source code. *Inf. Softw. Technol.*, 44(13):767–782, 2002. doi: 10.1016/S0950-5849(02)00103-9. URL https://doi.org/10.1016/S0950-5849(02)00103-9.

[R4] Letha H. Etzkorn, Lisa L. Bowen, and Carl G. Davis. An approach to program understanding by natural language understanding. *Nat. Lang. Eng.*, 5(3):219–236, 1999. URL http://journals.cambridge.org/action/displayAbstract?aid=48489.

[R5] Letha H Etzkorn, Carl G Davis, and Lisa L Bowen. The language of comments in computer software: A sublanguage of english. *Journal of Pragmatics*, 33(11):1731–1756, 2001. ISSN 0378-2166. doi: https://doi.org/10.1016/S0378-2166(00)00068-0. URL http://www.sciencedirect.com/science/article/pii/S0378216600000680.

[R6] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola O. Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In Foutse Khomh, Chanchal K. Roy, and Janet

Siegmund, editors, *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 286–296. ACM, 2018. doi: 10.1145/3196321.3196347. URL https://doi.org/10.1145/3196321.3196347.

[R7] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, page 234–245, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134630. doi: 10.1145/512529.512558. URL https://doi.org/10.1145/512529.512558.

[R8] Beat Fluri, Michael Würsch, and Harald C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007), 28-31 October 2007, Vancouver, BC, Canada*, pages 70–79. IEEE Computer Society, 2007. doi: 10.1109/WCRE.2007.21. URL https://doi.org/10.1109/WCRE.2007.21.

[R9] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. Analyzing the co-evolution of comments and source code. *Softw. Qual. J.*, 17(4): 367–394, 2009. doi: 10.1007/s11219-009-9075-x. URL https://doi.org/10.1007/s11219-009-9075-x.

[R10] Sonia Haiduc and Andrian Marcus. On the use of domain terms in source code. In René L. Krikhaar, Ralf Lämmel, and Chris Verhoef, editors, *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, pages 113–122. IEEE Computer Society, 2008. doi: 10.1109/ICPC.2008.29. URL https://doi.org/10.1109/ICPC.2008.29.

[R11] Dorsaf Haouari, Houari A. Sahraoui, and Philippe Langlais. How good is your comment? A study of comments in java programs. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, Banff, AB, Canada, September 22-23, 2011*, pages 137–146. IEEE Computer Society, 2011. doi: 10.1109/ESEM.2011.22. URL https://doi.org/10.1109/ESEM.2011.22.

[R12] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In Foutse Khomh, Chanchal K. Roy, and Janet Siegmund, editors, *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 200–210. ACM, 2018. doi: 10.1145/3196321.3196334. URL https://doi.org/10.1145/3196321.3196334.

[R13] Walid M. Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. On the relationship between comment update practices and software bugs. *J. Syst. Softw.*, 85(10):2293–2304, 2012. doi: 10.1016/j.jss.2011.09.019. URL https://doi.org/10.1016/j.jss.2011.09.019.

[R14] Mira Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *Empir. Softw. Eng.*, 10(1):31–55, 2005. doi:

10.1023/B:LIDA.0000048322.42751.ca. URL https://doi.org/10.1023/B:LIDA.0000048322.42751.ca.

[R15] Ninus Khamis, René Witte, and Juergen Rilling. Automatic quality assessment of source code comments: The javadocminer. In Christina J. Hopfe, Yacine Rezgui, Elisabeth Métais, Alun D. Preece, and Haijiang Li, editors, *Natural Language Processing and Information Systems, 15th International Conference on Applications of Natural Language to Information Systems, NLDB 2010, Cardiff, UK, June 23-25, 2010. Proceedings*, volume 6177 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2010. doi: 10.1007/978-3-642-13881-2_7. URL https://doi.org/10.1007/978-3-642-13881-2_7.

[R16] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: how developers see it. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1028–1038. ACM, 2016. doi: 10.1145/2884781.2884840. URL https://doi.org/10.1145/2884781.2884840.

[R17] Douglas Kramer. API documentation from source code comments: a case study of javadoc. In Johndan Johnson-Eilola and Stuart A. Selber, editors, *Proceedings of the 17th annual international conference on Documentation, SIGDOC 1999, New Orleans, Louisiana, USA, September 12-14, 1999*, pages 147–153. ACM, 1999. doi: 10.1145/318372.318577. URL https://doi.org/10.1145/318372.318577.

[R18] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 1–6, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450305471. doi: 10.1145/1937117.1937125. URL https://doi.org/10.1145/1937117.1937125.

[R19] Dawn J. Lawrie, Henry Feild, and David W. Binkley. Leveraged quality assessment using information retrieval techniques. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 149–158. IEEE Computer Society, 2006. doi: 10.1109/ICPC.2006.34. URL https://doi.org/10.1109/ICPC.2006.34.

[R20] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE Trans. Software Eng.*, 37(2):205–227, 2011. doi: 10.1109/TSE.2010.89. URL https://doi.org/10.1109/TSE.2010.89.

[R21] Haroon Malik, Istehad Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and Ahmed E. Hassan. Understanding the rationale for updating a function's comment. In *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, pages 167–176. IEEE Computer Society, 2008. doi: 10.1109/ICSM.2008.4658065. URL https://doi.org/10.1109/ICSM.2008.4658065.

[R22] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems.

IEEE Trans. Software Eng., 34(2):287–300, 2008. doi: 10.1109/TSE.2007.70768. URL https://doi.org/10.1109/TSE.2007.70768.

[R23] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In Chanchal K. Roy, Andrew Begel, and Leon Moonen, editors, *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 279–290. ACM, 2014. doi: 10.1145/2597008.2597149. URL https://doi.org/10.1145/2597008.2597149.

[R24] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, pages 23–32. IEEE Computer Society, 2013. doi: 10.1109/ICPC.2013.6613830. URL https://doi.org/10.1109/ICPC.2013.6613830.

[R25] E. Nurvitadhi, Wing Wah Leung, and C. Cook. Do class comments aid java program understanding? In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 1, pages T3C–T3C. IEEE, 2003. doi: 10.1109/FIE.2003.1263332.

[R26] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers - taxonomies and characteristics of comments in operating system code. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 331–341. IEEE, 2009. doi: 10.1109/ICSE.2009.5070533. URL https://doi.org/10.1109/ICSE.2009.5070533.

[R27] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In Dirk Beyer, Arie van Deursen, and Michael W. Godfrey, editors, *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pages 63–72. IEEE Computer Society, 2012. doi: 10.1109/ICPC.2012.6240510. URL https://doi.org/10.1109/ICPC.2012.6240510.

[R28] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans. Software Eng.*, 28(6): 595–606, 2002. doi: 10.1109/TSE.2002.1010061. URL https://doi.org/10.1109/TSE.2002.1010061.

[R29] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney K. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 390–401. ACM, 2014. doi: 10.1145/2568225.2568247. URL https://doi.org/10.1145/2568225.2568247.

[R30] Jelber Sayyad-Shirabad, Timothy Lethbridge, and Steve Lyon. A little knowledge can go a long way towards program understanding. In *5th International Workshop on Program Comprehension (WPC '97), May 28-30, 1997 - Dearborn, MI, USA*, pages 111–117. IEEE Computer Society, 1997. doi: 10.1109/WPC.1997.601275. URL https://doi.org/10.1109/WPC.1997.601275.

[R31] Daniel Schreck, Valentin Dallmeier, and Thomas Zimmermann. How documentation evolves over time. In Massimiliano Di Penta and Michele Lanza, editors, *9th International Workshop on Principles of Software Evolution (IWPSE 2007), in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia, September 3-4, 2007*, pages 4–10. ACM, 2007. doi: 10.1145/1294948.1294952. URL https://doi.org/10.1145/1294948.1294952.

[R32] Giriprasad Sridhara, Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In René L. Krikhaar, Ralf Lämmel, and Chris Verhoef, editors, *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, pages 123–132. IEEE Computer Society, 2008. doi: 10.1109/ICPC.2008.18. URL https://doi.org/10.1109/ICPC.2008.18.

[R33] Giriprasad Sridhara, Lori L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, pages 71–80. IEEE Computer Society, 2011. doi: 10.1109/ICPC.2011.28. URL https://doi.org/10.1109/ICPC.2011.28.

[R34] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. Code quality analysis in open source software development. *Inf. Syst. J.*, 12(1):43–60, 2002. doi: 10.1046/j.1365-2575.2002.00117.x. URL https://doi.org/10.1046/j.1365-2575.2002.00117.x.

[R35] Daniela Steidl, Benjamin Hummel, and Elmar Jürgens. Quality analysis of source code comments. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, pages 83–92. IEEE Computer Society, 2013. doi: 10.1109/ICPC.2013.6613836. URL https://doi.org/10.1109/ICPC.2013.6613836.

[R36] Margaret-Anne D. 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 Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 251–260. ACM, 2008. doi: 10.1145/1368088.1368123. URL https://doi.org/10.1145/1368088.1368123.

[R37] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17,*

2007, pages 145–158. ACM, 2007. doi: 10.1145/1294261.1294276. URL https://doi.org/10.1145/1294261.1294276.

[R38] Ted Tenny. Program readability: Procedures versus comments. *IEEE Trans. Software Eng.*, 14(9):1271–1279, 1988. doi: 10.1109/32.6171. URL https://doi.org/10.1109/32.6171.

[R39] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. CODES: mining source code descriptions from developers discussions. In Chanchal K. Roy, Andrew Begel, and Leon Moonen, editors, *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 106–109. ACM, 2014. doi: 10.1145/2597008.2597799. URL https://doi.org/10.1145/2597008.2597799.

[R40] Bradley L. Vinz and Letha H. Etzkorn. Improving program comprehension by combining code understanding with comment understanding. *Knowl. Based Syst.*, 21(8):813–825, 2008. doi: 10.1016/j.knosys.2008.03.033. URL https://doi.org/10.1016/j.knosys.2008.03.033.

[R41] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 397–407. ACM, 2018. doi: 10.1145/3238147.3238206. URL https://doi.org/10.1145/3238147.3238206.

[R42] Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 562–567. IEEE, 2013. doi: 10.1109/ASE.2013.6693113. URL https://doi.org/10.1109/ASE.2013.6693113.

[R43] Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik, editors, *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 380–389. IEEE Computer Society, 2015. doi: 10.1109/SANER.2015.7081848. URL https://doi.org/10.1109/SANER.2015.7081848.

[R44] Scott N. Woodfield, Hubert E. Dunsmore, and Vincent Yun Shen. The effect of modularization and comments on program comprehension. In Seymour Jeffrey and Leon G. Stucki, editors, *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*, pages 215–223. IEEE Computer Society, 1981. URL http://dl.acm.org/citation.cfm?id=802534.

[R45] 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 2005, Saint Louis, Missouri, USA, May 17, 2005, page 1–5. ACM, 2005. doi: 10.1145/1083142.1083152. URL https://doi.org/10.1145/1083142.1083152.

Appendix B

Implementation of comment features

This section provides detailed descriptions of the features that were implemented for Coalaty. These descriptions should make it easier to validate or reproduce our findings. Additionally, we have published the original source code for Coalaty on Figshare at https://figshare.com/s/d10cf6030ec7de1ef258.

B.1 On the interpretation of Java source code

Programming languages conform to the rules of a formal grammar. One might therefore expect that source code parsing is an exact science and that it can either be done correctly or *in*correctly. This is true to some extent: for machines, the source of an application is unambiguous and can only be parsed and understood in a single way. But for humans the less formal aspects of source code, like comments and spacing, can sometimes be interpreted in different ways. Our tool must be capable of conducting fully automated analyses of entire software repositories. This means that we have to decide beforehand how we choose to interpret these less formally defined aspects of source code.

First, it is important to consider what exactly constitutes a single comment. Our JavaParser library distinguishes between two major types of comments: line comments and block comments.

Definition 5 (Line comment)**.** A single-line comment that is preceded by `//`

Definition 6 (Block comment)**.** A multi-line comment that is enclosed `/*` and `*/`

In theory, line comments consist of a single line and describe an adjacent line of code, while block comments may – but do not have to – span multiple lines. In practice, developers sometimes use several consecutive line comments that together form a larger comment. An example of such (mis)use of line comments can be seen in Listing B.1.

Accurately determining which line comments possibly belong together and which parts of the code they describe exactly, is non-trivial [16, 20]. We currently choose to simply parse such comments according to definitions 5 and 6. This means that we treat consecutive line comments as separate line comments,

Listing B.1: A Java comment that consists of multiple consecutive line comments.

```
// The SMOG grade is a measure of readability that estimates the years
// of education needed to understand a piece of writing. SMOG is an
// acronym for “Simple Measure Of Gobbledygook”.
1.0430 * Math.sqrt(30.0 * complexWords / sentences) + 3.1291;
```

Listing B.2: A Java comment in which its text has been highlighted

```
/**
 * Check if a number is <b>odd</b>.
 *
 * * Examples include numbers like 1, 3, etc. (but not 2)
 *
 * <pre>
 * if (isOdd(4)) {
 *     // do something
 * }
 * </pre>
 *
 * @param number A number
 * @return True if the number is odd
 */
public boolean isOdd(int number) {
    return number % 2 != 0;
}
```

where the last line comment only describes the line of code that directly follows it.

Another non-trivial aspect of comment parsing concerns the text contained within comments, which usually only has to conform to the rules of some natural language. We call this the comment text.

Definition 7 (Comment text)**.** Any free-format text that is part of a comment

In most cases, the comment text should be equal to any natural language text that is contained within a comment. It includes HTML markup, but not Javadoc tags or references, asterisks (*) that precede comment lines, or pre-formatted text blocks (anything between and including <pre> tags), which contain example code[1]. Listing B.2 provides an example of a code snippet that visually shows what we do and do not consider to be part of the comment text.

Comment text can be split into character sequences, which we call tokens: words, numbers, HTML markup, punctuation, whitespace, and other symbols, like emoji. Some of these tokens are more useful than others. For most features that

[1]We currently make no attempt to detect comments that are actually code that has been commented out. This is a non-trivial problem that is beyond the scope of our work.

Listing B.3: A Java comment in which its tokens have been highlighted

```java
/**
 * Check if a number is <b>odd</b>.
 *
 * * Examples include numbers like 1, 3, etc. (but not 2)
 *
 * <pre>
 * if (isOdd(4)) {
 *     // do something
 * }
 * </pre>
 *
 * @param number A number
 * @return True if the number is odd
 */
public boolean isOdd(int number) {
    return number % 2 != 0;
}
```

our tool computes, we only consider words, numbers, symbols, and character sequences that would be valid identifiers (e.g. `BufferedReader` and `MAX_VALUE`) as tokens. We explicitly exclude markup tags as they are technically not part of the *content* of the comment text. Listing B.3 shows the same code snippet as in the previous listing, except this time we have highlighted the tokens that our tool would consider for its text-related features.

Definition 8 (Token)**.** A consecutive sequence of symbols, alphanumeric, underscore, or hyphen characters

Some features compare the contents of comment texts with those of the code to determine to what extent they are related to each other. Such comparisons are done using stems.

Definition 9 (Stem)**.** The base form of a word, as obtained by using the Porter stemming algorithm

For our tool it does not matter what these stems look like; comparing stems rather than raw tokens helps our tool recognise that words like ‘read’, ‘reads’, ‘reading’, and ‘reader’ are related words that refer to roughly the same semantic concepts.

Some researchers prefer to use the Levenshtein distance, which is the number character edits (additions, modifications, deletions) that are needed to transform one word into another. However, it is more expensive to compute and we believe that it is more likely to lead to false positives. For instance, the words ‘read’ and ‘dead’ would have a Levenshtein distance of only 1, even though the words are not actually related to each other.

Listing B.4: A copyright comment that is completely detached

```
/*
 * Copyright (C) 2009 Google Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * 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.
 */

package com.google.inject;
```

Listing B.5: A comment that was generated automatically

```
import io.reactivex.subjects.PublishSubject;

/**
 * Created by ukanth on 25/9/17.
 */

public class RxCommandEvent {
```

B.2 Excluded comments

Our tool requires that comments can be unambiguously matched to a specific piece of code. This is not always possible. In virtually all cases, it is because a comment does not immediately precede some code, but is separated by an empty line. Common examples include copyright notices (Listing B.4), comments that have been generated by a code editor (Listing B.5), section comments (Listing B.6), and code that has been commented out (Listing B.7).

Some comments are clearly scoped for human readers, but not for our parser due to limitations of our current setup, which treats consecutive single-line comments are separate comments that each have their own scope (Listing B.8). Other comments are completely empty and excluded by our parser as they are not really comments (Listing B.9). Finally, comments are sometimes also used as placeholders for constructor and method bodies that still need to or will never be implemented (Listing B.10).

Listing B.6: A comment that delimits sections of a source code file

```
    DBWriter.clearAllFlattrStatus();
}

// ------------------------------------------------ DIALOGS

private static void showRevokeDialog(final Context context) {
```

Listing B.7: A method that has been commented out

```
/*@Override
protected void onProgressUpdate(Integer... progress) {

if (progress[0] == 0 || progress[0] == -1) {
//do nothing
} else {
loadDialog.incrementProgress(progress[0]);
}
}*/

@Override
protected void onPostExecute(Boolean logPresent) {
```

Listing B.8: A comment that spans multiple line comments

```
// Provide initial value in case the speed list has changed
// out from under us
// and our current speed isn't in the new list
String newSpeed;
if (availableSpeeds.length > 0) {
    newSpeed = availableSpeeds[0];
} else {
    newSpeed = "1.00";
}
```

Listing B.9: A comment that is completely empty

```
/**
 *
 */
public class BuildCommand implements IBuildCommand {
```

Listing B.10: A constructor whose body consists entirely of a comment

```
public RxEvent() {
    // hidden constructor
}
```

B.3 Metric definitions

Here we explain each of the feature implementations (metrics) listed in Table 4.5. In order to make it easier to compare related metrics with each other, the range of output values follows these guidelines:

- Output values of formulas and common metrics are represented without any transformation, even when there is reason to doubt their validity[2];
- Counts of things, like tokens and number of lines, are also represented as is;
- Boolean values are always represented as either `0` or 1;
- Percentages are represented as values between `0` and 1;
- Ratios are presented in a similar way to percentages, except that values can be higher than 1.

Note that some metrics are only applicable in specific situations; for instance, the `method_length` (B.3.20) metric is only defined for methods.

B.3.1 overlap_percentage

The overlap percentage is defined as the number of tokens in the comment text that also appear in the described unit of code, divided by the total number of stems of meaningful (i.e. non-stop word) tokens in the comment text. For classes, we compare textual tokens with all identifiers that appear somewhere in the class, while for methods and constructors we look at identifiers and statements within their body.

The computation is done using stemmed forms of textual tokens and identifiers to allow approximate matches. This is necessary because code is usually always written using an imperative mood, while comments can often also be written in the third person.

B.3.2 cosine_similarity

The cosine similarity measure is a slightly more sophisticated method to determine how much the comment text and code are alike. We compute this value using Apache Commons Text's[3] `CosineSimilarity` class, with the same inputs as for our calculation of `overlap_percentage` (B.3.1).

[2]Readability formulas in particular are designed for texts with specific characteristics, which comment texts do not always meet.

[3]https://commons.apache.org/proper/commons-text/

B.3.3 flesch_ease

The Flesch reading-ease test is a formula that produces a score between 100.0 (very easy to read) to 0.0 (extremely difficult to read) for the readability of English texts. The formula works by summarising some overall properties of a text as follows:

$$206.835 - 1.015\left(\frac{\#\text{ tokens}}{\#\text{ sentences}}\right) - 84.6\left(\frac{\#\text{ syllables}}{\#\text{ tokens}}\right)$$

It is important to note that comment texts should be seen as a sublanguage of English. For instance, camel-cased identifiers typically are compounds, i.e. they consist of a combination of several words. Identifiers are much longer than most English words and thus *might* be harder to read. On the other hand, they reduce the number of tokens in the text. It is not clear how this affects the reliability of readability formulas like the Flesch reading-ease test.

B.3.4 flesch_kincaid

The Flesch–Kincaid Grade Level Formula is based on the same principles as the Flesch reading-ease score, but produces a score that is normally equivalent to a US grade level, which can make it easier to interpret. This is possibly why it is mentioned and used more often in software engineering literature. The feature is calculated as follows:

$$0.39\left(\frac{\#\text{ tokens}}{\#\text{ sentences}}\right) + 11.8\left(\frac{\#\text{ syllables}}{\#\text{ tokens}}\right) - 15.59$$

B.3.5 gunning_fog

The Gunning fog index is another readability test for English writing. Its goal is to estimate the number of years of education that are needed to understand a text. We calculate it as follows:

$$0.4\left[\left(\frac{\#\text{ tokens}}{\#\text{ sentences}}\right) + 100\left(\frac{\#\text{ complex tokens}}{\#\text{ tokens}}\right)\right]$$

where 'complex tokens' are tokens that consist of three or more syllables. Some descriptions of the formula also list exceptions to this rule, e.g. proper nouns (names), familiar jargon, and compound words should not be treated as complex words. Due to time constraints we have chosen to defer the implementation of these exceptions to a later time.

One limitation of the Gunning fog index is that it should be used for passages of around or at least 100 words. Comment texts often do not meet this criterion.

B.3.6 smog_index

The SMOG index (alternatively: SMOG grade) also tries to estimate the number of years of education to understand an English text. The formula can be seen as an improved version of the Gunning fog index and is widely used in medical fields. We calculate the SMOG index using the following formula:

$$1.0430\sqrt{\#\text{ complex tokens} \times \frac{30}{\#\text{ sentences}}} + 3.1291$$

B.3.7 automated_readability

The automated readability index (ARI) is the last readability test for English that we have implemented in our tool. It is primarily easier to calculate than the other formulas since it does not require any 'understanding' of how words are divided into syllables:

$$4.71\left(\frac{\#\text{ characters}}{\#\text{ tokens}}\right) + 0.5\left(\frac{\#\text{ tokens}}{\#\text{ sentences}}\right) - 21.43$$

The number of characters is sentence-based and therefore includes punctuation characters that are excluded from token-based counts.

B.3.8 tokens

This value is equal to the number of tokens that are present in the comment text. Punctuation and HTML markup tags are not included in this count. Periods that are part of abbreviations are treated as part of the abbreviation, which is counted as a single token. The `tokens` feature is also used in many of the readability formulas that we have discussed above.

B.3.9 sentences

We use the Stanford CoreNLP library to detect sentences in our comment text. The value of `sentences` shows the number of sentences that were identified using CoreNLP, and is also used in the readability formulas. Intuitively, sentences are delimited by punctuation like full stops, exclamation marks, question marks, and (semi)colons. Additionally, token sequences that appear in Javadoc tags without punctuation are also counted as sentences.

B.3.10 is_javadoc

This value is true if the comment is enclosed by `/**` and `*/` and appears in a place that supports Javadoc comments.

B.3.11 is_block_comment

This value is true if the comment is enclosed by `/*` and `*/` and is *not* Javadoc.

B.3.12 is_line_comment

This value is true if the comment is preceded by `//` and not part of a multi-line comment.

B.3.13 coverage

This calculates what percentage of a *Javadoc* comment is present. Each comment should at least have a free-format description text. Comments may also describe inputs and outputs using an appropriate Javadoc tag, like `@param` and `@return`, when necessary. Inputs cover method and constructor parameters, while outputs include return values and thrown exceptions.

B.3.14 omitted_full_stops

For simplicity we assume that in most cases sentence-ending punctuation ('.', '?' and '!') can only be omitted in the last sentence of comments, as readers need some way to determine where sentences terminate.

B.3.15 local_identifiers

This counts the number of tokens in the comment text that also appear as identifiers within the scope of the comment. This is done using the original, unstemmed tokens to avoid false positives when English words are similar to identifier names, but do not actually refer an identifier.

B.3.16 math_symbols

We currently use a naive method that checks the comment text for the presence of the following symbols: `=`, `+`, `-`, `*`, `/`, `^`. This value shows the number of distinct symbols, but only if the number is greater than 1.

By requiring that at least two types of symbols are present, we can drastically lower the number of false positives as some of these symbols (e.g. `-` and `/`) also commonly occur as part of English texts.

B.3.17 is_english

Most code and comments are written in English. One could therefore argue that a comment is more consistent if it is also written in English. This feature also makes it easier to interpret the various readability scores, as all of the formulas are specifically designed for the English language.

Accurately determining the language of a text is non-trivial, especially with a limited language model. We therefore choose an alternative approach, where we use a spelling checker on all words in the comment text, except those that appear as an identifier, e.g. class, method, and variable names. This allows us to 'classify' words as English or non-English: texts with a high ratio of correctly spelled words are very likely to be in English, while texts with a very low ratio are probably written in some other language.

B.3.18 identifiers

The number of unique identifiers that appear within the scope of the comment. We currently use a regular expression-based approach to extract these from expressions in the abstract syntax tree.

B.3.19 operators

The number of operators that appear in the described constructor or method. Examples of operators include assignment expressions (i.e. `=`), binary expressions (e.g. `+`), and unary expressions (e.g. `++`).

B.3.20 method_length

The number of physical lines in a method is calculated by simply counting the number of newline characters (`\n`). This value may therefore be affected by coding styles, superfluous whitespace and comments within methods, and is therefore best seen as a ballpark figure.

B.3.21 cyclomatic_complexity

McCabe's cyclomatic complexity is computed for methods by counting the number of `if`, `for`(each), `while`, `switch`, `throw`, and `catch` statements, conditional expressions (ternaries), and `||` and `&&` within statements that support them.

B.3.22 tasks

This value shows the number of occurrences of several commonly used keywords for task comments: `TODO`, `HACK`, `XXX`, `FIXME`, and `REVISIT`. We count occurrences using a case-insensitive search within the comment text.

B.3.23 method_name_similarity

This boolean value is true if more than 50% of the comment text is similar to tokens in the method name. Similarity in this case means that a stemmed, lowercased token appears both in the comment text and the method name as a token.

B.3.24 extra_info_score

This metric tries to capture several features about the extent that a comment provides extra information about a method using a single linear variable.

Comment texts that have a length of at most 2 almost certainly do not provide extra information and result in a score of 0. Comments texts with a length up to and including 30 characters may contain extra information and get a score of 1. Other comment texts receive a score that shows the length relative to the minimum threshold of 30, e.g. a comment text with length 45 will have a score of 45/30 = 1.5.

B.3.25 global_identifiers

This metric is computed by counting the number of tokens in the comment text that also appears elsewhere (i.e. not within the scope of the comment) as an identifier. This is done using the original, unstemmed tokens to avoid false positives when English words are similar to identifier names, but do not actually refer to some other place in the code. We also exclude stop words, as identifiers should rarely be named after stop words.

B.3.26 mentions_parents

True if the comment text includes any form of the words 'override' or 'overload'.

B.3.27 is_for_attribute

This value is true if the comment is placed directly above an attribute declaration. Attributes are also known as class variables or members.

B.3.28 is_for_class

This value is true if the comment is placed directly above a top-level class declaration. Inner and anonymous classes are not included in our analysis.

B.3.29 is_for_constructor

This value is true if the comment is placed directly above a class constructor.

B.3.30 is_for_method

This value is true if the comment is placed directly above a method declaration.

B.3.31 is_for_package

This value is true if the comment is placed directly above a package declaration in a `package-info.java` file.

B.3.32 is_for_enum

This value is true if the comment is placed directly above an enum declaration.

B.3.33 is_for_annotation

This value is true if the comment is placed directly above an annotation declaration.

B.3.34 is__for_interface

This value is true if the comment is placed directly above an interface declaration. This only covers actual `interfaces`, not similar concepts like abstract classes.

B.3.35 control_structures

Counts all occurrences of the following control structures: `break`, `continue`, `if`, `return`, or `switch`.

B.3.36 loop_structures

Counts all occurrences of the following loop structures: `do`, `while`, `foreach`[4] or `for`.

B.3.37 method_calls

The number of occurrences of method calls. For methods, we search the abstract syntax tree for method call expressions. For attributes, we approximate this check by detecting the presence of an opening parenthesis (`(`).

[4] Loops that look like `for (Type element : list)`, as opposed to loops that look like `for (int i = 0; i < someLimit; i++)`

B.3.38 variable_assignments

Counts all occurrences of variable assignments. For methods, we search the abstract syntax tree for assignment operators. For attributes, we approximate this check by detecting the presence of the assignment operator (`=`).

B.3.39 variable_declaration

Counts all occurrences of variable declarations.

B.3.40 try_catch_blocks

Counts all occurrences of try-catch-blocks. `catch` and `finally` are not counted separately, but included in this count.

B.3.41 describes_inputs

This metric works in a similar way as `coverage` (B.3.13), but only covers the inputs of a constructor or method, i.e. `@param`. Note that it does not include the free-format description.

B.3.42 describes_output

This metric works in a similar way as `coverage` (B.3.13), but only covers the outputs of a constructor or method, i.e. `@return` and `@throws`. Note that it does not include the free-format description.

B.3.43 hyperlinks

Number of hyperlinks in the comment text. For now, we use a naive implementation that only looks at tokens that start with 'http://', 'https://' or 'www.'.

B.3.44 issue_numbers

Count of possible issue numbers in the comment text. Based on our experience with commonly used issue trackers like GitHub and JIRA, we assume that most issue numbers look like '#12345' or 'ABC-12345'.

B.3.45 abbreviations

We detect abbreviations using a heuristic: an abbreviation is a token that includes full stops (e.g. 'e.g.') or only includes uppercase characters (e.g. 'MIT'). Tokens that include numbers (e.g. '3.14' or look like email or internet addresses (e.g. 'nos.nl') are not treated as abbreviations. Special keywords that are commonly used for task comments (e.g. 'TODO') are often written using uppercase characters; we do not consider these to be abbreviations.

Our heuristic does not recognise some special abbreviations, like 'Ph.D', and common abbreviations that are written without uppercase characters or full stops, like 'aka'.

To avoid false positives when the comment text is largely or completely written in uppercase characters (e.g. 'YOU MUST NOT…'), we only consider abbreviations that are next to a non-abbreviated token.

Appendix C

Survey design

In this appendix we describe the specifics of our survey design, which may be helpful to those who wish to reproduce our results or conduct a follow-up study.

C.1 Recruitment text

While respondents for our survey were recruited in various ways, all were sent to a web page that contained more information about the study. This web page is published as an unlisted page (i.e., it is only reachable for those who were given its URL) under a publicly accessible domain, and contains the following recruitment text:

> Every software developer knows that source code can be hard to read, especially if it was written by someone else – or yourself, but a very long time ago. Adding comments to code can make it more understandable. But not always.
>
> It is not entirely clear what makes some comments better than others. We therefore wish to gain a better understanding of the factors that make a comment more (or less) helpful to future software developers.
>
> You can help us by answering some questions about comments in code snippets in an online survey. The survey should only take about 20 minutes.
>
> In return, we'll raffle 3 $50 Amazon gift cards among respondents and pledge to donate €1 to the Dutch Cancer Society (*KWF Kankerbestrijding*) for each of the first 100 respondents!
>
> Are you just as excited about this study as we are? Send an email to cf.lung@studie.ou.nl titled "Survey" and we'll reply with the details!

Prospective respondents generally visit this page when they:

- were directly contacted by the researcher;
- see a short, informal advertisement for the survey; or
- were referred via someone who had already completed the survey.

C.2 Information letter

The details and conditions of the study are outlined in the information letter that prospective participants receive when they contact the researcher via the email address listed in the recruitment text.

Participants are told and expected to have read this letter before they open the link to the online survey.

> Thank you for expressing your interest in our survey on comments in Java source code. This survey is part of a research project at the Open University of the Netherlands (*Open Universiteit*).
>
> Please read the information below carefully. If you have any questions, you can ask the researcher and principal investigator for additional information. Their contact details can be found at the end of this letter.
>
> ### What the study is about
>
> Every software developer knows that source code can be hard to read, especially if it was written by someone else – or yourself, but a very long time ago. Adding comments to code can make it more understandable. But not always.
>
> It is not entirely clear what makes some comments better than others. We therefore wish to gain a better understanding of the factors that make a comment more (or less) helpful to future software developers.
>
> ### What participation means and is expected of you
>
> Our survey will show you Java code snippets with a comment. You will be asked to read and rate the comments in these snippets. We ask that you do this in a quiet environment that is free from distractions. The survey should take about 20 minutes. After completion of the survey, you get the opportunity to take part in a raffle for one of the three $50 Amazon gift cards. We will also donate €1 to the Dutch Cancer Society (*KWF Kankerbestrijding*) for each of the first 100 respondents.
>
> If you decide to participate in this study, you will be asked to indicate that you understand the information in this letter and agree to participate in the research by completing a consent form. You can find this consent form and the survey at softwareengineering.limequery.org/972851.
>
> Participation is voluntary. If you do not want to participate, this does not have any negative consequences for you. If you do participate, you can always change your mind and quit, by not submitting your survey responses. You do not have to explain why you quit.
>
> ### What will happen afterwards
>
> Your participation in the research ends when you submit your survey responses. The entire research is finished when all participants have

submitted their responses. If you wish to be informed about the results of the research, you can send an email to cf.lung@studie.ou.nl.

Your privacy is important to us

The survey is anonymous. However, we do collect, use, and store a small amount of data about your demographic background; specifically, your level of education and (programming) language proficiency. This is necessary to interpret your responses. We make sure that this data cannot be traced back for you.

All survey data are stored anonymously by the Open University of the Netherlands for 10 years, and may be shared with colleagues, e.g. for auditing purposes.

Moreover, if you take part in the raffle, your email address will be stored at least until the date of the raffle (1 October 2021).

For general information about your rights when processing personal data, you can consult the website of the Dutch Data Protection Authority (*Autoriteit Persoonsgegevens*). The privacy disclaimer of the Open University of the Netherlands can be found at www.ou.nl/en/persoonsgegevens-disclaimer.

Questions

If you have any questions about this study, please contact the researcher via cf.lung@studie.ou.nl.

Contact details

Researcher	Chun Fei Lung (cf.lung@studie.ou.nl)
Principal investigator	Ebrahim Rahimi (Ebrahim.Rahimi@ou.nl)
Data protection officer	Saskia van der Westen (FG@ou.nl)

C.3 Informed consent

As mentioned in the information letter, participants are required to complete a consent form before they can participate in the study. Due to technical reasons we present the informed consent form as the first page of the survey. Respondents can only proceed with the survey after completing the consent form. Our study uses the following template:

☐ I give permission for the data that is collected during this study to be used for this scientific research.

☐ I have read the information letter related to this study and I have had the opportunity to ask questions to the researcher if certain points were not clear.

☐ I understand that all the information that I supply in relation to this study will be collected in a safe manner, *will be published anonymously (if applicable)* and therefore will not lead back to me.

☐ I understand that I can pull out of the study at any time by exiting the survey. I do not have to provide a reason for doing so.

☐ I agree that collected data is stored for a period of 10 years, in accordance with the guidelines for the Association of Universities in the Netherlands (VSNU).

If you have read the above points and agree to participate in the study, please check all the boxes and enter today's date in the box below:

2021-12-31

C.4 Snippets

In the survey respondents are asked to rate the helpfulness of 44 code snippets with comments. Table C.1 lists the snippets that were included in our survey, along with the secondary statement that accompanied each snippet and their provenance.

The snippets presented in the survey were identical to how they appeared in the original source code. In this section, we have made a few small modifications that make them more suitable for a printable representation:

- Superfluous left indentations have been removed;
- Long lines have been wrapped (denoted by '↳');
- Long methods have been truncated (denoted by '. . .').

Table C.1: List of snippets

Snippet	Statement	Provenance
C.1		com.google.common.hash.AbstractByteHasher:51-53
C.2		org.eclipse.cdt.internal.ui.actions.IndentAction:457-461
C.3		com.owncloud.android.operations.SynchronizeFolderOperation:180-184
C.4	A	main.java.de.danoeh.antennapod.core.service.download.DownloadService:467-472
C.5	A	com.owncloud.android.utils.MimetypeIconUtil:180-185
C.6	A	com.vaadin.server.JsonCodec:782-797
C.7	A	org.eclipse.cdt.debug.ui.memory.floatingpoint.Rendering:1784-1792
C.8	A	org.eclipse.cdt.utils.EFSExtensionManager:98-107
C.9		com.owncloud.android.files.services.FileDownloader:110-112
C.10	B	org.eclipse.cdt.internal.ui.text.spelling.engine.ISpellChecker:37-41
C.11	B	org.apache.hadoop.security.authentication.util.KerberosUtil:103-119
C.12	C	com.stericson.roottools.RootTools:229-234
C.13	C	org.wordpress.android.util.StringUtils:185-188
C.14		org.apache.hadoop.registry.client.types.ServiceRecord:130-134
C.15	B	com.google.common.collect.ImmutableBiMap:71-75
C.16	B	com.owncloud.android.files.services.TransferRequester:99-101
C.17	A	org.apache.hadoop.io.compress.zlib.BuiltInZlibDeflater:57-64
C.18	A	org.apache.hadoop.hdfs.server.namenode.FSNamesystem:1868-1872
C.19		org.wordpress.android.ui.stats.StatsWidgetProvider:282-286
C.20		com.google.inject.Key:290-292
C.21	D	org.apache.hadoop.mapreduce.jobhistory.JobHistoryParser:773-773
C.22	D	com.stericson.rootshell.RootShell:104-110
C.23	D	org.apache.hadoop.util.ToolRunner:103-106
C.24	D	org.eclipse.cdt.debug.core.CDebugUtils:585-596
C.25		de.danoeh.antennapod.core.service.playback.PlaybackService:190-192
C.26		org.eclipse.cdt.dsf.internal.LoggingUtils:85-92
C.27	B	org.wordpress.android.util.StringUtils:100-104
C.28	B	org.wordpress.android.models.ReaderPost:496-499
C.29	B	org.wordpress.android.editor.EditorFragment:1547-1549
C.30	B	com.vaadin.data.util.sqlcontainer.SQLContainer:887-890
C.31	B	org.eclipse.cdt.internal.ui.editor.AddIncludeAction:175-177
C.32		org.apache.hadoop.hdfs.server.blockmanagement.BlockPlacementPolicy:216-226
C.33		org.apache.hadoop.hdfs.util.XMLUtils:65-75
C.34		de.danoeh.antennapod.core.storage.PodDBAdapter:557-560
C.35		org.apache.hadoop.hdfs.qjournal.server.Journal:179-184
C.36		org.wordpress.android.ui.posts.adapters.PostsListAdapter:796-799
C.37		org.wordpress.android.datasets.ReaderSearchTable:60-64
C.38		org.eclipse.cdt.internal.corext.util.Strings:86-92
C.39		com.amaze.filemanager.filesystem.FileUtil:637-642
C.40	B	org.apache.hadoop.hdfs.protocol.datatransfer.PacketHeader:187-191
C.41	B	org.eclipse.cdt.internal.corext.util.Strings:110-119
C.42		com.amaze.filemanager.utils.Utils:105-126
C.43	A	org.apache.hadoop.hdfs.DFSUtil:712-722
C.44	A	com.google.common.primitives.UnsignedLong:184-190

A = comment is easy to understand, B = contains right amount of information
C = just as easy to understand without the comment, D = comment meets expectations

Listing C.1: Comment that describes what the code does

```
/**
 * Updates this hasher with {@code len} bytes starting at {@code off} in
     ↳ the given buffer.
 */
protected void update(byte[] b, int off, int len) {
  for (int i = off; i < off + len; i++) {
    update(b[i]);
  }
}
```

Listing C.2: Comment that literally repeats the code

```
/**
 * Returns the editor's selection provider.
 *
 * @return the editor's selection provider or <code>null</code>
 */
private ISelectionProvider getSelectionProvider() {
    ITextEditor editor= getTextEditor();
    if (editor != null) {
        return editor.getSelectionProvider();
    }
    return null;
}
```

Listing C.3: Comment that describes the purpose of the code

```
/**
 * Performs the synchronization.
 *
 * {@inheritDoc}
 */
@Override
protected RemoteOperationResult run(OwnCloudClient client) {
    RemoteOperationResult result;
    mFailsInFileSyncsFound = 0;
    mConflictsFound = 0;
    mForgottenLocalFiles.clear();

    try {
        // get locally cached information about folder
        mLocalFolder = getStorageManager().getFileByPath(mRemotePath);

        if (mPushOnly) {
            // assuming there is no update in the server side, still need
                ↳ to handle local changes
            Log_OC.i(TAG, "Push only sync of " + mAccount.name +
                ↳ mRemotePath);
            preparePushOfLocalChanges();
            syncContents();
            //pushOnlySync();
            result = new RemoteOperationResult(ResultCode.OK);

        } else {
            // get list of files in folder from remote server
            result = fetchRemoteFolder(client);

            if (result.isSuccess()) {
                // success - merge updates in server with local state
                mergeRemoteFolder(result.getData());
                syncContents();

            } else {
                // fail fetching the server
                if (result.getCode() == ResultCode.FILE_NOT_FOUND) {
                    removeLocalFolder();
                }
                if (result.isException()) {
                    Log_OC.e(TAG, "Checked " + mAccount.name + mRemotePath
                        ↳  + " : " +
                        result.getLogMessage(), result.getException());
                } else {
                    ...
```

Listing C.4: Comment that is written in easy-to-understand English (1)

```
/**
 * Adds a new DownloadStatus object to the list of completed downloads and
 * saves it in the database
 *
 * @param status the download that is going to be saved
 */
private void saveDownloadStatus(DownloadStatus status) {
    reportQueue.add(status);
    DBWriter.addDownloadStatus(status);
}
```

Listing C.5: Comment that is written in easy-to-understand English (2)

```
/**
 * provides the file extension of a given filename.
 *
 * @param filename the filename
 * @return the file extension
 */
private static String getExtension(String filename) {
    String extension = filename.substring(filename.lastIndexOf(".") + 1).
        ↳ toLowerCase();
    return extension;
}
```

Listing C.6: Comment with English for professionals

```
/**
 * Compares two json values for deep equality.
 *
 * This is a helper for overcoming the fact that
 * {@link JsonValue#equals(Object)} only does an identity check and
 * {@link JsonValue#jsEquals(JsonValue)} is defined to use JavaScript
 * semantics where arrays and objects are equals only based on identity.
 *
 * @since 7.4
 * @param a
 *            the first json value to check, may not be null
 * @param b
 *            the second json value to check, may not be null
 * @return <code>true</code> if both json values are the same;
 *         <code>false</code> otherwise
 */
public static boolean jsonEquals(JsonValue a, JsonValue b) {
    assert a != null;
    assert b != null;

    if (a == b) {
        return true;
    }

    JsonType type = a.getType();
    if (type != b.getType()) {
        return false;
    }

    switch (type) {
    case NULL:
        return true;
    case BOOLEAN:
        return a.asBoolean() == b.asBoolean();
    case NUMBER:
        return a.asNumber() == b.asNumber();
    case STRING:
        return a.asString().equals(b.asString());
    case OBJECT:
        return jsonObjectEquals((JsonObject) a, (JsonObject) b);
    case ARRAY:
        return jsonArrayEquals((JsonArray) a, (JsonArray) b);
    default:
        throw new RuntimeException("Unsupported JsonType: " + type);
    }
}
```

Listing C.7: Comment with English that may be hard to understand (1)

```
/**
 * The scroll range is limited by SWT. Because it can be less than the
     ↳ number
 * of rows (of memory) that we need to display, we need an arithmetic
     ↳ mapping.
 *
 * @return ratio this function returns how many rows a scroll bar unit
 *         represents. The number will be some fractional value, up to but
 *         not exceeding the value 1. I.e., when the scroll range exceeds
 *         the row range, we use a 1:1 mapping.
 */
private final BigDecimal getScrollRatio()
{
    BigInteger maxRange = getMaxScrollRange();
    if (maxRange.compareTo(BigInteger.valueOf(Integer.MAX_VALUE)) > 0)
    {
        return new BigDecimal(maxRange).divide(BigDecimal.valueOf(Integer.
            ↳ MAX_VALUE), SCROLL_CONVERSION_PRECISION);
    }

    return BigDecimal.ONE;
}
```

Listing C.8: Comment with English that may be hard to understand (2)

```
/**
 * If the EFS store represented by locationURI is backed by a physical file
     ↳ , gets the path corresponding
 * to the underlying file.  The path returned is suitable for use in
     ↳ constructing a {@link Path} object.  This
 * method will return the corresponding path regardless of whether or not
     ↳ the EFS store actually exists.
 *
 *
 * @param locationURI
 * @return String representing the path, or <code>null</code> if there is
     ↳ an error or if the store
 * is not backed by a physical file.
 */
public String getPathFromURI(URI locationURI) {
    EFSExtensionProvider provider = fSchemeToExtensionProviderMap.get(
        ↳ locationURI.getScheme());

    if (provider == null) {
        provider = fDefaultProvider;
    }

    return provider.getPathFromURI(locationURI);

}
```

Listing C.9: A very short comment

```
/**
 * Service initialization
 */
@Override
public void onCreate() {
    super.onCreate();
    Log_OC.d(TAG, "Creating service");
    mNotificationManager = (NotificationManager) getSystemService(
        ↳ NOTIFICATION_SERVICE);

    // Configure notification channel
    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.
        ↳ O) {
        NotificationChannel mNotificationChannel;
        // The user-visible name of the channel.
        CharSequence name = getString(R.string.
            ↳ download_notification_channel_name);
        // The user-visible description of the channel.
        String description = getString(R.string.
            ↳ download_notification_channel_description);
        // Set importance low: show the notification everywhere but with no
            ↳  sound
        int importance = NotificationManager.IMPORTANCE_LOW;
        mNotificationChannel = new NotificationChannel(
            ↳ DOWNLOAD_NOTIFICATION_CHANNEL_ID,
                name, importance);
        // Configure the notification channel.
        mNotificationChannel.setDescription(description);
        mNotificationManager.createNotificationChannel(mNotificationChannel
            ↳ );
    }

    HandlerThread thread = new HandlerThread("FileDownloaderThread",
            Process.THREAD_PRIORITY_BACKGROUND);
    thread.start();
    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper, this);
    mBinder = new FileDownloaderBinder();

    // add AccountsUpdatedListener
    AccountManager am = AccountManager.get(getApplicationContext());
    am.addOnAccountsUpdatedListener(this, null, false);

    // create manager for local broadcasts
    mLocalBroadcastManager = LocalBroadcastManager.getInstance(this);
}
```

Listing C.10: A comment with fewer than 30 words

```
/**
 * Returns whether this spell checker accepts word additions.
 *
 * @return <code>true</code> if word additions are accepted, <code>false</
     ↳ code> otherwise
 */
boolean acceptsWords();
```

Listing C.11: A comment with more than 100 words

```
/*
 * For a Service Host Principal specification, map the host's domain
 * to kerberos realm, as specified by krb5.conf [domain_realm] mappings.
 * Unfortunately the mapping routines are private to the security.krb5
 * package, so have to construct a PrincipalName instance to derive the
     ↳ realm.
 *
 * Many things can go wrong with Kerberos configuration, and this is not
 * the place to be throwing exceptions to help debug them.  Nor do we
     ↳ choose
 * to make potentially voluminous logs on every call to a communications
     ↳ API.
 * So we simply swallow all exceptions from the underlying libraries and
 * return null if we can't get a good value for the realmString.
 *
 * @param shortprinc A service principal name with host fqdn as instance, e
     ↳ .g.
 *     "HTTP/myhost.mydomain"
 * @return String value of Kerberos realm, mapped from host fqdn
 *     May be default realm, or may be null.
 */
public static String getDomainRealm(String shortprinc) {
  Class<?> classRef;
  Object principalName; //of type sun.security.krb5.PrincipalName or IBM
      ↳ equiv
  String realmString = null;
  try {
    if (IBM_JAVA) {
      classRef = Class.forName("com.ibm.security.krb5.PrincipalName");
    } else {
      classRef = Class.forName("sun.security.krb5.PrincipalName");
    }
    int tKrbNtSrvHst = classRef.getField("KRB_NT_SRV_HST").getInt(null);
    principalName = classRef.getConstructor(String.class, int.class).
        newInstance(shortprinc, tKrbNtSrvHst);
    realmString = (String)classRef.getMethod("getRealmString", new Class
        ↳ [0]).
        invoke(principalName, new Object[0]);
  } catch (RuntimeException rte) {
    //silently catch everything
  } catch (Exception e) {
    //silently return default realm (which may itself be null)
  }
  if (null == realmString || realmString.equals("")) {
    return getDefaultRealmProtected();
  ...
```

Listing C.12: Comment that is formatted as Javadoc

```
/**
 * @param binaryName String that represent the binary to find.
 * @param singlePath boolean that represents whether to return a single
     ↳ path or multiple.
 *
 * @return <code>List<String></code> containing the paths the binary was
     ↳ found at.
 */
public static List<String> findBinary(String binaryName, boolean singlePath
    ↳ ) {
    return RootShell.findBinary(binaryName, singlePath);
}
```

Listing C.13: Comment that is written as a normal block comment

```
/*
 * Wrap an image URL in a photon URL
 * Check out http://developer.wordpress.com/docs/photon/
 */
public static String getPhotonUrl(String imageUrl, int size) {
    imageUrl = imageUrl.replace("http://", "").replace("https://", "");
    return "http://i0.wp.com/" + imageUrl + "?w=" + size;
}
```

Listing C.14: Javadoc comment that documents all attributes

```
/**
 * Look up an internal endpoint
 * @param api API
 * @return the endpoint or null if there was no match
 */
public Endpoint getInternalEndpoint(String api) {
  return findByAPI(internal, api);
}
```

Listing C.15: Javadoc comment that documents some attributes

```
/**
 * Returns an immutable map containing the given entries, in order.
 *
 * @throws IllegalArgumentException if duplicate keys or values are added
 */
public static <K, V> ImmutableBiMap<K, V> of(K k1, V v1, K k2, V v2, K k3,
    ↳ V v3, K k4, V v4) {
  return RegularImmutableBiMap.fromEntries(
      entryOf(k1, v1), entryOf(k2, v2), entryOf(k3, v3), entryOf(k4, v4));
}
```

Listing C.16: Javadoc comment that does not document attributes

```
/**
 * Call to upload a new single file
 */
public void uploadNewFile(Context context, Account account, String
    ↳ localPath, String remotePath, int
        behaviour, String mimeType, boolean createRemoteFile, int createdBy
            ↳ ) {

    uploadNewFiles(
            context,
            account,
            new String[]{localPath},
            new String[]{remotePath},
            new String[]{mimeType},
            behaviour,
            createRemoteFile,
            createdBy
    );
}
```

Listing C.17: Comment that reuses terms from code

```
/**
 * reinit the compressor with the given configuration. It will reset the
 * compressor's compression level and compression strategy. Different from
 * <tt>ZlibCompressor</tt>, <tt>BuiltInZlibDeflater</tt> only support three
 * kind of compression strategy: FILTERED, HUFFMAN_ONLY and
     ↳ DEFAULT_STRATEGY.
 * It will use DEFAULT_STRATEGY as default if the configured compression
 * strategy is not supported.
 */
@Override
public void reinit(Configuration conf) {
  reset();
  if (conf == null) {
    return;
  }
  setLevel(ZlibFactory.getCompressionLevel(conf).compressionLevel());
  final ZlibCompressor.CompressionStrategy strategy =
    ZlibFactory.getCompressionStrategy(conf);
  try {
    setStrategy(strategy.compressionStrategy());
  } catch (IllegalArgumentException ill) {
    LOG.warn(strategy + " not supported by BuiltInZlibDeflater.");
    setStrategy(DEFAULT_STRATEGY);
  }
  if(LOG.isDebugEnabled()) {
    LOG.debug("Reinit compressor with new compression configuration");
  }
}
```

Listing C.18: Comment that largely uses different terms than code

```
/**
 * stores the modification and access time for this inode.
 * The access time is precise up to an hour. The transaction, if needed, is
 * written to the edits log but is not flushed.
 */
void setTimes(String src, long mtime, long atime) throws IOException {
  HdfsFileStatus auditStat;
  checkOperation(OperationCategory.WRITE);
  writeLock();
  try {
    checkOperation(OperationCategory.WRITE);
    checkNameNodeSafeMode("Cannot set times " + src);
    auditStat = FSDirAttrOp.setTimes(dir, src, mtime, atime);
  } catch (AccessControlException e) {
    logAuditEvent(false, "setTimes", src);
    throw e;
  } finally {
    writeUnlock();
  }
  getEditLog().logSync();
  logAuditEvent(true, "setTimes", src, null, auditStat);
}
```

Listing C.19: Comment that contains full sentences

```
/**
 * This is called every time an App Widget is deleted from the App Widget
     ↳ host.
 * @param context The Context in which this receiver is running.
 * @param widgetIDs Widget IDs to set blank. We cannot remove widget from
     ↳ home screen.
 */
@Override
public void onDeleted(Context context, int[] widgetIDs) {
    setRemoteBlogIDForWidgetIDs(widgetIDs, 0);
}
```

Listing C.20: Comment that is written in telegraphic style

```
/**
 * Gets a key for an injection type and an annotation type.
 */
public static <T> Key<T> get(TypeLiteral<T> typeLiteral,
    Class<? extends Annotation> annotationType) {
  return new Key<T>(typeLiteral, strategyFor(annotationType));
}
```

Listing C.21: Comment that describes trivial code (1)

```
/** @return the ApplicationAttemptId */
public ApplicationAttemptId getAppAttemptId() {
  return appAttemptId;
}
```

Listing C.22: Comment that describes trivial code (2)

```
/**
 * Use this to check whether or not a file exists on the filesystem.
 *
 * @param file String that represent the file, including the full path to
     ↳ the
 *             file and its name.
 * @return a boolean that will indicate whether or not the file exists.
 */
public static boolean exists(final String file) {
    return exists(file, false);
}
```

Listing C.23: Comment that describes complex code (1)

```
/**
 * Print out a prompt to the user, and return true if the user
 * responds with "y" or "yes". (case insensitive)
 */
public static boolean confirmPrompt(String prompt) throws IOException {
  while (true) {
    System.err.print(prompt + " (Y or N) ");
    StringBuilder responseBuilder = new StringBuilder();
    while (true) {
      int c = System.in.read();
      if (c == -1 || c == '\r' || c == '\n') {
        break;
      }
      responseBuilder.append((char)c);
    }

    String response = responseBuilder.toString();
    if (response.equalsIgnoreCase("y") ||
        response.equalsIgnoreCase("yes")) {
      return true;
    } else if (response.equalsIgnoreCase("n") ||
        response.equalsIgnoreCase("no")) {
      return false;
    }
    System.err.println("Invalid input: " + response);
    // else ask them again
  }
}
```

Listing C.24: Comment that describes complex code (2)

```
/**
 * Returns the ICProject associated with the project setting in the Main
     ↳ tab
 * of a CDT launch configuration, or throws a CoreException providing a
 * reason (e.g., the setting is empty, the project no longer exists, the
 * isn't a CDT one, etc).
 *
 * @param config
 *            the launch configuration
 * @return an ICProject; never null.
 * @throws CoreException
 * @since 7.0
 */
public static ICProject verifyCProject(ILaunchConfiguration config) throws
    ↳ CoreException {
    String name = CDebugUtils.getProjectName(config);
    if (name == null) {
        throwCoreException(DebugCoreMessages.getString("CDebugUtils.
            ↳ C_Project_not_specified"),  //$NON-NLS-1$
                ICDTLaunchConfigurationConstants.ERR_UNSPECIFIED_PROJECT);
    }
    ICProject cproject = CDebugUtils.getCProject(config);
    if (cproject == null) {
        IProject proj = ResourcesPlugin.getWorkspace().getRoot().getProject
            ↳ (name);
        if (!proj.exists()) {
            throwCoreException(DebugCoreMessages.getFormattedString("
                ↳ CDebugUtils.Project_NAME_does_not_exist", name),  //$NON
                ↳ -NLS-1$
                    ICDTLaunchConfigurationConstants.ERR_NOT_A_C_PROJECT);
        } else if (!proj.isOpen()) {
            throwCoreException(DebugCoreMessages.getFormattedString("
                ↳ CDebugUtils.Project_NAME_is_closed", name), //$NON-NLS-1
                ↳ $
                    ICDTLaunchConfigurationConstants.ERR_NOT_A_C_PROJECT);
        }
        throwCoreException(DebugCoreMessages.getString("CDebugUtils.
            ↳ Not_a_C_CPP_project"),  //$NON-NLS-1$
                ICDTLaunchConfigurationConstants.ERR_NOT_A_C_PROJECT);
    }
    return cproject;
}
```

Listing C.25: Comment for code with many operators (1)

```
/**
 * Returns true if the sleep timer is currently active.
 */
public synchronized boolean isSleepTimerActive() {
    return sleepTimer != null
            && sleepTimerFuture != null
            && !sleepTimerFuture.isCancelled()
            && !sleepTimerFuture.isDone()
            && sleepTimer.getWaitingTime() > 0;
}
```

Listing C.26: Comment for code with many operators (2)

```
/**
 * General string utility for removing newline and space character from the
 * end of a string. Typically used when logging an object's toString()
 *
 * @param str
 *            the string
 * @return the string without trailing newlines
 */
public static String trimTrailingNewlines(String str) {
    final int strlen = str.length();
    if (strlen == 0) {
        return str;
    }

    int removeCount = 0;
    for (int i = strlen - 1; i >= 0 && Character.isWhitespace(str.charAt(i)
        ↳ ); i--) {
        removeCount++;
    }

    return (removeCount == 0) ? str : str.substring(0, str.length() -
        ↳ removeCount);
}
```

Listing C.27: Comment with an explicit ‘todo’

```
/*
 * nbradbury - adapted from Html.escapeHtml(), which was added in API Level
     ↳  16
 * TODO: not thoroughly tested yet, so marked as private - not sure I like
     ↳ the way
 * this replaces two spaces with " "
 */
private static String escapeHtml(final String text) {
    if (text == null) {
        return "";
    }

    StringBuilder out = new StringBuilder();
    int length = text.length();

    for (int i = 0; i < length; i++) {
        char c = text.charAt(i);

        if (c == '<') {
            out.append("&lt;");
        } else if (c == '>') {
            out.append("&gt;");
        } else if (c == '&') {
            out.append("&amp;");
        } else if (c > 0x7E || c < ' ') {
            out.append("&#").append((int) c).append(";");
        } else if (c == ' ') {
            while (i + 1 < length && text.charAt(i + 1) == ' ') {
                out.append(" ");
                i++;
            }

            out.append(' ');
        } else {
            out.append(c);
        }
    }

    return out.toString();
}
```

Listing C.28: Comment with an implicit ‘todo’

```
/*
 * attachments are stored as the actual JSON to avoid having a separate
     ↳ table for
 * them, may need to revisit this if/when attachments become more important
 */
public String getAttachmentsJson() {
    return StringUtils.notNullStr(attachmentsJson);
}
```

Listing C.29: Comment that summarises code

```
/**
 * Hide the action bar if needed.
 */
private void hideActionBarIfNeeded() {

    ActionBar actionBar = getActionBar();
    if (actionBar != null
            && !isHardwareKeyboardPresent()
            && mHideActionBarOnSoftKeyboardUp
            && mIsKeyboardOpen
            && actionBar.isShowing()) {
        getActionBar().hide();
    }
}
```

Listing C.30: Comment that provides summary and extra information

```
/**
 * Refreshes the container - clears all caches and resets size and offset.
 * Does NOT remove sorting or filtering rules!
 */
public void refresh() {
    refresh(true);
}
```

Listing C.31: Comment that only provides extra information

```
/**
 * For tests only.
 */
public void setAmbiguityResolver(IElementSelector fAmbiguityResolver) {
    this.fAmbiguityResolver = fAmbiguityResolver;
}
```

Listing C.32: Comment that mentions related classes

```
/**
 * Split data nodes into two sets, one set includes nodes on rack with
 * more than one  replica, the other set contains the remaining nodes.
 *
 * @param availableSet all the available DataNodes/storages of the block
 * @param candidates DatanodeStorageInfo/DatanodeInfo to be split
 *        into two sets
 * @param rackMap a map from rack to datanodes
 * @param moreThanOne contains nodes on rack with more than one replica
 * @param exactlyOne remains contains the remaining nodes
 */
public <T> void splitNodesWithRack(
    final Iterable<T> availableSet,
    final Collection<T> candidates,
    final Map<String, List<T>> rackMap,
    final List<T> moreThanOne,
    final List<T> exactlyOne) {
  for(T s: availableSet) {
    final String rackName = getRack(getDatanodeInfo(s));
    List<T> storageList = rackMap.get(rackName);
    if (storageList == null) {
      storageList = new ArrayList<>();
      rackMap.put(rackName, storageList);
    }
    storageList.add(s);
  }
  for (T candidate : candidates) {
    final String rackName = getRack(getDatanodeInfo(candidate));
    if (rackMap.get(rackName).size() == 1) {
      // exactlyOne contains nodes on rack with only one replica
      exactlyOne.add(candidate);
    } else {
      // moreThanOne contains nodes on rack with more than one replica
      moreThanOne.add(candidate);
    }
  }
}
```

Listing C.33: Comment that is placed above if-else

```
/**
 * Given a code point, determine if it should be mangled before being
 * represented in an XML document.
 *
 * Any code point that isn't valid in XML must be mangled.
 * See http://en.wikipedia.org/wiki/Valid_characters_in_XML for a
 * quick reference, or the w3 standard for the authoritative reference.
 *
 * @param cp      The code point
 * @return        True if the code point should be mangled
 */
private static boolean codePointMustBeMangled(int cp) {
  if (cp < 0x20) {
    return ((cp != 0x9) && (cp != 0xa) && (cp != 0xd));
  } else if ((0xd7ff < cp) && (cp < 0xe000)) {
    return true;
  } else if ((cp == 0xfffe) || (cp == 0xffff)) {
    return true;
  } else if (cp == 0x5c) {
    // we mangle backslash to simplify decoding... it's
    // easier if backslashes always begin mangled sequences.
    return true;
  }
  return false;
}
```

Listing C.34: Comment that is placed above for-loop

```
/**
 * Insert all FeedItems of a feed and the feed object itself in a single
 * transaction
 */
public void setCompleteFeed(Feed... feeds) {
    try {
        db.beginTransactionNonExclusive();
        for (Feed feed : feeds) {
            setFeed(feed);
            if (feed.getItems() != null) {
                for (FeedItem item : feed.getItems()) {
                    setFeedItem(item, false);
                }
            }
            if (feed.getPreferences() != null) {
                setFeedPreferences(feed.getPreferences());
            }
        }
        db.setTransactionSuccessful();
    } catch (SQLException e) {
        Log.e(TAG, Log.getStackTraceString(e));
    } finally {
        db.endTransaction();
    }
}
```

Listing C.35: Comment that is placed above while-loop

```
/**
 * Scan the local storage directory, and return the segment containing
 * the highest transaction.
 * @return the EditLogFile with the highest transactions, or null
 * if no files exist.
 */
private synchronized EditLogFile scanStorageForLatestEdits() throws
    ↳ IOException {
  if (!fjm.getStorageDirectory().getCurrentDir().exists()) {
    return null;
  }

  LOG.info("Scanning storage " + fjm);
  List<EditLogFile> files = fjm.getLogFiles(0);

  while (!files.isEmpty()) {
    EditLogFile latestLog = files.remove(files.size() - 1);
    latestLog.scanLog(Long.MAX_VALUE, false);
    LOG.info("Latest log is " + latestLog);
    if (latestLog.getLastTxId() == HdfsServerConstants.INVALID_TXID) {
      // the log contains no transactions
      LOG.warn("Latest log " + latestLog + " has no transactions. " +
          "moving it aside and looking for previous log");
      latestLog.moveAsideEmptyFile();
    } else {
      return latestLog;
    }
  }

  LOG.info("No files in " + fjm);
  return null;
}
```

Listing C.36: Comment that is placed above method calls

```
/*
 * called after the media (featured image) for a post has been downloaded -
     ↳  locate the post
 * and set its featured image url to the passed url
 */
public void mediaChanged(MediaModel mediaModel) {
    // Multiple posts could have the same featured image
    List<Integer> indexList = PostUtils.indexesOfFeaturedMediaIdInList(
        ↳ mediaModel.getMediaId(), mPosts);
    for (int position : indexList) {
        PostModel post = getItem(position);
        if (post != null) {
            String imageUrl = mediaModel.getUrl();
            if (imageUrl != null) {
                mFeaturedImageUrls.put(post.getId(), imageUrl);
            } else {
                mFeaturedImageUrls.remove(post.getId());
            }
            notifyItemChanged(position);
        }
    }
}
```

Listing C.37: Comment for code with many assignments

```
/**
 * Returns a cursor containing query strings previously typed by the user
 * @param filter - filters the list using LIKE syntax (pass null for no
     ↳ filter)
 * @param max - limit the list to this many items (pass zero for no limit)
 */
public static Cursor getQueryStringCursor(String filter, int max) {
    String sql;
    String[] args;
    if (TextUtils.isEmpty(filter)) {
        sql = "SELECT * FROM tbl_search_suggestions";
        args = null;
    } else {
        sql = "SELECT * FROM tbl_search_suggestions WHERE query_string LIKE
            ↳  ?";
        args = new String[]{filter + "%"};
    }

    sql += " ORDER BY date_used DESC";

    if (max > 0) {
        sql += " LIMIT " + max;
    }

    return ReaderDatabase.getReadableDb().rawQuery(sql, args);
}
```

Listing C.38: Comment for code with multiple declarations

```
/**
 * Converts the given string into an array of lines. The lines
 * don't contain any line delimiter characters.
 *
 * @return the string converted into an array of strings. Returns <code>
 *  null</code> if the input string can't be converted in an array of lines
     ↳ .
 */
public static String[] convertIntoLines(String input) {
    try {
        ILineTracker tracker= new DefaultLineTracker();
        tracker.set(input);
        int size= tracker.getNumberOfLines();
        String result[]= new String[size];
        for (int i= 0; i < size; i++) {
            IRegion region= tracker.getLineInformation(i);
            int offset= region.getOffset();
            result[i]= input.substring(offset, offset + region.getLength())
                ↳ ;
        }
        return result;
    } catch (BadLocationException e) {
        return null;
    }
}
```

Listing C.39: Comment for code with try-catch block

```
/**
 * Check if a file is writable. Detects write issues on external SD card.
 *
 * @param file The file
 * @return true if the file is writable.
 */
public static boolean isWritable(final File file) {
    if (file == null)
        return false;
    boolean isExisting = file.exists();

    try {
        FileOutputStream output = new FileOutputStream(file, true);
        try {
            output.close();
        } catch (IOException e) {
            // do nothing.
        }
    } catch (FileNotFoundException e) {
        return false;
    }
    boolean result = file.canWrite();

    // Ensure that file is not created during this process.
    if (!isExisting) {
        file.delete();
    }

    return result;
}
```

Listing C.40: Comment whose Javadoc only describes input

```
/**
 * Perform a sanity check on the packet, returning true if it is sane.
 * @param lastSeqNo the previous sequence number received - we expect the
 *                  current sequence number to be larger by 1.
 */
public boolean sanityCheck(long lastSeqNo) {
  // We should only have a non-positive data length for the last packet
  if (proto.getDataLen() <= 0 && !proto.getLastPacketInBlock()) return
      ↳ false;
  // The last packet should not contain data
  if (proto.getLastPacketInBlock() && proto.getDataLen() != 0) return false
      ↳ ;
  // Seqnos should always increase by 1 with each packet received
  return proto.getSeqno() == lastSeqNo + 1;
}
```

Listing C.41: Comment whose Javadoc only describes output

```
/**
 * Returns <code>true</code> if the given string only consists of
 * white spaces according to C. If the string is empty, <code>true
 * </code> is returned.
 *
 * @return <code>true</code> if the string only consists of white
 *  spaces; otherwise <code>false</code> is returned
 *
 * @see java.lang.Character#isWhitespace(char)
 */
public static boolean containsOnlyWhitespaces(String s) {
    int size= s.length();
    for (int i= 0; i < size; i++) {
        if (!Character.isWhitespace(s.charAt(i)))
            return false;
    }
    return true;
}
```

Listing C.42: Comment that refers to external documentation

```
/**
 *   Compares two Strings, and returns the portion where they differ.  (
     ↳ More precisely,
 *   return the remainder of the second String, starting from where it's
     ↳ different from the first.)
 *
 *   For example, difference("i am a machine", "i am a robot") -> "robot".
 *
 *   StringUtils.difference(null, null) = null
 *   StringUtils.difference("", "") = ""
 *   StringUtils.difference("", "abc") = "abc"
 *   StringUtils.difference("abc", "") = ""
 *   StringUtils.difference("abc", "abc") = ""
 *   StringUtils.difference("ab", "abxyz") = "xyz"
 *   StringUtils.difference("abcde", "abxyz") = "xyz"
 *   StringUtils.difference("abcde", "xyz") = "xyz"
 *
 *  @param str1 - the first String, may be null
 *  @param str2 - the second String, may be null
 *  @return the portion of str2 where it differs from str1; returns the
     ↳ empty String if they are equal
 *
 *  Stolen from Apache's StringUtils
 *  (https://commons.apache.org/proper/commons-lang/javadocs/api-2.6/org/
     ↳ apache/commons/lang/StringUtils.html#difference(java.lang.String
     ↳ ,%20java.lang.String))
 */
public static String differenceStrings(String str1, String str2) {
    if (str1 == null) return str2;
    if (str2 == null) return str1;

    int at = indexOfDifferenceStrings(str1, str2);

    if (at == INDEX_NOT_FOUND) return "";

    return str2.substring(at);
}
```

Listing C.43: Comment that uses multiple abbreviations

```
/**
 * Get a URI for each configured nameservice. If a nameservice is
 * HA-enabled, then the logical URI of the nameservice is returned. If the
 * nameservice is not HA-enabled, then a URI corresponding to an RPC
     ↳ address
 * of the single NN for that nameservice is returned, preferring the
     ↳ service
 * RPC address over the client RPC address.
 *
 * @param conf configuration
 * @return a collection of all configured NN URIs, preferring service
 *         addresses
 */
public static Collection<URI> getNsServiceRpcUris(Configuration conf) {
  return getNameServiceUris(conf,
      DFSConfigKeys.DFS_NAMENODE_SERVICE_RPC_ADDRESS_KEY,
      DFSConfigKeys.DFS_NAMENODE_RPC_ADDRESS_KEY);
}
```

Listing C.44: Comment that includes math

```
/**
* Returns the value of this {@code UnsignedLong} as a {@code long}. This is
    ↳  an inverse operation
* to {@link #fromLongBits}.
*
* <p>Note that if this {@code UnsignedLong} holds a value {@code >= 2^63},
    ↳ the returned value
* will be equal to {@code this - 2^64}.
*/
@Override
public long longValue() {
    return value;
}
```

Appendix D

Project metrics

Tables D.1–D.11 in this appendix list the metrics whose mean values differ most noticeably from the overall mean across all projects, i.e. they show a difference of at least 50% relative to the overall mean. Features are sorted in order of ascending relative difference from the overall mean across all projects.

Table D.1: Large differences in comment metrics for AFWall+

Metric	Mean	Diff (%)	Std.	Min	Max
mentions_parents	0.000000	-100.00	0.000000	0.000	0.000
is_for_package	0.000000	-100.00	0.000000	0.000	0.000
is_for_enum	0.000000	-100.00	0.000000	0.000	0.000
is_for_annotation	0.000000	-100.00	0.000000	0.000	0.000
is_for_interface	0.002433	-85.73	0.049326	0.000	1.000
hyperlinks	0.002433	-69.47	0.049326	0.000	1.000
is_block_comment	0.031630	-66.54	0.175227	0.000	1.000
variable_assignments	1.009732	56.95	3.045973	0.000	39.000
flesch_ease	63.401708	60.78	43.034664	-300.815	122.086
is_for_attribute	0.313869	68.91	0.464629	0.000	1.000
try_catch_blocks	0.145985	103.11	0.595186	0.000	8.000
is_line_comment	0.240876	113.76	0.428136	0.000	1.000

Table D.2: Large differences in comment metrics for Amaze File Manager

Metric	Mean	Diff (%)	Std.	Min	Max
mentions_parents	0.000000	-100.00	0.000000	0.000	0.000
is_for_package	0.000000	-100.00	0.000000	0.000	0.000
is_for_annotation	0.000000	-100.00	0.000000	0.000	0.000
is_for_interface	0.000000	-100.00	0.000000	0.000	0.000
is_block_comment	0.014414	-84.75	0.119299	0.000	1.000
flesch_ease	59.999829	52.15	49.163316	-469.999	122.108
operators	2.987387	53.90	7.492826	0.000	125.000
control_structures	2.034234	54.93	4.057428	0.000	42.000
variable_declaration	1.333333	60.05	2.749248	0.000	24.000
loop_structures	0.174775	61.44	0.600912	0.000	5.000
method_length	10.450450	61.91	18.715782	0.000	159.000
variable_assignments	1.108108	72.24	2.756520	0.000	47.000
method_calls	6.162162	78.42	12.813816	0.000	131.000
is_line_comment	0.201802	79.08	0.401707	0.000	1.000
tasks	0.018018	114.18	0.146066	0.000	2.000
is_for_enum	0.003604	144.16	0.059976	0.000	1.000
try_catch_blocks	0.190991	165.72	0.575923	0.000	5.000
hyperlinks	0.021622	171.29	0.188766	0.000	2.000

Table D.3: Large differences in comment metrics for AntennaPod

Metric	Mean	Diff (%)	Std.	Min	Max
tasks	0.000000	-100.00	0.000000	0.0	0.0
mentions_parents	0.000000	-100.00	0.000000	0.0	0.0
is_for_package	0.000000	-100.00	0.000000	0.0	0.0
is_for_annotation	0.000000	-100.00	0.000000	0.0	0.0
is_block_comment	0.021569	-77.18	0.145341	0.0	1.0
is_line_comment	0.041176	-63.46	0.198796	0.0	1.0
is_for_class	0.171569	62.47	0.377190	0.0	1.0

Table D.4: Large differences in comment metrics for Apache Hadoop

Metric	Mean	Diff (%)	Std.	Min	Max
is_for_annotation	0.000000	-100.00	0.000000	0.0	0.0
tasks	0.001631	-80.61	0.040363	0.0	1.0
is_block_comment	0.021533	-77.22	0.145178	0.0	1.0
hyperlinks	0.002610	-67.25	0.057069	0.0	2.0

Table D.5: Large differences in comment metrics for Apache Spark

Metric	Mean	Diff (%)	Std.	Min	Max
mentions_parents	0.000000	-100.00	0.000000	0.000	0.000
is_for_constructor	0.000000	-100.00	0.000000	0.000	0.000
is_for_enum	0.000000	-100.00	0.000000	0.000	0.000
hyperlinks	0.000000	-100.00	0.000000	0.000	0.000
is_block_comment	0.009709	-89.73	0.098533	0.000	1.000
omitted_full_stops	0.067961	-87.94	0.240491	0.000	1.000
describes_inputs	0.140777	-66.16	0.345967	0.000	1.000
describes_output	0.184466	-56.15	0.389760	0.000	1.000
math_symbols	0.135922	56.62	0.714672	0.000	5.000
abbreviations	0.368932	73.05	1.481853	0.000	14.000
flesch_ease	71.663126	81.73	21.714331	1.265	122.004
is_for_interface	0.087379	412.58	0.283770	0.000	1.000
is_for_annotation	0.009709	839.74	0.098533	0.000	1.000
is_for_package	0.048544	4960.12	0.215963	0.000	1.000

Table D.6: Large differences in comment metrics for Eclipse CDT

Metric	Mean	Diff (%)	Std.	Min	Max
is_for_package	0.000000	-100.00	0.000000	0.000	0.000
flesch_ease	3.119397	-92.09	163.204541	-1096.027	122.154
is_for_annotation	0.000247	-76.11	0.015710	0.000	1.000
is_for_enum	0.000494	-66.56	0.022214	0.000	1.000
hyperlinks	0.002715	-65.94	0.052039	0.000	1.000
is_for_interface	0.032330	89.65	0.176896	0.000	1.000

Table D.7: Large differences in comment metrics for Google Guava

Metric	Mean	Diff (%)	Std.	Min	Max
is_block_comment	0.007576	-91.99	0.086791	0.0	1.000
method_calls	1.547348	-55.20	2.664980	0.0	23.000
operators	0.903409	-53.46	3.167696	0.0	44.000
hyperlinks	0.003788	-52.47	0.061487	0.0	1.000
tokens	30.081439	52.67	43.494236	1.0	409.000
extra_info_score	3.650498	69.49	6.484463	0.0	54.733
tasks	0.015152	80.10	0.122271	0.0	1.000
is_for_package	0.001894	97.42	0.043519	0.0	1.000
math_symbols	0.200758	131.33	0.684221	0.0	4.000
is_for_enum	0.003788	156.65	0.061487	0.0	1.000
is_for_annotation	0.005682	449.96	0.075235	0.0	1.000
mentions_parents	0.005682	862.43	0.075235	0.0	1.000

Table D.8: Large differences in comment metrics for Google Guice

Metric	Mean	Diff (%)	Std.	Min	Max
is_block_comment	0.000000	-100.00	0.000000	0.0	0.0
tasks	0.000000	-100.00	0.000000	0.0	0.0
mentions_parents	0.000000	-100.00	0.000000	0.0	0.0
is_line_comment	0.013245	-88.25	0.114703	0.0	1.0
is_for_attribute	0.033113	-82.18	0.179526	0.0	1.0
try_catch_blocks	0.013245	-81.57	0.162758	0.0	2.0
describes_output	0.125828	-70.09	0.332759	0.0	1.0
math_symbols	0.026490	-69.48	0.229406	0.0	2.0
operators	0.609272	-68.61	1.478619	0.0	9.0
variable_assignments	0.231788	-63.97	0.778406	0.0	4.0
omitted_full_stops	0.215358	-61.80	0.309525	0.0	1.0
variable_declaration	0.331126	-60.25	0.991442	0.0	5.0
describes_inputs	0.165563	-60.21	0.372925	0.0	1.0
method_length	2.920530	-54.75	5.769487	0.0	40.0
method_calls	1.589404	-53.98	3.190134	0.0	21.0
abbreviations	0.099338	-53.41	0.443546	0.0	4.0
hyperlinks	0.013245	66.19	0.114703	0.0	1.0
is_for_enum	0.006623	348.71	0.081379	0.0	1.0
is_for_interface	0.079470	366.19	0.271371	0.0	1.0
is_for_package	0.033113	3351.60	0.179526	0.0	1.0
is_for_annotation	0.039735	3746.07	0.195986	0.0	1.0

Table D.9: Large differences in comment metrics for ownCloud

Metric	Mean	Diff (%)	Std.	Min	Max
is_for_package	0.000000	-100.00	0.000000	0.0	0.0
is_for_enum	0.000000	-100.00	0.000000	0.0	0.0
is_for_annotation	0.000000	-100.00	0.000000	0.0	0.0
is_for_interface	0.000803	-95.29	0.028330	0.0	1.0
is_block_comment	0.008828	-90.66	0.093581	0.0	1.0
math_symbols	0.023274	-73.18	0.220122	0.0	3.0
method_length	9.694222	50.19	15.829577	0.0	163.0
operators	2.917335	50.29	6.002709	0.0	83.0
describes_output	0.638844	51.86	0.480528	0.0	1.0
variable_assignments	1.043339	62.17	2.565082	0.0	40.0
method_calls	6.247191	80.88	14.204017	0.0	324.0
tasks	0.031300	272.06	0.203936	0.0	3.0

Table D.10: Large differences in comment metrics for Vaadin

Metric	Mean	Diff (%)	Std.	Min	Max
mentions_parents	0.000000	-100.00	0.000000	0.0	0.0
is_for_package	0.000000	-100.00	0.000000	0.0	0.0
is_line_comment	0.009346	-91.71	0.096271	0.0	1.0
hyperlinks	0.001038	-86.97	0.032225	0.0	1.0
variable_declaration	0.398754	-52.13	1.601350	0.0	23.0
is_for_annotation	0.003115	201.54	0.055757	0.0	1.0

Table D.11: Large differences in comment metrics for WordPress

Metric	Mean	Diff (%)	Std.	Min	Max
mentions_parents	0.000000	-100.00	0.000000	0.000	0.00
is_for_package	0.000000	-100.00	0.000000	0.000	0.00
is_for_annotation	0.000000	-100.00	0.000000	0.000	0.00
is_for_interface	0.001373	-91.95	0.037037	0.000	1.00
is_for_constructor	0.007550	-80.48	0.086590	0.000	1.00
describes_inputs	0.168725	-59.45	0.373043	0.000	1.00
coverage	0.202103	-51.17	0.348048	0.000	1.00
describes_output	0.207275	-50.73	0.405494	0.000	1.00
flesch_ease	61.193376	55.18	60.186056	-1315.976	122.09
control_structures	2.040494	55.41	3.190238	0.000	40.00
is_line_comment	0.203844	80.90	0.402992	0.000	1.00
is_for_enum	0.002745	86.01	0.052342	0.000	1.00
is_block_comment	0.367193	288.43	0.482205	0.000	1.00
hyperlinks	0.038435	382.25	0.228239	0.000	3.00