b
UNIVERSITÄT
BERN

Adherence of class comments to style guidelines

Bachelor's Thesis

Suada Abukar

from

University of Bern

Faculty of Science, University of Bern

31.08.2021

Prof. Dr. Oscar Nierstrasz

Pooja Rani

Software Composition Group

Institute of Computer Science

University of Bern, Switzerland

Abstract

Code comments are essential for program comprehension and maintenance tasks. They are written in natural language and are either semi-structured or unstructured. As a result, determining their quality is a difficult task. To control certain aspects of quality such as consistency, readability, or preciseness, programming languages provide comment-related conventions in the coding style guidelines. One of the ways to assess comment quality in the aforementioned aspects is to verify whether or not the code adheres to the respective coding style guidelines. However, what specific types of conventions they suggest related to code comments and if developers follow these conventions while writing comments is not yet explored.

Previous works have proposed to automatically assess code quality using various linters or static tools. However, the extent to which these tools support comments is limited and comment validation on a semantic level is not provided. Additionally, one project can follow more than one style guideline. Thus, verifying which convention is from which guideline and to what extent it is followed is an essential but nontrivial task. This thesis provides an empirical study investigation of the content of popular commenting style guidelines and commenting practices in Java and Python. We extract comment-related rules from style guidelines used by 13 open-source projects. Furthermore, we assess nearly 700 statistically significant samples of class comments originating from these 13 projects. The projects vary in domain, size, and number of contributors and are selected from two popular programming languages: Java and Python.

This thesis uncovers the quality of class comments written in open-source projects and the content of the comment style guide. We discovered that 57% of the comment conventions rules do not apply to the class comment samples. From the applicable portion, 83% of class comments follow the convention rules. The rules that are followed by the comments address predominantly the content and writing style of comments. On the other side, rules addressing the structure of a comment are often violated.

Our results highlight the importance of writing clear and straightforward rules in the style guidelines since they are used and interpreted by developers with different levels of coding experience. In addition, the high percentage of adherence proves that developers do consult style guidelines when coding.

Contents

1 Introduction

Software documentation assists developers in understanding various details about the software. Studies have shown that documentation is appreciated by developers and that it reduces the time required for maintenance tasks of the software [9]. It exists in various forms in software artifacts such as being separated from the source code (wikis, design documents) or embedded in the code (code comments). Code comments help developers gain an overview and understanding of the code without having to read the code beforehand.

Code comments are one of the main forms of code documentation. They can contain various types of information about the code. Previous work has highlighted these information types in Java, Python, and Smalltalk [4, 5, 13]. Some information types are written in comments using directives or tags, e.g., :return:, @param. In contrast, some are written implicitly in the comment without using any specific tag. As code comments are written in natural language, writing high-quality code comments is difficult. To write high-quality comments, several coding style guidelines are suggested by various programming languages and organizations.

Coding style guides, also known as code conventions or coding standards, are a set of rules or suggestions to write code including comments. They have the purpose to improve the consistency, readability, and maintainability of the code [8]. They provide comment guidelines or rules about where a comment should be located within the source code, for example, class comments in Java should precede the class definition. Similarly, they suggest how a comment should be formatted, for instance, whether it

should have the same indentation as the surrounding code and whether spaces or tabs should be used to indent. In addition, the guidelines describe how various components of a comment should be written and which writing style to chose when documenting the source code.

Projects can follow various standard guidelines for comments or choose to follow their own guidelines. Previous works have investigated code conventions in various projects but did not study comment conventions in detail. Therefore, it is unknown in the context of comments whether the various style guidelines complement each other or contain conflicting guidelines. To obtain this understanding, we formulate our first research question: **RQ_1:** *What do coding style guidelines suggest about comments?*
We want to find out which kinds of conventions various style guidelines suggest for comments. We analyze standard guidelines and project-specific guidelines from two popular languages, Java and Python. Rani *et al.* investigated comments of diverse Java and Python projects that vary in domain, size, and the number of contributors [5]. We use their projects to analyze project-specific guidelines. We identify all comment-related rules from these guidelines and categorize them into a taxonomy, extended from the taxonomy proposed by Rani *et al.* [6].

Even though the coding style guidelines suggest various comment-related guidelines, it is unknown if developers follow these guidelines in writing comments or not. To have high-quality comments, it is essential to know if comments adhere to the comment guidelines or not.

Linters, also known as style checkers, consist of rules from various style guidelines or introduce their own. They can validate code or its comments against the rules and point out rule violations [3]. However, these tools provide limited support to verify the quality of comments. For example, they can verify whether the comment for a code entity is present or not but they do not check its content. *i.e.*, Checkstyle can be configured to verify rules from the Google Java style guide.[1] However, only some rules from the style guide are covered partially and many are not covered at all by Checkstyle.[2] As they are not able to detect violations for the content of comments, there is no knowledge available regarding the quality of comment components and whether these components adhere to style guidelines or not.

To gain this understanding, we formulate another question in this thesis: **RQ_2:** *To what extent do developer's class comments follow style guidelines?*
As mentioned previously, the adherence of comments to the style guidelines is not yet explored, so we use the list of comment-related rules from **RQ_1** and validate them against the comments of Java and Python projects. Since various types of code comments exist in code, for instance, documentation or implementation comments, we set our focus on a specific type of documentation comments *i.e.*, class comments. Rani *et al.* investigated class comments of diverse Java and Python projects, and we use their dataset for this RQ [5]. In the validation of comments against the rules, we define various conditions for each rule that has to be fulfilled for the rule to be considered as *followed*, *not followed*, or *not applicable*.

Overall, this thesis provides an empirical study investigating various style guidelines for comments and

[1] `https://checkstyle.sourceforge.io/index.html` accessed August 31, 2021
[2] `https://checkstyle.org/google_style.html` accessed August 31, 2021

developers commenting practices in adopting them in their comments for Java and Python. In the first step, we extract comment-related rules from style guidelines suggested by Java and Python open-source projects and curate a list of all the code comment-related rules from them. In the second step, we categorize and prioritize the extracted rules and develop conditions under which they can be verified for comments. In the last step, we validate the comments from the dataset with the defined conditions.

We found that the comment-related rules that style guidelines contain are mostly of the type *Content* (32% for Java and 40% for Python) and *Formatting* (19% for Java and 21% for Python). According to our results, a significant portion of the rules does not apply to the comments, mainly due to rules addressing information that is missing in the comment. However, 83% of Java comments and 82% of Python comments adhere to the rules. The rules that are often followed by comments are of type *Writing style* and *Content*. On the other side, *Structure* rules are often violated.

Contribution of the thesis: The thesis provides insights into various comment aspects as following:

- we learn what popular Java and Python style guidelines say about code comments. We present this in the form of a taxonomy.
- we highlight various weak points in the style guidelines that need to be improved to have more coherent, adequate, and precise comment conventions.
- we show the adherence of certain rule types against comments and highlight what tools can focus on to improve the code comment quality.
- we provide a labeled dataset of comments indicating if they follow the rules or not.[3]

The rest of the thesis is structured as follows. In Chapter 2 we discuss related research. In Chapter 3 we take a close look at style guides and what rules they contain about comments. Chapter 4 follows up with a validation of our class comment data set with the in Chapter 3 extracted rules, to analyze the adherence to the style guide. Lastly, we point out threats to validity in Chapter 5 and conclude in Chapter 6.

[3] `https://doi.org/10.5281/zenodo.5296443`

2

Related Work

2.1 Comment analysis works

Understanding source code is important to developers to have well-maintained software. Code comments play a role in understanding source code. Previous research has analyzed code comments of a variety of programming languages and demonstrated how critical, high-quality code comments are for supporting program understanding activities and increasing the effectiveness of maintenance duties [6]. A few studies explored the content inside comments and developer commenting practice to write various information types in comments [4, 6, 13]. Pascarella *et al.* investigated Java comments [4] and Zhan *et al.* investigated Python comments [13]. Both devised a taxonomy for code comments and found that the summary is predominantly present in code comments.

Rani *et al.* lied in another work the focus especially on class comments and the different types of information they contain [5]. They investigated the class comments from Java, Python, and Smalltalk and produced a taxonomy for the classification of comment types. The classification consists of information types which are types of semantic information. They analyzed the characteristics of the information types present in the class comments written in the languages mentioned above and created a tool that can accurately and automatically identify such information types.

Though measuring the quality of these information types is critical, it is not well-supported or

investigated for many other popular languages. Steidl *et al.* measure the quality of comments based on specific quality attributes such as consistency and coherence [10]. They propose a model for comment quality, that is based on a categorization of comments done with machine learning. In the paper, they focused on comments from Java and C/C++ programs. They did a survey to test the validity of their metrics. Their metrics capture the coherence between comments and code and the length of comments.

2.2 Coding style guidelines

To control the quality of comments, various coding style guidelines are proposed. A few studies have analyzed the style guidelines in the context of code but none of them explored the style guidelines specifically for comments in Java and Python or compared the developer's commenting practices to standard guidelines [1–3, 7, 11].

Steinbeck and Koschke investigated what types of Javadoc violations exist and which elements in the source code (class, interface, annotation, enum, field, method, constructor) are prone to be affected by Javadoc violations [11]. The authors examined Javadoc comments from 163 different open-source projects and further analyzed the longevity of the Javadoc violations observed in the projects. Steinbeck and Koschke developed a tool for the detection of the Javadoc violations. However, the tool is limited to detecting violations caused by missing or incorrect components. For simplicity, the authors evaluated only the length of the description of comment components. Therefore, whether the descriptions are meaningful and relevant to the code is not evaluated, which leads to semantic violations caused by such components not being detected. In this thesis, we manually validate the descriptions of comment components up to a semantic level.

Bafatakis *et al.* evaluated the coding style compliance of Python code snippet fragments on Stack Overflow with the style checking tools, Pylint and Flake8 which cover the PEP 8 style guide and flag violations [1]. However, the authors considered docstrings as unnecessary to their evaluation and excluded missing-docstring violations detectable by Pylint. Their focus was dedicated to Python code and not necessarily Python code comments which is relatable since the interest in Stack Overflow lies more on code than comments.

2.3 Adherence of conventions to coding style guidelines

Smith *et al.* investigated whether adherence to code conventions influences software maintainability [9]. They studied 71 coding conventions adapted from Checkstyle documentation and surveyed developers to rate the importance of these 71 conventions. They examined the adherence of open-source software projects to these conventions over their project lifetime. For evaluation of adherence, the authors used the linter, Checkstyle. However, their focus was more on the general side of coding convention and not on the comment conventions. We focus on the conventions related to comments.

Ueda *et al.* point out bad practices in the use of coding rule violation detection tools [12]. They

highlight that from all detected violations 80% are not fixed by developers. One of the reasons is due to the negligence of tool customization to the software project. Thus, amongst all detected violations, those deemed irrelevant to the developers are featured as well. In this paper, Ueda *et al.* took up the task to create an automatic customization of ASATs (Automatic Static Analysis Tools) to project source codes. Their tool takes source code as an input and generates an ASAT with a configuration that might be of interest to the developer. According to their results, their tool assists developers to set their focus on rule violations relevant to them. This is particularly important since, according to Ueda *et al.*, a high percentage (80%) of detected code violations are ignored by developers. We will be investigating whether this applies to class comments too.

Simmons *et al.* investigated the adherence of open-source Data Science projects to coding standards and how they differ from non-Data Science projects [8]. They did a comparison between 1048 Data Science and 1099 non-Data Science projects, all written in Python, and evaluated the adherence by using Pylint. They found that Data Science projects have numerous functions rich with parameters and local variables. Additionally, Data Science projects use a different naming convention for variables than non-Data Science projects. Simmons *et al.* did not focus on whether code comments adhere to the code convention. Thus, the paper examined the adherence with a style checker which is currently unable to validate the semantic information in comments.

Rani *et al.* investigated the commenting practices in Pharo and found 23 information types from Smalltalk class comments [6]. From the 23 information types observed in Pharo class comments, only 7 were suggested by the default class comment template that is automatically generated when creating a new class. In addition, they assessed the extent to which developer commenting practices adhere to the default template. We use the same terminology when referring to information types and follow the same methodology to verify the adherence of class comments in Java and Python to the style guidelines. Similar to their results, we found that Java and Python class comments follow similar types of rules and violate structure and syntax rules.

3

Comment Conventions

3.1 Introduction

Code comments help developers gain an overview of the source code and make code maintainable. Therefore, if code comments are well-formatted and written consistently, developers will have a better understanding of the code and the software maintenance cost can be reduced [9]. To help developers write code comments, coding style guidelines suggest what code comment components are expected and what information a code comment should contain. Coding style guidelines consist of conventions and best practices on how to write good code comments and ensure consistency in style and formatting throughout the code base. However, what conventions the style guidelines suggest for comments is not yet explored. Therefore, we study the research question $\mathbf{RQ}_1$**:** *What do coding style guidelines suggest about comments?*

To find out what coding style guidelines suggest about code comments, we first need to investigate what kind of coding style guidelines are used by developers and applied to projects. For that, we identify the coding style guidelines various Java and Python open-source projects recommend to their developer to follow when contributing to the project. We consider six Java projects which were previously used by Pascarella *et al.* for their investigation in understanding what types of code comments developers write and the purpose those code comments have [4]. Additionally, we consider code comments from seven Python projects studied by Zhan *et al.* for the classification of Python code comments [13]. Rani *et al.* used projects of both papers and created a dataset with only class comments [5]. From each coding style

guideline suggested by these 13 projects, we extract all comment-related rules with the intention to filter afterward for comment-related rules that are applicable for class comments. In the next step, we categorize the extracted comment-related rules according to the initial rule type taxonomy proposed by Rani *et al.* [6]. Thus, we categorize all rules into five rule types as *Formatting*, *Content*, *Syntax*, *Structure*, and *Writing style*.

For the majority of the projects, we found the Sun Java Coding Convention and Google Java Style Guide to be standard coding style guidelines in Java. The Python projects mention PEP 8/257, NumpyDoc, and the Google Python Style Guide as standard style guidelines. Interestingly, some projects follow their own style guidelines on top of the standard style guidelines. Smit *et al.* referred to these project-specific rules as *self-imposed standards* [9]. The number of comment-related rules in the coding style guidelines vary as not all projects provide the same details for comment conventions. For instance, the Sun/Oracle Java Code convention has a whole coding style guideline document dedicated to code comments while the Google Java Style Guide has dedicated a small section to code comments. Lastly, after the categorization of the extracted rules, we discover that the *Content* and *Formatting* rules are predominantly present in the coding style guidelines of both languages. On the other hand, *Writing style* rules make a small fraction of all extracted rules in both languages, Java and Python.

3.2 Methodology

3.2.1 Data collection

To answer RQ_1, we use the dataset of projects studied by Rani *et al.* for the analysis of information types present in Java and Python class comments. The dataset consists of class comments extracted from six Java and seven Python projects. The selection of the open-source projects was done by Pascarella *et al.* [4] (for Java projects) and Zhan *et al.* [13] (for Python projects) based on the code base size, the number of contributors, domain, the number of stars on GitHub (for Python projects), all of them being publicly available on GitHub, and due to Java and Python being popular programming languages. However, Rani *et al.* [5] extracted the class comments from those 13 projects. Table 3.1 describes the dataset in terms of the projects and analyzed sample classes and their class comments. Notice that the number of classes and class comments are equal in Python but not in Java. Previous work combined the inner class comments with the main class comments while in our case we separate them to analyze if there is any difference. More information about the data pre-processing can be found in Section 3.2.2.

A snapshot was made from all projects at the time of the analysis done by Rani *et al.* (end of 2019). Further details are documented in Section 7.1.2. Since the web pages of the style guidelines display their latest version, we made sure to extract the rules from the style guideline version that was the latest at the time of Rani *et al.*'s [5] analysis.

Language	Project	# classes	# class comments
Java	Hadoop	153	195
	Eclipse.cdt	110	136
	Vaadin	41	85
	Guava	49	59
	Guice	10	19
	Spark	13	14
Python	Django	108	108
	Pipenv	107	107
	Pytorch	49	49
	Pandas	35	35
	Mailpile	25	25
	iPython	22	22
	Requests	4	4

Table 3.1: Java and Python projects used in our analysis

3.2.2 Tweaking the Dataset

In the dataset we use in this thesis, we have in addition to Java class comments also class comments of inner classes. Each Java file in a Java project has one entry with all class comments from that file. We split such entries of Java files by inner classes. To do this, we wrote a parser that splits class comments and preserves for each class comment the source file name. In Figure 3.1 you can see an example of an entry with several comments separated with a pipe character. All three comments were written in the *Utils.java* source file. The source file with the class and inner class comments depicted green is displayed in Figure 3.2. More details about the parser can be found in the replication package.

```
Utils.java

* A utility class. It provides
* A path filter utility to filter out output/part files in the output dir
|
* This class filters output(part) files from the given directory
* It does not accept files with filenames _logs and _SUCCESS.
* This can be used to list paths of output directory as follows:
* Path[] fileList = FileUtil.stat2Paths(fs.listStatus(outDir,
* new OutputFilesFilter()));
|
* This class filters log files from directory given
* It doesnt accept paths having _logs.
* This can be used to list paths of output directory as follows:
* Path[] fileList = FileUtil.stat2Paths(fs.listStatus(outDir,
* new OutputLogFilter()));
```

Figure 3.1: Class comment from Hadoop (not split)

```java
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you 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 org.apache.hadoop.mapred;
import org.apache.hadoop.classification.InterfaceAudience;
import org.apache.hadoop.classification.InterfaceStability;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.PathFilter;
/**
 * A utility class. It provides
 *    A path filter utility to filter out output/part files in the output dir
 */
@InterfaceAudience.Public
@InterfaceStability.Stable
public class Utils {
  public static class OutputFileUtils {
    /**
     * This class filters output(part) files from the given directory
     * It does not accept files with filenames _logs and _SUCCESS.
     * This can be used to list paths of output directory as follows:
     *   Path[] fileList = FileUtil.stat2Paths(fs.listStatus(outDir,
     *                                         new OutputFilesFilter()));
     */
    public static class OutputFilesFilter extends OutputLogFilter {
      public boolean accept(Path path) {
        return super.accept(path)
               && !FileOutputCommitter.SUCCEEDED_FILE_NAME
                   .equals(path.getName());
      }
    }

    /**
     * This class filters log files from directory given
     * It doesnt accept paths having _logs.
     * This can be used to list paths of output directory as follows:
     *   Path[] fileList = FileUtil.stat2Paths(fs.listStatus(outDir,
     *                                       new OutputLogFilter()));
     */
    public static class OutputLogFilter implements PathFilter {
      public boolean accept(Path path) {
        return !"_logs".equals(path.getName());
      }
    }
  }
}
```

Figure 3.2: Utils.java

3.2.3 Extracting comment related rules

For each project, we search for the mention of the style guidelines developers are expected to follow when contributing to a project. We mainly checked the project web pages and their GitHub repositories. For instance, for Hadoop and Eclipse.cdt the style guideline is mentioned on the project page.

Once we find which coding style guidelines the project recommends, we scan the whole guidelines and their associated pages to find comment-related rules or conventions. To preserve the context of the rules, section titles and examples given in the guideline were extracted as well. As one sentence can talk about different types of comments (class, function, variables) or parts of comments (summary, parameters), we split the rules based on types of comments. Thus, we extracted all comment-related rules.

After the extraction of all rules, we categorize the rules, adapting a taxonomy proposed by Rani *et*

al. [6]. They propose two main categories, namely *Content* and *Writing style*. The *Content* type defines the information types written inside the comment and the *Writing style* type defines the rules or practice about the grammar and punctuation. Besides these two types, we extended the taxonomy with three more categories: *Formatting*, *Syntax*, and *Structure*, to cover all aspects of comments. In case the rule does not fit in any of the mentioned categories, we classified it as *Other*. The rule categorization is discussed in more detail in Section 7.1.5.

3.3 Results

After finding the coding style guideline and extracting the comment-related rules, we categorized them according to the described taxonomy.

3.3.1 Style guidelines

We found that of the six Java projects four suggest its developers adhere to the Sun Java Code Convention and the other two projects suggest the Google Java Style Guide. Table 3.2 shows the selected Java and Python projects and the style guidelines they follow. For the Python projects, we found three main standard style guides, PEP 8/257, NumpyDoc, and Google Python Style Guide. The iPython project recommends developers follow two style guides, PEP 8/257 and NumpyDoc. There are specific projects that do not refer to any specific guidelines of their own. For example, Guava and Guice in Java and Pipenv and Mailpile in Python do not define project-specific guidelines as shown in Table 3.2.

Finding 1. *The Java projects suggest the Sun Java Code Convention and Google Java Style Guide for writing code comments. The Python projects suggest PEP 8/257, Google Python Style Guide, and Numydoc.*

Language	Project	Project-specific guideline	Style guideline
Java	Eclipse.cdt	✓	Oracle
	Hadoop	✓	Oracle
	Vaadin	✓	Oracle
	Spark	✓	Oracle
	Guava	×	Google
	Guice	×	Google
Python	Django	✓	PEP 8/257
	Requests	✓	PEP 8/257
	Pipenv	×	PEP 8/257
	Mailpile	×	PEP 8/257
	iPython	✓	PEP 8/257, Numpydoc
	Pandas	✓	Numpydoc
	Pytorch	✓	Google

Table 3.2: Overview of the Java and Python projects and the style guidelines they use.

3.3.2 Rule types

According to the taxonomy, we found that the majority of the rules for all projects were about *Formatting* and *Content*. Figure 3.3 displays a similar distribution of rules types in the Java and Python style guides with the exception of the percentage of Java *Structure* rules being slightly higher than the percentage of Java *Syntax* rules. For both languages *Writing style* rules are the least present in the style guides. The definition of each rule type can be found in Section 7.1.5.

Finding 2. *Most rules present in style guidelines, for Java and Python, are of type Content followed by Formatting. Writing style rules are the least present.*

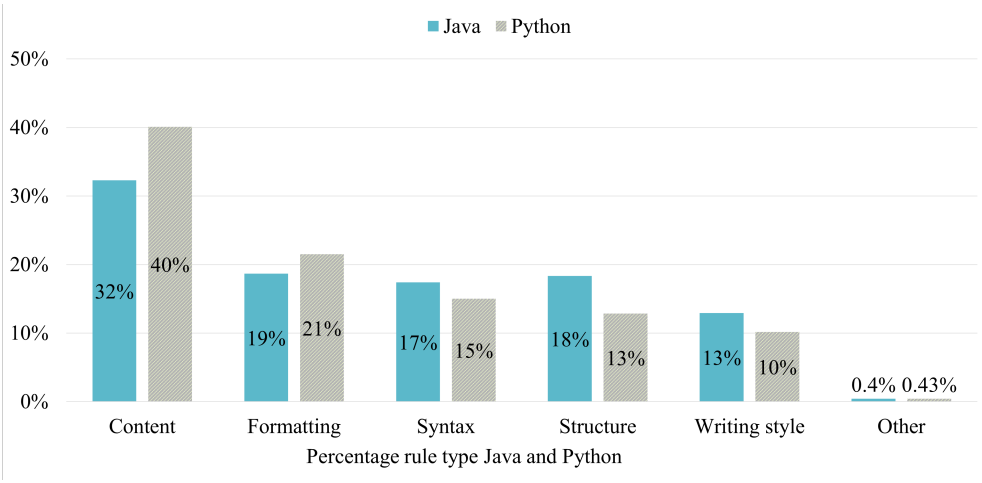

Figure 3.3: Average rule type percentages across all projects.

Figure 3.4 gives an overview of the rule type distribution for each project and standard style guideline. The projects marked with an asterisk have project-specific rules (*i.e.*, Spark). The style guidelines are underlined (*i.e.*, Oracle) The numbers in parentheses indicate the number of comment-related rules each project and style guide has in total. For the projects that follow a standard style guideline and have self-imposed standards defined, the number of rules of both standards is added into a sum. For instance, as shown in Figure 3.4, Hadoop has in total 152 comment-related rules that should be followed by its developers. In Table 3.2 we see that Hadoop uses the Sun Java Code Convention. With the 149 rules from the Sun Java Code Convention and the 3 rules in Hadoop, we get a total of 152 rules for Hadoop.

In both languages, the *Content* rules followed by *Formatting* rules are most represented. Furthermore, *Syntax* and *Structure* rules make out on average about the same percentage on Java whereas in Python the percentage of syntax rules is higher than in Java.

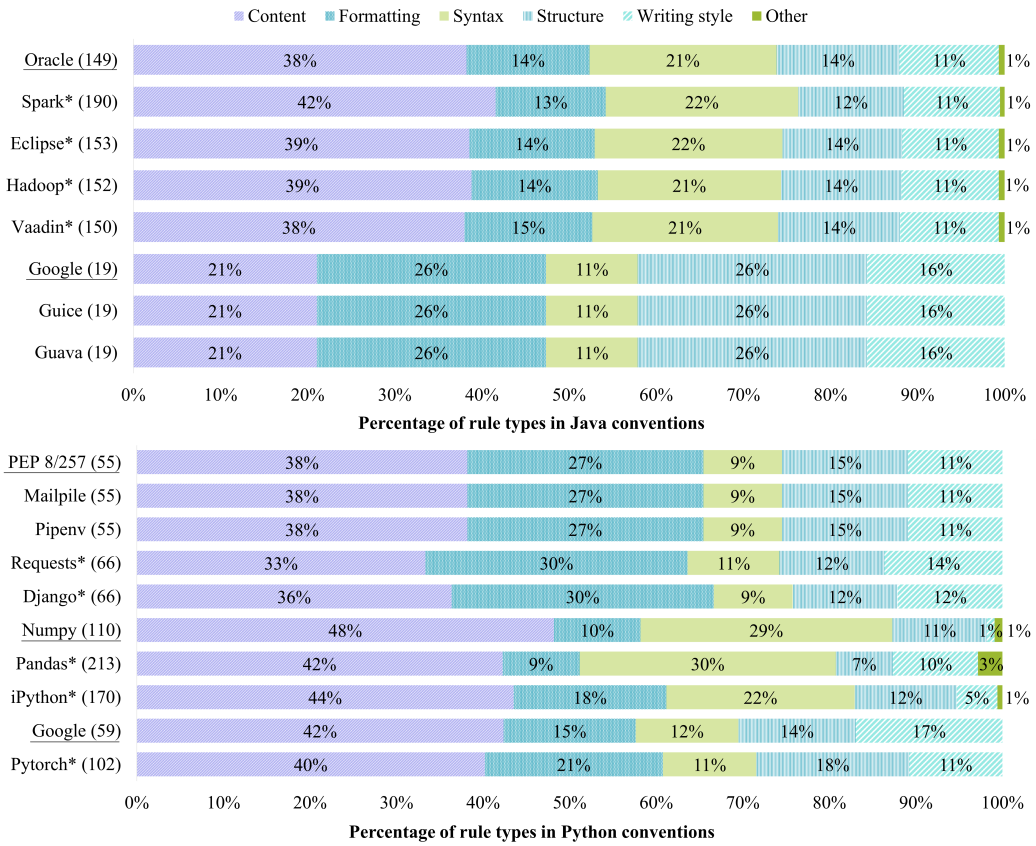

Figure 3.4: Rule type percentage of the Java and Python projects.

Figure 3.5 shows the rule type distribution for code comment-related rules that apply to class comments. The distribution of class comment related rules is also dominated by *Content* and *Formatting* rules. Otherwise, compared to Figure 3.4 there is no significant change of rule type ratio. Interestingly, the present rule type categories do not contain *Other* rules whereas we spot for Numpy, Pandas, and iPython rules of type *Other*. Taking a closer look, those rules come solely from Numpy and Pandas, since Pandas and iPython suggest following the Numpy style guide and Pandas' self-imposed rules contain *Other* rules, therefore the 3% in Pandas and 1% in iPython.

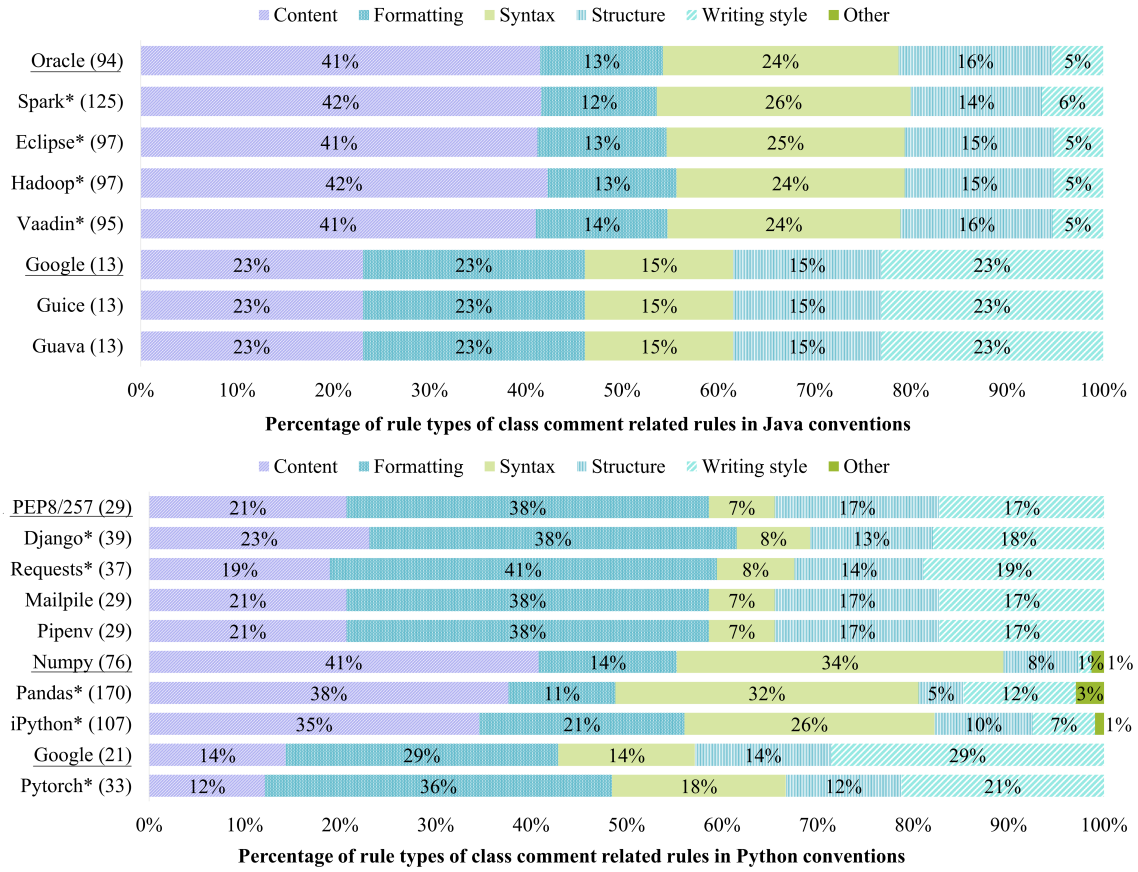

Figure 3.5: Rule type percentage of class comment related rules of the Java and Python projects.

3.4 Implication and Discussion

From the Java coding style guides only the Sun Java Code Convention contained a comment style guide dedicated to writing documentation comments. In Python, we found that PEP 257 contained rules specific to docstrings. The other conventions had rules about other things such as naming conventions, file structure, and programming practices and have a section dedicated to comments where implementation and documentation comments were differentiated. Interestingly, all style guides gave information about the dedicated location of the comments and how they should be indented. Not all style guides went into the same level of detail about what content should be written inside comments.

Since many rules are about formatting and indentation, this indicates that style guidelines are concerned with the look of code and how it is displayed. The second focus of the style guidelines seems to lie on the information inside the comments. *Formatting* rules usually discussed how comments within the source code and components inside the comment should be indented. They also defined where and how many blank lines should be inserted between the comment and the source code, and between the components

inside the comment. Lastly, for each project there was at least one *Formatting* rule, either defined in the standard style guide or the project convention, that defined a limit of characters allowed per line. *Content* rules usually gave general statements on what should be inside a comment *i.e.*, "Comments should be used to give overviews of code." but were in some cases more precise *i.e.*, "The first sentence of deprecated-text should at least tell the user when the API was deprecated and what to use as a replacement." In other cases, *Content* rules defined what should not be inside a comment *i.e.*, "Do not use @version tags."

Writing style rules make the smallest percentage of all extracted rules. This means when a comment is already well-formatted and contains relevant information, then how the information is written is not as important anymore. However, since even comments with good content, correct syntax, and formatting can be written in different writing styles, inconsistencies between code comments are not avoided. The lack of *Writing style* rules reveals a gap in the style guidelines on how to write the comment content concerning language and grammar.

As we can see that the style guidelines provide *Content* rule types more often than other rule types but these rules are not well covered by linters or style checkers. The linters are often limited to checking the presence of comments or their formatting but rarely contain the rules to check the content inside. Having linters configured in the projects can ensure that developers follow the formatting rules. In addition, there is a strong need to conduct studies on linters to see how well they support checking code comments.

There is no clear answer as to why *Syntax*, *Structure*, and *Writing style* rules have a low presence in both Java and Python style guides. This raises the question of whether there is a correlation between the number of rule types present in a style guide and the adherence of the comments to a specific rule type. Going one step further, are the predominantly present *Formatting* and *Content* rules in the style guideline more followed by developers than rule types that with less presence? Lastly, whether the small percentage of *Syntax* and *Structure* rules is related to the projects using linters, can also be explored in future work.

3.5 Conclusion

In this chapter, we consulted the style guides used by 13 open-source projects: 6 Java and 7 Python projects. We found that Java projects use the Sun Java Code Convention and Google Java Style Guide. The Python projects use PEP 8/257, Numpydoc, and Google Python Style Guide. Of the 13 projects, 9 (from which 4 Java and 5 Python projects) have additionally a convention specific for their project defined.

From the Java coding style guides only the Sun Java Code Convention contained a comment style guide dedicated to writing documentation comments. In Python, we found that PEP 257 contains rules specific to docstrings. The other conventions had rules about other things such as naming conventions, file structure, and programming practices and have a section dedicated to comments where implementation and documentation comments were differentiated. Interestingly, all style guides gave information about the dedicated location of the comments and how they should be indented. Not all style guides went into the same level of detail. Project conventions defined exceptions to the style guide and added additional

information which in some cases were already covered by the style guide itself.

From the style guidelines used by the projects, we extracted all comment-related rules and categorized them according to an adapted taxonomy: *Content*, *Formatting*, *Syntax*, *Structure*, *Writing style*, and *Other*. It turns out that *Content* and *Formatting* rules are the most frequent rule types. Only a small portion of the rules per each style guide were *Writing style* rules.

4

Convention Adherence

4.1 Introduction

The previous chapter described how programming languages, various communities, organizations, or projects provide style guidelines to write consistent, readable, and maintainable comments. They provide various conventions to dictate the formatting, style, or content aspect of comments. However, whether developers follow these conventions while writing comments is not known. In order to have high-quality comments, it is essential to verify the adherence of developer commenting practices to the style guidelines. In this chapter we answer **RQ2:** *To what extent do developers follow style guides when writing class comments?*

As verifying all comments against all rules is not a trivial task, we select sample class comments and carefully analyze each rule to develop the condition under which we can say that a comment follows a rule or not. Therefore, following the methodology given in Chapter 3, we validate each rule against the sample of the selected class comments.

We find that developers follow *Writing style* rules more often than other rule types. Moreover, developers follow *Content* rules often however the majority of the *Content* rules are not applicable to comments due to the absence of comment components targeted by the rules. For instance, the *Content* rule from the Sun Java Code Convention "The @deprecated description in the first sentence should at least

tell the user when the API was deprecated." is addressing the purpose of the description following the "@deprecated" tag. However, the comment adherence to this rule can only be verified if the comment contains the "@deprecated" tag addressed in the rule. If the comment does not have "@deprecated" tag, this aforementioned *Content* rule cannot be applied due to the absence of this tag. Of the applicable rules, not all rules are followed by developers. We find that developers often violate *Structure* rules. Our results confirm the finding from the previous work of Rani *et al.* [6] in which Pharo class comments adhere predominantly to *Content* and *Writing style* rules described in the Pharo class comment template.

4.2 Methodology

To answer RQ_2, we evaluate the rules extracted from the style guidelines (in the previous chapter) against comments to see if developers follow these rules when writing comments or not. However, verifying each comment against each rule is not a trivial task. We notice some open rules that are hard to verify. We carefully identify such rules.

Validated Rules. To evaluate code or comments against the style guidelines automatically, various tools known as linters or style checkers are available. However, not all rules are supported by these tools. There are already existing tools to validate and fix the formatting of source code.[1] However, they provide limited support for verifying comment rules. They provide limited support for checking the formatting of comments and do not support checking all other types of rules, therefore the majority of the rules require manual analysis. We, thus, first focus on the rules that require manual validation *i.e.*, all rules other than formatting rules. To restrict the scope of the work and reduce the manual effort, we focus on specific types of comments and specific types of rules. As explained in Section 3.2.1, we reuse the dataset prepared by Rani *et al.* [5]. They analyzed class comments and identified the information types sentence-wise. From the dataset, we consider the class comments extracted from open-source Java and Python projects. Additionally, we use the rule taxonomy formulated in Chapter 3. As we focus on class comments, we consider only the rules about class comments and the rules types (*Content*, *Syntax*, *Structure*, *Writing style*, *Other*) except *Formatting* type and the rules categorized as *Other*.

Table 4.1 and Table 4.2, displayed below, describe all rules that were validated, not validated due to various reasons such as some rules being of type *Formatting* (NV-Format) and some rules not being constrictive enough or not possible to be validated with the class comments in our dataset (NV-Other).

Language	# All rules	# CC rules	# Valid	# N-Valid	# NV-Format	# NV-Other
Java	217	145	83	62	21	41
Python	397	252	180	72	51	21

Table 4.1: Rules that are validated or not validated

[1] `www.jetbrains.com/help/idea/reformat-and-rearrange-code.html` accessed August 31, 2021

In Table 4.1 we see the total number of rules we extracted from the style guides per language. The column *All rules* shows all the comment-related rules, and the column *CC rules* shows all class comment related rules. The column *Valid* shows all the validated rules, the column *NValid* shows the total number of rules that are not validated due to any reason. The column *NValid* is further described with the reason for non-validation due to the formatting rules (the column *NV-Format*) or any other reason (the column *NV-Other*).

Out of all extracted rules, 69% (Java) and 64% (Python) are class comment related. As mentioned earlier, we excluded *Formatting* rules to focus on rules requiring manual analysis. The dataset consists of class comments and not any related code of the class or other classes mentioned in the comment. We find various rules (43% for Java and 37% for Python) that cannot be validated with class comments alone. Thus, in addition to excluding *Formatting* rules, we exclude rules that cannot be validated with our dataset and require the consultation of the source code. For instance, the rule "The closing quotes are on the same line as the opening quotes" cannot be validated due to the absence of comment delimiters (*i.e.*,/**...*/, """) and indentation in the dataset. Similarly, some rules are excluded due to being unprecise such as the rule "A doc comment may contain multiple @version tags." In this case, this rule is always considered as *followed* since there is no constriction on the number of @version tags used in the comment. Therefore, this rule cannot be violated. Cases, where the @version tag is missing are the exception, and the rule is considered not applicable to those comments leading to those comments being marked as *not applicable*.

	Style guide	**# All rules**	**# CC rules**	**# Valid**	**# N-Valid**	**# NV-Format**	**# NV-Other**
Java	Oracle	149	94	54	40	12	28
	Google	19	13	8	5	3	2
	Vaadin	1	1	0	1	1	0
	Eclipse.cdt	4	3	2	1	1	0
	Hadoop	3	3	1	2	1	1
	Spark	41	31	18	13	3	10
Python	PEP	55	29	12	17	11	6
	Google	59	21	12	9	6	3
	Numpy	110	76	63	13	11	2
	Django	11	10	6	4	4	0
	Pandas	103	94	78	16	8	8
	iPython	5	2	1	1	1	0
	Pytorch	43	12	5	7	6	1
	Requests	11	8	3	5	4	1

Table 4.2: Number of extracted rules from each style guide (including project-specific style guides)

Table 4.2 displays the number of rules that got extracted from each Java and Python code style guide and convention specific for the projects. The column *Style guide* show lists all style guides suggested

by the projects and the projects with project-specific rules. The style guides and projects are ordered by the programming language they are written in, starting with *Java* and below the horizontal line followed with *Python* style guides and projects. As in Table 4.1, the columns *All rules* and *CC rules* show all comment-related rules and how many rules are class comment related. The columns *Valid* and *N-Valid* show how many of the class comment related rules are validated and not validated. The column *NV-Format* shows how many not validated rules are *Formatting* rules. The column *NV-Other* shows the rules that were not validated for other reasons.

The Java projects Guava and Guice as well as the Python projects Pipenv and Mailpile are not listed in Table 4.2 since they do not have any project-specific rules defined. Overall the rules defined by projects tend to be more class comment-related.

Adherence of comments against the rules. We identified whether each comment or part of it follows the rules by labeling it with *followed*, *not followed*, or *not applicable*. We consider a comment or part of it following the rule (*followed*) when it satisfies all the conditions of the rule fully otherwise we label *not followed*. In case the rule is not applicable to the comment or the comment component targeted in the rule is missing, we label it as *not applicable*.

The validation was done manually with the occasional help of regular expressions. Let's consider the following rule: "If the member has no replacement, the argument to @deprecated should be 'No replacement'." First, we filter the dataset for class comments with the *deprecated* tag. After this step, we can already consider the rule to be *not applicable* to the class comments that are filtered out. Next, for the comments that contain the @deprecated tag, we filter again for the pattern "No replacement". Then we go through the comments and identify the sentence or part that is about the deprecation information. We check whether this part satisfies the rule. If the rule is satisfied, the comment is labeled as *"followed"*. Comments that did contain the @deprecated tag but not the "No replacement" pattern are labeled as *"not followed"*. Regular expressions could not be used for rules that target the description of a comment component, *e.g.*, for a rule like "the first sentence of the class comment needs to be a summary". In such cases, we manually examine each class comment and validated it accordingly. We use a search term to find the occurrence of the exact term and in case the condition/rule does not specify the exact term, we composed a regular expression.

One author validated all the rules against the selected comments. Another author reviewed all the evaluations independently. The second author reviewed the doubtful cases. In the end, using the majority voting mechanism, we resolved all the conflicts.

4.3 Results

This research question aims to verify the adherence of class comments to the style guidelines. In Figure 4.1 we see the adherence of the class comments to the comment-related rules. The Java and Python projects are listed on the y-axis and the percentage of each category is displayed on the x-axis. For each project,

the distribution of the class comments categorized as *followed*, *not followed*, or *not applicable* is displayed. The last category (*not applicable*) is due to a rule being not applicable to the class comment. For instance, in Guice, 53% of the comments follow the coding style guideline suggested by Guice. However, 2% of the class comments do not adhere to the coding style guideline and 45% of the comments could not be validated due to rules being not applicable to them. Generally, we see that a high percentage of class comments are labeled as *not applicable* since they often do not contain the components addressed by the rules. For instance, not all comments contain the *@deprecated* tag and therefore rules about this tag do not need to be taken into consideration. Pandas and iPython are two extreme cases here. Considering the labels *Followed* and *Not followed* more class comments do follow the style guidelines rule than the ones violating them. Noteworthy is the high percentages of the *not applicable* category. Out of the 263 rules we validated against the class comments, on average 55% of the Java class comments and 60% of the Python rules did not apply to the comments. In addition, the percentage of Java class comments violating the rules is lower (on average 2%) than Python 7%. Interestingly, Pandas, which has its own self-imposed rules and adopts the Numpydoc style guide, follows the highest number of rules from all 13 projects and has the highest percentage of the *not applicable* rules.

Not applicable. Rules that could not be validated against comments due to missing information inside the comment usually target optional or circumstantial comment components. For the latter, we consider for instance the deprecation of an option as circumstantial. Most rules that are not applicable are of type *Content*, *Syntax*, and *Structure*.

Followed. Rules such as "*Avoid Latin*", "*Comments should never include special characters such as form-feed and backspace.*", both from the Sun Java Code Convention, were followed by all comments of Projects that suggest following the Sun Java Code Convention.

Not followed. The percentage of comments violating rules is very low. Interestingly, there was never a rule that was violated by all comments within a project. To be more specific, the highest percentage of comments violating a rule lies by 35.7%. Compared to this we have several rules that are 100% *Followed* or 100% *Not applicable*. This 35.7% of rule violating comments, were recorded in Spark for the rule from the Sun Java Code Convention "*Doc comments are meant to describe the specification of the code, from an implementation-free perspective.*".

Finding 1. *A large portion of comments is not applicable to rules. Java shows a higher portion than Python.*

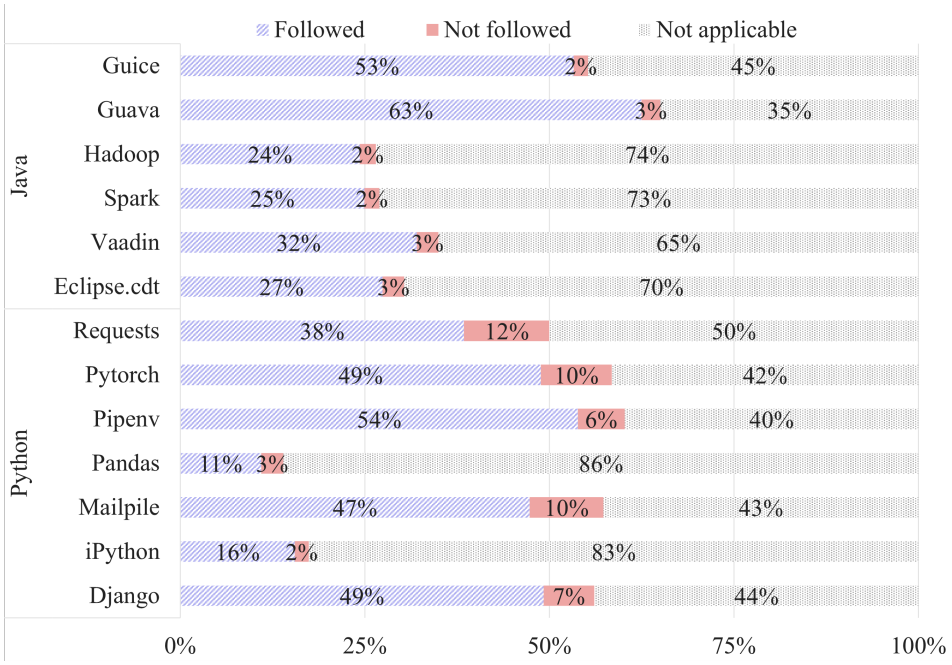

Figure 4.1: Comment adherence across all projects

4.3.1 Adherence to rule types

Figures 4.2 and 4.3 illustrate what rule types tend to be followed or not followed by the class comments. Missing rule type categories for a project in the graph indicate that no comments were validated against that specific category. In Figure 4.2 it is noticeable that across nearly all Java and Python projects rules of type *Writing style* are followed the most. As an exception, we see that for Pandas *Structure* rules are followed the most. On the other hand, in Figure 4.3, we see that rules of type *Structure* (on average 27%) tend to be violated the most.

Finding 2. *Writing style rules are followed the most across all projects except for Pandas in which comments tend to adhere more to* Structure *rules.*

Finding 3. *Structure rules are those rules that tend to be not followed by the majority of the projects, with Spark leading with 75% of its Structure rules not being followed.*

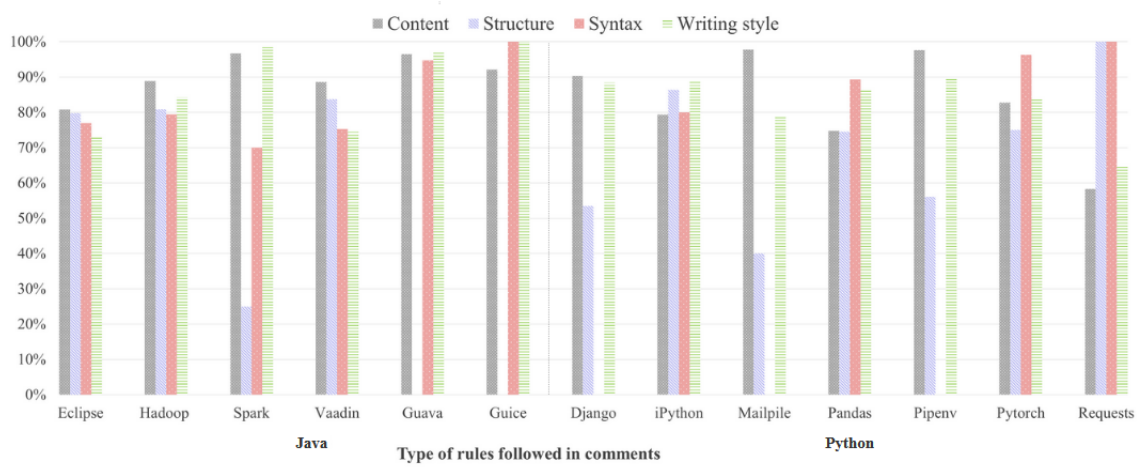

Figure 4.2: Types of rules that are followed

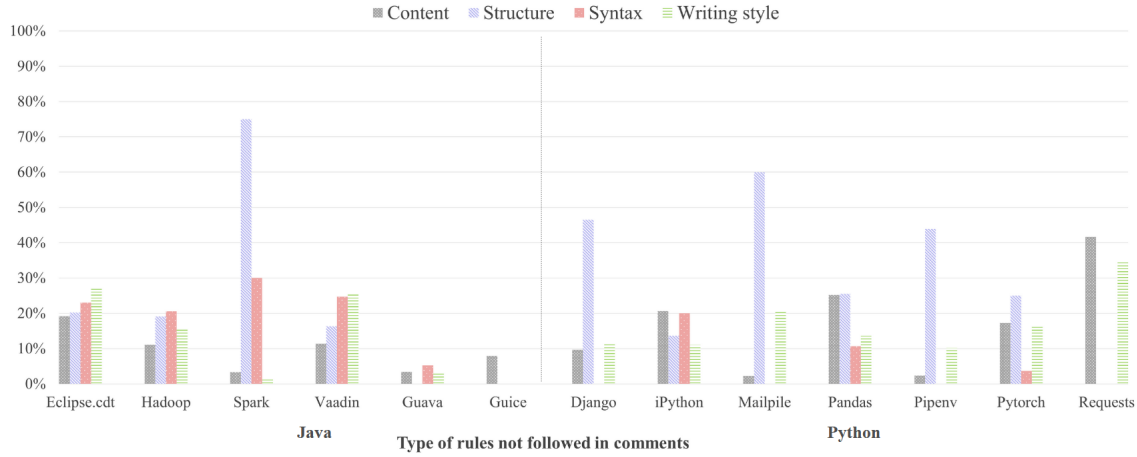

Figure 4.3: Types of rules that are not followed

4.3.2 Not applicable rules

In Table 4.3 you see Java rules that were for some or several projects never applicable to any class comment. The hyphen indicates that this rule does not belong to the style guide of the project. For instance, the first two rules *"Any of the standard 'block tags' that are used appear in the order @param, @return, @throws, @deprecated"* and *"Any of the standard 'block tags' @param, @return, @throws, @deprecated never appear with an empty description."* are from the Google Java Style Guide, which is the style guide for Guava and Guice. We have six rules from the Sun Java Code Convention and one rule from the Google Java Style Guide that were never applicable to the class comments in our dataset. For the Python rules, we found that the following PEP 8/257 rule was never applicable to any Python class comment: "The

docstring for a class should list the public methods."

Finding 4. *From the 83 Java class comment related rules we validated, 7 were never applicable to any class comment. For the Python class comments, only one rule was never applicable.*

Rule	Eclipse.cdt	Hadoop	Spark	Vaadin	Guava	Guice
Any of the standard "block tags" that are used appear in the order @param, @return, @throws, @deprecated	-	-	-	-	×	×
Any of the standard "block tags" @param, @return, @throws, @deprecated never appear with an empty description.	-	-	-	-	✓	×
@serial field-description.	×	×	×	×	-	-
The optional field-description describes the meaning of the field and its acceptable values. The field-description can span multiple lines.	×	×	×	×	-	-
{@linkplain package.class#member label}	×	×	×	×	-	-
@serial field-description | include | exclude	×	×	×	×	-	-
Use XXX in a comment to flag something that is bogus but works.	×	×	×	×	-	-
Use FIXME to flag something that is bogus and broken.	×	×	×	×	-	-

Table 4.3: Examples of not applicable Java rules.
(×) rule was not applicable to any class comments of that project, (✓) rule was applicable to some class comments of that project, (-) rule is from a style guideline not suggested by the project

4.4 Implication and Discussion

According to our results, rules of type *Writing style* are followed the most amongst all rule types. Considering the results of Chapter 3, where we found that from all rule types present in style guidelines *Writing style* rules make up a small portion. We assume this result may be due to grammar rules being followed by developers per default unrelated when it comes to writing code comments or other types of documentation. However, this can be further investigated in future work.

As we have seen in Figure 4.1, Pandas has with 86% the highest percentage of comments with non-applicable rules. We found that due to the possibility of comment components being optional and several rules addressing optional comment components, the percentage of not applicable comments rises. For

instance, the Pandas rule “This section has a header, ‘See Also’ (note the capital S and A), followed by the line with hyphens and preceded by a blank line.” is not applicable to 91% of the class comments in Pandas. Since the *See Also* section is optional many comments will be not applicable to this rule as shown with the previous example. We found that the same applies to various other optional sections and comment components.

4.5 Conclusion

The Java projects have higher adherence to the rules in the style guide than the Python projects. Moreover, the class comment related rules of the Java projects do not apply to 60% of the Java comments whereas the percentage lies by 55% for the Python comments. This high percentage is caused by certain parts of comments being optional or only used in certain scenarios (*i.e.*, object deprecation). Therefore, rules targeting such optional or circumstantial comment components will only apply to a small portion of all comments in a project. We even had in both languages rules that never applied to any class comments in our dataset. However, there is room for bias since the projects vary in sample size and rules. This may lead to projects with fewer rules show a higher adherence to the style guide. This might be the case for the Java projects Guava and Guice.

Additionally, rules of type *Writing style* tend to be followed more by developers. Contrary, *Structure* rules are more violated than other rule types. Lastly, projects tend to show more not applicable rules if the number of samples is close to or smaller than the number of rules.

5

Threats to Validity

In this chapter, we highlight various threats to the validity of our study.

5.1 Threats to construct validity

This concerns the measurement used in our study.

Selection of sample guidelines. We did not consider all style guidelines available in the Java and Python ecosystems and instead focused on the selected guidelines. To mitigate this concern, we focused on the popular standard guidelines used in Java and Python open-source projects. As these guidelines are considered as a baseline for many other guidelines and used frequently in projects, they should give a reliable overview of commenting conventions recommended by the style guidelines.

Selection of sample projects. We did not consider all projects in each language but considered a sample of them. In order to utilize the carefully considered heterogenous projects from various related works, we considered the same projects[4, 5, 13]. The projects originate from different ecosystems such as Google and Oracle, and thus follow different coding style guidelines for their comments.

Selection of sample comments. As we did not consider all comments of a project, this can be a concern related to subjectiveness and bias in the evaluation. However, we leveraged the dataset of sample comments from the previous work [5]. The sample comments represent the statistically significant sample set for

both languages. Additionally, they are selected using a hybrid approach of stratified and random sampling approaches. However, as we used the dataset from Rani *et al.*, we relied on their classified comments. Their focus was just on the content of the comments and not on other aspects such as syntax or structure of comments. Thus, the dataset worked well for their case study, but we noticed some issues for our analysis. For instance, there were reported cases of dangling comments, where the comments precede the class definition but are written using the block comment symbols rather than the documentation comment symbols. Additionally, we found very few cases of inline comments even though their focus is just on class comments. In such cases, we consulted the source file of the class and manually verified whether the extracted comment was in fact a class comment. Some source files for the classes do not exist in the latest version of the project. In future work, we plan to validate the comments with the latest version of the projects.

5.2 Threats to internal validity

This concerns confounding factors that could influence our results.

Rule taxonomy. Another important concern is the definition of the rule type taxonomy and mapping of the rules to the taxonomy are performed by two human subjects. To reduce this risk, we extend the taxonomy proposed by the earlier work following the same methodology [6]. However, as the system evolves, there are high chances that the style guidelines are updated and the conventions present in them are changed, replaced, or reformulated. This can affect the categorization of the rule type. Similarly, there can be additional external links added in the guidelines to point to further commenting conventions. For example, Python PEP 8 refers to the PEP 257 guideline for Docstring related conventions. To ensure the accuracy of the current version of the taxonomy, we iterated the style guidelines over the course of the thesis not to miss any comment-related rule. Two authors independently categorized each rule and in case of a conflict, the third author reviewed and marked the decision using the majority voting mechanism.

Comment adherence to the rules. When validating if a rule follows a comment or not, our understanding of the rule can be subjective and can lead to a wrong validation. To prevent this and provide reproducibility of the validation process, for each rule we mutually defined various validation conditions under which a comment can be considered as *followed*, *not followed*, or *not applicable*. We performed a two-step validation approach, where first the author independently validated each rule against each comment, and then another author independently reviewed the validations. Whenever the opinions diverged, we discussed until a final consensus was reached.

Additional research is needed to test whether our results generalize to closed-source projects that are maintained in a corporate setting with the same or a different set of developers.

5.3 Threats to external validity

This concerns the generalization of our results. The main goal of this thesis is to analyze the commenting conventions proposed by various style guidelines for the selected programming languages. The proposed approach may achieve different results in other programming languages or projects. To reduce this threat, we considered both static and dynamic programming languages, following different style guidelines. Additionally, the projects considered also vary in terms of size, domain, language, and contributors.

6

Conclusion and Future Work

We found that the six Java open-source projects in our dataset suggest their developers to use the Sun Java Code Convention and Google Java Style Guide. The seven Python projects suggest PEP 8/257, Numpydoc, and Google Python Style Guide. Besides the suggested style guide the projects often defined a project-specific convention where either new rules besides the ones present in the style guide were defined or rules from the style guide are changed or complemented. Interestingly, all style guides gave information about the location of the comments within the source code and how they should be indented. We discovered that not all style guides went into the same level of detail regarding code comments.

From the style guidelines used by the projects, we extracted all comment-related rules and categorized them according to an adapted taxonomy. Our results show that *Content* and *Formatting* rules are the most frequent rule types. Only a small portion of the rules per each style guide was *Writing style* rules.

We validated the class comment-related style guideline rules against the Java and Python class comments in our dataset and found that the Java class comments have a higher adherence to style guide rules than the Python projects. Moreover, the class comment related rules of the Java projects do not apply to 60% of the Java comments whereas the percentage lies by 55% for the Python comments. This high percentage in both languages is caused by optional comments components or comment components only used in certain scenarios. This leads to such rules only applying to a small portion of all comments in a project. Additionally, according to our results, rules of type *Writing style* tend to be followed more by developers. In contrast, *Structure* and *Content* rules are more violated than other rule types.

With this thesis, we point out the unbalanced distribution of the rule types present in Java and Python style guides. This finding might encourage writing better style guides for code comments. Future work might investigate whether there is a link between the content of style guides and coverage of linters.

For future work, we plan to extend our work to other programming languages such as C/C++, C# and analyze other comment types such as method comments, package comments, inline comments to see if developers follow the guidelines specific to a type. We also plan to involve developers via surveys and interviews to validate our rule taxonomy and labeled comments. This step is important to strengthen our results, such as where the current style guidelines lack, and design tools to enable the automatic adherence of comments to the style guidelines. Lastly, our analysis focused only on open-source Java and Python projects on GitHub. Additional research is necessary to confirm whether our results generalize to closed-source projects.

7

Anleitung zu wissenschaftlichen Arbeiten

This chapter provides various guides related to Chapter 3 in *Comment Conventions* and Chapter 4 in *Comment Adherence*. An overview of the thesis is displayed in Figure 7.1. In the following, we will provide more details for certain steps mentioned in the previous chapters. The *Comment Conventions* section provides a guide to the finding of the style guidelines (Section 7.1.1) for the various projects presented in *Data collection* (Section 3.2.1), identifying comment-related rules from the content of the style guidelines (*Extracting comment-related rules*), and organizing the rules into the taxonomy as discussed in Chapter 3.

The *Comment Adherence* section presents a guide to validate comments to the rules in Chapter 4. In Section 7.2.1 we describe the step we took before validating the style guide rules against our comments in the dataset. Moreover, we explain how we defined conditions under which a rule is considered as followed, not followed, or not applicable to a comment.

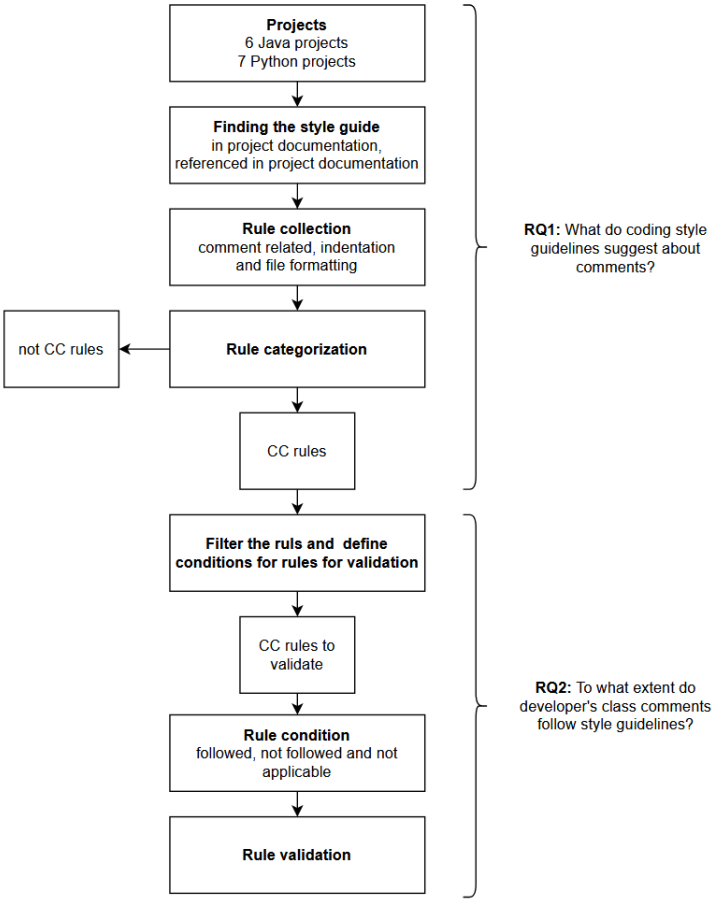

Figure 7.1: Thesis overview

7.1 Comment Conventions

7.1.1 Finding the style guides

Projects can, when suggesting developers which guideline to use, reference multiple guidelines or specific sections of guidelines. Generally, this information is documented on the project web page, or the version control repository (*e.g.*, GitHub) under the contribution guideline section or the documentation guideline. The purpose of these two guidelines is to tell developers which conventions to refer to for what part of the code when contributing to the open-source project. As all 13 projects in our dataset are popular, have their own web page, and are hosted on GitHub, we searched in both sources for a reference to style guides.

In most cases, we found this information located in the “Contribute” section of the project web page where the required material about the project set up, workflow, and collaboration rules is mentioned, or in

the README of the GitHub repository.[1]

In the case of the Vaadin project, we could not find a direct mention of the style guideline, so we consulted the linter file (Checkstyle file) in its GitHub repository.[2] Similarly, for the Python projects Mailpile and Pipenv, we found no mention of the usage of the style guidelines for those projects. Thus, finding the style guideline for both of them was not a trivial process. In the following paragraph, we detailed the strategy adopted to find which style guideline these projects follow.

Pipenv mentions on its project page that the documentation is written in restructuredText (described later on) and states *"When contributing documentation, please do your best to follow the style of the documentation files. This means a soft-limit of 79 characters wide in your text files and a semi-formal, yet friendly and approachable, prose style."*.[3] However, this statement does not refer to comments in Python source code but instead the documentation text files that reside in a designated directory in their GitHub repository. The plain text markup language, restructuredText, is used to create Python comments, simple web pages, and standalone documents. However, it does not contain rules on how Python comments should be written.[4] We checked the pull requests of the GitHub repository since they contain information about changes developers push to the repository. According to the pull requests, some developers pushed changes to the code to make their changes adhere to PEP 8, confirming that Pipenv developers use PEP 8/257 as a style guideline. [5] Similarly, we found that Mailpile uses the PEP 8/257 style guideline.[6] As we did not find any mention of a project-specific guideline neither on Pipenv's and Mailpile's project page nor in their GitHub repository, we assumed that Pipenv and Mailpile had not any project-specific style guideline defined. Thus, we analyzed the adherence of the comments from Pipenv and Mailpile solely for PEP 8/257 rules.[7]

Once we found out which style guideline the projects refer to, the next important thing is finding out which version of the style guide to use.

7.1.2 Identifying the style guide version

Over time, style guides evolve and new conventions are added, deleted, or changed. After determining the style guide the projects in our dataset suggest, we consulted the version of the style guide that is available online, which is usually the latest version. However, since the snapshot of our projects in the dataset dates to the end of 2019, we had to search for the version of the style guide that was available at that time and compare it to the online version.

[1] `https://cwiki.apache.org/confluence/display/HADOOP/How+To+Contribute` accessed August 31, 2021

[2] `https://github.com/vaadin/framework/blob/727accead38ec3a292900054764af59ea89e0fda/checkstyle/vaadin-checkstyle.xml` accessed August 31, 2021

[3] `https://pipenv.pypa.io/en/latest/dev/contributing/#documentation-contributions` accessed August 31, 2021

[4] `https://docutils.sourceforge.io/rst.html` accessed August 31, 2021

[5] `https://github.com/pypa/pipenv/pull/87` accessed August 31, 2021

[6] `https://github.com/mailpile/Mailpile/pull/62` accessed August 31, 2021

[7] `https://docs.python-requests.org/en/latest/dev/contributing/` accessed August 31, 2021

We hypothesized that in the time frame between the project snapshot and the time of writing, both style guide versions will differ slightly. We could trace back older versions of the style guides since the content of the style guides very often is stored also as a text file which is situated in a GitHub repository. We compared the online version from which we extracted the rules to the version dating to the end of 2019. Most style guidelines did not change at all or did not change any comment-related rules. Pandas has changed the wording, however, the meaning is still the same. We could not find any versioning of project-specific rules defined for the Eclipse.cdt project. Therefore, used the rules available on the website at the time of writing. To guarantee the reproducibility of the thesis, we have a copy of each style guide version that was considered in the thesis recorded in the replication package.

7.1.3 Analyzing the style guide content

Once the version of the style guide to use is determined, the next step is to analyze the content of the style guide and extract comment-related rules. In the following paragraphs, we describe the steps taken to achieve this.

7.1.3.1 Standard guidelines

All style guidelines are written in natural language, and provide examples as an addition to the text. The examples are usually code fragments or comments. Some style guides (Figure 7.2) provide examples of good and bad code or comments. The style guide content is ordered in sections for better orientation. All style guides are written in English and some have translations available. In this thesis, we only considered the English translation of the style guide.

Good:

```
def astype(dtype):
    """
    Cast Series type.

    This section will provide further details.
    """
    pass
```

Bad:

```
def astype(dtype):
    """
    Casts Series type.

    Verb in third-person of the present simple, should be infinitive.
    """
    pass
```

Figure 7.2: Pandas style guide good and bad example of short summary

We discovered that standard style guidelines address other things besides comments such as naming

conventions, file organization, indentation and formatting, code statement writing, programming practices, and comments. Additionally, some style guidelines, refer to other sources for more details about a certain comment component. For instance, the Sun Java Code Convention often refers when it comes to tags to another style guide.

Java. From all style guides, the Sun Java Code Convention has the most sources, from which rules were extracted. Among the sources, there is a style guide named *"How to Write Doc Comments for the Javadoc Tool"* that is specific to writing documentation comments. Google provides a style guide for the majority of popular programming languages. The Google Java Style Guide is one of the many style guides defined by Google. It serves as a complete convention of Google's coding standards for Java source code. and also includes rules about how to write good comments. Compared to the Sun Java Code Convention, the Google Style Guide does not go into as much detail regarding comments and the comment components (*i.e.*, summary, field description, examples, tags *etc.*) that exist and how they should be documented.

Python. Five out of the seven Python projects claim to follow the PEP style guide for writing documentation comments. This is not surprising since PEP is the standard Python-style convention. PEP 8 refers to PEP 257 to gain more details for documentation comments. We, therefore, always considered rules from PEP 257, even a project only claims to follow PEP 8. Contrary to the Sun Java Code Convention the summary line in multi-line docstrings is separated from the description that follows by a blank line. NumpyDoc follows PEP 8 and PEP 257 and has it extended with its own conventions as well. The style guide is structured by the sections a docstring can contain. Such sections are: Short summary, extended summary, parameters, returns/yields/receives, raises, warns, see also and notes section. NumpyDoc has rules specific to classes, methods, and modules. The Google Python Style Guide contains Python rules which include code formatting, indentations of code and comments, and docstrings. Similar to the NumpyDoc, the summary line is separated from the description by a blank line. Regarding sections, the Google Python Style Guide mentions only four sections: *Args*, *Returns/Yields*, *Raises*, and *Attributes* while Numpy mentions the ones listed earlier.

7.1.3.2 Project-specific guidelines

As stated before, projects can follow one or several style guidelines or can define their own. For both languages, Java and Python, we had two projects per language (Guava, Guice, Pipenv, Mailpile) (Table 3.2) that only refer to the style guidelines of the language. On the other hand, the project iPython, suggests its developers to follow the two style guides: PEP 8/257 and Numpydoc. To avoid overlapping or conflicting rules, projects can prioritize the conventions of one style guide over another. Generally, we gave rules defined in the project-specific guideline a higher priority than the rules present in the standard style guidelines.

Java. In the case of Eclipse.cdt, Hadoop, and Vaadin, only a few rules were defined as exceptions to the standard style guide suggested by the projects. The Eclipse.cdt style guideline defines that an indent should be four spaces wide, @version tags should not be used, and that all HTML tags must be terminated. Hadoop defines that comments should not contain @author tags and the indentation should

be two spaces per level instead of four. Additionally, Hadoop mentions that public classes and methods should be documented with informative Javadoc comments. However, this rule is already covered by the Sun Java Code Convention. Vaadin adds to the line length 80 rule from Sun Java Code Convention that member declarations are not wrapped onto the next line. Among all Java projects, Spark has the most project-specific rules. This is due to the fact that Spark has 65% of the code in Scala and thus uses Scaladoc. Although many rules in Scaladoc are similar to the Sun Java Code Convention, we carefully selected the rules applicable for Javadoc comments. For instance, rules about the @group tag used in Scala were not extracted, since this tag does not exist in the Java programming language.

Python. In Python, we also found exceptional cases where the rule in the project-specific guideline contradicts the style guideline suggested by the project. In those cases, the project either defined an exception or adaption or defines a rule that contradicts a rule in the suggested style guide. For example, the Django guideline recommends the line length for the documentation, comments, and docstrings to be 79 characters compared to 72 suggested by PEP eight. Similarly, iPython recommends using four spaces for indentation and never tabs. We documented all such exceptions and prioritized them over the rules suggested by the baseline guideline.

7.1.4 Extracting the rules

Finding the rules The aim is to identify the code comment-related rules. In each style guide, we searched for keywords like 'documentation', 'comment', and 'class'. As the web page presents the content divided into various sections, we extracted the section titles to preserve the context of the rule. It facilitated identifying which type of comment the rule targets. The rule target tells what type of comment the rule (inline, block, or documentation) is addressed. Moreover, it provided information about which object is addressed (class, interface, method, field, *etc.*) and therefore helped and identifying whether a rule is class comment related. In case a rule needed more context to make it understandable we extracted information that gives the rule more context. As code comments are written in source code files, some conventions given for the code also apply to comments *e.g.*, file formatting, indentation, or the maximum number of characters allowed per line. Therefore, We collected the rules that can apply to comments as well.

Extraction order We extracted rules in the order they appeared on the style guide's web page to facilitate their traceability for the future, and to help understand the context of the comments. As the rules can be scattered across multiple sections, paragraphs, or sentences within a paragraph, we gathered which component of each comment the rule refers to and thus grouped the rules according to the comment type. We kept the wording as used in the style guide to avoid changing the interpretation of the rule during the validation. Some rules were phrases and were extracted from a sentence that had multiple rules within a sentence. In such cases, we added the necessary and missing information, which usually was the comment type or component the rule targets. For example, the rule from the Sun Java Code convention *"Include paragraphs marked with @link or @see tags that refer to the new versions of the same functionality."* recommends the user to add replacement information in the description of the Javadoc *@deprecated* tag. However, this is not clear from the rule alone, so we attached a note to the rule explaining that

the @*deprecated* tag is addressed here. If the solution above was not applicable, we added the missing information directly to the rule but wrote it in *italics* to visually distinguish it from the rest of the rule.

Interpreting rules In addition, to directly mentioning the guideline, sometimes the style guideline gave examples on how to write a comment without further description or elaboration. In such cases, we formulated rules from the examples. For instance, Figure 7.3 shows two examples from Request, and based on these examples, we formulated two rules:

- Single-line docstring, opening and, closing triple quotes should be on the same line.
- Multi-line docstring, opening triple quotes should be followed by text. Closing triple quotes should be on a separate line.

```
def the_earth_is_flat():
    """NASA divided up the seas into thirty-three degrees."""
    pass
```

```
def fibonacci_spiral_tool():
    """With my feet upon the ground I lose myself / between the sounds
    and open wide to suck it in. / I feel it move across my skin. / I'm
    reaching up and reaching out. / I'm reaching for the random or
    whatever will bewilder me. / Whatever will bewilder me. / And
    following our will and wind we may just go where no one's been. /
    We'll ride the spiral to the end and may just go where no one's
    been.

    Spiral out. Keep going...
    """
    pass
```

Figure 7.3: Docstring examples of Requests

As another example, the Google Java Style Guide stated that the formatting of Javadoc single-line comments should look as follows: *"/** An especially short bit of Javadoc. */"* In this case, we formulate that *"before and after the comment delimiters should have a space"* and *"the comment should fit on a line, and should start and end with the comment delimiters"*.

Rule exceptions If rules mention cases where it is permissible to not follow them, we considered such exceptions in validating the comments. For instance, the Google Java Style Guide states that the number of characters on a line should not exceed 100 but lines with more than 100 characters are accepted if they contain shell commands that may be copy-pasted into a shell are an exception. Thus, if a comment contained a shell command, and it exceeded the line length limit, we considered the comment as *followed*.

Rule splitting After having extracted all rules from the style guide and the secondary sources (if mentioned), we split the rules (if possible) so that each rule can be validated individually. Each split rule was documented to preserve the traceability to its main rule (or sentence). For instance the rule "the

@deprecated description in the first sentence should at least tell the user when the API was deprecated and what to use as a replacement." is split into "the **@deprecated** description in the first sentence should at least tell the user when the API was deprecated." and "the **@deprecated** description in the first sentence should at least tell the user what to use as a replacement.".

A single style guide has, in most cases, rules across several web pages. We considered references to other web pages and extracted the rules from them as well. We paid close attention to preserve the source of each extracted rule for traceability. We followed the same process for extracting the rules as we used for the main style guide.

Once we gathered all the rules from all the style guides, the next step is to categorize the rules into various rule types. The following section describes various categories and their intention.

7.1.5 Categorizing the rules

We categorized the rules into various categories so that future studies interested in focusing on a specific aspect of comments, can focus on a category. Rani *et al.* defined two categories (*Content* and *Writing style*) of the rules, we extended their taxonomy by adding three additional categories.

- **Content.** Rani *et al.* defined the content type rules as the rules that provide which types of information comments should contain [6]. For example, the rule *"If the member has no replacement, the argument to @deprecated should be 'No replacement'."* from the Sun Java Code convention is a rule of type content.
- **Structure.** This type tells the organization of the information type inside the comment. They clarify how the different information types inside the comment should be organized and informs about where they should be located. Structure rules are not only about the information organization inside the comment but also around the comment and in the source code itself. A typical example would be how the information in the comment should be listed, eg. the order of tags, sections, etc.
- **Formatting.** The rules provide information about how comments should be indented within the source code, whether the components of the comment should be separated by a blank line, and if so by how many. Rules of this type usually complement structure rules and add details to the formations and use of spacing, indentions of the information types. In case a rule suggests white space in combination with another character, we consider this rule as syntax. *e.g.*, *"This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tag"* (from Oracle).
- **Syntax.** Syntax rules tell you how to specifically write something. *"If you have more than one paragraph in the doc comment, separate the paragraphs with a <p> paragraph tag"*
- **Writing style.** This type of rule entails anything about language-specific like grammar, punctuations, and capitalization. Such rules explain which words to use which to avoid. For example the rule from

Django *"Avoid use of 'we' in comments, e.g. 'Loop over' rather than 'We loop over'."* is a *Writing style* rule.

- **Others.** Rules that do not fit in any of the categories mentioned above.

7.2 Convention Adherence

7.2.1 Rule conditions

Depending on the rule, we adapted our validation strategy. Some rules can be either considered as *followed* or *not followed*. Such rules typically suggest specific information should be present in the comment. Rules that can fall under all three categories usually suggest how a comment component should be written. Depending on whether the component is written as recommended the comment is considered to be following the rule or not. In case of the absence of the comment component, the rule cannot be applied.

For instance, for the rule from the Sun Java Code Convention "*If the member has no replacement, the argument to @deprecated should be 'No replacement'.*" we defined the following set of conditions: The class comment follows this rule if the description of the @deprecated tag contains the text "No replacement" and thus labeled *followed*. The comment does not follow the rule (*not followed*) if something besides "No replacement" is present in the corresponding text. We decided to be strict here since the rule clearly states what should be written. The rule is not applicable if a replacement of the deprecated instance is provided or if the @deprecated tag is not present in the class comment.

Bibliography

[1] N. Bafatakis, N. Boecker, W. Boon, M. C. Salazar, J. Krinke, G. Oznacar, and R. White. Python Coding Style Compliance on Stack Overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 210–214. IEEE, 2019.

[2] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan. On the Relationship between Comment Update Practices and Software Bugs. *Journal of Systems and Software*, 85(10):2293–2304, 2012.

[3] M. Ochodek, R. Hebig, W. Meding, G. Frost, and M. Staron. Recognizing lines of code violating company-specific coding guidelines using machine learning. *Empirical Software Engineering*, 25(1):220–265, 2020.

[4] L. Pascarella and A. Bacchelli. Classifying code comments in Java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 227–237. IEEE, 2017.

[5] P. Rani, S. Panichella, M. Leuenberger, A. Di Sorbo, and O. Nierstrasz. How to Identify Class Comment Types? A Multi-language Approach for Class Comment Classification. *Journal of Systems and Software*, 181:111047, 2021.

[6] P. Rani, S. Panichella, M. Leuenberger, M. Ghafari, and O. Nierstrasz. What do class comments tell us? An investigation of comment evolution and practices in Pharo. *CoRR*, abs/2005.11583, 2020.

[7] E. Rodrigues and L. Montecchi. Towards a Structured Specification of Coding Conventions. In *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 168–16809. IEEE, 2019.

[8] A. J. Simmons, S. Barnett, J. Rivera-Villicana, A. Bajaj, and R. Vasa. A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2020.

[9] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia. Maintainability and source code conventions: An analysis of open source projects. *University of Alberta, Department of Computing Science, Tech. Rep. TR11*, 6, 2011.

[10] D. Steidl, B. Hummel, and E. Jürgens. Quality Analysis of Source Code Comments. *2013 21st International Conference on Program Comprehension (ICPC)*, pages 83–92, 2013.

[11] M. Steinbeck and R. Koschke. Javadoc Violations and Their Evolution in Open-Source Software. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 249–259. IEEE, 2021.

[12] Y. Ueda, T. Ishio, and K. Matsumoto. Automatically Customizing Static Analysis Tools to Coding Rules Really Followed by Developers. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 541–545. IEEE, 2021.

[13] J. Zhang, L. Xu, and Y. Li. Classifying Python code comments based on supervised learning. In *International Conference on Web Information Systems and Applications*, pages 39–47. Springer, 2018.