UNIVERSITÄT
BERN

Tool Support for Commenting Conventions

Bachelor's Thesis

Michael Dooley

from

University of Bern

Faculty of Science, University of Bern

July 30, 2021

Prof. Dr. Oscar Nierstrasz

Pooja Rani, Nataliia Stulova

Software Composition Group

Institute of Computer Science

University of Bern, Switzerland

Abstract

Code comments play an important role in program comprehension and maintenance tasks. They are written in natural language, and are of a semi-structured or unstructured nature. Due to this, assessing their quality is a difficult task. One of the ways to assess comment quality is to verify if comments follow the respective coding style guidelines or not. Previous works have proposed to automatically assess comment quality using various linters or static tools. However, the extent to which these tools support various comment conventions is unknown.

We thus assessed how well-known style checkers for Python and Java check comments and compare these to style guidelines.

Even the best tool is useless if not used; as such we also analyzed how style checkers are used in practice by analyzing 48 well-known open source projects written in Python and Java respectively. The projects vary in domain, size, and contributors.

We discovered that current style checkers generally do not check every convention found in style guidelines, with coverage being typically below 25%. We further found that style checker usage for comments is largely individualistic for some style checkers, while others most commonly use their default configuration. We also found that style checker violations do not always seem to be corrected, or usage of style checkers seems to have been disregarded or forgotten.

Contents

1 Introduction

It is well known that code comments help developers understand the code they are working on. However it can be confusing if individual developers do not write or update documentation or do so in different ways. As such, there is an aim to have most code documented and keep the comments consistent. To write consistent comments, there are various style guidelines for programming languages.

Style guidelines provide conventions for where and how to write documentation comments, in addition to detailing best practices on how to write code. They provide a baseline example for what a "good" comment is. However there are usually multiple major guidelines; for Java there are code conventions from Sun/Oracle and from Google, while for Python there are style guidelines laid out in PEP, but also by Google and Numpy.

Style checkers are designed to ensure consistent coding style across a project and often include checks for comments. Most examples of style checkers are linters. They are static analysis tools that are typically run via the command line and they usually report a list of violations of the checks that are reported in a terminal (section 3.1.2 discusses this in detail). Commonly used style checkers are Checkstyle and PMD for Java and Pylint, Flake8, and Black for Python.

There has been some prior research conducted to analyze the quality of comments [4, 9], but these largely focused on writing tools to evaluate quality themselves, and are rather limited in terms of broadness when it comes to the quality of comments. To our knowledge, there has been no scientific analysis of how

style guidelines and style checkers compare to each other for code comments specifically. As such it is unknown how well style checkers actually support comment conventions laid out in style guidelines.

Beller *et al.* have analyzed how often style checkers are used and how they are configured in open-source projects [2]. They found that style checkers are used commonly, but not ubiquitously and that their configuration is usually only slightly changed from the default values. However, this work does not specifically look at comments; we thus cannot say if the more general findings of Beller *et al.* also apply to comments, so we analyze how style checkers are used for comments in practice.

We thus stipulate two research questions:

1. *How well do style checkers cover comment conventions from style guidelines?*

   In order to answer this question, we manually mapped all rules for comments found in style guidelines or style checkers together to see what conventions are covered by what style checker, what is not covered and what is covered despite not being in any guidelines. We applied a taxonomy by Abukar and Rani [1] to the rules, which categorizes the rules into various categorizes (formatting, content, syntax, structure, and writing style), in order to see if there were any significant differences in what types of rules are checked or not checked by style checkers.

2. *To what extent are style checkers used and how are they configured for code comments in practice?*

   For this we gathered various repositories from GitHub and extracted the usage and configurations of the style checkers used within. We then ran the style checkers in the configuration specified by the project on the repository to see if the specifications set by the project were actually followed or not.

We found that style checkers do not currently offer great coverage of comment conventions found in style guidelines in general. However, results for what types of rules are checked and what types are not checked differed too much by language for us to draw a definitive conclusion from the data we gathered.

We also found that the usage of style checkers varies greatly depending on the style checker, with some usually being used in close to a default configuration, while others are commonly individualized to a high degree.

1.1 Organization

In this thesis, we first introduce style guidelines and style checkers in more details (3), then detail our methodology, results and discussion for each research question individually (3, and 4). We lastly note our threats to validity (5) and what kind of future work might be done in this direction (6).

2

Related Work

There have been scientific tools written to analyze the quality of comments; however these tools do not help in evaluating other style checkers that are available to developers. Khamis *et al.* have written a tool called *JavadocMiner* which assesses the quality of inline Javadoc comments based on how easy it is to understand a comment (using natural language processing) and how complete a comment is [4]. This tool however does not verify the numerous other rules laid out in style guidelines. Steinbeck and Koschke analyzed Javadoc violations in open-source projects using their own tool, but looked only for Javadoc that is entirely missing, that does not contain recommended tags or that contains incorrect tags [9]. Thus, they missed many other comment conventions related to the content of the comments such as if comment is formatted according to the style guidelines or not.

There has been research into how many issues style checkers find in software projects; these however do not focus on comments specifically, but rather on stylistic issues in general. Simmons *et al.* ran Pylint on a large number of Python projects in order to compare data science projects and non-data science projects in terms of stylistic adherence, however do not question the completeness of Pylint for such purpose and also do not focus on comments specifically [7]. Beller *et al.* analyze the usage of style checkers in open source software projects at a large scale [2]. They take both a quantitative approach to see how many projects use style checkers and a qualitative approach to see how they are used by analyzing configuration files and surveying the developers. However, their work does not focus explicitly on comments, and in fact their results regarding configuration differs from our results when focusing specifically on comments.

Furthermore our research includes additional tools that Beller *et al.* did not analyze.

Ochodek *et al.* in their work validated if code conforms to guidelines by using a machine learning approach [6]. This is the only work we could find that specifically tries to match code to guidelines written for humans; they however do not specifically focus on comments. They mainly focused on the feasibility of the use of machine learning in this context, coming to the conclusion that it works better for rules relating only to a single line of code.

Ueda *et al.* describe a proof of concept to generate configuration files for static analysis tools using the style checker ESLint for JavaScript as an example [10]. They found that configurations for style checkers do not always include all rules actually followed by projects in the process of analyzing their tool, assuming that this is due to developers not recognizing what rules their project actually follows.

As such while there has been work focusing on style checkers, we found no work specifically combining style checkers and comments; indicating that this is currently a research gap. As comments may differ from code in general, our work tries to fill this gap.

3

Style Checkers' Support of Comment Guidelines

3.1 Introduction

3.1.1 Style Guidelines

Developers often read code for program comprehension and maintenance tasks. Apart from strict syntax conventions, there are various aspects of code that compilers do not check but which play an important role in high quality code that is consistent, readable, and maintainable. To control these aspects, various programming language communities such as those of Java and Python provide their language specific guidelines. Similarly, organizations such as Google, Apache and projects such as Numpy provide their own customized style guidelines. These style guidelines dictate various conventions, *e.g.*, naming conventions, formatting conventions, or comment conventions. (Figure 3.1[1]) For example, "*Maximum line length is* 80 characters."[2] is an example of a formatting convention. Similarly, "*The docstring for a module should generally list the classes, exceptions and functions (and any other objects) that are exported by the module, with a one-line summary of each.*"[3] is an example of a comment convention.

There is no universal style guideline that provides all the guidelines related to all aspects of code. Different guidelines typically have a different focus and thus differ in how strict they are in certain areas.

[1] `https://google.github.io/styleguide/javaguide.html#s7-javadoc` accessed Jun 30, 2021
[2] `https://google.github.io/styleguide/pyguide.html` accessed Jun 30, 2021
[3] `https://www.python.org/dev/peps/pep-0257/` accessed Jun 30, 2021

7 Javadoc

7.1 Formatting

7.1.1 General form

The *basic* formatting of Javadoc blocks is as seen in this example:

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String p1) { ... }
```

... or in this single-line example:

```
/** An especially short bit of Javadoc. */
```

The basic form is always acceptable. The single-line form may be substituted when the entirety of the Javadoc block (including comment markers) can fit on a single line. Note that this only applies when there are no block tags such as `@return`.

7.1.2 Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (`*`)—appears between paragraphs, and before the group of block tags if present. Each paragraph but the first has `<p>` immediately before the first word, with no space after.

7.1.3 Block tags

Any of the standard "block tags" that are used appear in the order `@param`, `@return`, `@throws`, `@deprecated`, and these four types never appear with an empty description. When a block tag doesn't fit on a single line, continuation lines are indented four (or more) spaces from the position of the `@`.

Figure 3.1: Excerpt of *Google Java Style Guide*

Sometimes they also contradict each other, as they are usually not designed to complement each other.

Typically, there are a few well-known style guidelines for a given language. These act as a baseline from which projects lay out their own rules regarding programming style. They do this by adding, disregarding or tweaking a few of the conventions from the guideline.

These style guidelines usually contain conventions for how to write comments. Examples of style guidelines are the *Python Enhancement Proposals* (PEP)[4] for Python or Code Conventions for the Java TM Programming Language[5] for Java. PEP is a collection of guidelines; comment conventions are found in PEP257[6] and, to a lesser extent, PEP8.[7] The Code Conventions for the Java TM Programming Language contain comment conventions in Section 5.[8]

[4] `https://www.python.org/dev/peps/` accessed Jun 30, 2021
[5] `https://www.oracle.com/java/technologies/javase/codeconventions-contents.html` accessed Jun 30, 2021
[6] `https://www.python.org/dev/peps/pep-0257/` accessed Jun 30, 2021
[7] `https://www.python.org/dev/peps/pep-0008/` accessed Jun 30, 2021
[8] `https://www.oracle.com/java/technologies/javase/codeconventions-comments.html` accessed Jun 30, 2021

3.1.2 Style Checkers

As described in the previous chapter, style guidelines dictate various conventions. Developers find it important to ensure their code adheres to the conventions in order to ensure consistency, readability, and maintainability of their code [8]. However, ensuring the adherence of these conventions manually is a tedious and time-consuming task. To ensure it automatically, various static analysis based tools, commonly referred as style checkers or linters are introduced. They check for violations of stylistic practices in the source code of a project. Typically they output a list of violations to the console that they are run in, though some can also be configured to output the result in other formats such as XML or json, or have plugins available that check the code live in an IDE while it is under development. As most style checkers are command-line based it is possible to use them in continuous integration pipelines.

The checks that a style checker performs are not necessarily identical to the style prescribed by a given project or guideline. For comments, usually there are some checks implemented, but not necessarily all that are required by any style guideline.

However, some style checkers claim to support one or multiple particular guidelines: Pylint, pycodestyle and pydocstyle all support PEP, with the latter also having configurations for Google's and Numpy's Python style guides and Checkstyle supporting both Sun's and Google's conventions on how to write Java code.

3.1.3 Differences in Style Guidelines and Style Checkers

Style guidelines provide conventions on how to write "good" code comments. Style checkers often use the style guidelines as a baseline to define their checks and raise a warning when these rules are not followed. As a result style checkers are often used to automate checking code quality across a large project.

There has however been little study as to the completeness of such checks by style checkers when compared against the comment conventions found in style guidelines. We thus formulate and try to answer our first research question *RQ1: How well do style checkers cover comment conventions from style guidelines?*

We further investigate if there is any correlation between what a rule says and if it is checked by a style checker as there has been similarly little research done in this direction. We do this by applying a taxonomy described by Abukar and Rani [1], which separates the rules into five categories based on formatting, writing style, content, syntax, and structure.

3.2 Methodology

As a first step to answering our question, we gathered commonly known style guidelines and style checkers for both Python and Java. We selected these two languages based upon them being the most popular, as

reported by PYPL PopularitY of Programming Language Index.[9] For style guidelines for Python, we found PEP8[10] and PEP257[11] (which we grouped together, as they are meant to be used alongside one another), Google Python Style Guide[12] and Numpy's Style guide,[13] while for Java we found "Code Conventions for the Java TM Programming Language",[14] hosted by Oracle but not updated since 1999, and the Google Java Style Guide.[15]

Abukar and Rani [1] had already extracted the rules in comment conventions out of all of these guidelines and categorized them according to their own taxonomy, whereby a rule belongs to at least one of the categories "Structure", "Content", "Syntax", "Formatting" or "Writing Style".

For style checkers, we found Checkstyle and PMD for Java (Table 7.1), and Pylint, Pyflakes, pycodestyle, Flake8, pydocstyle and Black for Python (Table 7.2). Flake8 is a wrapper around pycodestyle, Pyflakes and a tool to calculate cyclomatic complexity — the last of which is irrelevant to our thesis as cyclomatic complexity cannot be applied to comments. We selected these tools based on them supporting checks for comments as well as being "stand-alone" in terms of interoperability, as defined by Ochodek *et al.* [6]. It should be noted that Black terms itself a code formatter rather than a style checker, as it does not only report violations of its rules, but strives to fix them as well. We also added codespell to the Python corpus at a later stage, as we found Python projects using codespell during our work on Research Question 2 (*To what extent are style checkers used and how are they configured for code comments in practice?*). Although codespell is a language-independent spell checker, proper spelling is mentioned in the Google Python Style Guide, and as such codespell can enforce this rule.

We then extracted all rules from the selected style checkers related to comments. For pydocstyle this is the entirety of the ruleset as pydocstyle is a style checker specifically for Python docstrings. Our approach in this matter was to check the official documentation of all of these tools (Table 7.3) and see which rules related specifically to comments or otherwise corresponded to a convention from a style guideline.

We established the mapping between the rules provided by style checkers and those provided by conventions from style guidelines by considering as equal those rules that describe the same style for the same scope.

For those rules from style checkers that did not correspond to any comment convention from a style guideline we applied the taxonomy employed by Abukar and Rani [1]. We then tallied up the results and interpreted them.

[9] `https://pypl.github.io/PYPL.html` accessed Jul 1 2021
[10] `https://www.python.org/dev/peps/pep-0008/` accessed Jun 21, 2021
[11] `https://www.python.org/dev/peps/pep-0257/` accessed Jun 21, 2021
[12] `https://google.github.io/styleguide/pyguide.html` accessed Jun 21, 2021
[13] `https://numpydoc.readthedocs.io/en/latest/format.html` accessed Jun 21, 2021
[14] `https://www.oracle.com/java/technologies/javase/codeconventions-contents.html` accessed Jun 21, 2021
[15] `https://google.github.io/styleguide/javaguide.html` accessed Jun 21, 2021

3.3 Results

3.3.1 Python Rule Coverage

We found a total of 252 rules for Python; 30 of these were found only in style checkers, while the other 222 are found in style guidelines. Of these latter rules, only 35 (16%) are checked by at least one style checker; meaning that the other 187 (84%) are not checked by any style checker.

In the following paragraphs, we discuss our results specific to each style checker, with a column chart for all of the style checkers found in Figure 3.2.

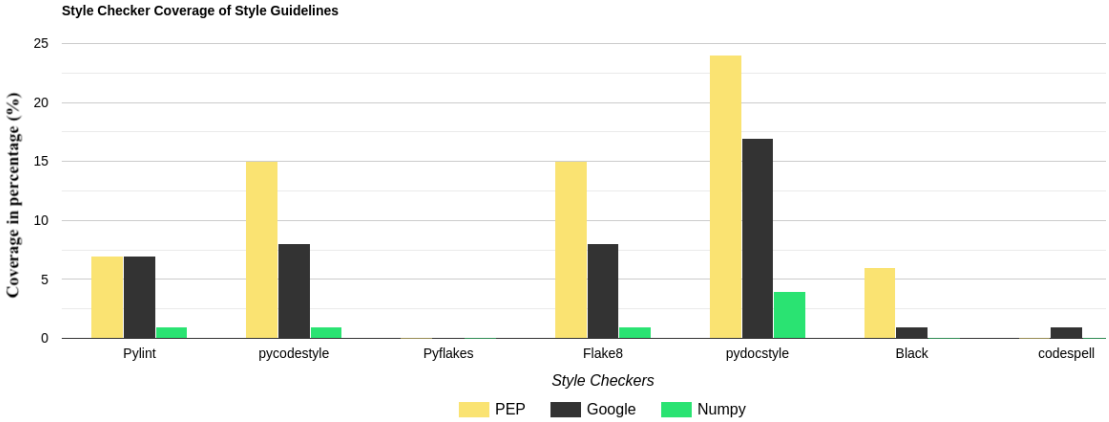

Figure 3.2: Coverage of Python rules

Pylint mentions that it mainly supports the PEP8 guideline,[16] which mentions that docstrings should follow PEP257. In total, there are only eight rules related to code comments that Pylint can check, only five of which are found in PEP (amounting to a coverage of 7%); of the other three rules one each is found in Google's and Numpy's guidelines, and one found in none.

Flake8 is a wrapper around pycodestyle, Pyflakes and mccabe — a tool which calculates cyclomatic complexity. In total Flake8 covers 14 rules related to comments and docstrings — two of these come from Pyflakes with the other twelve stemming from pycodestyle. Neither of Pyflakes's checks come from any guideline, with them simply checking for syntactic correctness of examples and type comments. Of the twelve rules checked by pycodestyle, ten are based upon comment conventions from PEP guidelines (15% coverage). This is not surprising, as pycodestyle specifically tries to implement the style described in PEP8.

pydocstyle checks 48 comment- and docstring-related rules (26% of all rules we found) — significantly more than any other style checker for Python. This however is not surprising, given that pydocstyle is

[16] `http://pylint.pycqa.org/en/latest/intro.html` accessed Jun 21, 2021

a style checker specifically for docstrings. pydocstyle tries to support all three of the style guidelines (PEP257, Google, Numpy), its default configuration however being based on PEP257 specifically. Yet of all of pydocstyle's rules, only 16 correspond to comment conventions found in PEP, while 13 correspond to comment conventions described by Google and only four correspond to conventions found in Numpy's Style guide. This amounts to 24% coverage of PEP, the most out of any Python style checker/style guideline coverage, but still not even a quarter of all rules found within this style guideline. Google's style guidelines are covered to 17% and those of Numpy to 4%.

Black checks only four rules related to comments; which is not surprising given that Black needs to be able to also correct violations, not only report them, as it is a code formatter rather than a style checker.[17] All of Black's corrections are based upon PEP.

codespell only checks spelling, which is to be expected of a spell checker. It does so by flagging all instances of a word found anywhere in the files it is run on that are also found in its list of common misspellings, which is based upon a list curated by Wikipedia editors.[18] We only included codespell in this thesis as Google's Style Guide specifically mentions that comments should be spelled correctly.

Among Python Style Checkers, we found surprisingly little overlap of rules checked, apart from the obvious overlap between Flake8 and its dependencies. Notably pydocstyle and Flake8 have only a single rule that is checked by both (disallowing tabs), while the maximum line length is the only rule checked by three style checkers. Other than the line-length check, there is also no overlap between Pylint and Flake8 when it comes to comments and docstrings, although Pylint does overlap on three rules with pydocstyle, and can check spelling if PyEnchant is installed, thereby overlapping with codespell. Other than codespell, Black is the only tool that does not check anything comment-related that is not also covered by another style checker — with the maximum line-length also covered by Pylint and Flake8, two more rules checked by Flake8 and one covered by pydocstyle.

In some cases, the default configurations of the style checkers do not exactly correspond to the style guidelines they claim to support; see 7.1.2 for the details of these.

Finding 1. Python coverage of comment conventions is below 25% for all comment conventions when checked by a single tool.

Finding 2. 42% of Python rules found in guidelines are content-related while only 3% of rules additionally added by style checkers fit this category.

Finding 3. pydocstyle has the best coverage of comment conventions among all Python tools analyzed — however there is little overlap with other tools, meaning that even better results can be achieved when using multiple tools.

[17] `https://github.com/psf/black/blob/main/docs/the_black_code_style/current_style.md` accessed Jun 21, 2021

[18] `https://github.com/codespell-project/codespell/blob/master/README.rst` accessed Jul 5, 2021

3.3.2 Java Rule Coverage

We found 216 rules for Java, of which 44 were found only in style checkers. However, 80% of the rules found in guidelines were not checked by any style checker we analyzed.

In the following paragraphs, we discuss our results specific to each style checker; a visual presentation of these results can be found in Figure 3.3.

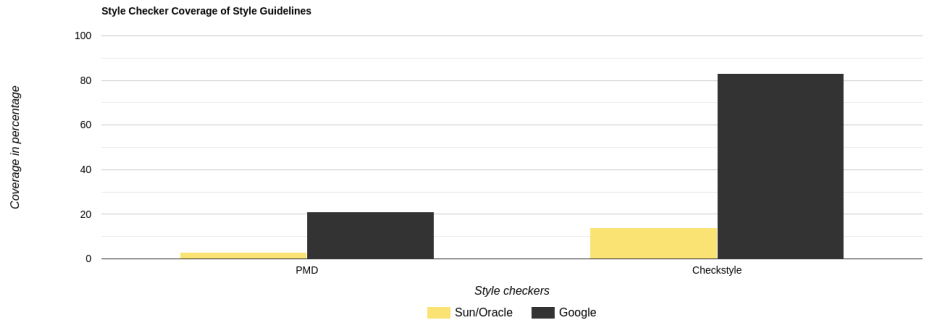

Figure 3.3: Coverage of Java rules

Checkstyle covers far more rules than PMD, with Checkstyle checking 73[19] and PMD eleven.[20] However 39 of Checkstyle's rules do not correspond to a comment convention in a style guideline, while 22 correspond to a rule in Oracle/Sun's style guide and 20 correspond to one set by Google. As Google's guide only contains 24 rules related to comments, this is a coverage of 83% — the highest among all style checkers for a guideline. For the Oracle/Sun guidelines, the coverage is only 14%. However, we also found that Checkstyle's default configurations are not perfect matches for the guidelines that they are based on even when only looking at what Checkstyle can already do.

PMD only checks eleven rules, and of these only six actually correspond to a style guideline. Comparing the style checkers, PMD does not check any rule from a convention in a guideline that Checkstyle cannot also check. There is no overlap between PMD's and Checkstyle's additional checks.

Finding 4. Checkstyle offers a higher coverage of comment-related rules than PMD.

Finding 5. Default configurations of style checkers do not always match the guideline they supposedly correspond to.

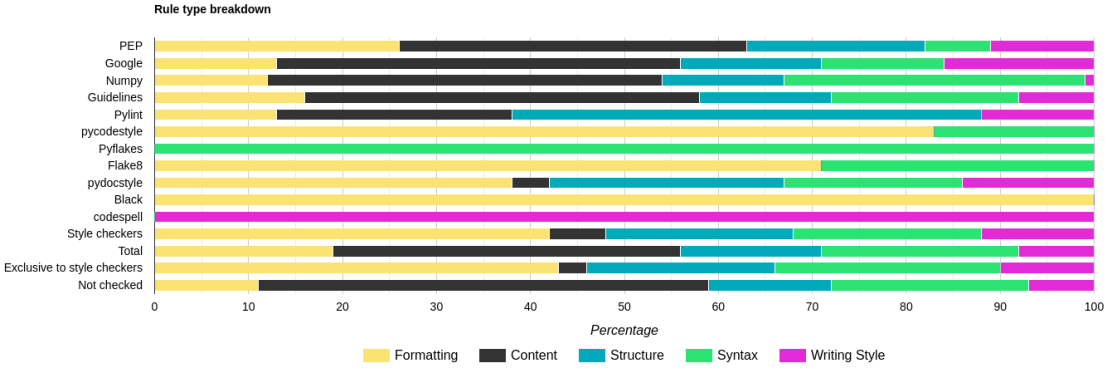

Figure 3.4: Python rule types

3.3.3 Types of rules checked by Python style checkers

As can be seen in Figure 3.4, we found that rules exclusive to style checkers are often formatting rules (43%), and are very rarely content rules (3%). We however found no significant difference in the ratio of rules from conventions and the ratio of rules from conventions not checked by any style checker. However we noticed that formatting conventions found in guidelines were the most likely to be checked (40% of formatting guidelines being checked) by any style checker while only 3% of content conventions found in guidelines were checked. Writing style conventions were covered above average at 28%. However, these results differ significantly when only looking at individual style checkers.

3.3.4 Types of rules checked by Java style checkers

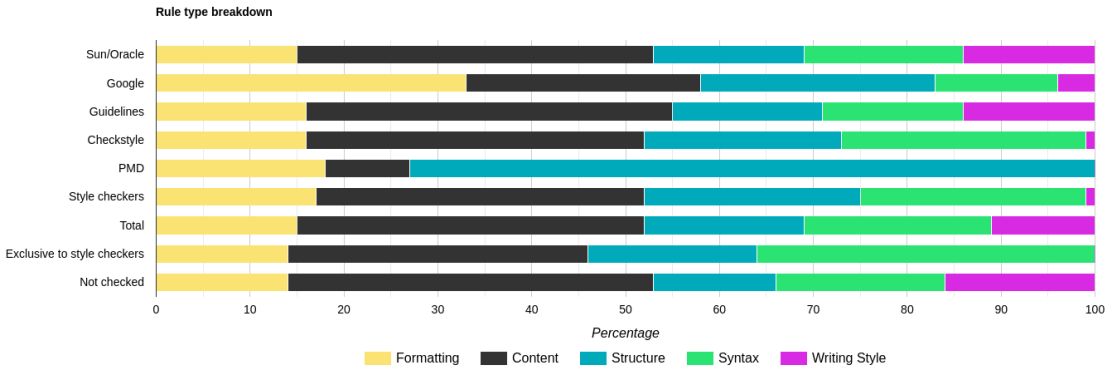

Figure 3.5: Java rule types

We found that rules added by style checkers are more likely to be syntax-related, and less likely to describe content or writing style. We visualize the breakdown of the types of rules in Figure 3.5.

[19] `https://checkstyle.sourceforge.io/checks.html` accessed Jun 21, 2021
[20] `https://pmd.github.io/latest/pmd_rules_java.html` accessed Jun 21, 2021

Once again we found no significant deviation in the ratios between all rules and rules not covered by style checkers. However, only a single writing style convention was covered by a style checker — the one introduced by Google. Structure conventions are checked in 36% of the cases compared to an average of 20% while syntax rules are only covered to 11%. Formatting rules are covered to 26%, while content rules are close to the average at 19%.

Finding 6. Structure and formatting rules are slightly more likely to be checked by style checkers than other types of rules, but for other types we cannot draw a conclusion from only Python and Java.

3.3.5 Implication and Discussion

We found that style checkers currently do not generally support enough rules to be considered a satisfactory validation of comments against a particular guideline. The exception is Checkstyle checking Google's Java guideline, but in that case this can be largely accredited to Google's guideline being fairly loose. We also found that some style checkers are already better than others, with pydocstyle and Checkstyle offering significantly better results than other style checkers for Python and Java respectively. For Python it may also be beneficial to use multiple style checkers to check comments, while for Java PMD adds little value when used alongside Checkstyle.

We found that some types of rules are checked more frequently than others; however there is little correlation in the extremes between Java and Python style checkers, meaning we cannot present a generalized finding from looking at only these two cases. Formatting and structure rules are however checked slightly above average in both languages.

We also found that default configurations of style checkers sometimes do not correspond to what style checkers themselves claim to support, even if the relevant checks are implemented, such as Pylint defaulting to an allowed line length of 100. This can be confusing for developers, as if they just want to add a style checker to a project they are unlikely to do extensive research on whether the default style checker configuration is working as advertised. It is possible that this is the result of style guidelines changing over time with style checkers not adapting the changes, though we did not verify this. Checkstyle's documentation however specifically mentions the term "at-clause" when referring to tags that should be located at the end of a Javadoc comment, which it states had been used in Google's Style guideline until 2017, but acknowledges that this is no longer the case and thus explains it. However both of the "at-clause" rules still map to a rule in Google's Guidelines, only that these now use the term "block tag" instead. This finding alone thus cannot prove this assumption. In any case it is likely that this would not be the case for all such inconsistencies, thus we have little other explanation for why these persist.

3.4 Conclusion

We found that current style checkers do generally not offer a high decree of coverage of comment-related conventions found in style guidelines. There were however significant differences in the degree different

tools cover different guidelines. We did however not find any significant deviation in what types of rules are checked more or less frequently that is present in both Python and Java. Both languages do have types of rules checked more or less often; but they do not correlate.

4

Usage of Style Checkers in Practice

4.1 Introduction

Several research studies showed that developers are interested in various automation strategies like automatic generation of comments, detection of bad comments and verification of comment quality by assessing their content or style. However, a limited set of tools support such automation for comments, especially verifying their quality. The previous chapter highlighted various conventions to ensure their quality and some of the tools which facilitate checking adherence of comments to the conventions. It also listed the conventions extracted from various style guidelines for comments and associated rules formulated in the style checkers to verify if comments adhere to the conventions.

To verify comments' adherence to the conventions, style checkers provide a list of checks. These checks often can be customized to incorporate project or developers' specific requirements and thus provide flexibility. Generally these rules are presented to the developers using a default configuration file along with the style checker. The rules can be customized by manually enabling or disabling. For instance, RuboCop and ESLint[1] provide such a flexible approach, which might be the reason for their wide adoption [3]. However, we still do not know to what extent developers use various style checkers in their projects, what configuration they adopt, what rules they prefer and which rules they disable in the context of code comments. This is important to know in order to understand what challenges developers

[1] `https://eslint.org/`

face in using style checkers for code comments. We thus formulate our second research question: *To what extent are style checkers used and how are they configured for code comments in practice?* To answer this research question, we investigated various popular open source projects available on GitHub. We explain the criteria to choose these projects later in the methodology section. We extracted the usage of style checkers and their associated configuration from these projects and also ran the style checkers in order to understand how they are used.

We found that some style checkers are often used close to their default configuration for comments, while others are usually heavily individualized. We also found that *pydocstyle*, the tool that offers the best coverage for Python, is rarely used (8% of projects). For Java we found that a significant portion of projects (44%) do not use a style checker that can check comments.

4.2 Methodology

Our goal was to understand how style checkers are used in practice for comments. In order to achieve this, we first gathered the 100 most-starred repositories on GitHub in both the Python and Java languages (the most popular programming languages according to the PYPL PopularitY of Programming Languages Index,[2]) as reported by a GitHub search sorted by most stars. GitHub provides a large number of open-source applications which we could analyze; we believed 100 projects would provide a sufficiently sized sample while still being manageable to handle manually. We chose the most-starred ones as we believe this to be the tangible criterion most likely associated with a project that is maintained according to the most recent practices, but acknowledge that it is not a perfect metric for this purpose. We then discarded all projects that were not suitable for our analysis.

This ultimately left us with 48 projects each in Java and Python.

As a next step, we gathered style checker(s) the project uses, versions they use as well as configurations for the style checker(s) used. We considered a project to be using a style checker if it was listed as a requirement (for example in a pom.xml file for Java projects), loaded during continuous integration or when it had any configuration file for a specific style checker present.

Using this approach, we found all nine of the tools analyzed in Research Question 1 (*How well do style checkers cover comment conventions from style guidelines?*) used in at least three projects, enabling us to analyze all of them.

We then cloned the repositories that used style checkers and installed the versions of the style checkers used by the projects.

We then ran the style checker with the version and configuration specified by the project in the project repository, but as our focus is on code comments, we selected only those rules of the style checkers that refer to code comments.

[2] `https://pypl.github.io/PYPL.html` accessed Jul 1 2021

We only ran the style checker against those files specified by the project, if there was any such specification, or all relevant files if there was no such configuration.

We then interpreted the gathered data and the output of the style checkers. For Checkstyle we grouped the violations by module as that was the level that is configurable; for the others we grouped them by check.

4.3 Results

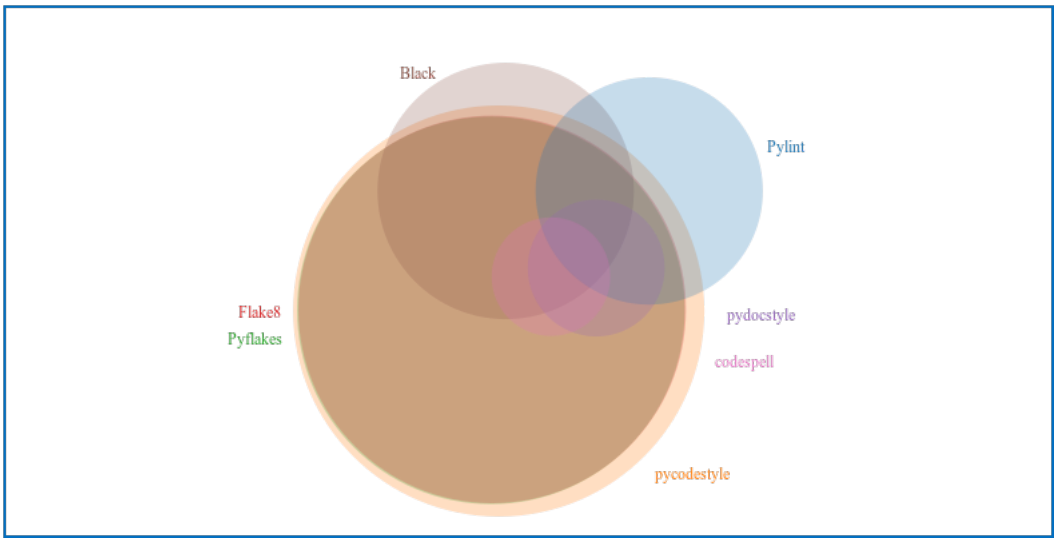

Figure 4.1: Python Style Checker Usage

Among the 48 Python projects investigated (as listed in Table 7.13), we found only seven that did not use any style checker that checks comments. Of our projects, 32 use Flake8, 14 use Black, eleven use Pylint, four use pydocstyle and three use codespell. pycodestyle is already included in Flake8, yet some projects use both. Only three projects use pycodestyle without Flake8. The same is true for Pyflakes, but no project uses Pyflakes standalone. We present a view of the overlap in Figure 4.1.

For Java, 21 of our 48 projects (listed in Table 7.14) do not use a style checker, while 18 use Checkstyle only, one uses PMD only and eight use both. This is visualized in Figure 4.2.

4.3.1 Versions

We found in general that Python projects either do not specify a version for their style checkers, or keep the version relatively recent. For Java, we found that style checkers are often used through a plugin, which is updated less often; but even with less updates they are not always used in the most recent versions.

Finding 1. Python projects either do not specify a version for their style checkers, or keep the version relatively recent.

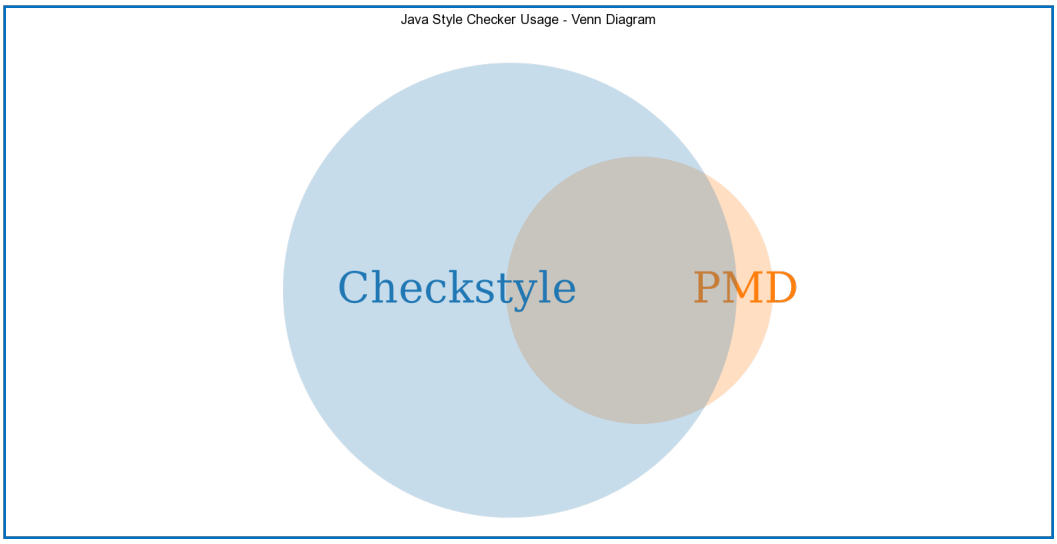

Figure 4.2: Java Style Checker Usage

4.3.2 Configuration and Reported Violations

4.3.2.1 Python

Pylint has nine checks that can be enabled that relate to comments; three of these require PyEnchant and check spelling; they were not used by any of the projects we looked at. Of the eleven projects, two did not check anything related to comments, one does not have a configuration and one uses the default configuration. Commonly changed options include disabling the different checks for missing docstrings (between three and four projects), as well as changing the max line length — done by seven projects, including both that then do not check it. We found values of 88, 100, 110 and 1000 used.

None of the projects we checked — other than the two that do not check comments (*sqlmap* and *glances*) — fully satisfied the configuration they set out for Pylint for comments themselves (see Table 7.23). Our table also provides Pylint's names for codes.[3] The results were largely project-dependent; the only consistencies across projects being that C0112 (*empty-docstring*) and C0144 (*non-ascii-name*) were never violated, while C0301 (*line-too-long*) was almost always violated (six times out of seven that we could check).

Flake8 has 19 checks, one of which is deactivated by default. This was not activated in any of the 32 projects we checked, however almost all of the other checks were active in almost all projects (see Table 7.24, with Table 7.25[4] [5] for the error codes). The exception to this trend was E501, which checks

[3] `http://pylint.pycqa.org/en/latest/technical_reference/features.html` accessed Jun 21, 2021
[4] `https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes` accessed Jun 21, 2021
[5] `https://flake8.pycqa.org/en/latest/user/error-codes.html` accessed Jun 21, 2021

line length — this was only active in 19 projects, however eight of the projects that have it deactivated instead use either Black or Flake8 Bugbear to check line length. The maximum line length used by E501 was also commonly changed to a different value. Other than that, *YouCompleteMe* disables three checks, *tornado* 2 and *spaCy* disables one; only E266 (*Too many leading '#' for block comment*) is disabled in more than one project, however.

We found that 20 out of 32 projects (including *spaCy* in the latter number only) did not report any violations for their comment-related checks, with one project (*you-get*) accounting for 841 out of 1140 violations alone. Most checks were violated by two or three projects, with notable deviations being E302 (*expected 2 blank lines, found 0*) in ten projects (including *spaCy*) and E303 (*too many blank lines*) and E501 (*line too long*) in seven projects each. Four checks were never violated, those being E112 (*expected an indented block*), E113 (*unexpected indentation*), F721 (*syntax error in doctest*) and F723 (*syntax error in type comment*). Notably this includes both checks implemented in Pyflakes (those prefixed with *F*).

Of the projects using pycodestyle without Flake8, as seen in Table 7.26 (with Table 7.25[6] for the error codes) *httpie* disables E501 while *numpy* disables E265 and E266. *ansible* also changes the maximum line length to 160. When run, violations are only reported for *numpy*. The breakdown of violations was similar to the one found for the same rules when checking Flake8.

pydocstyle was used with the config based upon the PEP convention for all projects that used it; in the case of *celery* we could not locate a configuration at all, however. As seen in Table 7.27 (with Table 7.28[7] providing an overview of the error codes) *core* did not change any settings, while *airflow* deactivated eight checks, and *tqdm* deactivated two. When run, we found violations in all four projects, with some violations being more common than others. D202 (*No blank lines allowed before function docstring*) was the most common violation, being reported 1003 times across three projects — this however being largely due to a high number of violations in a single project, which has very little other violations. Different checks for missing docstrings (D100-D107) accounted for over half of the violations, with D401 (*First line should be in imperative mood*) being the only other check with a somewhat noteworthy number of violations. We also found ten checks that were checked, but never violated.

Black has barely anything that is configurable in an attempt to ensure stylistic consistency across projects that use it; as such there is very little that could even be changed to begin with. However, five projects out of 14 increased the maximum line length. We did not run Black, as we did not find a possibility of having it only report what we were interested in, or having it report these cases differently than any others.

Running *codespell*, we found 36 misspellings in *celery* and none in *core* and *pandas*. As codespell checks more than just Python files it should be noted however that only 23 of these misspellings were found in Python files — and in those not necessarily in comments or docstrings.

[6] `https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes` accessed Jun 21, 2021
[7] `https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes` accessed Jun 21, 2021

Finding 2. For Python we found that the tool that offers the best coverage (pydocstyle) is only rarely used and if it is used we found it reports a comparatively high number of violations.

4.3.2.2 Java

Checkstyle is the most configurable style checker we looked at, with 30 modules doing some checks on comments. We quickly found that Checkstyle configurations are often highly customized, as is evident from Table 7.29, Table 7.30 and Table 7.31, with only two out of 26 projects using one of the default configurations supplied by Checkstyle (both the Google one). Six projects did not activate any comment-related modules in their configuration. The most commonly used module was *JavadocMethod* — used in ten projects, which is not even 40% of the projects using Checkstyle. Eight modules were not used by any of the projects. Five projects also use custom Regexp checks to ban something related to comments.

Due to the varied configurations, results for individual modules can hardly be compared across projects as their usage heavily varies. Indeed in almost all cases of modules being violated a large percentage of the violations stem from a single project, thus one cannot claim there to be any larger trend present. When looking only at the number of violations for the different repositories, five out of 19 projects that have at least one active module that checks comments and ran to completion had no violations, while a further six had fewer than five violations. On the other side of the spectrum, *elasticsearch* reported 48699 violations, by far the most out of any project using any style checker. All of these violations stemmed from a configuration file that seemed to be intended to be loaded by an IDE rather than the regular configuration also present within the project, which did not report any violations.

Six out of the nine projects that use PMD do not have any checks for comments activated. In the remaining three projects *CommentRequired* and *CommentDefaultAccessModifier* are never loaded (though it should be noted that *shardingsphere* would load the latter if its version of PMD already included the check), while *CommentSize* and *UncommentedEmptyMethodBody* are loaded once and *CommentContent* and *UncommentedEmptyConstructor* twice, as illustrated in Table 7.32. In the one case where *CommentSize* is loaded, the maximum comment size is increased to 40 lines of 160 characters each, more than eleven times the default size.

RxJava is the only of the three project that passes all of its checks, although this is less impressive when one considers that the only rule it actually checks is that comments do not contain the words “idiot” or “jerk”. Indeed this rule is also the only one checked by projects that is never violated. *tinker* and *shardingsphere* report a total of 37 violations across both projects, with *UncommentedEmptyMethodBody* being violated the most — 21 times, albeit only in one project.

Finding 3. Some style checkers are usually used close to their default settings, while the others are usually heavily customized.

Finding 4. Style checkers report a high number of comment-related violations for some projects seemingly using them.

4.3.3 Implication and Discussion

We found that projects tend to configure comment-related rules for Checkstyle, PMD and Pylint, but mostly only change the maximum line length for others. This contradicts previous findings by Beller *et al.* [2], who found that only few rules in style checker default configurations seem to be the subject of major disagreement, with the style checkers they analyzed including Checkstyle, PMD and Pylint. This indicates that this behavior is either particular to comment-checking, or that there has been a shift in how style checkers are configured since Beller *et al.* published their findings. It is notable that the former three style checkers all work with their own configuration files while Flake8, its components, pydocstyle and Black mostly use command-line flags that can be stored in specific files like tox.ini. With the former three style checkers, this mostly results in only a few checks being run, while in the latter cases most checks are run. We cannot definitively prove why the differences in configuration architecture leads to these results; but we theorize that this could be due to style checkers that largely rely on command-line flags have most of their checks enabled by default, while those that come with a default configuration file usually only have a few selected rules active in the default configuration. When configuring the style checkers one would thus need to check the documentation for what kinds of checks exist that need to specifically be enabled that are not in the default file; while if almost all are enabled to begin with one can simply run the style checkers and see which checks result in undesired violations. We suspect the second approach would leave rarely seen violations enabled. Furthermore, this second approach would leave more checks enabled with minimal effort, as not every developer installing a style checker is necessarily going to read through the documentation of every check the style checker offers to see what they want. Furthermore if the style checker updates to have new checks, these would be enabled by default if one needs a flag to disable them, while they would not be enabled if they need to be specified in a configuration file which is already in use. In the latter case, developers may not even notice a style checker being updated if this is done through an automated process such as DependaBot,[8] a tool which automatically updates dependencies installed in a project.

We also found that all projects resorting to a default configuration file for PMD use the maven plugin, thus using its default rather than PMD's quickstart configuration. Contrary to PMD's quickstart config, the maven default configuration does not check any comment-related rules. It thus seems that if PMD should check comments, adding this to the default configuration of the maven plugin would be beneficial.

For Python we found that the tool that offers the best coverage (pydocstyle) is only rarely used, and if it is used we found it reports a comparatively high number of violations.

However when it comes to usage, we often found cases where a style checker produced numerous results, indicating that despite the style checker being used the style checker violations were not followed. There can be two reasons for this: either the violations are simply ignored, or the style checker is not actually used. In some cases the style checker is only run on changed code, meaning that we may have found violations only in code that was committed before the introduction of the style checker. We did not verify this however, so this remains a speculation. We also found multiple cases of a style checker not

[8] `https://dependabot.com/` accessed Jul 1 2021

running due to an outdated configuration file — we assume in this case an automated update occurred to an unused style checker, with nobody noticing that the update broke the style checker.

We did not track precisely how a style checker was used by a project, so we can only speculate as to the reasons for these violations. Perhaps tracking different ways a style checker is used in relation to the number of violations it found could be the basis of further study.

We tracked the number of violations any used style checker found in the projects, but find it difficult to come to a convincing conclusion regarding the individual results, as we often did not have enough data points to make any form of generalized claim, and those data points that we do have seem to often be statistical extremes rather than indicative of a general trend. We thus summarize that violation behavior is very individual to the project.

4.4 Conclusion

We find that usage of style checkers that support comments is common, but that there are some projects that do not use style checkers, and that usage of style checkers is more prevalent among Python projects than Java projects. However, for Python the style checkers used more commonly are not the best at detecting nonconformity with style guidelines in comments. For Java, rules that check comments are often deactivated, but which ones are active is largely individual to the specific projects. We theorize that this may be due to differences in the types of configuration that these style checkers offer.

We further find that just because a style checker is there, it does not necessarily mean that it is used, as in some projects a style checker raised numerous issues with comments even in the configuration that the project had specified.

5

Threats to Validity

We used manual analysis in almost all crucial steps; this means that it is possible that our own interpretation of the given data may have influenced the findings which are presented as fact.

5.1 Research Question 1

The mapping of rules from style guidelines and style checkers was done manually. As we used the wording of style guidelines and the rules of style checkers as a base it is likely that some rules could be split into multiple rules if one were to look at them more granularily while others could be combined due to them covering the same aspects, therefore altering the number of rules, influencing the numbers derived from the mapping. It is also possible that we missed a mapping due to misunderstanding of a rule or a check and thus considered it to be a new rule. This threat could have been weakened if we had had more researchers available to do the mapping and then compare their results.

While in all cases of disagreement over the type of a rule we either had multiple authors discuss the type of a rule or discussed it with the main researcher from Abukar and Rani's thesis [1], for some rules the type remained somewhat unclear; for these we took a decision, but it is possible that other people may have different views on these cases.

5.2 Research Question 2

As a matter of convenience, we only analyzed open source projects. This might introduce a bias, as practice in closed-source environments may be different.

We attempted to find all used style checkers and associated configuration files, but it is possible that we missed some. While configuration files are usually present in one of a few commonly used ways for any given style checker, some project use custom build tools, or ways of configuring a style checker not mentioned in the style checker's documentation. It is possible that we missed one or multiple such cases. It is also possible that we missed some mentions of files a style checker should not be run on, meaning we included files specifically excluded by the project in our count of violations reported by the style checker. We tried to mitigate this issue by conducting a more thorough search if a style checker reported violations.

Similarly it is possible (and in fact we consider it likely) that we were too broad in categorizing a project as "using" a style checker, as they might only be installed and either never been used or no longer are in active use by the project. Beller *et al.* [2] similarly note that this is likely not a reliable definition of the term. In their work, they surveyed developers whether style checkers are used in the projects they work on; however they also state that this probably introduces a bias towards developers who use style checkers, as they are more likely to answer such a survey. We could not think of any other way of determining such a definition.

6

Conclusion and Future Work

In conclusion our primary finding is that style checkers in general do not check style guidelines to a high degree of completeness; the average value among Python style checkers claiming to support a particular guideline is 13% and for Java this value is 52%, but this result is heavily influenced by Google's guide being very loose and thus well-supported by only a small number of checks. pydocstyle provides the highest coverage among Python style checkers, but has little overlap with others, meaning that using multiple style checkers may be beneficial. For Java, Checkstyle has significantly better coverage than PMD when it comes to comments. In general when comparing style checkers, they differ widely in terms of what types of rules they check and what they do not check. However, structure and formatting rules are consistently checked more often than other types of rules, albeit not always by a significant margin.

We further found that style checkers that cover comments are rarely used, as we found pydocstyle used in only four of 48 projects we analyzed (8%), while for Java a significant portion of projects do not use any style checker — and for those that do use one, they often have most rules inactive that check comments.

We also found that style checker usage heavily depends on the project for style checkers that use dedicated configuration files, while those that use command-line arguments tend to use the default settings for comments more often. We also found that the reported violations of style checkers are sometimes ignored, be it deliberately or due to non-use of the style checkers.

As we have found that the available style checkers do not check a large proportion of rules, future work

might include writing a tool that does.

Also, as our results found little overlap between programming languages when it came to the type results, it might be beneficial to look at more programming languages to see if one of the two languages analyzed here is a statistical anomaly, and whether there is actually some kind of trend at large as to what type of rules are checked by style checkers.

Third, we suggest to conduct an analysis of how style checkers are used, as we found more cases of outdated configurations or seemingly unused configurations than expected, but did not investigate as to how this happened. We also found projects report large numbers of violations, but did not check if this was related to *e.g.*, projects not using the style checker in continuous integration as this was not the focus of our thesis.

7

Anleitung zu wissenschaftlichen Arbeiten

7.1 Mapping

7.1.1 Creating the Mapping

We created a mapping of all rules from style guidelines and style checkers to one another to identify instances of a rule in a style checker corresponding to a rule laid out in one or multiple guidelines. We did this separately for Java and Python, as we were not interested in finding rules that apply to multiple programming languages.

We defined a rule from a style checker to be equivalent to a rule from a convention when both cover the same scope of the code, and the same content. For example, "*Limit all lines to a maximum of 79 characters*" from PEP and "*Maximum line length is 80 characters*" from Google's style guidelines for Python correspond to C0301 (*line-too-long*) in Pylint, E501 (*line too long*) from pycodestyle, which is also inherited by Flake8, and "*Black defaults to 88 characters per line*" from Black. In this case, all of the style checkers provide ways to change the maximum line length allowed, so they can cover both 79 and 80 character limits.

In order to create the mapping, we needed to sometimes split rules based on the scope of what style checkers check or what the rules entail. For example, Google's Python style guidelines state that one should "*[p]ay attention to punctuation, spelling, and grammar*". There are style checkers that check

spelling (Pylint and codespell), but none that check punctuation or grammar. As such we decided to add our own definition for each rule along with its scope, and then map all of the rules on to this. In our example, this resulted in three rules being created — "*Pay attention to punctuation*", "*Pay attention to spelling*" and *Pay attention to grammar*". This allowed us to have a base list of existing rules, to which we then could compare the individual style checkers and style guidelines. It further allowed us to ensure that each rule was in fact unique, as some of the style guidelines cover the same content in multiple sections, leading to duplicate rules in the dataset. This approach allowed us to merge these into only a single rule.

The full mapping for both Java and Python can be found at `https://github.com/Mecanoman/replicationPackage`.

7.1.2 Python Mapping Results

We found a total of 252 rules for Python as shown in the column *Total* of Table 7.4. There are 30 Python rules that we found only within style checkers, in other words they do not cover any convention we found in a guideline. On the other hand, 187 rules from comment conventions found in Python style guidelines are not checked by any style checker, representing 84% of all rules from guidelines.

A full overview of the number of rules for each style checker can be found in Table 7.4 and the percentages in Table 7.5.

In the following subsections, we discuss differences of the style checkers when compared to the guidelines they claim to support.

7.1.2.1 Pylint

There is a single instance where we found a rule being implemented, but with the default settings not corresponding to PEP8: the maximum line length allowed by Pylint defaults to 100, while PEP8 sets it at 79. One rule (spelling, covered by three Pylint checks) requires *PyEnchant*[1] (a package that provides bindings for the Enchant spelling library) to be installed and properly set up, something not mentioned upon installation of Pylint. C0112 (*empty-docstring*) is not found in any guideline, though the rationale for checking for this is immediately clear: the output message includes *"it would be too easy"*, indicating that this check is mainly included to prevent developers from outsmarting Pylint's *missing-module-docstring*, *missing-class-docstring* and *missing-function-docstring* checks by including empty docstrings rather than actually writing documentation.

7.1.2.2 Flake8 / pycodestyle

Only one of the rules checked by pycodestyle was not found in the conventions extracted by Abukar and Rani; [1] that being the indentation of comments. This was checked by E112 (*expected an indented block*), E113 (*unexpected indentation*) and E117 (*over-indented*). The only other rule checked that does not match PEP is that spaces and tabs may not be mixed; this is only specifically mentioned in the Google Python

[1] `https://pypi.org/project/pyenchant/` accessed Jul 1 2021

Style Guide as PEP states that spaces are generally preferred to tabs. We further found that there is a check that comments may not be longer than a certain line-length (79 by default) which is deactivated by default due to it already being covered by the more generic check for a maximum line-length on all lines. However PEP8 specifically states that "*long blocks of text with fewer structural restrictions (docstrings or comments)*" should not exceed 72 characters, while the limit for regular lines of code is 79 characters. This means that in its default configuration, which should check PEP8, a check that is implemented and covers a point in the guideline it should be checking is specifically deactivated.

7.1.2.3 pydocstyle

The default configurations of pydocstyle only match the guidelines for 17 out of the 48 rules (35%). The PEP configuration of pydocstyle contains 15 rules that are not found in the PEP guidelines, while not checking that "*The summary line may be on the same line as the opening quotes or on the next line*". pydocstyle is able to enforce either, but there is no way to allow both — accordingly, neither of the checks is activated in the PEP configuration. For Google, 26 rules are checked despite them not being found in the guidelines. Only one rule is implemented in pydocstyle and listed in Google's Style Guide, but is not activated in pydocstyle's Google configuration. This is for module docstrings' first phrases to end with a period — which one can verify with pydocstyle check *D400*. The most likely reason for this is that the rule cannot be configured to only check module docstrings, as other summary lines may also end with exclamation points or question marks when following Google's guidelines. Numpy has 35 rules checked that are not in their guideline, however upon further research we found that Numpy takes PEP as a base and expands upon their conventions, bringing this number down to 22.

7.1.2.4 Black

Black sets the maximum line-length to 88 — 10% more than 80, although the documentation suggests that 90 might be an even better threshold citing a speech by Raymond Hettinger at PyCon 2015.[2] This is contrary to PEP, which sets the maximum line length at 79.

7.1.3 Java Mapping Results

We found 216 rules for Java as shown in the column *Total* in Table 7.6. Of these, 44 were found only in style checkers — not corresponding to conventions set out in any guideline. Conversely we found 138 rules from style guidelines that are not verified by style checkers, representing 80% of all rules from style guidelines.

All numbers are presented in Table 7.6, with the associated percentages in Table 7.7.

In the following subsections, we discuss some noteworthy points of the style checkers.

[2] `https://www.youtube.com/watch?v=wf-BqAjZb8M&t=260s` accessed Jun 21, 2021

7.1.3.1 Checkstyle

It should be noted that the Checkstyle documentation explains rule modules, rather than rules directly, but further lists all rules that a module contains without further details at the rule level. A module is a bundle of rules that share some similarity in what they check; for example the *SummaryJavadoc* module includes rules to check that the summary of a Javadoc comment exists, ends with a period, does not contain specified phrases and is valid HTML. If a rule was documented to be included in a module but with no explanation as to what it does, we looked at Checkstyle's test cases to establish what it does.

Out of the 22 rules that correspond to a Sun convention, six are not activated in the default Sun configuration of Checkstyle, while an additional 16 rules are activated that the guideline does not call for. This is even more extreme with the Google configuration, where 22 rules not found in Google's style guide are added by Checkstyle. One rule implemented in Checkstyle is also not active despite being in Google's style guide (comments being present for every member of a class, with only methods, constructors, annotation fields and compact constructors being checked in the Google configuration). In some cases these additional rules are justified, for example an @param tag not corresponding to a parameter is not something that is likely to be done due to a lack of awareness of a particular style guideline but rather due to a mistake during refactoring; it would also seem odd to mention that @param should only be used for existing parameters in a style guideline.

However in some cases these additional rules being included in the default configuration for a particular guideline can be considered incorrect as the guideline does not call for them. For example, Checkstyle's Google configuration checks that there is exactly one line break between a Javadoc comment and a statement; however the Google Java Style Guide only states that a statement must be preceded by a line break, not that there must be a line break after a Javadoc comment, nor that there may not be more than one.

While not related directly to comments, we found a comment by a Google developer reasoning why Guava does not use Checkstyle and listing — among other reasons — that "*Either Checkstyle or its 'Google style' config is frequently incorrect*".[3]

Among the 39 checks that do not correspond to conventions from style guidelines that are checked by Checkstyle, a significant portion are either highly customizable rules designed for custom styles (21%) or validate HTML in the Javadoc (21%) (which no style guideline explicitly mentions need be valid — though a reader is likely to assume that when a guideline calls for HTML it is in fact calling for *valid* HTML). There are two rules specifically justified for inclusion: the first is disallowing trailing comments — comments that use /* */ delimiters located after a statement on the same line — referring to Steve McConnell's *Code Complete* [5] rather than any guideline as the source for this rule. The other is disallowing comments containing *TODO*, with Checkstyle reasoning that when these comments cause a violation they are less likely to be forgotten than when they do not.

The remaining 54% of rules not found in guidelines are composed of easy-to-validate rules, such as

[3] `https://github.com/google/guava/issues/5362` accessed Jun 21, 2021

there being a line break between a javadoc comment and a statement or an @deprecated tag only being used once within a comment. Checkstyle also has the possibility to add checks for the existence of any Regexp in a file; we did not include these in the overall total of rules that Checkstyle checks as they can check virtually any rule, as long as it is possible to write a Regexp corresponding to the result.

Furthermore, a lot of checks in Checkstyle are highly customizable; *TodoComment* for example can be reconfigured to disallow any Regexp in a comment. In some cases, this high degree of customization can cause certain checks to overlap.

7.1.3.2 PMD

PMD checks far fewer rules than Checkstyle: only eleven, and of these only six actually correspond to a style guideline. The first one of these rules is checking line length for comments, while another rule corresponds to five rules that a Javadoc comment must exist for a certain element. Curiously this rule can also be inverted, specifically disallowing comments for certain elements. The additional checks added are somewhat unorthodox: it is possible to enforce a maximum number of lines a comment may have (by default: six), require to comment empty constructors and methods within the constructor or method, require comments for default access modifiers and keep comments inoffensive. The last check mentioned in its default setting means checking that a comment contains neither the words "*idiot*" nor "*jerk*". Using this check in its default configuration may be misleading, as it is easy to think of other offensive words in the English language that developers could reasonably expect to be included in the filter. It however is the developers' responsibility to add these words to the filter.

In its quickstart configuration, only *UncommentedEmptyConstructor* and *UncommentedEmptyMethodBody* are checked which means that in this configuration PMD does not validate any comment-related rules in any guideline.

Comparing the style checkers, PMD does not check any rule from a convention in a guideline that Checkstyle cannot also check. There is no overlap between PMD's and Checkstyle's additional checks.

7.2 Taxonomy

Abukar and Rani present a taxonomy to categorize the rules from guidelines into five categories such as formatting, structural, syntax, content, and writing style [1]. This taxonomy is defined as follows: a rule is categorized as a formatting rule if it describes whitespace-related styling, *i.e.*, indentation, spacing between different parts of a comment, line breaks and empty lines, both within and surrounding the comment. An example would be "*Leave one blank line after the one-line summary*", found in Google's Python style guide. The structure category relates to the internal structure of the comment, *i.e.*, the ordering of certain elements, as well as the position of the comment as a whole within the program (mainly if it should exist in the first place). This includes for example "*The attributes section is below the parameters section*" from Numpy's guidelines or "*Write docstrings for all public functions*" from PEP. Content rules describe *what* a comment should contain, for example "*The docstring for a class should list the instance variables*", found

in PEP. Syntax rules typically describe *how* something should be documented; for example "*If you have more than one paragraph in the doc comment, separate the paragraphs with a <p> paragraph tag*" from Sun/Oracle's guidelines falls in this category. Writing style rules typically relate to language-specific issues. This includes cases relating to aspects such as grammar, spelling and punctuation, but also instructions on what linguistic concepts to use or not use. For instance "*Comments should be complete sentences*" from PEP and "*Use 3rd person (descriptive) not 2nd person (prescriptive)*" from the Sun/Oracle guidelines are found in this category.

As Abukar and Rani [1] were still at a preliminary stage, we independently applied this taxonomy to the same rules and then compared the results. We resolved the differences in opinion regarding the categorization of specific rules through discussion. Most of the differences were the result of either a mistaken understanding of either the taxonomy or of the rule. For those that were not, some of the cases were rules that do not fit any category particularly well (like the encoding of a file in Python — something we eventually categorized as a Content rule). Others were rules that were not particularly clearly defined, or that fit multiple categories; we were however eventually able to settle on a single category for most rules.

7.2.1 Python Rule Types

Of the Python rules we found by combining the rules extracted by Abukar and Rani [1] with those that we found exclusively in style checkers, 48 are formatting-related, 94 are content-related, 37 regard structure, 53 describe syntax and 21 detail writing style (see section 7.2 for the taxonomy). We present a full overview of the numbers in Table 7.8 and the associated ratios in Table 7.9. Of the rules, 13 formatting, one content, six structure, seven syntax and three writing style rules are exclusive to style checkers, while the others are found in guidelines. This means that 43% of rules added by style checkers belong to the formatting category, while for style guidelines this number is only 16%. Inversely, 42% of rules found in guidelines were content-related while only 3% of rules added by style checkers fit this category (numbers rounded to the next integer).

On an individual basis, we found roughly the same percentage of content and structure rules in all three guidelines. Numpy has significantly more syntax conventions than the other two at 32% compared to 7% and 13%, but has significantly less writing style conventions with only 1% compared to 12% and 16% for PEP and Google respectively. This could be the result of Numpy being an extension of PEP and thus having its focus where PEP is loose.

Among rules checked by style checkers, we found significant differences between almost all style checkers in all categories; this means that which type of rules a style checker implements is heavily individual.

7.2.2 Java Rule Types

For Java, we found 33 rules regarding formatting, 81 regarding content, 36 regarding structure, 44 regarding syntax and 24 detailing writing style from both the rules found in the rules extracted by Abukar and Rani [1] and the checks from style checkers, as detailed in Table 7.10. Thereof, six formatting rules, 14 describing content, eight describing structure and 16 describing syntax were exclusive to style checkers. The ratio of rules implemented exclusively by style checkers was thus similar to guidelines for formatting and structure-related rules, while writing style and content rules were added fewer times (0% and 32% compared to 14% and 39% in guidelines) and syntax rules made up a significantly larger portion of rules added by style checkers than those introduced in style guidelines (36% compared to 16%). Table 7.10 contains all relevant numbers while the associated ratios can be found in Table 7.11.

The Sun/Oracle and the Google guidelines had some differences when it comes to their makeup, with Google notably having only one writing style convention. The largest difference was found in formatting conventions, which are more than twice as prevalent in Sun/Oracle as in Google.

For the style checkers, we note that 73% of rules in PMD regard structure, while PMD does not have any writing style or syntax checks. Checkstyle has more rules and it is generally closer to the ratios from the guidelines, with the exception of writing style, where it checks only a single rule — the same one as added by Google.

7.3 Project Selection

We discarded all projects that did not match our criteria described in Table 7.12.

- First, we discarded all repositories that did not contain an actual software project, but rather teaching material, examples, reference lists or anything else that cannot be considered a proper project.
- We next discarded repositories with fewer than five contributors, as adherence to style may be less important when working only within a small group of developers.
- We then discarded all repositories with fewer than 500 commits, as repositories with few commits may not yet be at a stage considered mature.
- We also discarded repositories with no commits to the default branch in 2021 (at the time a span of slightly over four months) to ensure that the repository was still active. It should be noted that we did not discard *keras* despite it claiming itself to be inactive in its README.md file, as *keras* had commits on almost a weekly basis for a few months.
- We further discarded repositories documented in languages other than English as not all rules applicable to English documentation may be applicable to documentation in other languages. Chinese was the only language other than English where we found projects that we then discarded.
- Lastly we discarded projects that are style checkers themselves, as we believed including them

would introduce a bias in our data, as a style checker is likely to use itself to check the style it is written in.

7.4 Determining Style Checker Usage

Some style checkers can be turned off for individual lines of code using noqa comments. If there were only very few noqa comments turning off specific checks in few files, we assumed that this was a sole contributor using the style checker to check his own commits, and not a project-wide attempt at using the style checker. The exception to this rule was *keras*, which disables Pylint in over 200 files despite us finding no indication that it is ever loaded, nor there being any configuration file present or it being mentioned in any documents. Given the number of files referencing Pylint, we considered *keras* to be using Pylint.

In one case (*ExoPlayer*) we found Checkstyle listed in two ignore lists only which seemed to have been generated by the Eclipse IDE. We did not consider *ExoPlayer* to be using Checkstyle.

In case we were unable to determine the style checker version used in the project, we used the latest available version of the style checker. If the project specified a range of the style checker, we assumed the newest version within that range. Similarly if the project did not contain a configuration file, we ran the style checker with the default configuration if the style checker came with one. There was one project (*apollo*) that did not have a configuration, but also used a version of PMD based on a maven plugin that had not yet implemented its default configuration. We did not run PMD on this one as we had no idea as to what checks should and what checks should not be run. We also found one project, *ansible*, (using Pylint) with three configuration files that differed from one another, but could not establish which one was used in what circumstances or on what directories; in this case we ran the configuration file that seemed most likely and compared the results to those by *ansible*'s sanity tests. As these results were completely different we discarded the project, assuming a mistake on our side.

7.5 Running Style Checkers

The git hashes of the projects we ran style checkers on can be found at `https://github.com/Mecanno-man/replicationPackage`.

7.5.1 Style Checker Results Discarded

If there was no configuration present in a project, we ran the style checker(s) against all files in the project, unless this turned up files for which the style checker should obviously not be used on (in one case for example, Checkstyle reported a match with an undesirable regexp in a .png file). If this was the case, we specified that the style checker be only run against files whose file extension matched the language we were checking, *i.e.*, .java for Java style checkers and .py for Python style checkers.

In one case (*graal*), we could not establish for what files a style checker should be used for. We tried building the project as a developer in an attempt to find out how graal uses Checkstyle, but were not successful in building it. We thus discarded this project as well.

7.5.2 Running Pyflakes

We decided to only run pycodestyle and Pyflakes on projects not using Flake8, as Flake8 already includes those two. This resulted in us not running the unbundeled version of Pyflakes against any project. However, the results for running these tools should be identical to the results of Flake8.

7.5.3 Handling of Style Checker Crashes

In some cases, the style checker crashed. We experienced three different reasons for style checkers to crash:

- The first type of crash was caused by a configuration using hooks no longer present in the style checker. Typically this was due to them being renamed. If this was the issue, we renamed the hook in the configuration and then ran the style checker, as the intention of the config file remains clear. However this issue being present probably means that the style checker was not actively used by those projects.
- A second type of crash was due to the style checker being unable to parse a specific file, in which case we discarded the project as we could not determine if the style checkers continued running after the parsing error or not.
- The third type of crash was due to the version of the style checker not running in our environment. There was only one case of this (Flake8 in *spaCy*). We determined this to be due to an incompatibility of an old version of Pyflakes with Python 3.8, which we used for these tests. In this case, we ran the newest version of the style checker, but separated those results from the others as newer versions of the style checker may find issues that the developers would not be aware of with the older version that they actually use.

7.5.4 Old Checkstyle Versions

apollo and *rocketmg* used a version of Checkstyle that did not yet report which module triggered a particular rule; we either attributed each violation to the only module where they could come from, or when they could come from multiple modules we ran each module individually to find out which one triggered a particular check to fail.

7.6 Python Style Checker Usage

Among the 48 Python projects investigated, we found only seven that did not use any style checker that checks comments. Four projects use only Pylint, 14 use Flake8, one uses only Black and another two use only pycodestyle, with the remaining 21 using multiple style checkers. Ten projects use Flake8 and Black, one uses Pylint and Black, one uses Pylint, pycodestyle and Black, one uses Flake8, codespell, Black and pycodestyle, one uses Flake8, pycodestyle and Pyflakes, one uses Flake8 and pycodestyle (but referring to the old name *pep8*), one uses Black, Pylint, Flake8 and pydocstyle, two use Flake8 and Pylint, one uses Flake8 and pydocstyle, one uses Flake8, pydocstyle and codespell and one uses every style checker we analyzed. In total this means we have 32 projects using Flake8, 14 using Black, eleven using Pylint, four using pydocstyle and three using codespell. pycodestyle is already included in Flake8, yet some projects use both. Only three projects use pycodestyle without Flake8. The same is true for Pyflakes, but no project uses Pyflakes standalone.

7.7 Version Results

7.7.1 Python

The newest version of Flake8 at the time of writing was version 3.9.2. This version is explicitly mentioned in only three projects (see Table 7.15), but falls into the range specified in nine further ones. We found that eleven projects do not specify a version, meaning that only nine out of 32 projects using Flake8 specifically use an older version. Of these, one uses version 3.9.1 and two use 3.9.0, which only differ by minor bug fixes from version 3.9.2. Two projects use version 3.8.4, one uses 3.8.3, one 3.8.2 and one 3.8.1, which were all released in 2020. The final project however uses version 3.5.0, which was almost four years old at the time of writing.

The newest version of Black at the time we checked versions was 21.5b1. This was used in only two projects specifically, with another specification including it in its range (see Table 7.16). A further two use 21.4b2 and two use 21.4b0, which do not have significant changes compared to the current version. Four projects use 20.8b1, while two projects use 19.10b0 — the oldest version still in use within our projects. One project uses the version *stable*, which we believe mapped to version 21.5b1 at the time of writing, but may be updated by Black to point to newer versions when they are released.

For Pylint, the version was only specified in four out of eleven projects (see Table 7.17). At the time we checked the versions, 2.8.2 was the most current version, and was used by two projects; in one explicitly while in the other by range. The other versions used were 2.6.0 and 2.4.3 with one project each.

Of the three projects using pycodestyle without also using Flake8, one uses version 2.7.0, one 2.6.0 and one does not specify a version (see Table 7.18). Version 2.7.0 was the newest version at the time of writing.

pydocstyle's newest version at the time of checking versions was 6.0.0 — used by two of four projects

using this style checker. One project did not specify a version, while the last one specified 5.1.1 in their continuous integration, and major release 5 in general outside of continuous integration (see Table 7.19).

Two out of three projects using codespell specified version 2.0.0, the newest at the time of writing; the third project did not specify a version (see Table 7.20).

7.7.2 Java

On the Java side, Checkstyle's newest version when we extracted version numbers was 8.42; at the time this version was used by only two of the 26 projects (see Table 7.21). It should be noted that version 8.43, released shortly after we had wrapped up most of our work additionally supports @summary tags in Javadoc comments to identify the summary sentence. A lot of projects use a maven plugin to load Checkstyle, the newest version of which defaults to loading Checkstyle 8.29, which is accordingly used by six projects. We further found one project using version 8.39, one project using 8.38, two projects using 8.37 (the default loaded by Gradle), one project using 8.36.2, one project using 8.28, three projects using 8.19, one project using 8.17, one project using 8.14, one project using 8.12, one project using 8.11, one project using 8.5 and two projects using 6.11.2. The remaining two projects had conflicting versions specified, with *skywalking* loading both versions 8.19 and 6.18, but using 8.38 in continuous integration, and *graal* referring to version 8.8 in some directories but 8.36.1 in others.

We were unable to determine the version of PMD used in two projects (see Table 7.22); the other seven projects loaded PMD using a maven plugin, which at its newest loads PMD version 6.29.0. The newest version at the time of checking versions was 6.34.0. Nonetheless, none of the projects referred to the newest version of the maven plugin, though one did not specify a version at all (instead referring to a *parent pom*, which we could not locate). As such, we found one instance of 6.21.0, three of 5.6.1, one of 5.3.5 and one of 5.3.2 in use. It should be noted that major release 6 was released in December 2017, meaning that most of the projects that specify a version use a rather old version of PMD. The oldest version we found was 5.3.2, which was released in 2015.

For both Checkstyle and PMD it should be noted that the maven plugins commonly used are updated significantly fewer times than the actual style checkers loaded by the plugin. During the entirety of 2020 for example, the Checkstyle plugin was updated once[4] and the PMD plugin twice;[5] in the same period Checkstyle released 13[6] updates and PMD ten.[7]

[4] `https://maven.apache.org/plugins/maven-checkstyle-plugin/history.html` accessed Jul 1 2021
[5] `https://github.com/apache/maven-pmd-plugin/releases` accessed Jul 1 2021
[6] `https://github.com/checkstyle/checkstyle/releases` accessed Jul 1 2021
[7] `https://github.com/pmd/pmd/releases` accessed Jul 1 2021

7.8 Results of Running Style Checkers

7.8.1 Pylint

We were unable to fully determine how the project *ansible* runs Pylint as we found three different configs in that project. We ran Pylint in the configuration we believed most likely for the project and found 13337 violations; we then also ran the test suite that *ansible* runs without configuring it to only run Pylint and only for comments, with Pylint reporting no violations therein. We thus believe our results for this project to be erroneous and discarded them.

For one project (*ray*) Pylint crashed trying to parse a file in both the version used in *ray* as well as the newest version of Pylint. We thus discarded this project from the results, leaving us with nine projects out of eleven that we could run Pylint on. We should however note that Pylint found two violations of C0301 (*line-too-long*) in *ray* before crashing despite the maximum line length being set to 1000.

It should be noted that checks C0144 (*non-ascii-name*) and C0301 (*line-too-long*) do not necessarily relate to comments, as these are checks that can relate to both comments and code. We did not investigate how often these checks relate to comments.

7.8.2 Flake8 and pycodestyle

Checks E101, E111, E112, E113, E117, E302, E303, E501 and W191 may also relate to code rather than comments. We did not investigate how often they are comment-related.

7.8.3 pydocstyle

The number of checks never violated does not include check D302, which has been deprecated as it only pertains to Python 2 code, and we are unsure if it was actually run despite not being mentioned in the documentation as being ignored by default.

7.8.4 codespell

We did not gather configuration data as for codespell this essentially is a list of allowed words from codespell's misspellings list — something which we believe adds little value to this thesis as this is obviously tailored to the project's needs.

7.8.5 Checkstyle

We found two projects loading a plugin that supposedly validates comments against the spring guidelines — assumedly another set of style guidelines. We did not analyze this plugin as we did not analyze the spring guidelines. Only one of these loads regular Checkstyle comment checks as well.

MissingJavadocType — one of the eight modules never used — was forked by *ElasticSearch*, which then uses the forked version. We did not include this fork in our research.

SingleSpaceSeparator and *IllegalTokenText* are used by multiple projects, but never have comment-checking turned on. For the context of our research we did not consider these modules as used for comments. We also found that *tinker* has configured multiple modules but then set the severity level to *ignore*, thereby causing these modules to not generate any output. We did not consider these modules as being used.

Two projects use custom RegExp checks to ban TODO comments despite there being a module in Checkstyle specifically designed for this. That module (*TodoComment*) can also be repurposed to check for something else, which was done in one case: *flink* uses the module to disallow @fixme, @xxx and @author tags, while allowing to-dos. Table 7.29, Table 7.30 and Table 7.31 contain an overview of the violations we found, with fields marked *N/A* standing for a module not used in the respective project.

Checkstyle was unable to process a file in *skywalking* and crashed after finding two *NonEmptyAtclauseDescription* and one *EmptyLineSeparator*. We did not include this project further in the results section. We were also forced to fix the configuration of *spring-framework* as it uses an outdated configuration that no longer works with the version of Checkstyle it uses.

It should be noted that *SingleSpaceSeparator*, *LineLength*, *IllegalTokenText*, *AnnotationLocation*, *EmptyLineSeparator* as well as the additional RegExp Modules do not necessarily relate to comments, as they either cover both code and comments or contain both rules relating to comments and rules not relating to comments.

7.8.6 PMD

Out of the nine projects using PMD, three do not have a config and three only use PMD to load a plugin to check conformance with Alibaba's style guide. Similar to the Checkstyle plugin for Spring, we did not investigate this plugin any further. The three projects that use the default configuration all use the maven plugin, which has its own default configuration that does not check anything in comments.

We had to fix one configuration file for PMD, as *shardingsphere* used keywords not found in the version of PMD it uses.

7.9 Detailed Tables

Table 7.13: Python projects and style checkers used

Repository	Style Checker(s)
https://github.com/ytdl-org/youtube-dl	Flake8
https://github.com/nvbn/thefuck	Flake8
https://github.com/django/django	Flake8

Continued on next page

Table 7.13 – continued from previous page

Repository	Style Checker(s)
https://github.com/pallets/flask	Flake8, Black
https://github.com/keras-team/keras	Pylint
https://github.com/httpie/httpie	pycodestyle
https://github.com/ansible/ansible	Pylint, pycodestyle, Black
https://github.com/scikit-learn/scikit-learn	Flake8
https://github.com/huggingface/transformers	Flake8, Black
https://github.com/psf/requests	Flake8
https://github.com/home-assistant/core	Pylint, Flake8, pydocstyle, Pyflakes, pycodestyle, codespell, Black
https://github.com/scrapy/scrapy	Pylint, Flake8
https://github.com/soimort/you-get	Flake8
https://github.com/apache/superset	Pylint, Black
https://github.com/python/cpython	None
https://github.com/deepfakes/faceswap	Flake8
https://github.com/3b1b/manim	None
https://github.com/tiangolo/fastapi	Black, Flake8
https://github.com/localstack/localstack	Flake8
https://github.com/pandas-dev/pandas	Flake8, codespell, Black, pycodestyle,
https://github.com/certbot/certbot	Pylint
https://github.com/getsentry/sentry	Flake8, Black
https://github.com/iperov/DeepFaceLab	None
https://github.com/willmcgugan/rich	Black
https://github.com/sherlock-project/sherlock	Flake8
https://github.com/chubin/cheat.sh	None
https://github.com/openai/gym	None
https://github.com/ycm-core/YouCompleteMe	Flake8
https://github.com/hankcs/HanLP	None
https://github.com/docker/compose	Flake8, pycodestyle, Pyflakes

Continued on next page

Table 7.13 – continued from previous page

Repository	Style Checker(s)
https://github.com/mitmproxy/mitmproxy	Flake8
https://github.com/pypa/pipenv	Flake8, Black
https://github.com/apache/airflow	Black, Pylint, Flake8, pydocstyle
https://github.com/encode/django-rest-framework	Flake8
https://github.com/trailofbits/algo	None
https://github.com/explosion/spaCy	Flake8
https://github.com/sqlmapproject/sqlmap	Pylint
https://github.com/tornadoweb/tornado	Flake8, Black
https://github.com/getredash/redash	Flake8, pep8
https://github.com/nicolargo/glances	Flake8, Pylint
https://github.com/tqdm/tqdm	Flake8, pydocstyle
https://github.com/StevenBlack/hosts	Flake8
https://github.com/celery/celery	codespell, Flake8, pydocstyle
https://github.com/numpy/numpy	pycodestyle
https://github.com/magenta/magenta	Pylint
https://github.com/facebookresearch/detectron2	Flake8, Black
https://github.com/locustio/locust	Flake8, Black
https://github.com/ray-project/ray	Pylint, Flake8

Table 7.14: Java projects and style checkers used

Repository	Style Checker(s)
https://github.com/spring-projects/spring-boot	Checkstyle
https://github.com/elastic/elasticsearch	Checkstyle
https://github.com/ReactiveX/RxJava	PMD, Checkstyle
https://github.com/spring-projects/spring-framework	Checkstyle
https://github.com/google/guava	None
https://github.com/square/okhttp	Checkstyle

Continued on next page

Table 7.14 – continued from previous page

Repository	Style Checker(s)
https://github.com/square/retrofit	None
https://github.com/apache/dubbo	Checkstyle
https://github.com/bumptech/glide	Checkstyle
https://github.com/airbnb/lottie-android	None
https://github.com/Blankj/AndroidUtilCode	None
https://github.com/zxing/zxing	Checkstyle
https://github.com/netty/netty	Checkstyle
https://github.com/NationalSecurityAgency/ghidra	None
https://github.com/skylot/jadx	Checkstyle
https://github.com/alibaba/arthas	None
https://github.com/ctripcorp/apollo	Checkstyle, PMD
https://github.com/alibaba/fastjson	None
https://github.com/SeleniumHQ/selenium	None
https://github.com/signalapp/Signal-Android	None
https://github.com/dbeaver/dbeaver	None
https://github.com/google/gson	None
https://github.com/ReactiveX/RxAndroid	None
https://github.com/seata/seata	PMD, Checkstyle
https://github.com/apache/kafka	Checkstyle
https://github.com/alibaba/spring-cloud-alibaba	Checkstyle
https://github.com/libgdx/libgdx	None
https://github.com/square/picasso	Checkstyle
https://github.com/google/ExoPlayer	None
https://github.com/alibaba/nacos	PMD, Checkstyle
https://github.com/jenkinsci/jenkins	PMD, Checkstyle
https://github.com/nostra13/Android-Universal-Image-Loader	None
https://github.com/apache/skywalking	Checkstyle
https://github.com/bazelbuild/bazel	None
https://github.com/facebook/fresco	None
https://github.com/redisson/redisson	Checkstyle, PMD

Continued on next page

Table 7.14 – continued from previous page

Repository	Style Checker(s)
https://github.com/apache/flink	Checkstyle
https://github.com/alibaba/Sentinel	PMD
https://github.com/Tencent/tinker	PMD, Checkstyle
https://github.com/mybatis/mybatis-3	None
https://github.com/oracle/graal	Checkstyle
https://github.com/brettwooldridge/HikariCP	None
https://github.com/openzipkin/zipkin	None
https://github.com/apache/rocketmq	Checkstyle
https://github.com/lottie-react-native/lottie-react-native	None
https://github.com/apache/shardingsphere	Checkstyle, PMD
https://github.com/TeamNewPipe/NewPipe	Checkstyle
https://github.com/LMAX-Exchange/disruptor	Checkstyle

Table 7.27: Reported pydocstyle violations

	core	airflow	tqdm	celery	Used in Projects	Violations	Total projects violating
D100	2	N/A	18	38	3	58	3
D101	0	41	15	8	4	64	3
D102	0	N/A	64	855	3	919	2
D103	3	141	23	74	4	241	4
D104	0	N/A	3	25	3	28	2
D105	0	N/A	24	223	3	247	2
D106	0	0	0	0	4	0	0
D107	0	N/A	13	135	3	148	2
D200	0	145	23	0	4	168	2
D201	0	0	0	0	4	0	0
D202	986	13	4	0	4	1003	3

Continued on next page

Table 7.27 – continued from previous page

	core	airflow	tqdm	celery	Used in Projects	Violations	Total projects violating
D203	N/A	N/A	N/A	N/A	0	0	0
D204	0	0	32	1	4	33	2
D205	3	N/A	58	2	3	63	3
D206	0	0	0	0	4	0	0
D207	0	0	0	0	4	0	0
D208	0	0	0	1	4	1	1
D209	0	0	2	2	4	4	2
D210	0	0	0	0	4	0	0
D211	0	0	0	0	4	0	0
D212	N/A	N/A	N/A	N/A	0	0	0
D213	N/A	N/A	N/A	N/A	0	0	0
D214	N/A	N/A	N/A	N/A	0	0	0
D215	N/A	N/A	N/A	N/A	0	0	0
D300	0	0	0	0	4	0	0
D301	0	1	0	0	4	1	1
D302 (deprecated)	0	0	0	0	4?	0	0
D400	1	N/A	N/A	9	2	10	2
D401	3	N/A	45	72	3	120	3
D402	0	1	3	0	4	4	2
D403	1	2	25	0	4	28	3
D404	N/A	N/A	N/A	N/A	0	0	0
D405	0	0	0	0	4	0	0
D406	N/A	N/A	N/A	N/A	0	0	0
D407	N/A	N/A	N/A	N/A	0	0	0
D408	N/A	N/A	N/A	N/A	0	0	0
D409	N/A	N/A	N/A	N/A	0	0	0
D410	N/A	N/A	N/A	N/A	0	0	0
D411	N/A	N/A	N/A	N/A	0	0	0

Continued on next page

Table 7.27 – continued from previous page

	core	airflow	tqdm	celery	Used in Projects	Violations	Total projects violating
D412	0	0	1	8	4	9	2
D413	N/A	N/A	N/A	N/A	0	0	0
D414	0	0	0	0	4	0	0
D415	N/A	N/A	N/A	N/A	0	0	0
D416	N/A	N/A	N/A	N/A	0	0	0
D417	N/A	N/A	N/A	N/A	0	0	0
D418	0	0	0	N/A	3	0	0
Total	999	344	353	1453	N/A	3149	4

Table 7.28: pydocstyle error codes

D100	*Missing docstring in public module*
D101	*Missing docstring in public class*
D102	*Missing docstring in public method*
D103	*Missing docstring in public function*
D104	*Missing docstring in public package*
D105	*Missing docstring in magic method*
D106	*Missing docstring in public nested class*
D107	*Missing docstring in __init__*
D200	*One-line docstring should fit on one line with quotes*
D201	*No blank lines allowed before function docstring*
D202	*No blank lines allowed after function docstring*
D203	*1 blank line required before class docstring*
D204	*1 blank line required after class docstring*
D205	*1 blank line required between summary line and description*
D206	*Docstring should be indented with spaces, not tabs*
D207	*Docstring is under-indented*

Continued on next page

Table 7.28 – continued from previous page

D208	*Docstring is over-indented*
D209	*Multi-line docstring closing quotes should be on a separate line*
D210	*No whitespaces allowed surrounding docstring text*
D211	*No blank lines allowed before class docstring*
D212	*Multi-line docstring summary should start at the first line*
D213	*Multi-line docstring summary should start at the second line*
D214	*Section is over-indented*
D215	*Section underline is over-indented*
D300	*Use """triple double quotes"""*
D301	*Use r""" if any backslashes in a docstring*
D302	*Deprecated: Use u""" for Unicode docstrings*
D400	*First line should end with a period*
D401	*First line should be in imperative mood*
D402	*First line should not be the function's "signature"*
D403	*First word of the first line should be properly capitalized*
D404	*First word of the docstring should not be* This
D405	*Section name should be properly capitalized*
D406	*Section name should end with a newline*
D407	*Missing dashed underline after section*
D408	*Section underline should be in the line following the section's name*
D409	*Section underline should match the length of its name*
D410	*Missing blank line after section*
D411	*Missing blank line before section*
D412	*No blank lines allowed between a section header and its content*
D413	*Missing blank line after last section*
D414	*Section has no content*
D415	*First line should end with a period, question mark, or exclamation point*
D416	*Section name should end with a colon*
D417	*Missing argument descriptions in the docstring*
D418	*Function/ Method decorated with @overload shouldn't contain a docstring*

Table 7.1: Style Checker Documentation

Style Checker	Interface	Integration	Output Formats	Configuration Options
Checkstyle	command-line, Ant	multiple IDEs, CI, Maven, Gradle, Codacy, some other tools	text, XML, SARIF	rules can be activated/deactivated; rules often highly customizable
PMD	command-line, Maven, Ant, Gradle	CI, Codacy, multiple IDEs	text, SARIF, codeclimate, csv, Emacs, HTML, ideaj, json, Summary HTML, colored text, Textpad, vbHTML, XML, XSLT, YAHTML	rules can be activated/deactivated; some rules customizable

Table 7.2: Style Checker Documentation

Style Checker	Interface	Integration	Output Formats	Configuration Options
Pylint	command-line	multiple IDEs and text editors, CI	text, json, parseable (Emacs), colorized, msvs	rules can be activated/deactivated; few rules customizable
pycodestyle	command-line	–	text, custom	rules can be activated/deactivated; only line length customizable
Pyflakes	command-line	–	text	none
Flake8	command-line	pre-commit	text	rules can be activated/deactivated; only line length customizable
pydocstyle	command-line	pre-commit	text	rules can be activated/deactivated
Black	command-line	–	text, automatic fixing	maximum line-length can be customized
codespell	command-line	–	text, automatic fixing, interactive	users can create their own dictionaries

Table 7.3: Style Checker Documentation

Style Checker	Documentation
Pylint	`https://pylint.readthedocs.io/en/latest/technical_reference/features.html`
pycodestyle/Flake8	`https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes`
Pyflakes/Flake8	`https://flake8.pycqa.org/en/latest/user/error-codes.html` (Flake8 adds error codes for Pyflakes)
pydocstyle	`http://www.pydocstyle.org/en/stable/error_codes.html`
Black	`https://github.com/psf/black/blob/main/docs/the_black_code_style/current_style.md`
codespell	`https://github.com/codespell-project/codespell/blob/master/README.rst`
PMD	`https://pmd.github.io/latest/pmd_rules_java.html`
Checkstyle	`https://checkstyle.sourceforge.io/checks.html`

Table 7.4: Overlap of Python rules

Overlap	Number of rules	PEP	Google	Numpy	Exclusive to style checkers
Number of rules	252	68	75	110	30
Pylint	8	5	5	2	1
Flake8	14	10	6	1	3
pydocstyle	48	16	13	4	26
Black	4	4	1	0	0
pycodestyle	12	10	6	1	1
Pyflakes	2	0	0	0	2
codespell	1	0	1	0	0
Not checked by any style checker	187	42	56	104	N/A

Table 7.5: Coverage of Python rules

Coverage	Pylint	Flake8	pydocstyle	Black	pycodestyle	Pyflakes	codespell
PEP	7.35%	14.71%	23.53%	5.88%	14.71%	0.00%	0.00%
Google	6.67%	8.00%	17.33%	1.33%	8.00%	0.00%	1.33%
Numpy	1.82%	0.91%	3.64%	0.00%	0.91%	0.00%	0.00%

Table 7.6: Overlap of Java rules

Overlap	Total	PMD	Checkstyle	None
Total	216	11	73	138
Oracle/Sun	160	5	22	138
Google	24	5	20	4
None	44	5	39	N/A

Table 7.7: Coverage of Java rules

Coverage	PMD	Checkstyle
Oracle/Sun	3.13%	13.75%
Google	20.83%	83.33%

Table 7.8: Python rule coverage by type

	Formatting	Content	Structure	Syntax	Writing Style	Total
PEP	18	25	13	5	8	68
Google	10	32	11	10	12	75
Numpy	13	47	14	35	1	110
Guidelines	35	93	31	46	18	222
Pylint	1	2	4	0	1	8
pycodestyle	10	0	0	2	0	12
Pyflakes	0	0	0	2	0	2
Flake8	10	0	0	4	0	14
pydocstyle	18	2	12	9	7	48
Black	4	0	0	0	0	4
codespell	0	0	0	0	1	1
Style checkers	27	4	13	13	8	65
Total	48	94	37	53	21	252
Exclusive to style checkers	13	1	6	7	3	30
Not checked	21	90	24	40	13	187
Not checked %	60.00%	96.77%	77.42%	86.96%	72.22%	84.23%
Checked %	40.00%	3.23%	22.58%	13.04%	27.78%	15.77%

Table 7.9: Python rule coverage by type: ratios

	Formatting	Content	Structure	Syntax	Writing Style
PEP	26.47%	36.76%	19.12%	7.35%	11.76%
Google	13.33%	42.67%	14.67%	13.33%	16.00%
Numpy	11.82%	42.73%	12.73%	31.82%	0.91%
Guidelines	15.77%	41.89%	13.96%	20.72%	8.11%
Pylint	12.50%	25.00%	50.00%	0.00%	12.50%
pycodestyle	83.33%	0.00%	0.00%	16.67%	0.00%
Pyflakes	0.00%	0.00%	0.00%	100.00%	0.00%
Flake8	71.43%	0.00%	0.00%	28.57%	0.00%
pydocstyle	37.50%	4.17%	25.00%	18.75%	14.58%
Black	100.00%	0.00%	0.00%	0.00%	0.00%
codespell	0.00%	0.00%	0.00%	0.00%	100.00%
Style checkers	41.54%	6.15%	20.00%	20.00%	12.31%
Total	19.05%	37.30%	14.68%	21.03%	8.33%
Exclusive to style checkers	43.33%	3.33%	20.00%	23.33%	10.00%
Not checked	11.23%	48.13%	12.83%	21.39%	6.95%

Table 7.10: Java rule coverage by type

	Formatting	Content	Structure	Syntax	Writing Style	Total
Sun/Oracle	24	62	26	27	23	160
Google	8	6	6	3	1	24
Guidelines	27	67	28	28	24	172
Checkstyle	12	26	15	19	1	73
PMD	2	1	8	0	0	11
Style checkers	13	27	18	19	1	78
Total	33	81	36	44	24	216
Exclusive to style checkers	6	14	8	16	0	44
Not checked	20	54	18	25	23	138
Not checked %	74.07%	80.60%	64.29%	89.29%	95.83%	80.23%
Checked	25.93%	19.40%	35.71%	10.71%	4.17%	19.77%

Table 7.11: Java rule coverage by type: ratios

	Formatting	Content	Structure	Syntax	Writing Style
Sun/Oracle	15.00%	38.75%	16.25%	16.88%	14.38%
Google	33.33%	25.00%	25.00%	12.50%	4.17%
Guidelines	15.70%	38.95%	16.28%	16.28%	13.95%
Checkstyle	16.44%	35.62%	20.55%	26.03%	1.37%
PMD	18.18%	9.09%	72.73%	0.00%	0.00%
Style checkers	16.67%	34.62%	23.08%	24.36%	1.28%
Total	15.28%	37.50%	16.67%	20.37%	11.11%
Exclusive to style checkers	13.64%	31.82%	18.18%	36.36%	0.00%
Not checked	14.49%	39.13%	13.04%	18.12%	16.67%

Table 7.12: Project selection process

Criteria	Remaining projects (Python)	Remaining Projects (Java)
100 Most-starred repos	100	100
Actual project	64	78
5 or more contributors	64	75
500 or more commits	54	64
Commits in 2021	51	60
English	49	48
Not a style checker	48	48

Table 7.15: Flake8 versions

Project	Version declared	Assumed version used
youtube-dl	–	3.9.2
thefuck	–	3.9.2
django	3.9.0	3.9.0
flask	3.9.2	3.9.2
scikit-learn	3.7.8 and 3.8.2	3.8.2
transformers	>=3.8.3	3.9.2
requests	–	3.9.2
core	3.9.2	3.9.2
scrapy	–	3.9.2
you-get	–	3.9.2
faceswap	–	3.9.2
fastapi	>=3.8.3,<4.0.0	3.9.2
localstack	>=3.6.0	3.9.2
pandas	3.9.2	3.9.2
sentry	–	3.9.2
sherlock	–	3.9.2
YouCompleteMe	>= 3.0.0	3.9.2
compose	3.8.3	3.8.3
mitmproxy	>=3.8.4,<4	3.9.2
pipenv	>=3.3.0,<4.0	3.9.2
airflow	>=3.6.0	3.9.2
django-rest-framework	3.9.0	3.9.0
spaCy	3.5.0	3.5.0
tornado	3.8.4	3.8.4
redash	–	3.9.2
glances	–	3.9.2
tqdm	3.8.4	3.8.4
hosts	>=3.8,<=4.0	3.9.2
celery	>=3.8.3	3.9.2
detectron2	3.8.1	3.8.1
locust	–	3.9.2
ray	3.9.1	3.9.1

Table 7.16: Black versions

Project	Version declared
flask	21.5b1
ansible	19.10b0
transformers	21.4b0
core	21.5b1
superset	19.10b0
fastapi	20.8b1
pandas	20.8b1
sentry	20.8b1
rich	^20.8b1
pipenv	stable
airflow	21.4b2
tornado	20.8b1
detectron2	21.4b2
airflow	21.4b2
locust	21.4b0

Table 7.17: Pylint versions

Project	Version declared	Assumed version used
keras	–	2.8.2
ansible	2.6.0	2.6.0
core	2.8.2	2.8.2
scrapy	–	2.8.2
superset	–	2.8.2
certbot	2.4.3	2.4.3
airflow	∼=2.8.1	2.8.2
sqlmap	–	2.8.2
glances	–	2.8.2
magenta	–	2.8.2
ray	–	2.8.2

Table 7.18: pycodestyle versions

Project	Version declared	Assumed version used
httpie	–	2.7.0
ansible	2.6.0	2.6.0
numpy	2.7.0	2.7.0

Table 7.19: pydocstyle versions

Project	Version declared	Assumed version used
core	6.0.0	6.0.0
airflow	6.0.0	6.0.0
tqdm	–	6.0.0
celery	5.1.1	5.1.1

Table 7.20: codespell versions

Project	Version declared	Assumed version used
core	2.0.0	2.0.0
pandas	2.0.0	2.0.0
celery	–	2.0.0

Table 7.21: Checkstyle versions

Project	Version used
spring-boot	8.11
elasticsearch	8.39
RxJava	8.37
spring-framework	8.42
okhttp	8.28
dubbo	8.29
glide	8.5
zxing	8.29
netty	8.19
jadx	8.37
apollo	6.11.2
seata	8.29
kafka	8.36.2
sping-cloud-alibaba	8.29
picasso	8.17
nacos	8.29
jenkins	8.42
skywalking	8.38
redisson	8.29
flink	8.14
tinker	8.19
graal	8.8 and 8.36.1
rocketmg	6.11.2
shardingsphere	8.19
NewPipe	8.38
disruptor	8.12

Table 7.22: PMD versions

Project	Version specified	Version assumed
RxJava	–	6.34.0
apollo	5.3.5	5.3.5
seata	5.6.1	5.6.1
nacos	5.6.1	5.6.1
jenkins	–	6.29.0
redisson	6.21.0	6.21.0
Sentinel	5.6.1	5.6.1
tinker	–	6.34.0
shardingsphere	5.3.2	5.3.2

Table 7.23: Usage of Pylint checks and reported violations

	C0112 (*empty-docstring*)	C0114 (*missing-module-docstring*)	C0115 (*missing-class-docstring*)	C0116 (*missing-function-docstring*)	C0144 (*non-ascii-name*)	C0301 (*line-too-long*)	C0401 (*wrong-spelling-in-comment*)	C0402 (*wrong-spelling-in-docstring*)	C0403 (*invalid-characters-in-docstring*)	Total
keras	0	2	816	3757	0	57	N/A	N/A	N/A	4632
core	0	0	0	0	0	362	N/A	N/A	N/A	362
scrapy	0	N/A	N/A	N/A	0	427	N/A	N/A	N/A	427
superset	0	N/A	N/A	N/A	0	6	N/A	N/A	N/A	6
certbot	0	0	68	1969	0	32	N/A	N/A	N/A	2069
airflow	0	210	4	165	0	0	N/A	N/A	N/A	379
sqlmap	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0
glances	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0
magenta	0	0	0	N/A	0	14	N/A	N/A	N/A	14
Total Violations	0	263	888	5891	0	898	0	0	0	7940
Total projects violating	0	2	3	3	0	6	0	0	0	7
ansible	used	used	used	used	used	used	unused	unused	unused	N/A
ray	used	unused	unused	unused	used	used	unused	unused	unused	N/A
Used in Projects	9	6	6	5	9	9	0	0	0	N/A

Table 7.24: Reported Flake8 violations

	E101	E111	E112	E113	E114	E115	E116	E117	E302	E303	E261	E262	E265	E266	E501	W191	W505	F721	F723	Total
youtube-dl	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	N/A	0	0	0
thefuck	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	N/A	0	0	0
Django	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
Flask	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	N/A	0	0	0
scikit-learn	0	0	0	0	2	1	2	1	37	2	7	6	48	0	81	0	N/A	0	0	187
transformers	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	N/A	0	0	0
requests	0	2	0	0	0	0	0	0	1	2	0	0	0	0	N/A	0	N/A	0	0	5
core	0	0	0	0	0	0	0	0	1	0	0	0	0	0	N/A	0	N/A	0	0	1
scrapy	0	0	0	0	0	0	0	0	1	0	3	0	27	0	1	0	N/A	0	0	32
you-get	20	7	0	0	0	6	13	2	292	12	64	40	185	2	168	30	N/A	0	0	841
faceswap	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
fastapi	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	N/A	0	0	0
localstack	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
pandas	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
sentry	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	N/A	0	0	0
sherlock	0	4	0	0	0	0	0	2	1	5	1	0	17	0	2	0	N/A	0	0	32
YouCompleteMe	0	N/A	0	0	N/A	0	0	0	2	N/A	N/A	0	0	0	1	0	N/A	0	0	3
compose	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
mitmproxy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
pipenv	0	0	0	0	0	0	0	0	3	1	0	0	0	0	N/A	0	N/A	0	0	4
airflow	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
django-rest-framework	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	N/A	0	0	0
tornado	0	0	0	0	0	0	0	0	0	0	0	0	N/A	N/A	0	0	N/A	0	0	0
redash	0	0	0	0	0	0	0	0	15	2	1	0	0	0	N/A	0	N/A	0	0	18
glances	1	0	0	0	1	0	0	2	0	2	0	0	1	0	43	1	N/A	0	0	51
tqdm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
hosts	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
celery	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	N/A	0	0	1
detectron2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
locust	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	N/A	0	0	0
ray	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N/A	0	0	0
Violations	21	13	0	0	3	7	15	7	353	26	78	46	278	2	262	31	0	0	0	1142
Total projects violating	2	3	0	0	2	2	2	4	10	7	5	2	5	1	7	2	0	0	0	11
spaCy	0	0	0	0	0	0	0	0	2	0	2	0	0	N/A	N/A	0	N/A	0	0	4
Used in Projects	32	31	32	32	31	32	32	32	32	31	31	32	31	30	19	32	0	32	32	N/A

Table 7.25: Relevant Flake8 and pycodestyle error codes

E101	*indentation contains mixed spaces and tabs*
E111	*indentation is not a multiple of four*
E112	*expected an indented block*
E113	*unexpected indentation*
E114	*indentation is not a multiple of four (comment)*
E115	*expected an indented block (comment)*
E116	*unexpected indentation (comment)*
E117	*over-indented*
E302	*expected 2 blank lines, found 0*
E303	*too many blank lines*
E261	*at least two spaces before inline comment*
E262	*inline comment should start with '# '*
E265	*block comment should start with '# '*
E266	*too many leading '#' for block comment*
E501	*line too long*
W191	*indentation contains tabs*
W505	*doc line too long*
F721	*syntax error in doctest*
F723	*syntax error in type comment*

Table 7.26: Reported pycodestyle violations

	httpie	ansible	numpy	Used in Projects	Violations	Total projects violating
E101	0	0	3	3	3	1
E111	0	0	4	3	4	1
E112	0	0	0	3	0	0
E113	0	0	0	3	0	0
E114	0	0	1	3	1	1
E115	0	0	0	3	0	0
E116	0	0	0	3	0	0
E117	0	0	10	3	10	1
E302	0	0	1031	3	1031	1
E303	0	0	86	3	86	1
E261	0	0	101	3	101	1
E262	0	0	15	3	15	1
E265	0	0	N/A	2	0	0
E266	0	0	N/A	2	0	0
E501	N/A	0	3900	2	3900	1
W191	0	0	2	3	2	1
W505	N/A	N/A	N/A	0	0	0
Total	0	0	5153	N/A	5153	1

Table 7.29: Reported Checkstyle violations

	spring-boot	elasticsearch	RxJava	spring-framework	okhttp	dubbo	glide	zxing	netty
AtclauseOrder	N/A	N/A	N/A	0	0	N/A	N/A	N/A	N/A
InvalidJavadocPosition	N/A	N/A	N/A	N/A	0	N/A	N/A	N/A	N/A
JavadocBlockTagLocation	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocContentLocation	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocMethod	N/A	15691	0	0	0	N/A	N/A	N/A	N/A
JavadocMissingLeadingAsterisk	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocMissingWhitespaceAfterAsterisk	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocPackage	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocParagraph	N/A	N/A	N/A	N/A	20	N/A	N/A	N/A	N/A
JavadocStyle	N/A	N/A	N/A	0	N/A	N/A	14	0	N/A
JavadocTagContinuationIndentation	N/A	N/A	N/A	0	1	N/A	N/A	N/A	N/A
JavadocType	N/A	N/A	N/A	0	N/A	N/A	223	0	N/A
JavadocVariable	N/A	N/A	N/A	1	N/A	N/A	N/A	N/A	N/A
MissingJavadocMethod	N/A	33007	0	N/A	149	N/A	N/A	N/A	N/A
MissingJavadocPackage	N/A	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
MissingJavadocType	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Continued on next page

Table 7.29 – continued from previous page

	spring-boot	elasticsearch	RxJava	spring-framework	okhttp	dubbo	glide	zxing	netty
NonEmptyAtclauseDescription	N/A	N/A	N/A	0	0	N/A	N/A	0	N/A
RequireEmptyLineBeforeBlockTagGroup	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SingleLineJavadoc	N/A	N/A	N/A	N/A	10	N/A	N/A	N/A	N/A
SummaryJavadoc	N/A	N/A	N/A	N/A	3	N/A	N/A	N/A	N/A
WriteTag	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SingleSpaceSeparator	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
LineLength	N/A	0	N/A	N/A	38	N/A	N/A	N/A	N/A
CommentsIndentation	N/A	N/A	N/A	0	37	N/A	N/A	N/A	N/A
TodoComment	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
TrailingComment	N/A	N/A	N/A	N/A	N/A	N/A	145	N/A	N/A
MissingDeprecated	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0	N/A
IllegalTokenText	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AnnotationLocation	N/A	N/A	N/A	0	13	N/A	N/A	N/A	N/A
EmptyLineSeparator	N/A	N/A	N/A	N/A	146	N/A	N/A	N/A	N/A
Additional RegExp checks	N/A	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Total	0	48699	0	1	417	0	382	0	0

Table 7.30: Reported Checkstyle violations

	jadx	apollo	seata	kafka	spring-cloud-alibaba	picasso	nacos	jenkins	redisson	flink
AtclauseOrder	N/A	1	N/A	N/A	0	N/A	0	N/A	N/A	N/A
InvalidJavadocPosition	N/A	N/A	N/A	N/A	N/A	N/A	0	N/A	N/A	N/A
JavadocBlockTagLocation	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocContentLocation	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocMethod	N/A	679	N/A	N/A	N/A	N/A	0	N/A	N/A	0
JavadocMissingLeadingAsterisk	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocMissingWhitespaceAfterAsterisk	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocPackage	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JavadocParagraph	N/A	63	N/A	N/A	N/A	N/A	0	N/A	N/A	0
JavadocStyle	N/A	N/A	N/A	N/A	0	N/A	N/A	N/A	N/A	1
JavadocTagContinuationIndentation	N/A	3	N/A	N/A	0	N/A	N/A	N/A	N/A	N/A
JavadocType	N/A	N/A	N/A	N/A	0	N/A	N/A	N/A	N/A	0
JavadocVariable	N/A	N/A	N/A	N/A	0	N/A	N/A	N/A	N/A	N/A
MissingJavadocMethod	N/A	N/A	N/A	N/A	N/A	N/A	17	N/A	N/A	N/A
MissingJavadocPackage	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Continued on next page

Table 7.30 – continued from previous page

	jadx	apollo	seata	kafka	spring-cloud-alibaba	picasso	nacos	jenkins	redisson	flink
MissingJavadocType	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
NonEmptyAtclauseDescription	N/A	42	N/A	N/A	0	N/A	0	N/A	N/A	N/A
RequireEmptyLineBeforeBlockTagGroup	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SingleLineJavadoc	N/A	1	N/A	N/A	N/A	N/A	1	N/A	N/A	N/A
SummaryJavadoc	N/A	249	N/A	N/A	N/A	N/A	37	N/A	N/A	N/A
WriteTag	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SingleSpaceSeparator	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
LineLength	N/A	2141	N/A	N/A	N/A	17	46	N/A	N/A	N/A
CommentsIndentation	N/A	2	N/A	N/A	0	N/A	0	N/A	N/A	N/A
TodoComment	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0
TrailingComment	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
MissingDeprecated	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
IllegalTokenText	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AnnotationLocation	0	0	N/A	N/A	0	N/A	0	N/A	N/A	N/A
EmptyLineSeparator	0	N/A	N/A	N/A	N/A	N/A	1	N/A	N/A	0
Additional RegEx checks	N/A	N/A	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Continued on next page

Table 7.30 – continued from previous page

	jadx	apollo	seata	kafka	spring-cloud-alibaba	picasso	nacos	jenkins	redisson	flink
Total	0	3181	4	0	0	17	102	0	0	1

Table 7.31: Reported Checkstyle violations

	tinker	rocketmq	shardingsphere	NewPipe	disruptor	Violations	Total projects violating	skywalking	graal	Projects Using
AtclauseOrder	N/A	N/A	0	N/A	N/A	1	1	unused	unused	6
InvalidJavadocPosition	N/A	N/A	N/A	0	N/A	0	0	unused	unused	3
JavadocBlockTagLocation	N/A	N/A	N/A	N/A	N/A	0	0	unused	unused	0
JavadocContentLocation	N/A	N/A	N/A	N/A	N/A	0	0	unused	unused	0
JavadocMethod	N/A	N/A	94	0	0	16464	3	unused	unused	10
JavadocMissingLeadingAsterisk	N/A	N/A	N/A	N/A	N/A	0	0	unused	unused	0

Continued on next page

Table 7.31 – continued from previous page

	tinker	rocketmq	shardingsphere	NewPipe	disruptor	Violations	Total projects violating	skywalking	graal	Projects Using
JavadocMissingWhitespaceAfterAsterisk	N/A	N/A	N/A	N/A	N/A	0	0	unused	unused	0
JavadocPackage	N/A	N/A	N/A	N/A	N/A	0	0	unused	unused	0
JavadocParagraph	N/A	N/A	0	N/A	0	83	2	unused	unused	6
JavadocStyle	N/A	N/A	0	0	N/A	15	2	unused	used	8
JavadocTagContinuationIndentation	N/A	N/A	0	N/A	N/A	4	2	unused	unused	5
JavadocType	N/A	N/A	N/A	0	0	223	1	unused	unused	7
JavadocVariable	N/A	N/A	N/A	N/A	0	1	1	unused	unused	3
MissingJavadocMethod	N/A	N/A	N/A	N/A	N/A	33173	3	unused	unused	4
MissingJavadocPackage	N/A	N/A	N/A	N/A	N/A	1	1	unused	unused	1
MissingJavadocType	N/A	N/A	N/A	N/A	N/A	N/A	N/A	unused	unused	0
NonEmptyAtclauseDescription	N/A	N/A	0	N/A	N/A	42	1	used	unused	8
RequireEmptyLineBeforeBlockTagGroup	N/A	N/A	N/A	N/A	N/A	0	0	unused	unused	0
SingleLineJavadoc	N/A	N/A	0	N/A	0	12	3	unused	unused	5
SummaryJavadoc	N/A	N/A	0	N/A	N/A	289	3	unused	unused	4
WriteTag	N/A	N/A	N/A	N/A	N/A	0	0	unused	unused	0
SingleSpaceSeparator	N/A	N/A	N/A	N/A	N/A	N/A	N/A	unused	unused	0

Continued on next page

Table 7.31 – continued from previous page

	tinker	rocketmq	shardingsphere	NewPipe	disruptor	Violations	Total projects violating	skywalking	graal	Projects Using
LineLength	N/A	N/A	0	0	0	2250	5	unused	used	9
CommentsIndentation	4	N/A	N/A	N/A	N/A	43	3	unused	unused	6
TodoComment	N/A	N/A	0	N/A	4	4	1	unused	unused	3
TrailingComment	N/A	N/A	0	N/A	N/A	145	1	unused	unused	2
MissingDeprecated	N/A	N/A	0	N/A	N/A	0	0	unused	unused	2
IllegalTokenText	N/A	N/A	N/A	N/A	N/A	N/A	N/A	unused	unused	0
AnnotationLocation	0	N/A	0	N/A	N/A	13	1	unused	unused	8
EmptyLineSeparator	N/A	N/A	1	N/A	N/A	148	3	used	unused	6
Additional RegEx checks	0	4	N/A	N/A	N/A	8	2	used	unused	5
Total	4	4	95	0	4	52919	13	N/A	N/A	N/A

Table 7.32: Reported PMD violations

	CommentSize	CommentContent	CommentRequired	Uncommented Empty Constructor	Uncommented Empty MethodBody	Comment Default Access Modifier	Total
RxJava	N/A	0	N/A	N/A	N/A	N/A	0
apollo	N/A	N/A	N/A	N/A	N/A	N/A	0
seata	N/A	N/A	N/A	N/A	N/A	N/A	0
nacos	N/A	N/A	N/A	N/A	N/A	N/A	0
jenkins	N/A	N/A	N/A	N/A	N/A	N/A	0
redisson	N/A	N/A	N/A	N/A	N/A	N/A	0
Sentinel	N/A	N/A	N/A	N/A	N/A	N/A	0
tinker	N/A	N/A	N/A	4	21	N/A	25
shardingsphere	5	0	N/A	7	N/A	N/A	12
Used in Projects	1	2	0	2	1	0	N/A
Violations	5	0	0	11	21	0	37
Total projects violating	1	0	0	2	1	0	2

Bibliography

[1] S. Abukar and P. Rani. Adherence of class comments to the commenting style guidelines. Unpublished, 2021.

[2] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481, 2016.

[3] D. Kavaler, A. Trockman, B. Vasilescu, and V. Filkov. Tool choice matters: Javascript quality assurance tools and usage outcomes in GitHub projects. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 476–487. IEEE, 2019.

[4] N. Khamis, R. Witte, and J. Rilling. Automatic quality assessment of source code comments: The javadocminer. In C. J. Hopfe, Y. Rezgui, E. Métais, A. Preece, and H. Li, editors, *Natural Language Processing and Information Systems*, pages 68–79, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] S. McConnell. *Code Complete*. Microsoft Press.

[6] M. Ochodek, R. Hebig, W. Meding, G. Frost, and M. Staron. Recognizing lines of code violating company-specific coding guidelines using machine learning: A method and its evaluation. *Empirical Software Engineering*, 25, 01 2020.

[7] 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)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[8] M. Smit, B. Gergel, and H. J. Hoover. Code convention adherence in evolving software. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 504–507. IEEE, 2011.

[9] 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, 2021.

[10] 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, 2021.