The Journal of Systems & Software 181 (2021) 111047

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

How to identify class comment types? A multi-language approach for class comment classification

Pooja Rani [a,*], Sebastiano Panichella [b], Manuel Leuenberger [a], Andrea Di Sorbo [c], Oscar Nierstrasz [a]

[a] Software Composition Group, University of Bern, Switzerland
[b] Zurich University of Applied Science, Switzerland
[c] Department of Engineering, University of Sannio, Italy

ARTICLE INFO

Article history:
Received 16 December 2020
Received in revised form 5 June 2021
Accepted 9 July 2021
Available online 19 July 2021

Keywords:
Natural language processing technique
Code comment analysis
Software documentation

ABSTRACT

Most software maintenance and evolution tasks require developers to understand the source code of their software systems. Software developers usually inspect class comments to gain knowledge about program behavior, regardless of the programming language they are using. Unfortunately, (i) different programming languages present language-specific code commenting notations and guidelines; and (ii) the source code of software projects often lacks comments that adequately describe the class behavior, which complicates program comprehension and evolution activities.

To handle these challenges, this paper investigates the different language-specific class commenting practices of three programming languages: Python, Java, and Smalltalk. In particular, we systematically analyze the similarities and differences of the information types found in class comments of projects developed in these languages. We propose an approach that leverages two techniques – namely Natural Language Processing and Text Analysis – to automatically identify *class comment types, i.e.,* the specific types of semantic information found in class comments. To the best of our knowledge, no previous work has provided a comprehensive taxonomy of class comment types for these three programming languages with the help of a common automated approach.

Our results confirm that our approach can classify frequent class comment information types with high accuracy for the Python, Java, and Smalltalk programming languages. We believe this work can help in monitoring and assessing the quality and evolution of code comments in different programming languages, and thus support maintenance and evolution tasks.

1. Introduction

Software maintenance and evolution tasks require developers to perform program comprehension activities (Fjeldstad and Hamlen, 1983; Haiduc et al., 2010). To understand a software system, developers usually refer to the software documentation of the system (Bavota et al., 2013; de Souza et al., 2005). Previous studies have demonstrated that developers trust code comments more than other forms of documentation when they try to answer program comprehension questions (Maalej et al., 2014; Woodfield et al., 1981; de Souza et al., 2005). In addition, recent work has also demonstrated that *"code documentation"* is the most used source of information for bug fixing, implementing features, communication, and even code review (Müller and Fritz, 2013). In particular, well-documented code simplifies software maintenance activities, but many programmers often overlook or delay code commenting tasks (Curiel and Collet, 2013).

Class comments play an important role in obtaining a high-level overview of the classes in object-oriented languages (Cline, 2015). In particular, when applying code changes, developers using object-oriented programming languages can inspect class comments to achieve most or the majority of the high-level insights about the software system design, which is critical for program comprehension activities (Khamis et al., 2010; Nurvitadhi et al., 2003; Steidl et al., 2013). Class comments contain various types of information related to the usage of a class or its implementation (Haouari et al., 2011), which can be useful for other developers performing program comprehension (Woodfield et al., 1981) and software maintenance tasks (de Souza et al., 2005). Unfortunately, (i) different object-oriented programming languages adopt language-specific code commenting notations and guidelines (Farooq et al., 2015), (ii) they embed different kinds of information in the comments (Scowen and Wichmann,

* Corresponding author.
E-mail addresses: pooja.rani@inf.unibe.ch (P. Rani), panc@zhaw.ch (S. Panichella), manuel.Leuenberger@inf.unibe.ch (M. Leuenberger), disorbo@unisannio.it (A. Di Sorbo), oscar.nierstrasz@inf.unibe.ch (O. Nierstrasz).

https://doi.org/10.1016/j.jss.2021.111047

1974; Ying et al., 2005); and (iii) software projects commonly lack comments to adequately describe the behavior of classes, which complicates program comprehension and evolution activities (Moreno et al., 2013; Panichella et al., 2016, 2012). The objective of our work is to examine developer class commenting practices (*e.g.,* comment content and style) across multiple languages, investigating the way in which developers write information in comments, and to establish an approach to identify that information in a language-independent manner. To achieve this objective, we select three representative programming languages based on (i) whether the language is statically- or dynamically-typed, (ii) the level of detail embedded in its class comments, (iii) whether it supports live programming, and (iv) the availability of a code comment taxonomy for the language (investigated in previous work). The motivation behind each criterion is explained in the following paragraphs.

As mentioned, programming languages adopt different conventions to embed various types of information in class comments. For example, in Java (a statically-typed language), a class comment provides a high-level outline of a class, *e.g.,* a summary of the class, what the class actually does, and other classes it cooperates with (Nurvitadhi et al., 2003). In Python (a dynamically-typed language), the class comment guidelines suggest adding low-level details about public methods, instance variables, and subclasses, in addition to the high-level summary of the class (Python Documentation Guidelines, 2020).[1] In Smalltalk (another dynamically-typed language), class comments are a primary source of code documentation, and thus they contain high-level design details as well as low-level implementation details of the class, *e.g.,* rationale behind the class, its instance variables, and important implementation points of its public methods. By analyzing multiple languages that vary in the details of their class comments, we can provide a more general overview of class commenting practices than by focusing on a single language.

Programming languages offer various conventions and tools to express these different information types in class comments. For example, Java supports Javadoc documentation comments in which developers use specific annotations, such as *@param* and *@author* to denote a given information type. In Python, developers write class comments as docstrings containing similar annotations, such as *param:* and *args:* to denote various information types, and they use tools such as Pydoc and Sphinx to process these docstrings. In contrast to Java and Python, class comments in Smalltalk neither use annotations nor the writing style of Javadoc or Pydoc, thus presenting a rather different perspective on commenting practices, and particular challenges for existing information identification approaches. Additionally, Smalltalk is considered to be a pure object-oriented programming language and supports live programming since its inception, therefore, it can present interesting insights into code documentation in live programming environments.

We argue that the extent to which class commenting practices vary across different languages is an aspect only partially investigated in previous work. Given the multi-language nature of contemporary software systems, this investigation is critical to monitor and assess the quality and evolution of code comments in different programming languages, which is relevant to support maintenance and evolution tasks. Therefore, we formulate the following research question:

> **RQ**$_1$ "*What types of information are present in class comments? To what extent do information types vary across programming languages?*"

We focus on addressing this question, as extracting class comment information types (or simply *class comment types*) can help in providing custom details to both novice and expert developers, and can assist developers at different stages of development. Hence, we report the first empirical study investigating the different language-specific class commenting practices of three different languages: Python, Java, and Smalltalk. Specifically, we quantitatively and qualitatively analyze the class comments that characterize the information types typically found in class comments of these languages. Thus, we present a taxonomy of class comment types based on the mapping of existing comment taxonomies, relevant for program comprehension activities in each of these three languages, called *CCTM* (***C**lass **C**omment **T**ype **M**odel*), mined from the actual commenting practices of developers. Four authors analyzed the content of comments using card sorting and pair sorting (Guzzi et al., 2013) to build and validate the comment taxonomy.

In the cases a code comment taxonomy was already available from previous works (Pascarella and Bacchelli, 2017; Zhang et al., 2018; Rani et al., 2021), we used that taxonomy and refined it according to our research goals.

Our work provides important insights into the types of information found in the class comments of the investigated languages, highlighting their differences and commonalities. In this context, we conjecture that these identified information types can be used to explore automated methods capable of classifying comments according to CCTM, which is a relevant step towards the automated assessment of code comments quality in different languages. Thus, based on this consideration, we formulate a second research question:

> **RQ**$_2$ "*Can machine learning be used to automatically identify class comment types according to* CCTM*?*"

To answer this research question, we propose an approach that leverages two techniques – namely Natural Language Processing (NLP) and Text Analysis (TA) – to automatically classify class comment types according to CCTM. Specifically, by analyzing comments of different types using NLP and TA we infer relevant features characterizing the class comment types. These features are then used to train machine learning models enabling the automated classification of the comment types composing CCTM.

Our results confirm that the use of these techniques allows class comments to be classified with high accuracy, for all investigated languages. As our solution enables the presence or absence of different types of comment information needed for program comprehension to be determined automatically, we believe it can serve as a crucial component for tools to assess the quality and evolution of code comments in different programming languages. Indeed, this information is needed to improve code comment quality and to facilitate subsequent maintenance and evolution tasks.

In summary, this paper makes the following contributions:

1. an empirically validated taxonomy, called *CCTM*, characterizing the information types found in class comments written by developers in three different programming languages,
2. an automated approach (available for research purposes) able to classify class comments with high accuracy according to CCTM, and
3. a publicly available dataset of manually dissected and categorized class comments in the replication package.[2]

[1] https://www.python.org/dev/peps/pep-0257/.

[2] https://github.com/poojaruhal/RP-class-comment-classification.

```java
/**
* A class representing a window on the screen.
* For example:
* <pre>
*   Window win = new Window(parent);
*   win.show();
* </pre>
*
* @author Sami Shaio
* @version 1.13, 06/08/06
* @see java.awt.BaseWindow
* @see java.awt.Button
*/

class Window extends BaseWindow {
  ...
}
```

Fig. 1. A class comment in Java.

```python
class OneHotCategorical:
    """
    Creates a one-hot categorical distribution parameterized by :attr:`probs` or
    :attr:`logits`.

    Samples are one-hot coded vectors of size ``probs.size(-1)``.

    .. note:: :attr:`probs` must be non-negative, finite and have a non-zero sum,
              and it will be normalized to sum to 1.

    See also: :func:`torch.distributions.Categorical` for specifications of
    :attr:`probs` and :attr:`logits`.

    Example::

        >>> m = OneHotCategorical(torch.tensor([ 0.25, 0.25, 0.25, 0.25 ]))
        >>> m.sample()  # equal probability of 0, 1, 2, 3
        tensor([ 0.,  0.,  0.,  1.])

    Args:
        probs (Tensor): event probabilitie
        logits (Tensor): event log probabilities
    """
```

Fig. 2. A class comment in Python.

2. Background

Code commenting practices vary across programming languages, depending on the language paradigm, the communities involved, the purpose of the language, and its usage in different domains *etc*

While Java is a general-purpose, statically-typed object-oriented programming language with wide adoption in industry, Python, on the other hand, is dynamically-typed and supports object-oriented, functional, and procedural programming. We can observe differences in the notations used by Java and Python developers for commenting source code elements. For instance, in Java, a class comment, as shown in Fig. 1, is usually written above the class declaration using annotations (*e.g.,* @param, @version, *etc*), whereas a class comment in Python, as seen in Fig. 2, is typically written below the class declaration as "docstrings".[3] In Java, developers use dedicated Javadoc annotations, such as *@author* and *@see*, to denote a given information type. Similarly, in Python, developers use similar annotations in docstrings, such as *See also:*, *Example:* and *Args:*, and they use tools such as Pydoc and Sphinx to process them.

Smalltalk is a pure, object-oriented, dynamically-typed, reflective programming language. Pharo is an open-source and live

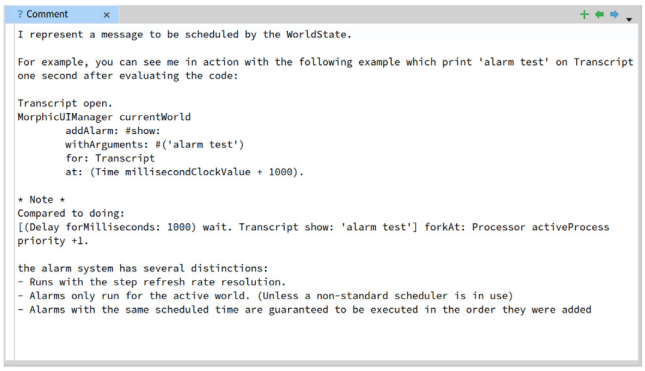

Fig. 3. A class comment in Smalltalk.

development environment incorporating a Smalltalk dialect. The Pharo ecosystem includes a significant number of projects used in research and industry (Pharo Consortium, 2020). A typical class comment in Smalltalk environment (Pharo) is a source of high-level design information about the class as well as low-level implementation details. For example, the class comment of the class *MorphicAlarm* in Fig. 3 documents (i) the intent of the class (mentioned in the first line), (ii) a code example to instantiate the class, (iii) a note explaining the corresponding comparison, and (iv) the features of the alarm system (in the last paragraph). The class comment in Pharo appears in a separate pane instead of being woven into the source code of the class. The pane contains a default class comment template, which follows a CRC (Class-Responsibility-Collaboration) model, but no other standard guidelines are offered for the structure and style of the comments. The comments follow a different, and informal writing style compared to Java, Python, and C/C++. For instance, a class comment uses complete sentences, often written in the first-person form, and does not use any kind of annotations, such as *@param* or *@see* to mark the information type, as opposed to class comments in other languages (Nurvitadhi et al., 2003; Padioleau et al., 2009; Zhang et al., 2018). As a descendant of Smalltalk-80, Smalltalk has a long history of class comments being isolated from the source code (Goldberg and Robson, 1983). Class comments are the main source of documentation in Smalltalk. Additionally, Smalltalk supports live programming for more than three decades, and therefore can present interesting insights into code documentation in a live programming environment.

Given the different commenting styles and the different types of information found in class comments from heterogeneous programming languages, the current study has the aim of (i) systematically examining the class commenting practices of different environments in details, and (ii) proposing an automated approach to identify the information contained in class comments.

3. Study design

The goal of our study is to understand the class commenting practices of developers across different object-oriented programming languages. With the obtained knowledge, we aim to build a recommender system that can automatically identify the different types of information in class comments. Such a system can provide custom details to both novice and expert developers, and assist them at various stages of development. Fig. 4 illustrates the research approach we followed to answer research questions RQ_1 and RQ_2.

[3] https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html.

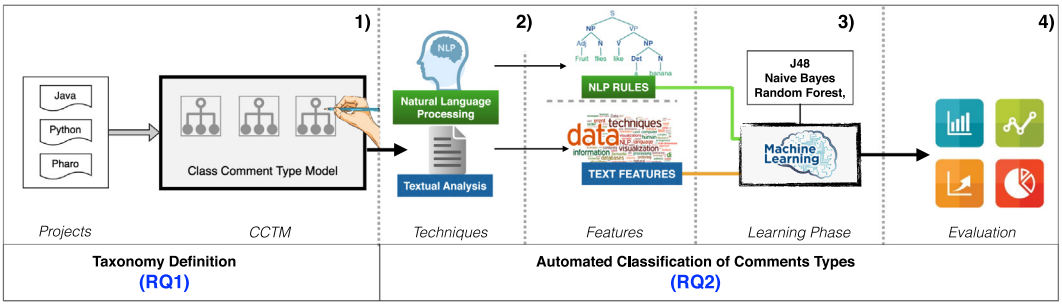

Fig. 4. Overview research approach.

3.1. Data collection

For our investigation, we selected (i) Java and Python, two of the top programming languages according to Google Trend popularity index[4] and TIOBE index,[5] and (ii) Smalltalk, as its class commenting practices emerged from Smalltalk-80 (Pharo Consortium, 2020; Goldstein and Bobrow, 1980). Other criteria to select these languages are explained in the Introduction (Section 1) and Background section (Section 2). Smalltalk is still widely used and it gained second place for *most loved programming language* in the Stack Overflow survey of 2017.[6] For each language, we chose popular, open-source, and heterogeneous projects. Such projects vary in terms of size, contributors, domains, ecosystems, and coding style guidelines (or comment guidelines).

As not all classes contain a class comment, we identified the classes having class comments for each project. Afterward, we extracted a statistically significant sample of class comments to conduct a manual analysis. This analysis aimed at identifying the semantic information developers embed in class comments. To determine the minimum size of the statistically significant sample for each language dataset, we set the confidence level to 95% and the margin of error to 5% (Triola, 2006).

After determining the sample size, we selected the number of comments to consider from each project based on the proportion of the project's class comments of all comments (from all projects). For instance, class comments from an Eclipse project in Java contribute to 29% of total comments (comments from all Java projects) therefore we selected the same proportion of sample comments, *i.e.,* 29% of Java sample size from the Eclipse project (110 class comments), as shown in Table 1. To select representative sample comments from each project, we applied the stratified random sampling strategy and selected a proportional number of comments from each stratum. The strata were defined based on the length (in terms of lines) of comments. In particular, for each project, we first computed the quintiles based on the distribution of comments' length and treated the quintiles as strata. For example, to choose 110 sample comments for Eclipse as shown in Table 1, we explored the distribution of comments lines and obtained quintiles as follows 1, 3, 4, 5, 7, and 1473. Hence the five strata are 1–3, 4–4, 5–5, 6–7, and 8–1473. Then from each stratum, we selected the proportional number of comments.

Java: we selected six open-source projects analyzed in previous work (Pascarella and Bacchelli, 2017) to ease the comparison of our work with previous achievements.[7] Modern complex projects are commonly developed using multiple programming languages. For example, Apache Spark contains 72% Scala classes, 9% Java classes, and 19% classes from other languages.[8] In the context of our study, we considered the classes from the language under investigation, and discarded the classes from other programming languages. For each class, we parsed the Java code and extracted the code comments preceding the class definition using the AST (Abstract Syntax Tree) based parser. During extraction, we found instances of block comments (comments starting with /* symbol) in addition to Javadoc class comments (comments starting with /** symbol) before the class definition. In such cases, the AST-based parser detects the immediately preceding comment (being it is a block comment or Javadoc comment) as a class comment and treats the other comment above it as a dangling comment. To not miss any kinds of class comment, we adapted our parser to join both comments (the AST detected class comment and the dangling comment) as a whole class comment.

Table 1
Overview of Java projects.

Project	% of Java classes	#Java classes	#Class comments	% of dataset	#Sample comments
Eclipse	98%	9128	6253	29%	110
Spark	9.3%	1090	740	3.4%	13
Guava	100%	3119	2858	13%	50
Guice	99%	552	466	2.1%	10
Hadoop	92%	11855	8846	41%	155
Vaadin	55%	5867	2335	11%	41

We present the overview of the projects selected for Java in Table 1 and raw files in the replication package.[9] We established 376 class comments as the statistically significant sample size based on the total number of classes with comments. For each project, the sample of class comments (the column *Sample comments*) is measured based on the proportion of class comments in the total dataset of class comments (shown in the column *"% of dataset"* of Table 1).

The number of lines in class comments varies from 1 to 4605 in Java projects. For each project, we established strata based on the identified quintiles from the project's distribution. From each stratum, we picked an equal number of comments using the random sampling approach without replacement. For example, in the Eclipse project, we identified the five strata as 1-3, 4-4, 5-5, 6-7, and 8-1473. We picked 25 comments from each stratum summing to the required 110 sample comments. We followed the same approach for Python and Smalltalk to select the representative sample comments.

Python: We selected seven open-source projects, analyzed also in the previous work (Zhang et al., 2018), to ease the comparison of our work with previous achievements. To extract class comments from Python classes, we implemented a Python AST based parser and extracted the comments preceding the class definition.

4 https://pypl.github.io/PYPL.html.
5 https://www.tiobe.com/tiobe-index/.
6 https://insights.stackoverflow.com/survey/2017/ verified on 4 Feb 2020.
7 Folder "RP/Dataset/RQ1/Java" in the Replication package.
8 https://github.com/apache/spark.
9 Folder "RP/Dataset/RQ1/Java" in the Replication package.

Table 2
Overview of Python projects.

Project	#Python classes	#Class comments	% of dataset	#Sample comments
Requests	79	43	1.1%	4
Pandas	1753	377	9.9%	35
Mailpile	521	283	7.5%	26
IPython	509	240	6.3%	22
Djnago	8750	1164	30%	107
Pipenev	1866	1163	30%	107
Pytorch	2699	520	13%	48

Table 3
Overview of Smalltalk projects.

Projects	Total classes	#Classes comments	% of dataset	#Sample comments
GToolkit	4191	1315	43%	148
Seaside	841	411	14%	46
Roassal	830	493	16%	56
Moose	1283	316	10%	36
PolyMath	300	155	5%	17
PetitParser	191	99	3%	11
Pillar	420	237	8%	27

The metadata related to the selected projects for Python are reported in Table 2, while the class comments are found in our replication package.[10] We measured 349 sample comments to be the statistically significant sample size based on total classes with comments.

Smalltalk: We selected seven open-source projects. As with Java and Python, we selected the same projects investigated in previous research (Rani et al., 2021). Table 3 shows the details of each project with the total numbers of classes, as well as classes with comments,[11] their proportion in the total comments, and the numbers of sample comments selected for our manual analysis. We extracted the stable version of each project compatible with Pharo version 7 except for GToolkit, due to the lack of backward compatibility. We, therefore, used Pharo 8 for GToolkit.

3.2. Analysis method

RQ_1: What types of information are present in class comments? To what extent do information types vary across programming languages?

The purpose of RQ_1 is to quantitatively and qualitatively investigate the language-specific class commenting practices characterizing programs written in Python, Java, and Smalltalk.

Fig. 5 depicts the research approach followed to answer RQ_1. The outcome of this research, as explained later, consists of a mapping taxonomy and a comprehensive taxonomy of class comment types, called *CCTM* (***C**lass **C**omment **T**ype **M**odel*), mined from the actual commenting practices of developers (see Fig. 5).

Mapping categories: Before preparing the *CCTM*, we analyzed various earlier comment taxonomies, such as the code comment taxonomy for Java (Pascarella and Bacchelli, 2017) and Python (Zhang et al., 2018), and the class comment taxonomy for Smalltalk (Rani et al., 2021), to analyze their reported categories. We mapped the categories from each existing taxonomy, since they were formulated using different approaches and based on different comment scope and categories. For instance, the Python code comment taxonomy by Zhang et al. (2018) is inspired by the Java code comment taxonomy (Pascarella and Bacchelli, 2017), whereas the Smalltalk class comment taxonomy is formulated

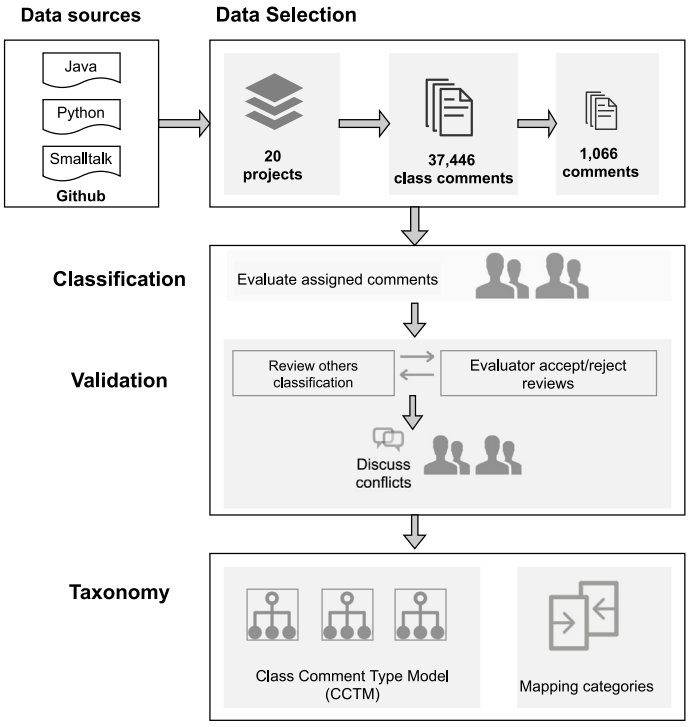

Fig. 5. Taxonomy study (RQ_1).

using an open-card sorting technique (Rani et al., 2021). Given the importance of mapping categories from heterogeneous environments (Choi et al., 2006), we established semantic interoperability (Euzenat, 2001) of the categories from each taxonomy in this step. One evaluator mapped the categories from Smalltalk to Java and Python categories. Two other evaluators validated the mapping by reviewing each mapping and proposing the changes. The original evaluator accepted or rejected the changes. All the disagreement cases were reviewed by the fourth evaluator and discussed among all to reach the consensus.

The categories that did not map to other taxonomy, we added them as new categories in that taxonomy. For example, the *Precondition* category from Smalltalk did not map to Java and Python, and thus we added it as a new category in Java and Python. Thus, we proposed the *CCTM* taxonomy highlighting the existing and new categories for class comments in the next step *Preparing CCTM*.

Preparing CCTM: In this step, we analyzed various ecosystems from each language. We built our dataset[12] by mining data from 20 GitHub projects (see Section 3.1 for further details). In the *Data Selection* step, we extracted the classes and their comments, collecting a total of 37 446 class comments. We selected a statistically significant sample set for each language summing 1066 total class comments. We then qualitatively classified the selected class comments (see the following *Classification* step) and validated them (see following *Validation* step) by reviewing and refining the categorization.

Classification: Four evaluators (two Ph.D. candidates, and two faculty members, all authors of this paper) each having at least four years of programming experience, participated in the study. We partitioned Java, Python, and Smalltalk comments equally among all evaluators based on the distribution of the language's dataset to ensure the inclusion of comments from all projects and diversified lengths. Each evaluator classified the assigned class

10 Folder "RP/Dataset/RQ1/Python" in the Replication package.
11 Folder "RP/Dataset/RQ1/Pharo" in the Replication package.
12 Folder "RP/Dataset" in the Replication package.

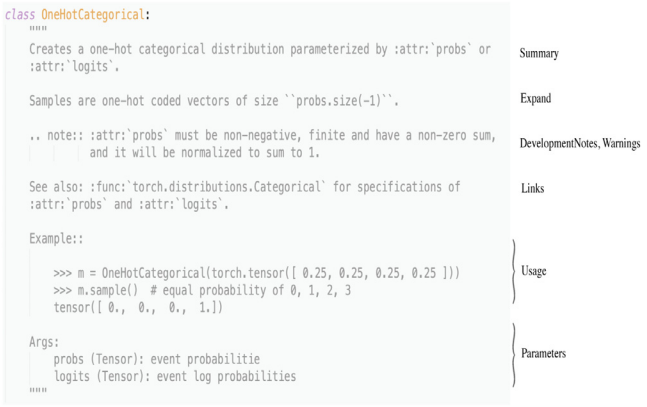

Fig. 6. Class comment of Python (Fig. 2) classified in various categories.

comments according to the CCTM taxonomy of Java, Python and Smalltalk (Pascarella and Bacchelli, 2017; Zhang et al., 2018; Rani et al., 2021).

For example, the Python class comment in Fig. 2 is classified in the categories such as *Summary*, *Warnings*, and *Parameters etc*, as shown in Fig. 6. **Validation:** After completing their individual evaluations, the evaluators continued with the validation step. Thus, the evaluators adopted a three-step method to validate the correctness of the performed class comments classification. In the first iteration, called *"Review others' classification"* in Fig. 5, every evaluator was tasked to review 50% of the comments (randomly assigned and) classified by other evaluators. This step allowed us to confirm that each evaluator's classification is checked by at least one of the other evaluators. In reviewing the classifications, the reviewers indicated their judgment by labeling each comment with *agree* or *disagree* labels.

In the second iteration (called *"Evaluator accept or reject reviews"* in Fig. 5), the original evaluator examined the disagreements and the proposed changes. They indicated their opinion for the changes by accepting the change or rejecting it, stating the reason. In case the reviewer's changes were accepted, the classification was directly fixed, otherwise the disagreements were carried to the next iteration. The third iteration assigned all identified disagreements for review to a new evaluator, who had not yet looked at the classification. Based on a majority voting mechanism, a decision was made and the classification was fixed according to agreed changes. The levels of *agreement and disagreement* among the evaluators for each project and language are available in the replication package.[13]

After arriving at a decision on all comment classifications, we merged overlapping categories or renamed the classes by applying the majority voting mechanism, thus converging on a final version of the taxonomy, *i.e.,* CCTM. This way, all the categories were discussed by all the evaluators to select the best naming convention, and whenever required, unnecessary categories were removed and duplicates were merged.

RQ_2: Automated Classification of Class Comment Types in Different Programming Languages

Motivation. Previous work has focused on identifying information types from code comments scattered throughout the source code, from high-level class overviews to low-level method details (Steidl et al., 2013; Pascarella and Bacchelli, 2017; Zhang et al., 2018; Geist et al., 2020). These works have focused individually on code comments in Java, Python, C++, and COBOL. Differently from our study, none of these works attempted to identify information types in class comments automatically across multiple languages. In our work, we are interested in exploring strategies that are able to achieve this goal in multiple languages such as Python, Java and Smalltalk. Hence, for this research question, we explore to what extent term-based strategies and techniques based on NLP patterns (Di Sorbo et al., 2019, 2016; Panichella et al., 2015) help to automatically identify the different information types composing CCTM.

Automated classification of class comment types. After the definition of CCTM, we propose an approach, called TESSERACT (au**T**omated multi-language**E** cla**SS**ifi**ER** of cl**A**ss **C**ommen**T**s), which automatically classifies class comment according to CCTM. To achieve this goal, we considered all the comments manually validated in answering RQ_1. Specifically, our approach leverages machine learning (ML) techniques and consists of four main steps:

1. ***Preprocessing***: All the manually-labeled class comments from RQ_1 in our dataset are used as ground truth to classify the unseen class comments.[14] It is important to mention that, since the classification is sentence-based, we split the comments into sentences. As a common preprocessing step, we change the sentences to lower case and remove all special characters.[15] We apply typical preprocessing steps on sentences (Baeza-Yates and Ribeiro-Neto, 1999) (*e.g.,* stop-word removal) for TEXT features but not for NLP features to preserve the word order to capture the important n-gram patterns observed in the class comments (Rani et al., 2021).
2. ***NLP Feature Extraction***: In this phase we focus on extracting NLP features to add to an initial term-by-document matrix *M* shown in Fig. 7 where each row represents a *comment sentence* (*i.e.,* a sentence belongs to our language dataset composing CCTM) and each column represents the extracted feature. To extract the NLP features, we use NEON, a tool proposed in previous work (Di Sorbo et al., 2019), which is able to automatically detect NLP patterns (*i.e.,* predicate–argument structures recurrently used for specific intents Di Sorbo et al., 2015) present in natural language descriptions composing various types of software artifacts (*e.g.,* mobile user reviews, emails *etc*) (Di Sorbo et al., 2019). In our study, we use NEON to infer all NLP patterns characterizing the comment sentences modeled in the matrix *M*. Then, we add the identified NLP patterns as feature columns in *M*, where each of them models the presence or absence (using binomial features) of an NLP pattern in the comment sentences. More formally, the boolean presence or absence of a jth NLP pattern (or feature) in a generic ith sentence in *M* is modeled by 0 (absence) and 1 (presence) values, respectively. The output of this phase consists of the matrix *M* where each ith row represents a comment sentence and jth represents an NLP feature.
3. ***TEXT Features***: In this phase, we add additional features (TEXT features) to the matrix *M*. To get the TEXT features, we preprocess the comment sentences by applying stop-word removal[16] and stemming (Lovins, 1968).[17] The output of this phase corresponds to the matrix *M* where each row represents a *comment sentence* (*i.e.,* a sentence belonging to our language dataset composing CCTM) and each

13 Folder "RP/Result/RQ1/Manually-classified-comments" in the Replication package.

14 File "RP/Dataset/RQ2/Java/raw.csv" in the Replication package.

15 File "RP/Results/RQ2/All-steps-result.sqlite" in the Replication package.

16 http://www.cs.cmu.edu/~mccallum/bow/rainbow/.

17 https://weka.sourceforge.io/doc.stable/weka/core/stemmers/IteratedLovinsStemmer.html.

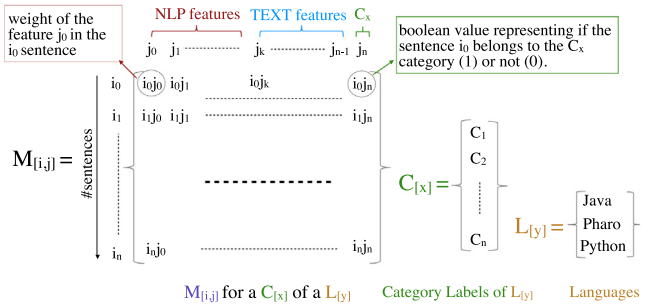

Fig. 7. Matrix representation for a classifier.

column represents a term contained in it. More formally, each entry $\mathbf{M}_{[i,j]}$ of the matrix represents the weight (or importance) of the jth term contained in the ith *comment sentence*.

For the TEXT features, terms in M are weighted using the *TF–IDF* score (Baeza-Yates and Ribeiro-Neto, 1999), which identifies the most important terms in the sentences in matrix M. In particular, we use *TF–IDF* as it downscales the weights of frequent terms that appear in many sentences. Such a weighting scheme has been successfully adopted in recent work (Misra et al., 2020) for performing code comment classification. The output of this phase consists of the weighted matrix M where each row represents a comment sentence and a column represents the weighted term contained in it.

It is important to note that a generic ith comment sentence can be classified into multiple categories according to CCTM. To model this, we prepare the matrix M for each category of each language. The generic (last) column $\mathbf{M}[j_n]$ of the matrix $\mathbf{M}$ (where $n-1$ is the total number of features extracted from all sentences) represents the category $\mathbf{C}_{[x]}$ of a language $\mathbf{L}_{[y]}$ as shown in Fig. 7. More formally, each entry $\mathbf{M}_{[i,j_n]}$ of the matrix represents the boolean value if the ith sentence belongs to the C_x (1) or not(0).

4. ***Classification***: We automatically classify class comments by adopting various machine learning models and a 10-fold cross-validation strategy. These machine learning models are fed with the aforementioned matrix M. Specifically, to increase the generalizability of our findings, we experiment (relying on the Weka tool[18]) with several machine learning techniques, namely, the standard probabilistic Naive Bayes classifier, the J48 tree model, and the Random Forest model. It is important to note that the choice of these techniques is not random but based on their successful usage in recent work on code comment analysis (Steidl et al., 2013; Pascarella and Bacchelli, 2017; Zhang et al., 2018; Shinyama et al., 2018) and classification of unstructured texts for software maintenance purposes (Panichella et al., 2015; Di Sorbo et al., 2016).

Evaluation metrics & statistical tests. To evaluate the performance of the tested ML techniques, we adopt well-known information retrieval metrics, namely precision, recall, and F-measure (Baeza-Yates and Ribeiro-Neto, 1999). During our empirical investigation, we focus on investigating the best configuration of features and machine learning models as well as alleviating concerns related to overfitting and selection bias. Specifically, (i) we investigate the classification results of the aforementioned machine learning models with different combinations of features (NLP, TEXT, and TEXT+NLP features), by adopting a 10-fold validation strategy on the term-by-document matrix M; (ii) to avoid potential bias or overfitting problems, we train the model for the categories having at least 40 manually validated instances in our dataset. The average number of comments belonging to a category varies from 43 comments to 46 comments across all languages, therefore we selected the categories with a minimum of 40 comment instances. The top categories selected from each language with the number of comments are shown in Table 4. In order to determine whether the differences between the different input features and classifiers were statistically significant or not we performed a Friedman test, followed by a post-hoc Nemenyi test, as recommended by Demšar (Demšar, 2006). Results concerning RQ_2 are reported in Section 4.

[18] http://waikato.github.io/weka/.

Table 4
Top frequent categories with at least 50 comments.

Language	Categories	#Comments
Java	Summary	336
	Expand	108
	Ownership	97
	Pointer	88
	Usage	87
	Deprecation	84
	Rationale	50
Python	Summary	318
	Usage	92
	Expand	87
	Development Notes	67
	Parameters	57
Smalltalk	Responsibility	240
	Intent	190
	Collaborator	91
	Examples	81
	Class reference	57
	Key message	48
	Key implementation point	46

4. Results

This section discusses the results of our empirical study.

4.1. RQ_1: Class comment categories in different programming languages

As a first step to formulating the CCTM taxonomy we systematically map the available taxonomies from previous works and identify the unmapped categories as shown in Fig. 8. The mapping taxonomy shows a number of cases in which Java and Python taxonomies do not entirely fit the Smalltalk taxonomy, as shown by pink and violet edges in Fig. 8. The figure shows the information types particular to a language (highlighted with red edges and unmapped nodes) such as *Subclass Explanations*, *Observation*, *Precondition*, and *Extension* are found in the Smalltalk taxonomy but not in Python and Java. However, our analysis shows that these information types are present in the class comments of Java and Python projects. We introduce such categories to the existing taxonomies of Java and Python, and highlight them in green in Fig. 9. On the other hand, the categories such as *Commented code*, *Exception*, and *Version* are found in Java and Python class comments but not in Smalltalk. One reason can be that the commented code is generally found in inline comments instead of documentation comments. However, information about *Exception* and *Version* is found in class comments of Java and Python but not in Smalltalk.

The mapping taxonomy also highlights the cases where categories from different taxonomies match partially. We define such

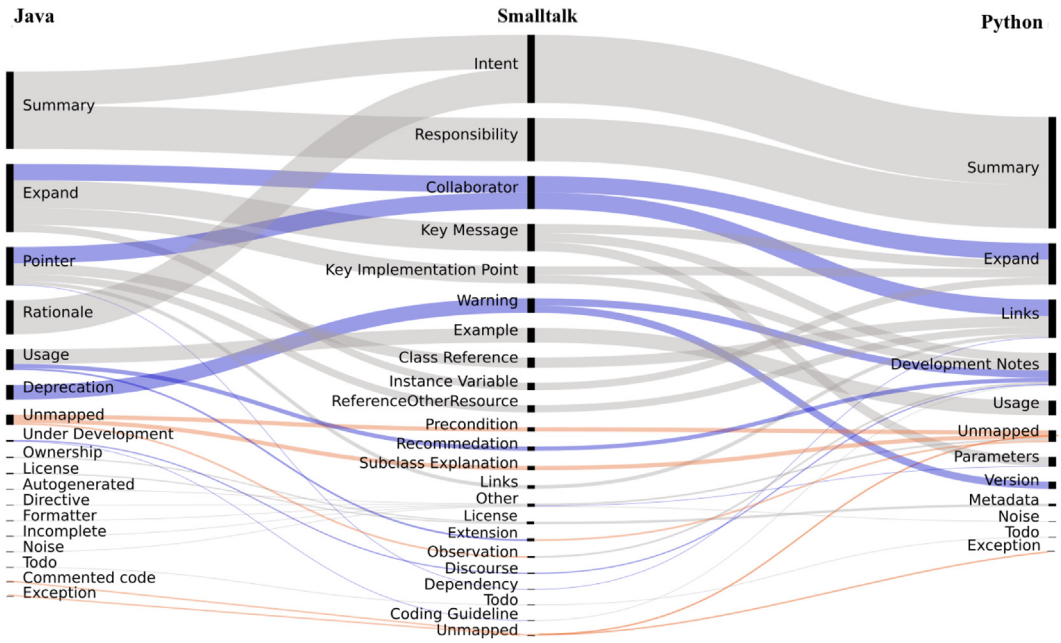

Fig. 8. Mapping of Smalltalk categories to Java and Python. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

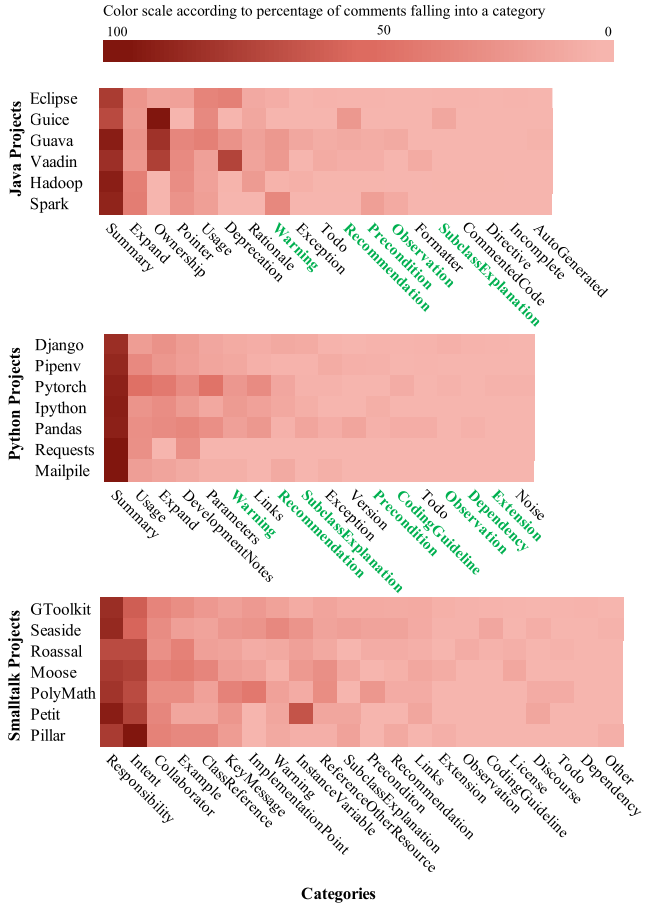

Fig. 9. Categories found in class comments (*CCTM*) of various projects (shown on the y-axis) of each programming language. The *x*-axis shows the categories inspired from existing work (highlighted in black) and the new categories (highlighted in green)). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

categories as subset categories and highlight them with violet edges. For example, the *Deprecation* category in Java and the *Version* category in Python are found under the *Warning* category in Smalltalk but their description given in the respective earlier work covers only a subset of that information type according to our investigation. Pascarella et al. define the *Deprecation* category as "*it contains explicit warnings used to inform the users about deprecated interface artifacts. The tag comment such as @version, @deprecated, or @since is used*" whereas Zhang et al. define the category *Version* as "*identifies the applicable version of some libraries*" but do not mention the deprecation information in this category or any other category in their taxonomy (Pascarella and Bacchelli, 2017; Zhang et al., 2018). Thus, we define the *Version* category as a subset or a partial match of the *Deprecation* category. On the other hand, the *Warning* category in Smalltalk ("*Warn readers about various implementation details of the class*") covers a broader aspect of warnings than the *Deprecation* category but does not mention the *Version* information type (Rani et al., 2021). We mark these categories as subset categories, and highlight them with violet edges. Similarly, the *Collaborator* category in Smalltalk matches partially *Expand* and *Links* in Python. The categories such as *Expand*, *Links*, and *Development Notes* in Python combine several types of information under them compared to Smalltalk. For example, *Expand* includes collaborators of a class, key methods, instance variables, and implementation-specific details. Such categories formulate challenges in identifying a particular type of information from the comment.

Finding 1. The Python taxonomy focuses more on high-level categories, combining various types of information into each category whereas the Smalltalk taxonomy is more specific to the information types.

Using the categories from the mapping taxonomy, we analyzed class comments of various languages and formulated the taxonomy for each language to answer the RQ_1. Fig. 9 shows the frequency of information types per language per project. The categories shown in green are the newly-added categories in each

taxonomy. The categories in each heatmap are sorted according to the frequency of their occurrence in total. For example, in Java, *Summary* appeared in a total of 336 comments (88%) of six Java projects out of 378 comments. Pascarella (Pascarella and Bacchelli, 2017) proposed a hierarchical taxonomy, grouping the lower-level categories within the higher-level categories such as grouping the categories *Summary*, *Rationale*, and *Expand* within *Purpose*. We show only lower-level categories that correspond with identified information types from other languages.

Fig. 9 shows that a few types of information, such as the summary of the class (*Summary*, *Intent*), the responsibility of the class (*Responsibility*, *Summary*), links to other classes or sources (*Pointer*, *Collaborator*, *Links*), developer notes (*Todo*, *Development Notes*), and warnings about the class (*Warning*) are found in class comments of all languages. Summary being the most prevalent category in all languages affirms the value of investing effort in summarizing the classes automatically (Haiduc et al., 2010; Nazar et al., 2016; Binkley et al., 2013; Moreno et al., 2013; Dragan et al., 2010). As a majority of summarization techniques focus on generating the intent and responsibilities of the class for program comprehension tasks (Moreno et al., 2013), other information types such as *Warning*, *Recommendation*, and *Usage* are generally ignored even though coding style guidelines suggest to write them in documentation comments. Our results indicate that developers mention them frequently, but whether they find these information types important to support specific development tasks, or they write just to adhere to the coding guidelines, requires more analysis. Such information types present an interesting aspect to investigate in future work. For example, usage of a class (*Usage*), its key responsibilities (*Responsibility*), warnings about it (*Warning*), and its collaborators (*Collaborator*) are found in significant numbers of comments in all languages. These information types are often suggested by the coding guidelines as they can support developers in various development and maintenance tasks. These information types can be included in the customized code summaries based on the development tasks a developer is doing. For example, a developer seeking dependent classes can quickly find such classes from the class comment without reading the whole comment. Similarly, a developer expected to refactor a legacy class can quickly go through the warnings, if present, to understand the specific conditions better and thus can save time. We plan to investigate such information types with developers. Specifically, we are interested in exploring how various categories are useful to developers, and for which kinds of tasks *e.g.,* program comprehension tasks or maintenance tasks.

Finding 2. Developers embed various types of information in class comments, varying from the high-level overview of the class to the low-level implementation details of the class across the investigated languages.

According to Nurvitadhi et al. (2003), a class comment in Java should describe the purpose of the class, its responsibilities, and its interactions with other classes. In our study, we observe that class comments in Java often contain the purpose and responsibilities of the class (*Summary* and *Expand*), but its interactions with other classes (*Pointer*) less often. On the other hand in Smalltalk, the information about interactions with other classes, *i.e., Collaborator*, is the third most frequent information type after *Intent* and *Responsibility* compared to Java and Python. One of the reasons can be that Smalltalk class comments are guided by a CRC (Class, Responsibility, Collaborator) design template and developers follow the template in writing these information types Rani et al. (2021). Class comments in Java also contain many other types of information. The most frequent type of information present in class comments is *Summary*, which shows that developers summarize the classes in the majority of the cases. Pascarella et al. found *Usage* to be the second most prevalent category in code comments, while we find overall *Expand* to be the second most prevalent category in class comments, and *Usage* to be the fifth most prevalent type of information (Pascarella and Bacchelli, 2017). However, the most prevalent categories vary across projects of a language, and also across programming languages. For example, *Usage* is mentioned more often than *Expand* in Google projects (Guice and Guava) whereas in Apache projects (Spark, Hadoop) it is not. In contrast to Java, Python class comments contain *Expand* and *Usage* equally frequently thus showing that Python targets both end-user developers and internal developers. We notice that Python and Smalltalk class comments contain more low-level implementation details about the class compared to Java. For example, Python class comments contain the details about the class attributes and the instance variables of a class with a header "*Attributes*" or "*Parameters*", its public methods with a header "*Methods*", and its constructor arguments in the *Parameters* and *Expand* categories. Additionally, Python class comments often contain explicit warnings about the class (with a header "*warning:*" or "*note:*" in the new line), making the information easily noticeable whereas such behavior is rarely observed in Java. Whether such variations in the categories across projects and languages are due to different project comment style guidelines or due to developer personal preferences is not known yet. We observe that developers use common natural language patterns to write similar types of information. For example, a Smalltalk developer described the collaborator class of the "PMBernoulliGeneratorTest" class in Listing 1 and a Java developer as shown in Listing 2 thus showing a pattern *"[This class] for [other class]"* to describe the collaborating classes. This information type is captured in the categories *Collaborator* in Smalltalk and *Pointer* in Java.

```
A BernoulliGeneratorTest is a test class for testing the
      behavior of BernoulliGenerator
```

Listing 1: Collaborator mentioned in the PMBernoulliGeneratorTest class in Smalltalk

```
An {@link RecordReader} for {@link SequenceFile}s.
```

Listing 2: Collaborator mentioned in the SequenceFileRecordReader class in Java

Identifying such patterns can help in extracting the type of information from the comment easily, and can support the developer by highlighting the required information necessary for a particular task, *e.g.,* to modify the dependent classes in a maintenance task.

In contrast to earlier studies, we observe that developers mention details of their subclasses in a parent class comment in all languages. We group this information under the *Subclass Explanation* category. In the Javadoc guidelines, this information is generally indicated by a special `@inherit` tag in the method comments, but we did not find such a guideline for Java class comments. Similarly we found no such guideline to describe subclasses for Smalltalk class comments or method comments. In contrast, the standard Python style guideline (Python Documentation Guidelines, 2020) suggests adding this information in the class comment but other Python style guidelines such as those from Google[19] and Numpy[20] do not mention this information type. However, we find instances of class comments containing

[19] https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html.

[20] https://numpydoc.readthedocs.io/en/latest/format.html.

subclass information in the *IPython* and *Pytorch* projects that follow the Numpy and Google style guidelines respectively. Investigating which information types each project style guidelines suggest for the comments, and to what extent developers follow these style guidelines in writing class comments is future work.

Finding 3. Not all information types found in class comments are suggested by corresponding project style guidelines.

Discussion: In the process of validating the taxonomies, reviewers (evaluator reviewing the classification) marked their disagreement for the classification, stating their reason, and proposing the changes. A majority of disagreements in the first Smalltalk iteration were due to the long sentences containing different types of information (*Intent* and *Responsibility* information types were commonly interweaved), and assignment of information to categories *Key Implementation Point* and *Collaborator*. In Java and Python, we observed that disagreements were due to the broad and loosely defined categories such as *Expand* in Java and *Development Notes* in Python. Several information types such as *Warning*, *Recommendation*, *Observation*, and *Development Notes* are not structured by special tags and thus pose a challenge for automatic identification and extraction. On the other hand, a few categories such as *Example* and *Instance Variable* in Smalltalk, and *Deprecation*, *Links*, and *Parameters* in Java and Python are explicitly marked by the headers or the tags such as *Usage*, *Instance Variable*, *@since*, *@see*, *@params* respectively. We observed that developers use common keywords across languages to indicate a particular information type. For example, notes are mentioned in the comment with a keyword "note" as a header as shown in Listing 3, Listing 4, and Listing 5.

```
Note that even though these
* methods use {@link URL} parameters, they are usually not
     appropriate for HTTP or other
* non-classpath resources.
```

Listing 3: Explicit note mentioned in the Resources class in Java

```
.. note::

Depending of the size of your kernel, several (of the last)
columns of the input might be lost, because it is a valid `
     cross-correlation`_,
and not a full `cross-correlation`_.
It is up to the user to add proper padding.
```

Listing 4: Explicit note mentioned in the Conv3d class in Python

```
Note: position may change even if an element has no parent
```

Listing 5: Explicit note mentioned in the BlElementPositionChangedEvent class in Smalltalk

Several information types are suggested by the project-specific style guidelines but not the exact syntax whereas a large number of information types are not mentioned by them. Due to the lack of conventions for these information types, developers use their own conventions to write them in the comments.

Maalej et al. (2014) demonstrates that developers consult comments in order to answer their questions regarding program comprehension. However, different types of information are interweaved in class comments and not all developers need to know all types of information. Cioch et al. presented the documentation information needs of developers depending on the stages of expertise (Cioch et al., 1996). They showed that experts need design details and low-level details whereas novice developers require a high-level overview with examples. Therefore, identifying and separating these information types is essential to address the documentation needs. Rajlich presented a tool that gathers important information such as a class's responsibilities, its dependencies, member functions, and authors' comments to facilitate the developer's need to access the particular information types (Rajlich, 2000). We advance the work by identifying and extracting several other frequent information types from the class comments automatically.

4.2. RQ_2: Automated classification of class comment categories in different programming languages

Haiduc et al. (2010) performed a study on automatically generating summaries for classes and methods and found that the experimented summarization techniques work better on methods than classes. More specifically, they discovered that while developers generally agree on the important attributes that should be considered in the method summaries, there were conflicts concerning the types of information (*i.e.,* class attributes and method names) that should appear in the class summaries. In our study, we found that while Smalltalk and Python developers frequently embed class attributes or method names in class comments, it rarely happens in Java. Automatically identifying various kinds of information from comments can enable the generation of customized summaries based on what information individual developers consider relevant for the task at hand (*e.g.,* maintenance task). To this aim, as described in Section 3.2, we empirically experiment with a machine learning-based multi-language approach to automatically recognize the types of information available in class comments.

Table 5 provides an overview of the average precision, recall, and F-Measure results considering the (top frequent) categories for all languages shown in Table 4. The results are obtained using multiple machine learning models and various combination of features[21]: (i) TEXT features only, (ii) NLP features only, (iii) both NLP and TEXT features.[22] The results in Table 5 show that the NLP+TEXT configuration achieves the best results with the Random Forest algorithm with relatively high precision (ranging from 78% to 92% for the selected languages), recall (ranging from 86% to 92%), and F-Measure (ranging from 77% to 92%). Fig. 10 shows the performance of the different algorithms with NLP+TEXT features for the most frequent categories of each language.

Finding 4. Our results suggest that the Random Forest algorithm fed by the combination of NLP+TEXT features achieves the best classification performance over the different programming languages.

According to Table 5, NLP features alone achieve the lowest classification performance for both Java and Python, while we observe that this type of feature works well when dealing with Smalltalk class comments. On the one hand, class comments can often contain mixtures of structured (*e.g.,* code elements, such as class and attribute names) and unstructured information (*i.e.,* natural language). NEON (i) leverages models trained on general-purpose natural language sentences to construct the parse tree of the sentences, and (ii) relies on the generated parse trees to identify common NLP patterns (Di Sorbo et al., 2019). Therefore, the presence of code elements degrades NEON's capability to generate accurate parse trees, and consequently complicates its

[21] File "RP/Result/RQ2/CV-10-results.xlsx" in the Replication package.
[22] File "RP/Result/RQ2/All-steps-results.sqlite" in the Replication package.

Table 5
Results for Java, Python, and Smalltalk obtained through different machine learning models and features.

Language	ML models	TEXT			NLP			NLP + TEXT		
		Precision	*Recall*	*F-Measure*	*Precision*	*Recall*	*F-Measure*	*Precision*	*Recall*	*F-Measure*
Java	J48	**0.91**	**0.92**	**0.91**	**0.84**	0.81	0.81	0.89	0.90	0.88
	Naive Bayes	0.86	0.83	0.83	**0.84**	0.81	0.81	0.86	0.84	0.84
	Random forest	**0.91**	**0.92**	**0.91**	**0.84**	**0.87**	**0.82**	**0.92**	**0.92**	**0.92**
Python	J48	0.73	0.83	0.73	0.68	0.80	0.66	0.81	0.83	0.79
	Naive Bayes	0.78	0.69	0.72	0.75	0.77	0.75	0.79	0.72	0.74
	Random forest	**0.84**	**0.85**	**0.83**	**0.78**	**0.81**	**0.78**	**0.85**	**0.86**	**0.84**
Smalltalk	J48	0.60	0.87	0.58	0.61	**0.88**	0.59	0.72	0.88	0.70
	Naive Bayes	**0.84**	0.80	**0.82**	**0.85**	0.83	**0.83**	**0.86**	0.82	**0.84**
	Random forest	0.75	**0.90**	0.73	0.82	**0.88**	0.82	0.78	**0.90**	0.77

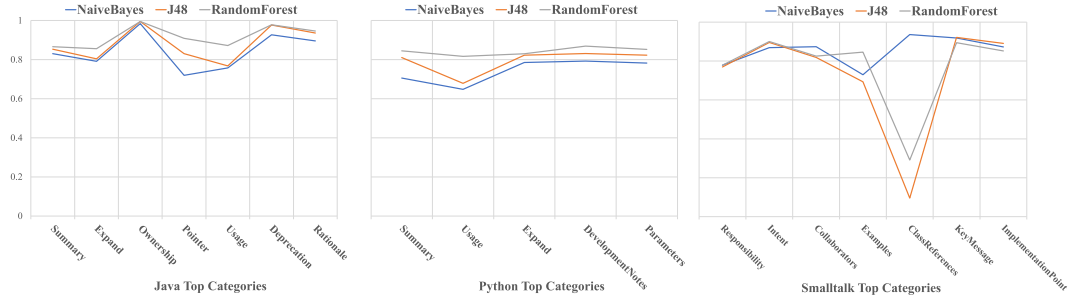

Fig. 10. Performance of the different classifiers based on F-measure for the TEXT + NLP feature set.

pattern recognition task. On the other hand, code in Smalltalk resembles natural language (English) phrases, *e.g.,* the method, shown in Listing 6, "addString: using:" takes two parameters "string" and "CamelCaseScanner new" written in a natural language sentence style. Similarly, class comments in Smalltalk are written in a more informal writing style (often using the first-person form, and written in complete sentences as shown in Fig. 3), compared to Java and Python (which suggest to write in a formal way using the third-person form). As demonstrated in previous work (Panichella et al., 2015; Di Sorbo et al., 2015), the usage of predicate–argument patterns is particularly well-suited when dealing with classification problems in highly unstructured and informal contexts.

```
Terms subclasses Bag with support for handling stop words etc.

example: string
| terms |
terms := Terms new.
terms addString: string using: CamelCaseScanner new.
terms withCountDo: [ :term :count |term -> count ].
```

Listing 6: (Smalltalk) Code in MalTerms comment resembles natural language

Finding 5. When dealing with sentences containing mixtures of code elements and natural language texts, NLP tools based on parse trees fail to correctly identify well-suited NLP features. The usage of these features is otherwise recommended when the class comments are mostly unstructured.

For the sake of brevity, we base the following analysis on the NLP+TEXT configuration when used with the Random Forest classifier. Table 6, Table 7, and Table 8 respectively report the precision, recall, and F-Measure for the top frequent categories (see Section 4.2) obtained through the Random Forest algorithm with the NLP+TEXT features.

In the case of Java, as shown in Table 6, *Deprecation*, *Ownership*, and *Rationale* achieve high F-measure scores ($\geq$ 95%), while

Table 6
Results for Java using the random forest classification model.

Category	NLP + TEXT		
	Precision	Recall	F-Measure
Summary	0.87	0.88	0.87
Expand	0.86	0.87	0.86
Ownership	0.99	0.99	0.99
Pointer	0.91	0.91	0.91
Usage	0.88	0.88	0.87
Deprecation	0.98	0.98	0.98
Rationale	0.95	0.95	0.95

Expand, *Summary* and *Usage* are the categories with the lowest F-measure values (but still higher than 85%). This means that for most Java categories Random Forest achieves very accurate classification results. However, we observe a certain variability in the results, depending on the category (see Table 4). While in our manually validated sample *Summary* and *Usage* occur more frequently than other categories, we observe that they achieve a classification performance lower than *Deprecation* and *Ownership* (Table 6). This outcome can be due to the presence of specific annotations or words that often occur in sentences belonging to the *Deprecation* (*e.g., @since*) and *Ownership* (*e.g., @author*) categories. Although we removed all the special characters (annotation symbols) from the sentences, the techniques based on NLP+TEXT features can well capture the specific terms that frequently occur and are useful for identifying these categories. For instance, in the Vaadin project, the *Ownership* category always contains "Vaadin" in the *@author* field. Similarly, in the other Java projects, author name patterns are included in the NLP+TEXT feature set.

In contrast with *Deprecation* and *Ownership* categories, we do not observe recurrent annotations or words introducing sentences of the *Rationale* type. However, they are more accurately classified compared to sentences in the *Summary*, and *Expand* categories. This could depend on the quantity and quality of the NLP features captured in these categories, jointly with a lower variability in the structure of comments falling in the *Rationale* class. In particular, we observed that for the *Rationale* category

Table 7
Results for Python using the random forest classification model.

Category	NLP + TEXT		
	Precision	Recall	F-Measure
Summary	0.86	0.86	0.85
Usage	0.83	0.83	0.82
Expand	0.83	0.86	0.83
Development Notes	0.87	0.89	0.87
Parameters	0.86	0.86	0.85

Table 8
Results for Smalltalk using the random forest classification model.

Category	NLP + TEXT		
	Precision	Recall	F-Measure
Responsibility	0.79	0.82	0.78
Intent	0.92	0.92	0.90
Collaborator	0.83	0.94	0.83
Example	0.85	0.84	0.85
Class references	0.29	0.98	0.29
Key messages	0.92	0.92	0.89
Key implementation points	0.87	0.89	0.85

twelve unique NLP features have *information gain* values higher than 0.01, whereas two unique NLP features for the *Summary* category and only one NLP feature for the *Expand* category have *information gain* scores higher than 0.01. In terms of quality, the top-ranked NLP feature (or pattern) of the *Summary* category (*i.e.,* "*Represents [something]*") occurs in only 3% of the overall comments falling in this category, whereas the top-ranked feature of the *Rationale* category (*i.e.,* "*Has [something]*") occurs in 8% of the total comments belonging to such category. Nevertheless, the NLP heuristics that occur in the sentences belonging to the *Expand* class are also frequent in the instances of the *Pointer* and *Usage* categories, making it harder to correctly predict the type of the sentences falling in these categories. Specifically, the NLP feature with the highest *information gain* score (*i.e.,* "*See [something]*") for the *Expand* category (information gain = 0.00676) is also relevant for identifying sentences of the *Pointer* category, exhibiting an information gain value higher than the one observed for the *Expand* category (*i.e.,* 0.04104).

In the case of Python (see Table 7), the F-measure results are still positive (> 80%) for all the considered categories. Similar to Java, more frequent categories do not achieve the best performance. For example, the category *Parameter* is the least frequent among the categories considered but achieves higher performance than most of the categories, as shown in Table 7. In contrast to Java, Python developers frequently use specific words (*e.g.,* "params", "args", or "note"), rather than annotations, to denote a specific information type. We observe that these words frequently appear in sentences of the *Parameter* and *Development note* categories and these terms are captured in the related feature sets. In the Python classification, the *Usage* category reports the lowest F-measure due to its maximum ratio of incorrectly classified instances, *i.e.,* 17% among all categories. This outcome can be partially explained by the small number of captured NLP heuristics (one heuristic *"[something] defaults"* is selected according to the information gain measure with threshold 0.005). We also observe that instances of the *Usage* category often contain code snippets mixed with informal text, making it hard to identify features that would be good predictors of this class. Similarly, the instances in the *Expand* category also contain mixtures of natural language and code snippets. Separating code snippets from natural language elements and treating each portion of the mixed text with a proper approach can help (i) to build more representative feature sets for these types of class comment sentences, and (ii) to improve overall classification performance.

Concerning Smalltalk, the Random Forest model provides slightly less stable results compared to Python and Java (see Table 5). As shown in Table 8, in the case of Smalltalk, *Intent* is the comment type with the highest F-measure. However, for most categories F-measure results are still positive (> 78%), except for the *Class references* category. The *Class references* category captures the other classes referred to in the class comment. Random Forest is the ML algorithm that achieves the worst results (F-Measure of 29%) for this category. However, the Naive Bayes algorithm achieves an F-Measure score of 93% for the same category. Similarly for the *Collaborator* category the Naive Bayes model achieved better results compared to the Random Forest model. Both categories can contain similar information, *i.e.,* the name of other classes the class interacts with. We observe that in Smalltalk, camel case class names are generally split into separate words in comments, thus making it hard to identify them as classes from the text. Nevertheless, as demonstrated in previous work (Pascarella and Bacchelli, 2017), the Naive Bayes algorithm can achieve high performance in classifying information chunks in code comments containing code elements (*e.g., Pointer*), while its performance degrades when dealing with less structured texts (*e.g., Rationale*). We observe similar behavior for the *Links* category in Python taxonomy. Fig. 8 shows that all these categories such as *Links*, *Pointer*, *Collaborator*, and *Class references* contain similar types of information. In contrast, developers follow specific patterns in structuring sentences belonging to other categories such as *Intent* and *Example*, as reported in previous work (Rani et al., 2021). In future work, we plan to explore combining various machine learning algorithms to improve the results (Alexandre et al., 2001).

To qualitatively corroborate the quantitative results and understand the importance of each considered NLP heuristic, we computed the popular statistical measure *information gain* for the NLP features in each category, and ranked these features based on their scores. We use the default implementation of the information gain algorithm and the ranker available in Weka with the threshold value 0.005 (Quinlan, 1986). Interestingly, for each category, the heuristics having the highest information gain values also exhibit easily explainable relations with the intent of the category itself. For instance, for the *Responsibility* category in Smalltalk (which lists the responsibilities of the class) we observe that *"Allows [something]"* and *"Specifies [something]"* are among the best-ranked heuristics. Similarly, we report that *"[something] is used"* and *"[something] is applied"* are among the heuristics having the best information gain values for the *Collaborator* category (*i.e.,* interactions of the class with other classes), while the heuristics *"[something] is class for [something]"* and *"Represents [something]"* have higher information gain values when used to identify comments of the *Intent* type (*i.e.,* describing the purpose of the class). These heuristics confirm the patterns identified by Rani et al. in their manual analysis of Smalltalk class comments (Rani et al., 2021). Similar results are observed for the other languages. More specifically, for the *Summary* category in Java, the heuristics *"Represents [something]"* and *"[something] tests"* are among the NLP features with the highest information gain. Instead, in the case of the *Expand* and *Pointer* categories, we observe a common relevant heuristic: *"See [something]"*.. The analysis of the top heuristics for the considered categories highlight that developers follow similar patterns (*e.g., "[verb] [something]"*) to summarize the purpose and responsibilities of the class across

the different programming languages. However, no common patterns are found when discussing specific implementation details (*Expand* and *Usage* in Java and Python and *Example* in Smalltalk).

Finding 6. In all the considered programming languages, developers follow similar patterns to summarize the purpose and the responsibilities of a class. No common patterns are observed when implementation details are discussed.

Discussion: To further confirm the reliability of our results, we complement previous results with relevant statistical tests. In particular, the *Friedman test* reveals that the differences in performance among the classifiers is statistically significant in terms of F-Measure. Thus, we can conclude that when comparing the performance of classifiers and using different input configurations, the choice of the classifier significantly affects the results. Specifically, to gain further insights about the groups that statistically differ, we performed the Nemenyi test. Results of the test suggest that the Naive Bayes and the J48 models do not statistically differ in terms of F-Measure, while the Random Forest model is the best performing model, with statistical evidence (p-value < 0.05).

To analyze how the usage of different features (NLP, TEXT, NLP+TEXT) affects the classification results, we also executed a Friedman test on the F-Measure scores obtained by the Random Forest algorithm for each possible input combination. The test concluded that the difference in the results with different inputs is statistically significant (p-value < 0.05). To gain further insight into the groups that statistically differ, we performed a Nemenyi test. The test revealed that there is significant difference between the NLP and NLP+TEXT combinations (p-value < 0.05). This result confirms the importance of both NLP and TEXT features when classifying class comment types in different languages. The input data and the scripts used for the tests are provided in the Replication package.[23] Currently, we use the TF–IDF weighting scheme for TEXT features but we plan to experiment with other weighting schemes for future work, for instance, TF–IDF-ICSDF (Inverse Class Space Density Frequency) as it considers also the distribution of inter-class documents when calculating the weight of each term (Dogan and Uysal, 2020).

4.3. Reproducibility

In order to automate the study, we developed a Command Line Interface (CLI) application in Java. The application integrates the external tool NEON, and the required external libraries (Standford NLP) to process and analyze the data (WEKA) using Maven. The input parameters for the application are the languages (*e.g.,* Java, Python) to analyze, their input dataset path, and the tasks to perform. Various tasks fetch the required input data from the database, perform the analysis, and store the processed output data back in it. The results of intermediate steps are stored in the database[24] and the final results are exported as CSV automatically using *Apache Commons CSV*.

5. Threats to validity

We now outline potential threats to the validity of our study. *Threats to construct validity* mainly concern the measurements used in the evaluation. To answer the RQs, we did not consider the full ecosystem of projects in each language but selected a sample of projects for each language. To alleviate this concern to some extent, we selected heterogeneous projects used in the earlier comment analysis work of Java, Python, and Smalltalk (Pascarella and Bacchelli, 2017; Zhang et al., 2018; Rani et al., 2021). The projects in each language focus on different domains such as visualization, data analysis, or development frameworks. They originate from different ecosystems, such as Google and Apache in Java, or Django Foundation, or community project in Python. Thus, the projects follow different comment guidelines (or coding style guidelines). Additionally, the projects are developed by many contributors (developers), which further lowers the risk toward a specific developer commenting style.

Another important issue could be due to the fact that we sampled only a subset of the extracted class comments. However, the sample size limits the estimation imprecision to 5% of error for a confidence level of 95%. To further mitigate concerns related to subjectiveness and bias in the evaluation, the truth set was built based on the judgment of four annotators (four authors of this work) who manually analyzed the resulting sample. Moreover, an initial set of 50 elements for each language was preliminarily labeled by all annotators and all disagreements were discussed between them. To reduce the likelihood that the chosen sample comments are not representative of the whole population, we used a stratified sampling approach to choose the sample comments from the dataset, thus considering the quartiles of the comment distribution for each language.[25]

Another threat to construct validity concerns the definition of the *CCTM* and the mappings of the different language taxonomies performed by four human subjects. To counteract this issue, we used the categories defined by the earlier works in the comment analysis (Pascarella and Bacchelli, 2017; Zhang et al., 2018; Rani et al., 2021).

Threats to internal validity concern confounding factors that could influence our results. The main threat to internal validity in our study is related to the manual analysis carried out to prepare the *CCTM* and the mapping taxonomy. Since it is performed by human subjects, it could be biased. Indeed, there is a level of subjectivity in deciding whether a comment type belongs to a specific category of the taxonomy or not, and whether a category of one language taxonomy maps to a category in another language taxonomy or not. To counteract this issue, the evaluators of this work were two Ph.D. candidates and two faculty members, each having at least four years of programming experience. All the decisions made during the evaluation process and validation steps are reported in the replication package (to provide evidence of the non-biased evaluation), and described in detail in the paper.[26] Also, we performed a two-level validation step. This validation step involved further discussion among the evaluators, whenever their opinions diverged, until they reached a final consensus.

Threats to conclusion validity concern the relationship between treatment and outcome. Appropriate statistical procedures have been adopted to draw our conclusions. To answer RQ_2, we investigate whether the differences in the performance achieved by the different machine learning models with different combination of features were statistically significant. To perform this task, we used the Friedman test, followed by a post-hoc Nemenyi test, as recommended by Demšar (2006).

Threats to external validity concern the generalization of our results. The main aim of this paper is to investigate the class commenting practices for the selected programming languages. The

[23] Folder "RP/Result/RQ2/Statistical-analysis" in the Replication package.
[24] File "RP/Results/RQ2/All-steps-result.sqlite" in the Replication package.
[25] File "RP/Dataset/RQ1/Java/projects-distribution.pdf" in the Replication package.
[26] Folder "RP/Result/RQ1/Manually-classified-comments" in the Replication package.

proposed approach may achieve different results in other programming languages or projects. To limit this threat, we considered both static and dynamic types of object-oriented programming languages having different commenting style and guidelines. To reduce the threat related to the project selection, we chose diverse projects, used in previous studies about comment analysis and assessment. The projects vary in terms of size, domain, contributors, and ecosystems. Finally, during the definition of our taxonomy (*i.e., CCTM*) we mainly rely on a quantitative analysis of class comments, without directly involving the actual developers of each programming language. Specifically, for future work, we plan to involve developers, via surveys and interviews. This step is particularly important to improve the results of our work and to design and evaluate further automated approaches that can help developers achieve a high quality of comments.

6. Related work

Comment classification. Comments contain various information types (Ying et al., 2005) useful to support numerous software development tasks. Recent work has categorized the links found in comments (Hata et al., 2019), and proposed comment categories based on the actual meaning of comments (Padioleau et al., 2009). Similar to these studies, our work is aimed at supporting developers in discovering important types of information from class comments.

Steidl et al. classified the comments in Java and C/C++ programs automatically using machine learning approaches. The proposed categories are based on the position and syntax of the comments, *e.g.,* inline comments, block comments, header comments *etc* (Steidl et al., 2013). Differently from Steidl et al. our work focuses on analyzing and identifying semantic information found in class comments in Java, Python, and Smalltalk. Pascarella et al. presented a taxonomy of code comments for Java projects (Pascarella and Bacchelli, 2017). In the case of Java, we used the taxonomy from Pascarella et al. to build our Java CCTM categories. However, our work is different from the one of Pascarella et al. as it focuses on class comments in three different languages, which makes our work broader in terms of studied languages and more specific in terms of type of code comments studied. Complementary to Pascarella, et al.'s work, Zhang et al. (2018) reported a code comment taxonomy in Python. Compared to the Python comment taxonomy of Zhang et al. we rarely observed the *Version*, *Todo*, and *Noise* categories in our Python class comment taxonomy. More importantly, we found other types of information in Python class comments that developers embed in the class comments but were not included in the Python comment taxonomy of Zhang et al. such as the *Warning*, *Observation*, *Recommendation* and *Precondition* categories of the proposed CCTM. More in general, our work complement and extend the studies of Pascarella et al. and Zhang et al. by focusing on class comments in three different languages, which makes our work broader in terms of studied languages as well as the types of code comments reported and automatically classified.

Several studies have experimented with numerous approaches to identify the different types of information in comments (Dragan et al., 2010; Ying and Robillard, 2014; Shinyama et al., 2018; Geist et al., 2020). For instance, Dragan et al. used a rule-based approach to identify the Stereotype of a class based on the class signature (Dragan et al., 2010). Their work is aimed at recognizing the class type (*e.g.,* data class, controller class) rather than the type of information available within class comments, which is the focus of our work. Shinyama et al. (2018) focused on discovering specific types of local comments (*i.e.,* explanatory comments) that explain how the code works at a microscopic level inside the functions. Similar to our work, Shinyama et al. and Geist et al. considered recurrent patterns, but crafted them manually as extra features to train the classifier. Thus, our approach is different, as it is able to *automatically* extract different natural language patterns (heuristics), combining them with other textual features, to classify class comment types of different languages.

Further comment analysis. Apart from identifying information types within comments, analyzing comments for other purposes has also gained a lot of attention in the research community in the past years. Researchers are investigating methods and techniques for generating comments (Haiduc et al., 2010; Nielebock et al., 2019), assessing their quality (Khamis et al., 2010; Steidl et al., 2013; Yu et al., 2016), detecting inconsistency between code and comments (Ratol and Robillard, 2017; Wen et al., 2019; Zhou et al., 2017; Liu et al., 2018), examining co-evolution of code and comments (Jiang and Hassan, 2006; Fluri et al., 2007, 2009; Ibrahim et al., 2012), identifying bugs using comments (Tan et al., 2007), or establishing traceability between code and comments (Marcus and Maletic, 2003; Antoniol et al., 2000).

Other work has focused on experimenting with rule-based and machine-learning-based techniques and identified further program comprehension challenges. For instance, for labeling source code, De Lucia et al. (2012) found that simple heuristic approaches work better than more complex approaches, *e.g.,* LSI and LDA. Moreno et al. (2013) proposed NLP and heuristic-based techniques to generate summaries for Java classes. Finally, recent research by Fucci et al. (2019) studied how well modern text classification approaches can identify the information types in API documentation automatically. Their results have shown how neural network outperforms traditional machine learning approaches and naive baselines in identifying multiple information types (multi-label classification). In the future, we plan to experiment with class comment classification using neural networks.

7. Conclusion

Class comments provide a high-level understanding of the program and help one to understand a complex program. Different programming languages have their own commenting guidelines and notations, thus identifying a particular type of information from them for a specific task or evaluating them from the content perspective is a non-trivial task.

To handle these challenges, we investigate class commenting practices of a total of 20 projects from three programming languages, Java, Python, and Smalltalk. We identify more than 17 types of information class comments of each programming language. To automatically identify the most frequent information types characterizing these languages, we propose an approach based on natural language processing and text analysis that classifies, with high accuracy, the most frequent information types of all the investigated languages.

The contributions of our work are (i) an empirically validated taxonomy for class comments in three programming languages; (ii) a mapping of taxonomies from previous works; (iii) a common automated classification approach able to classify class comments according to a broad class comment taxonomy using various machine learning models trained on top of different feature sets; and (iv) a publicly available dataset of 37 446 class comments from 20 projects and 1066 manually classified class comments.[27]

Our results highlight the different kinds of information class comments contain across languages. We found many instances of specific information types that are not suggested or mentioned by their respective coding style guidelines. To what extent developer commenting practices adhere to these guidelines is not known yet and is part of our future work agenda. We argue that such

[27] Folder "RP/Dataset/RQ1/Java" in the Replication package.

an analysis can help in evaluating the quality of comments, also suggested by previous works (Padioleau et al., 2009; Haouari et al., 2011; Steidl et al., 2013). To investigate this aspect, we plan to extract the coding style guidelines of heterogeneous projects related to comments and compare the extracted guidelines with the identified comment information types (using our proposed approach). Moreno et al. used a template-based approach to generate Java comments (Moreno and Marcus, 2017) where the template includes specific types of information they deem important for developers to understand a class. Our results show that frequent information types vary across systems. Using our approach to identify frequent information types, researchers can customize the summarization templates based on their software system.

CRediT authorship contribution statement

Pooja Rani: Conceptualization, Data curation, Software, Methodology, Investigation, Validation, Writing – original draft, Writing – review & editing, Visualization, Project administration. **Sebastiano Panichella:** Conceptualization, Software, Methodology, Investigation, Validation, Formal analysis, Writing – review & editing, Project administration. **Manuel Leuenberger:** Conceptualization, Software, Methodology, Investigation. **Andrea Di Sorbo:** Conceptualization, Software, Methodology, Investigation, Validation, Writing – review & editing. **Oscar Nierstrasz:** Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assistance" (SNSF project No. 200020-181973, Feb. 1, 2019–April 30, 2022). We also thank Ivan Kravchenko for helping us to extract the data for Java and Python.

References

Alexandre, L.A., Campilho, A.C., Kamel, M., 2001. On combining classifiers using sum and product rules. Pattern Recognit. Lett. 22 (12), 1283–1289.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., 2000. Information retrieval models for recovering traceability links between code and documentation. In: Proceedings of the International Conference on Software Maintenance (ICSM 2000). pp. 40–49. http://dx.doi.org/10.1109/ICSM.2000.883003.

Baeza-Yates, R., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley, URL http://sunsite.dcc.uchile.cl/irbook.

Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., Panichella, S., 2013. An empirical investigation on documentation usage patterns in maintenance tasks. In: 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 210–219.

Binkley, D., Lawrie, D., Hill, E., Burge, J., Harris, I., Hebig, R., Keszocze, O., Reed, K., Slankas, J., 2013. Task-driven software summarization. In: 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 432–435.

Choi, N., Song, I.-Y., Han, H., 2006. A survey on ontology mapping. ACM Sigmod Rec. 35 (3), 34–41.

Cioch, F.A., Palazzolo, M., Lohrer, S., 1996. A documentation suite for maintenance programmers. In: Proceedings of the 1996 International Conference on Software Maintenance. In: ICSM '96, IEEE Computer Society, Washington, DC, USA, pp. 286–295, URL http://dl.acm.org/citation.cfm?id=645544.655870.

Cline, A., 2015. Testing thread. In: Agile Development in the Real World. Springer, pp. 221–252.

Curiel, A., Collet, C., 2013. Sign language lexical recognition with propositional dynamic logic. In: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 2: Short Papers. The Association for Computer Linguistics, pp. 328–333, URL https://www.aclweb.org/anthology/P13-2059/.

De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., Panichella, S., 2012. Using IR methods for labeling source code artifacts: Is it worthwhile?. In: 2012 20th IEEE International Conference on Program Comprehension (ICPC). IEEE, pp. 193–202.

Demšar, J., 2006. Statistical comparisons of classifiers over multiple data sets. J. Mach. Learn. Res. 7, 1–30.

Di Sorbo, A., Panichella, S., Visaggio, C.A., Di Penta, M., Canfora, G., Gall, H.C., 2015. Development emails content analyzer: Intention mining in developer discussions (T). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 12–23.

Di Sorbo, A., Panichella, S., Visaggio, C.A., Di Penta, M., Canfora, G., Gall, H., 2016. Deca: development emails content analyzer. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C). IEEE, pp. 641–644.

Di Sorbo, A., Panichella, S., Visaggio, C.A., Di Penta, M., Canfora, G., Gall, H.C., 2019. Exploiting natural language structures in software informal documentation. IEEE Trans. Softw. Eng..

Dogan, T., Uysal, A.K., 2020. A novel term weighting scheme for text classification: tf-mono. J. Informetr. 14 (4), 101076. http://dx.doi.org/10.1016/j.joi.2020.101076, URL https://www.sciencedirect.com/science/article/pii/S1751157720300705.

Dragan, N., Collard, M.L., Maletic, J.I., 2010. Automatic identification of class stereotypes. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance. In: ICSM '10, IEEE Computer Society, USA, pp. 1–10. http://dx.doi.org/10.1109/ICSM.2010.5609703.

Euzenat, J., 2001. Towards a principled approach to semantic interoperability. In: Proc. IJCAI 2001 Workshop on Ontology and Information Sharing. In: Proc. IJCAI 2001 workshop on ontology and information sharing, No commercial editor., Seattle, United States, pp. 19–25, URL https://hal.inria.fr/hal-00822909, euzenat2001b.

Farooq, M., Khan, S., Abid, K., Ahmad, F., Naeem, M., Shafiq, M., Abid, A., 2015. Taxonomy and design considerations for comments in programming languages: A quality perspective. J. Qual. Technol. Manag. 10 (2).

Fjeldstad, R.K., Hamlen, W.T., 1983. Application Program Maintenance Study: Report to Our Respondents. In: Proceedings GUIDE 48.

Fluri, B., Wursch, M., Gall, H.C., 2007. Do code and comments co-evolve? On the relation between source code and comment changes. In: Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on. IEEE, pp. 70–79.

Fluri, B., Würsch, M., Giger, E., Gall, H.C., 2009. Analyzing the co-evolution of comments and source code. Softw. Qual. J. 17 (4), 367–394.

Fucci, D., Mollaalizadehbahnemiri, A., Maalej, W., 2019. On using machine learning to identify knowledge in API reference documentation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 109–119.

Geist, V., Moser, M., Pichler, J., Beyer, S., Pinzger, M., 2020. Leveraging machine learning for software redocumentation. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 622–626.

Goldberg, A., Robson, D., 1983. Smalltalk 80: The Language and Its Implementation. Addison Wesley, Reading, Mass., URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

Goldstein, I.P., Bobrow, D.G., 1980. Extending object-oriented programming in smalltalk. In: Proceedings of the Lisp Conference, pp. 75–81.

Guzzi, A., Bacchelli, A., Lanza, M., Pinzger, M., van Deursen, A., 2013. Communication in open source software development mailing lists. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, pp. 277–286.

Haiduc, S., Aponte, J., Moreno, L., Marcus, A., 2010. On the use of automated text summarization techniques for summarizing source code. In: 2010 17th Working Conference on Reverse Engineering. IEEE, pp. 35–44.

Haouari, D., Sahraoui, H.A., Langlais, P., 2011. How good is your comment? A study of comments in java programs. In: Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, Banff, AB, Canada, September 22-23, 2011. IEEE Computer Society, pp. 137–146. http://dx.doi.org/10.1109/ESEM.2011.22.

Hata, H., Treude, C., Kula, R.G., Ishio, T., 2019. 9.6 million links in source code comments: Purpose, evolution, and decay. In: Proceedings of the 41st International Conference on Software Engineering. IEEE Press, pp. 1211–1221.

Ibrahim, W.M., Bettenburg, N., Adams, B., Hassan, A.E., 2012. On the relationship between comment update practices and software bugs. J. Syst. Softw. 85 (10), 2293–2304.

Jiang, Z.M., Hassan, A.E., 2006. Examining the evolution of code comments in PostgreSQL. In: Proceedings of the 2006 International Workshop on Mining Software Repositories. ACM, pp. 179–180.

Khamis, N., Witte, R., Rilling, J., 2010. Automatic quality assessment of source code comments: the JavadocMiner. In: International Conference on Application of Natural Language To Information Systems. Springer, pp. 68–79.

Liu, Z., Chen, H., Chen, X., Luo, X., Zhou, F., 2018. Automatic detection of outdated comments during code changes. In: Reisman, S., Ahamed, S.I., Demartini, C., Conte, T.M., Liu, L., Claycomb, W.R., Nakamura, M., Tovar, E., Cimato, S., Lung, C., Takakura, H., Yang, J., Akiyama, T., Zhang, Z., Hasan, K. (Eds.), 2018 IEEE 42nd Annual Computer Software and Applications Conference, COMPSAC 2018, Tokyo, Japan, 23-27 July 2018, Volume 1. IEEE Computer Society, pp. 154–163. http://dx.doi.org/10.1109/COMPSAC.2018.00028.

Lovins, J.B., 1968. Development of a stemming algorithm. Mech. Transl. Comput. Linguist. 11 (1–2), 22–31.

Maalej, W., Tiarks, R., Roehm, T., Koschke, R., 2014. On the comprehension of program comprehension. ACM TOSEM 23 (4), 31:1–31:37. http://dx.doi.org/10.1145/2622669, URL http://mobis.informatik.uni-hamburg.de/wp-content/uploads/2014/06/TOSEM-Maalej-Comprehension-PrePrint2.pdf.

Marcus, A., Maletic, J.I., 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 125–135.

Misra, V., Reddy, J.S.K., Chimalakonda, S., 2020. Is there a correlation between code comments and issues?: an exploratory study. In: SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, Online Event, [Brno, Czech Republic], March 30 - April 3, 2020. pp. 110–117. http://dx.doi.org/10.1145/3341105.3374009.

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L.L., Vijay-Shanker, K., 2013. Automatic generation of natural language summaries for Java classes. In: IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013. pp. 23–32.

Moreno, L., Marcus, A., 2017. Automatic software summarization: the state of the art. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume, pp. 511–512.

Müller, S.C., Fritz, T., 2013. Stakeholders' information needs for artifacts and their dependencies in a real world context. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance. In: ICSM '13, IEEE Computer Society, Washington, DC, USA, pp. 290–299. http://dx.doi.org/10.1109/ICSM.2013.40.

Nazar, N., Hu, Y., Jiang, H., 2016. Summarizing software artifacts: A literature review. J. Comput. Sci. Tech. 31 (5), 883–909.

Nielebock, S., Krolikowski, D., Krüger, J., Leich, T., Ortmeier, F., 2019. Commenting source code: Is it worth it for small programming tasks? Empir. Softw. Eng. 24 (3), 1418–1457.

Nurvitadhi, E., Leung, W.W., Cook, C., 2003. Do class comments aid Java program understanding?. In: 33rd Annual Frontiers in Education, 2003. FIE 2003., Vol. 1. IEEE, p. T3C.

Padioleau, Y., Tan, L., Zhou, Y., 2009. Listening to programmers — Taxonomies and characteristics of comments in operating system code. In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, pp. 331–341.

Panichella, S., Aponte, J., Penta, M.D., Marcus, A., Canfora, G., 2012. Mining source code descriptions from developer communications. In: Beyer, D., van Deursen, A., Godfrey, M.W. (Eds.), IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012. IEEE Computer Society, pp. 63–72. http://dx.doi.org/10.1109/ICPC.2012.6240510.

Panichella, S., Di Sorbo, A., Guzman, E., Visaggio, C.A., Canfora, G., Gall, H.C., 2015. How can I improve my app? Classifying user reviews for software maintenance and evolution. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 281–290.

Panichella, S., Panichella, A., Beller, M., Zaidman, A., Gall, H.C., 2016. The impact of test case summaries on bug fixing performance: An empirical investigation. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). pp. 547–558. http://dx.doi.org/10.1145/2884781.2884847.

Pascarella, L., Bacchelli, A., 2017. Classifying code comments in Java open-source software systems. In: Proceedings of the 14th International Conference on Mining Software Repositories. In: MSR '17, IEEE Press, pp. 227–237. http://dx.doi.org/10.1109/MSR.2017.63.

2020. Pharo consortium. URL http://consortium.pharo.org verified on 10 Jan 2020.

2020. Python documentation guidelines. URL https://www.python.org/doc/ https://www.python.org/doc/ verified on 10 April 2020.

Quinlan, J.R., 1986. Induction of decision trees. Mach. Learn. 1 (1), 81–106.

Rajlich, V., 2000. Incremental redocumentation using the web. IEEE Softw. 17 (5), 102–106.

Rani, P., Panichella, S., Leuenberger, M., Ghafari, M., Nierstrasz, O., 2021. What do class comments tell us? An investigation of comment evolution and practices in pharo smalltalk. Empir. Softw. Eng. (in press). arXiv preprint arXiv:2005.11583.

Ratol, I.K., Robillard, M.P., 2017. Detecting fragile comments. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, pp. 112–122.

Scowen, R., Wichmann, B.A., 1974. The definition of comments in programming languages. Softw. - Pract. Exp. 4 (2), 181–188.

Shinyama, Y., Arahori, Y., Gondow, K., 2018. Analyzing code comments to boost program comprehension. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 325–334.

de Souza, S.C.B., Anquetil, N., de Oliveira, K.M., 2005. A study of the documentation essential to software maintenance. In: Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information. In: SIGDOC '05, ACM, New York, NY, USA, pp. 68–75. http://dx.doi.org/10.1145/1085313.1085331.

Steidl, D., Hummel, B., Juergens, E., 2013. Quality analysis of source code comments. In: Program Comprehension (ICPC), 2013 IEEE 21st International Conference on. IEEE, pp. 83–92.

Tan, L., Yuan, D., Zhou, Y., 2007. Hotcomments: How to make program comments more useful?. In: HotOS.

Triola, M., 2006. Elementary Statistics. Addison-Wesley.

Wen, F., Nagy, C., Bavota, G., Lanza, M., 2019. A large-scale empirical study on code-comment inconsistencies. In: Proceedings of the 27th International Conference on Program Comprehension. IEEE Press, pp. 53–64.

Woodfield, S.N., Dunsmore, H.E., Shen, V.Y., 1981. The effect of modularization and comments on program comprehension. In: Proceedings of the 5th International Conference on Software Engineering. IEEE Press, pp. 215–223.

Ying, A.T., Robillard, M.P., 2014. Selection and presentation practices for code example summarization. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 460–471.

Ying, A.T.T., Wright, J.L., Abrams, S., 2005. Source code that talks: An exploration of eclipse task comments and their implication to repository mining. SIGSOFT Softw. Eng. Notes 30 (4), 1–5. http://dx.doi.org/10.1145/1082983.1083152, URL http://doi.acm.org/10.1145/1082983.1083152.

Yu, H., Li, B., Wang, P., Jia, D., Wang, Y., 2016. Source code comments quality assessment method based on aggregation of classification algorithms. J. Comput. Appl. 36 (12), 3448–3453.

Zhang, J., Xu, L., Li, Y., 2018. Classifying python code comments based on supervised learning. In: Meng, X., Li, R., Wang, K., Niu, B., Wang, X., Zhao, G. (Eds.), Web Information Systems and Applications - 15th International Conference, WISA 2018, Taiyuan, China, September 14-15, 2018, Proceedings. In: Lecture Notes in Computer Science, vol. 11242, Springer, pp. 39–47. http://dx.doi.org/10.1007/978-3-030-02934-0_4.

Zhou, Y., Gu, R., Chen, T., Huang, Z., Panichella, S., Gall, H., 2017. Analyzing APIs documentation and code to detect directive defects. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press, pp. 27–37.

Pooja Rani is a Ph.D. student at the University of Bern (Switzerland). Her focus areas involve developing methodology and building tools to support developers in understanding code. Specifically, she studies code comments from various software systems and building tools to improve the quality of comments. She finished her masters at the Birla Institute of Technology and Science-Pilani (India) in 2017.

Sebastiano Panichella is a Computer Science Researcher at Zurich University of Applied Science (ZHAW). His main research goal is to conduct industrial research, involving both industrial and academic collaborations, to sustain the Internet of Things (IoT) vision, where future smart cities. Currently he is technical coordinator of H2020 and Innosuisse projects concerning DevOps for Complex Cyber–physical Systems. He authored (or co-authored) around sixty papers appeared in International Conferences and Journals. He serves and has served as program committee member of various international conference and as reviewer for various international journals in the fields of software engineering. He was selected as one of the top-20 Most Active Early Stage Researchers Worldwide in SE.

Manuel Leuenberger is a research visitor at the University of Bern (Switzerland). He is interested in inter-project dependencies, the evolution of these relations, and the challenges arising from API changes. He finished his masters at the University of Bern in 2017.

Andrea Di Sorbo is a research fellow at the University of Sannio, Italy. He received a Ph.D. in information technology from the University of Sannio, in 2018. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, text analysis, and software security and privacy. He co-authored several papers that appeared in flagship international conferences (ICSE, FSE, ASE) and journals (TSE, JSS, IST, JSEP). He serves and has served as review editor and guest associate editor for several journals in the field of software engineering. He is also a program committee member of some international conferences (ARES, MOBILESoft, SEAA).

Oscar Nierstrasz is Professor of Computer Science at the Institute of Computer Science (INF) in the Faculty of Science of the University of Bern, where he founded the Software Composition Group in 1994. He is co-author of over 300 publications and co-author of the open-source books Object-Oriented Reengineering Patterns and Pharo by Example.