# Code Comment Classification Taxonomies

Marija Kostić, Aleksa Srbljanović, Vuk Batanović and Boško Nikolić, *Member, IEEE*

***Abstract*—Code comments have become an increasingly important kind of software development metadata, due to the possibilities of automated code comment analysis and generation. Different downstream tasks inherently prioritize certain kinds of code comments over others, making it necessary to properly define and identify different comment classes. In this paper, we analyze, compare, and systematize previously proposed code comment classification taxonomies according to their comment classes and applicability. We also present a new taxonomy designed for the tasks of semantic code search and semantic text similarity, and we contrast it to the existing approaches.**

***Index Terms*—code comments; code comment taxonomy; comparison of classification taxonomies.**

## I. Introduction

Code comments represent an invaluable source of metadata regarding software implementation. They describe code functionalities and algorithmic specifics, provide usage instructions, point towards additional resources, denote potential or observed programming bugs and issues, etc. In short, code comments play a vital role in helping developers comprehend source code [1]. In this manner, code comments greatly increase code maintainability, particularly when dealing with large software projects and development teams.

Depending on the downstream task in focus, not all code comments are of equal importance. For instance, if one wishes to compare the functionality of two methods, comments which provide authorship information are of little consequence, whereas those describing program implementation are of much greater significance. However, distinct kinds of code comments can be difficult to distinguish, particularly when no clear keywords for each comment type exists. A further complication in identifying relevant comments is the fact that a standardized code comment taxonomy does not exist. Instead, multiple different code comment categorization solutions have been proposed so far, most often designed with a specific programming language and downstream task in mind.

In this paper, we first present a survey of the existing code comment taxonomies and their applications in downstream tasks. Afterward, we present a new comment classification schema suitable for the tasks of semantic code search and semantic textual similarity and applicable to various programming languages. We then compare our approach with previous code comment taxonomies and conclude with some pointers regarding the future use of our comment schema.

Marija Kostić is with the School of Electrical Engineering and the Innovation center of School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: marija.kostic@ic.etf.bg.ac.rs), (https://orcid.org/0000-0003-4923-3748).

Aleksa Srbljanović is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia, (e-mail: aleksa.srbljanovic@etf.bg.ac.rs).

Vuk Batanović is with the Innovation center of School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade Serbia, (e-mail: vuk.batanovic@ic.etf.bg.ac.rs), (https://orcid.org/ 0000-0003-2639-9091).

Boško Nikolić is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia, (e-mail: bosko.nikolic@etf.bg.ac.rs), (https://orcid.org/ 0000-0003-1142-9243).

## II. A Survey of Existing Taxonomies

In this section, we review existing code comment classification taxonomies and describe how those systems were applied to specific tasks. A similar, but much shorter survey of this kind was previously presented within [2].

Zhai et al. [3] wanted to leverage program analysis to systematically propagate comments, so that they can be passed on to uncommented code entities and help detect bugs. A bi-directional analysis was designed where: (1) program analysis propagates and updates code comments, and (2) comments provide additional semantic hints to enrich program analysis. For effective propagation, it was vital to understand what kind of information comments convey and to which code elements they refer to, as comments of different categories require different propagation rules. The authors introduced a code comment taxonomy in which there are two dimensions of interest: code entity and content perspective. Code entities commonly commented on are *class*, *method*, *statement*, and *variable*. As for the content perspective, five of them were identified: 1) *what* – a definition or a summary of the code entity's functionality; 2) *why* – the reason why the code entity is provided or its design rationale; 3) *how-it-is-done* – description of the implementation details; 4) *property* – properties of the entities such as pre-condition and post-conditions; 5) *how-to-use* – description of the usage, expected set-up, or the environment of the entity. A set of 5,000 comments from four popular Java libraries were classified at the sentence level. The agreement between two coders, measured using Cohen's Kappa metric [4], was 0.826.

Chen et al. [1] investigated the use of the relationship between code blocks and the categories of the corresponding comments to improve code summarization, where the aim is to automatically generate a code comment based on the given block of source code. They showed that a composite approach, where the most suitable summarization model is selected based on the comment category, outperforms other approaches. Comments were classified into six categories – *what*, *why*, *how-to-use*, *how-it-is-done*, *property* and *others*. Five of them are the same as categories of content perspective in [3], while the sixth one, named *others*, covers unspecified or ambiguous comments. For this task, 20,000 Java methods and their corresponding comments were manually classified.

 ISBN 978-86-7466-930-3

The overall agreement of the three annotators, expressed in the terms of Fleiss' Kappa score [5], is 0.79.

Padioleau et al. [6] studied code comments to understand developers' needs regarding the creation of new tools and languages or the improvement of the existing ones. Comments were classified along four dimensions according to the question of interest: 1) *What?* – content: What is in the comment? Does it contain useful information? Its categories *type*, *interface*, *code relationship* and *pastFuture* and subcategories are closely related to the specific usage of the C programming language for operating systems; 2) *Who?* – people involved: Who or which tool can benefit from a comment? Who is the comment author?; 3) *Where?* – code entity: Where in a file is a comment located?; 4) *When?* – time: When was a comment written? How did the comment evolve over time? The authors considered 1050 comments randomly sampled from the code of three operating systems written in the C programming language: Linux, FreeBSD and OpenSolaris. The *What?* and *Who?* dimensions were manually annotated for each comment, while the other two dimensions were labeled automatically.

Haouari et al. [7] investigated developers' commenting habits via an empirical study. For the quantitative aspect of the study, the authors determined the distribution of the comments with respect to the program construct type that follows it. This allowed them to see what program construct types are documented more often than others. Some of the observed constructs are *package declaration*, *import declaration*, *class declaration*, *method*, *constructor*, *for*, *while*, etc. For the qualitative aspect of the study a new comment taxonomy was designed. This taxonomy has four high-level dimensions: 1) *Object of the comment* which can be a single subsequent instruction (*follow*); the following block of instructions (*block*), no code in the vicinity (*nocode*), or any other situation (*other*); 2) *Comment type* which can be the description of the code functionality (*explanation*), future task to be completed like TODO items (*working*), old code that is commented out instead of being removed (*code*), or any other comments (*other*); 3) *Style* dimension is only observed in the case of explanatory comments (*type=explanation*) and can be either *explicit* or *implicit*; 4) *Quality* dimension is also specific only to explanatory comments. It involves three categories: *fair+* where comments describe functionalities of related code and give other information; *fair* where code functionality is adequately described; and *poor* where some or none of the functionality is described. Analysis for the quantitative aspect was fully automated and applied to all comments within three open-source Java projects. For the qualitative aspect, the authors had 49 developers manually classify 407 comments.

Steidl et al. [8] developed a model for comment quality analysis with four criteria (*coherence*, *usefulness*, *consistency*, and *completeness*). Their comment taxonomy consists of seven high-level categories: 1) *copyright* – copyright or license; 2) *header* – overview of the class functionality; 3) *member* – functionality of a method/field; 4) *inline* – implementation decisions within a method body; 5) *section* – group of methods/fields that belong to the same functional aspect; 6) *code* – commented out code; 7) *task* – developer notes with a remaining todo, a bug, or an implementation hack. Authors created a training set by classifying 830 Java and 500 C++ comments from twelve open-source projects.

Pascarella et al. [9]–[11] focused on increasing the empirical understanding of the types of comments that developers write in source code. After an iterative process of analyzing code files, the authors defined a fine-grained hierarchical taxonomy with two layers: the outer one consisting of six top-level categories (*purpose*, *notice*, *under development*, *style & IDE*, *metadata*, and *discarded*) and the inner one consisting of 16 subcategories. The *purpose* category contains comments that describe the functionality of the related source code. Its three subcategories *summary*, *expand*, and *rationale* respond to the question words what, how, and why, similarly to the categories in [1], [3]. The *notice* category covers comments about warnings, alerts, messages, or functionalities that should be used with care. Its subcategories are *deprecation*, *usage*, and *exception*. Subcategories *todo* (explicit actions to be done), *incomplete* (partial or empty comment bodies), and *commented code* belong to the *under development* top-level category. The *style & IDE* comments are used for communication with the IDE (*directive*) and logical separation of the code (*formatter*). The *metadata* comments define meta information about the code such as license, terms of use, authors, links to external resources (subcategories *license*, *ownership*, and *pointer*). All other comments that do not fit in the previous categories belong to the *discarded* category (subcategories *automatically generated* and *unknown*). Authors decided to classify comments at the character level. That means that annotators had to specify the starting and the ending character of each comment block and its category. After this process, it was found that in only 4% of cases one line had to be classified into more than one category. The study was conducted on more than 6,000 source code files with more than 40,000 lines of Java comments in open source and industrial software projects. To validate the proposed taxonomy, three developers were asked to manually classify 138 lines of comments in three Java source code files. They achieved a Fleiss' Kappa value of 0.9.

Unlike previously mentioned efforts, Shinyama et al. [12] worked only on comments inside functions or methods that explain code at the microscopic level i.e., on *local comments*. These comments are not visible in the documentation and often give insight into developers' minds. They are often crucial for understanding nontrivial parts of the code. As there usually is a relationship between a comment and the code it describes, the authors represented that relationship as an arc with three elements: (1) source – the comment itself; (2) destination – target code; and (3) type of relationship – comment category. They independently made a list of categories suitable for local comments: 1) *postcondition* – conditions that hold after the code is executed, typically used to explain what the code does; 2) *precondition* – conditions that hold before the code is executed, typically used for explaining why the code is needed; 3) *value description* – a

 ISBN 978-86-7466-930-3

phrase that can be equated with a variable, constant or expression; 4) *instruction* – instructions for code maintainers (todo comments); 5) *guide* – guides and examples for code users; 6) *interface* – description of a function, type, class, or interface; 7) *meta information* – author, date, or copyright; 8) *comment out* – commented out code; 9) *directive* – compiler directives that are not directed to human readers; 10) *visual cues* – comments inserted just for the ease of reading; 11) *uncategorized* – all other comments. For each arc element, a statistical classifier was built and trained on 1,000 manually classified Java comments. Classifiers were applied on large corpora of Java and Python comments. Annotation agreement was measured on a separate set of 100 Java comment-code pairs and reached Fleiss' Kappa value of 0.491.

Zhang et al. [13] used supervised learning to automatically classify Python code comments. Since most of the related work is based on Java and C/C++ programming languages, the authors conducted an iterative content analysis session to devise a Python-specific classification taxonomy. Their taxonomy contains 11 categories: 1) *metadata* – license and copyright; 2) *summary* – description of the functionality of the related code; 3) *usage* – explanations on how to use the code that can contain examples; 4) *parameters* – explanations of function parameters; 5) *expand* – detailed explanations of the purpose of a small block of code, usually inline comments; 6) *version* – library version information; 7) *development notes* – comments for developers concerning ongoing work, temporary tips, explanations of functions etc.; 8) *todo* – explicit actions to be done in the future like bug fixing or feature improving; 9) *exception* – indications that a function throws exceptions or suggestions how to prevent unwanted behaviors; 10) *links* – links to external resources; 11) *noise* – meaningless symbols which may be used for separation. The training set consisted of 330 annotated comments from seven popular Python open-source projects on GitHub.

## III. A New Code Comment Taxonomy

We explored code comment classification taxonomies in order to differentiate between different kinds of comments for the tasks of semantic code search and cross-level sematic textual similarity. In semantic code search (SCS), the goal is to construct a system which returns the most relevant code block(s) from a software repository for a given query in a natural language. To do so, most models rely on the accompanying code comments which describe the functionality of their respective code blocks [14]. Cross-level semantic similarity (CLSS) is the task in which a computational model ought to return a numerical semantic similarity score for a given pair of texts written in a natural language, where the length of the texts can be dissimilar (e.g., one text is a paragraph, the other is a sentence) [15]–[16]. Solving CLSS is of great use for semantic code search, since SCS implies finding semantic links between texts of different lengths – queries are usually limited to a couple of words or a sentence, whereas the length of code comments can range from a phrase to a paragraph. Obviously, these tasks are not limited to a particular programming or natural language.

We found that no previous code comment taxonomy was designed with these two downstream tasks in mind, so it was necessary to consider the previous classification systems and devise a suitable set of comment categories. Furthermore, we wanted to create a classification taxonomy that would be applicable to various programming languages, including C, C++, C#, Java, JavaScript/TypeScript, PHP, Python, and SQL. Upon reviewing the papers presented in the previous section, we decided to develop our own taxonomy using the approaches of Pascarella et al. [9]–[11] and Steidl et al. [8] as a starting point. This choice was based on the emphasis these works placed on the comments that describe code functionality, since such comments are the most relevant ones for SCS. We therefore distinguish between functional and non-functional comments via two top-level categories. These two categories are then subdivided into eight subcategories. In the remainder of this section, we will present the definition and scope of each of them.

### A. Functional

The *Functional* category contains comments that describe the functionality of the corresponding source code. Descriptions can be short, or they can extend over multiple lines. These comments are usually written in a natural language and are used to describe the purpose, behavior, or the reason why something is implemented in a particular way. They can respond to questions *What?*, *Why?*, and *How?*. We do not distinguish between these aspects of functionality because all of them are relevant for the SCS task. However, we do differentiate between three subcategories based on the type of the corresponding source code: 1) *Functional-Module* comments describe the functionality of a particular module like a class or an interface. If a programming language does not use the object-oriented paradigm, these comments pertain to entire files or scripts; 2) *Functional-Method* comments describe the functionality of a function or a method. They are usually located above or at the beginning of a function/method definition or declaration; 3) *Functional-Inline* are all the other comments that describe some functionality. They can describe the functionality of a variable or an expression and can be located inside a method body.

### B. Non-Functional

The *Non-Functional* category covers all comments that do not describe code functionality. Subcategories in this top-level category are not relevant for the SCS and CLSS tasks, but we still decided to include them because we wanted to make the annotated datasets usable for other downstream tasks as well. We distinguish between the following five subcategories: 1) The *Notice* category encompasses warnings, alerts, and messages intended for other developers or users of the source code. It also covers information about deprecated artifacts and instructions about alternative methods or classes that should be used. Comments that explain something is implemented in a certain way because of a bug, or a known issue, also belong to this category. Finally, examples or explicit suggestions how

 ISBN 978-86-7466-930-3

to use a functionality are classified as *Notice* comments as well; 2) *General* comments usually define meta-information about the code such as license, copyright, authorship, module/class version, timestamps, the name or path of the file, information about the libraries used in the source code, etc. These comments are usually located at the top of the file; 3) The *Code* category is composed of comments that contain source code that is commented out by developers. This is usually done during testing or debugging. This code may represent new or hidden features, work in progress, features being tested, temporarily removed code or older variants of the code; 4) *IDE* comments are used for communication with the IDE or the compiler to change their default behavior. Comment content is usually of limited value to human

TABLE 1 CLASSIFICATION OF EXAMPLE CODE COMMENTS ACCORDING TO DIFFERENT TAXONOMIES

| Class | Class | Example | Haouari [7] | Steidl [8] | Shinyama [12] | Zhang [13] | Pascarella [11] | Zhai [3] | Chen [1] | Our proposal |
|---|---|---|---|---|---|---|---|---|---|---|
| Functionality by type | What | `Pushes an item onto the top of this stack.` [3] | Type-Explanation | Header\|Inline\|Member | Postcondition | Summary | Summary | What | What | Functional |
| | Why | `It eliminates the need for explicit range operations.` [3] | | | Precondition | - | Rationale | Why | Why | |
| | How | `Shifts any subsequent elements to the left.` [3] | | | Uncategorized | Expand | Expand | How-it-is-done | How-it-is-done | |
| | Property | `The index must be a value greater than or equal to 0.` [3] | | | | - | - | Property | Property | |
| Functionality by position | Module | `This class is a member of the Java Collections Framework.` [3] | | Header | Interface | Summary | Purpose | Class | What\|Why How-it-is-done\|Property | Functional-module |
| | Method | `Check for symmetry, then construct the eigenvalue decomposition @param A square matrix` [8] | | Member | | Summary\|parameters | | Method | | Functional-method |
| | Inline | *`Increment this when there's a change requiring caches to be invalidated.`* | | Inline | (Post\|pre) condition | Expand | | Statement | | Functional-inline |
| | Variable/Field | `The number of characters to skip.` [3] | | Member | Value description | | | Variable | | |
| Section | | `--- Getter and Setter Methods ---` [8] | Type-Other | Section | Visual cue | - | Formatter | - | Others | Functional-inline |
| Automatically Generated | | *`COLORCORRECTION_HPP`* | | - | Uncategorized | Noise | Automatically generated | - | | |
| Notes | | `Caution: setting a new service manager stub won't replace the existing one.` [7] | | Task | | Development notes | Todo | - | | Notice |
| Usage | | `Example: renderText(100, 100, FONT, 12, "Hello");` [12] | Type-Working | - | | Usage | Usage | How-to-use | How-to-use | |
| Deprecation | | *`DEPRECATED: the following property is no longer in use, but defined until 2.0 to prevent conflicts`* | | - | Guide | Development notes | Deprecation | - | Others | |
| Exception | | *`@throws TransportExceptionInterface When an unsupported option is passed`* | | - | | Exception | Exception | - | | Functional-method |
| Link | | *`See https://github.com/symfony/symfony/pull/5582`* | | - | | Links | Pointer | - | | Notice |
| Bug | | *`Skip due to crash bug: https://support.microsoft.com/en-us/help/2908087`* | | Task | | Development notes | Todo | - | | Todo |
| Todo | | `TODO: Check synchronization.` [7] | | | Instruction | Todo | | - | | |
| Code | | `_mainFrame.hourglassOff();` [7] | Type-Code | Code | Commented out | Noise | Commented code | - | | Code |
| Ownership | | *`@author Ben Ramsey <ben@benramsey.com>`* | Type-other | Header | Meta information | | Ownership | - | | General |
| Meta | | *`Copyright(c) 2019 Intel Corporation.`* | | Copyright | | Metadata | License | - | | |
| Version | | *`@version $Revision: 1.0`* | | Header | Uncategorized | Version | Unknown | - | | |
| IDE | | `CHECKSTYLE:OFF` [12] | | - | Directive | - | Directive | - | IDE | |
| Other | | `The implementation is awesome.` [1] | | - | Uncategorized | Noise | Unknown | - | | Notice |

 ISBN 978-86-7466-930-3

readers; 5) The *Todo* category covers explicit tasks to be done and notes about bugs that need to be fixed.

## IV. Comparison With Previous Code Comment Classification Taxonomies

In this section, we present a comparison between the previously described taxonomies. Table 1 shows examples of code comments and their classes according to different taxonomies. In places where we were not sure what category the authors would choose for a specific comment, we put a dash symbol. Comments in italic are new examples, while others are taken from the cited papers. We have omitted the taxonomy presented in [6] from the table since it is extremely specific regarding the type of code it is applied to (operating systems code). Additionally, its authors have not disclosed all the categories they have devised.

All the comments which describe code functionality belong to one of the following classes - *What*, *Why*, *How*, *Property*, *Module*, *Method*, *Inline* and *Variable/Field*. However, these classes can be classified based on two perspectives: (1) according to the type of the commented functionality – categories *What*, *Why*, *How*, and *Property*, or (2) the comment's structural position within the code – categories *Module*, *Method*, *Inline*, *Variable/Field*. Depending on the downstream task, some taxonomies use the functionality type classification [1], [13], others use the comment place classification [8], some use both [3], [12], or neither [7]. The new taxonomy we propose takes into account the placement of a comment within the code. It should be emphasized we do not differentiate between *Inline* and *Variable/Field* functional comments like in some taxonomies [3], [8], [12], but rather group them together under the *Functional-Inline* class.

In some papers there is a separate class for comments which visually divide the code into sections [8], [11]–[12]. We classify such comments as *Functional-Inline* as well.

Some comments do not describe code functionality, but rather contain notes to developers and code users. Several different classes for these kinds of comments have been previously proposed. There are authors [1], [3], [7]–[8] who recognize only some of these comments because not all of them are relevant for their downstream task. In other papers [11]–[13] most of these comments are recognized, but every paper uses a different approach concerning their classification. Some authors [11], [13] use a higher level of granularity while others [12] perceive some or all such comments as one class. In [12] the authors also differentiate between the comments meant for developers and those meant for users. Regarding TODO comments, some taxonomies [12]–[13] clearly separate them from the other comments, while in others [8], [11] there is an overlap between TODO and other comment categories. Our taxonomy distinguishes between *Notice* and *Todo* classes. All the notes for developers/users, use cases, warnings about deprecation, exceptions and links are classified as *Notice*. Messages about missing/unfinished parts of code and bugs are classified as *TODO*.

Several taxonomies [7]–[8], [11]–[12], treat comments which represent parts of code as a separate class, while others do not mention them. In [11], code which is commented out is classified as a *Todo* comment, along with comments for bugs and unfinished code. In our taxonomy parts of code which are commented out are placed in a separate class – *Code*.

Most previous approaches use a separate class for meta-information comments. Some [8], [11], [13] utilize a more granular classification according to license information, authorship, version information etc. In our approach all the meta-information is classified into the *General* category.

A few authors [11]–[12] have proposed a separate class for comments which represent some instructions for the compiler or the development environment. We also include an *IDE* category in our taxonomy.

Some papers [1], [7], [11]–[13] use a separate category for all other comments which are not of interest. However, our taxonomy does not employ such a category, because we do not want to allow annotators to easily dismiss ambiguous comments which are difficult to categorize.

Additional information about the presented comment taxonomies is shown in Table 2. It contains the number of comments that are/will be annotated for each taxonomy, the used comment granularity, the natural and programing languages each taxonomy is applied to, the annotation agreement (if reported), and the downstream task. Data from the table shows that annotations are typically done on small sets of code comments written in English, and that comments are taken from one or two programming languages at most. It is hard to compare annotation agreements because agreement measures differ from paper to paper and relate to different numbers of annotators and different comment set sizes.

Some of the earliest taxonomies were specific for the task they were solving [6]–[7] and observed more than two perspectives (e.g., object, style, beneficiary etc.). Because of their complexity, they are not useful for tasks other than the ones they were designed for. But, over time, two perspectives became prominent: (1) what is the entity of a comment, and (2) how a comment describes the functionality of the entity.

All papers in this survey worked with comments in English and in one of the following programming languages: C/C++, Java, or Python. As mentioned in [11] many object-oriented languages have very similar functionalities, and it is reasonable to expect that their comments will behave the same. We can see that in more recent works [1], [3], [8], [11]–[13] taxonomies designed for Java, Python, and C++ are similar. For other programming paradigms (e.g., functional), further research must be done.

Although some authors wanted only to empirically study and understand the types of code comments [11]–[13], most of the times classification was done as a first step in solving a particular downstream task. In a couple of papers, it is shown that that kind of approach is fruitful. For example, Chen et al. [1] have found that different summarization models work best for different categories of comments. By including comment

 ISBN 978-86-7466-930-3

TABLE 2 GENERAL INFORMATION ABOUT CODE COMMENT CLASSIFICATION TAXONOMIES

| Paper | Year | Number of comments | Granularity | Language | Programming Languages | Annotation Agreement | Downstream task applicability |
|---|---|---|---|---|---|---|---|
| Padioleau [6] | 2009 | 1050 | Comment | English | C | - | Understanding developers' needs regarding the creation or improvement of tools and languages. |
| Haouari [7] | 2011 | 407 | Comment | English | Java | - | Investigation of developers commenting habits. |
| Steidl [3] | 2013 | 1330 | Comment | English | Java, C++ | - | Quality analysis of source code comments. |
| Shinyama [12] | 2018 | 1000 | Comment | English | Java, Python | Fleiss' Kappa = 0.491 | Analysis of comments inside functions or methods that often give insight into the developers' minds. |
| Zhang [13] | 2018 | 330 | Comment | English | Python | - | Classification of Python code comments. |
| Pascarella [11] | 2019 | 40000 | Character | English | Java | Fleiss' Kappa = 0.9 | Increasing the empirical understanding of the types of comments that developers write. |
| Zhai [3] | 2020 | 5000 | Sentence | English | Java | Cohen's Kappa = 0.826 | Code-comment propagation. |
| Chen [1] | 2021 | 20000 | Comment | English | Java | Fleiss' Kappa = 0.79 | Code summarization. |
| Our proposal | 2022 | ~10000 | Character | English | C/C++, C#, Java, JavaScript/ TypeScript, PHP, Python, SQL | To be determined | Semantic code search and cross-level semantic textual similarity. |

classification, they were able to design a composite summarization model that outperforms a standard approach where one model is applied to all comments. Another example is the work of Zhai et al. [3] focused on code comment propagation. Here, comment classification was necessary because comments with different content related to different programming entities cannot be propagated in the same way.

## V. CONCLUSION

In this paper, we have analyzed and compared previously proposed code comment classification taxonomies. We have systematized them according to the comment classes they use as well as according to their applicability to different programming languages and downstream tasks. We have also presented a new comment taxonomy, designed for the tasks of semantic code search and semantic textual similarity, applicable to various programming languages.

In order to validate the usefulness of our taxonomy, we are currently engaged in the creation of a code comment corpus which will encompass around 10,000 comments written in English or Serbian, and taken from a spectrum of programming languages(C/C++, C#, Java, PHP, Python, SQL, and JavaScript/TypeScript). We aim to manually annotate this corpus using the proposed taxonomy and use it to enable automated comment classification, both as a stand-alone task and as a first step within the mentioned downstream tasks.

## ACKNOWLEDGMENT

This work was supported by the Science Fund of the Republic of Serbia, grant no. 6526093, AI-AVANTES.

## REFERENCES

[1] Q. Chen, X. Xia, H. Hu, D. Lo, S. Li, "Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. *30*, no. *2*, pp. *1-29*, Feb 2021.

[2] B. Yang, Z. Liping, Z. Fengrong, "A Survey on Research of Code Comment," Proc. 2019 3rd International Conference on Management Engineering, Software Engineering, and Service Sciences-ICMSS 2019, Wuhan, China, pp. 45-51, Jan. 12, 2019.

[3] J. Zhai, X. Xu, Y. Shi, G. Tao, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, X. Zhang, "CPC: automatically classifying and propagating natural language comments via program analysis," Proc. ACM/IEEE 42nd International Conference on Software Engineering, Seoul, South Korea, pp. 1359-1371, Oct. 1, 2020.

[4] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. *20*, no. *1*, pp. *37-46*, 1960.

[5] J. L. Fleiss, "Measuring nominal scale agreement among many raters," *Psychological Bulletin*, vol. *76*, no. *5*, pp. *378-382*, 1971.

[6] Y. Padioleau, L. Tan, Y. Zhou, "Listening to programmers — Taxonomies and characteristics of comments in operating system code," Proc. 2009 IEEE 31st International Conference on Software Engineering, Vancouver, Canada, pp. 331-341, May 16-24, 2019.

[7] D. Haouari, H. Sahraoui, P. Langlais, "How Good is Your Comment? A Study of Comments in Java Programs," Proc. 2011 International Symposium on Empirical Software Engineering and Measurement, Banff, Canada, pp. 137-146, Sept. 22-23, 2011.

[8] D. Steidl, B. Hummer, E. Juergens, "Quality analysis of source code comments," Proc. 2013 21st International Conference on Program Comprehension, San Francisco, USA, pp. 83-92, May 20-21, 2013.

[9] L. Pascarella, "Classifying code comments in Java mobile applications," Proc. 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems, Gothenburg, Sweden, pp. 39-40, May 27-June 3, 2018.

[10] L. Pascarella, A. Bacchelli, "Classifying Code Comments in Java Open-Source Software Systems," Proc. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, Buenos Aires, Argentina, pp. 227-237, May 20-21, 2017.

[11] L. Pascarella, M. Bruntink, A. Bacchelli, "Classifying code comments in Java software systems," *Empirical Software Engineering*, vol. *24*, no. *3*, pp. *1499-1537*, Jun 2019.

[12] Y. Shinyama, Y. Arahori, K. Gondow, "Analyzing Code Comments to Boost Program Comprehension," Proc. 2018 25th Asia-Pacific Software Engineering Conference , Nara, Japan, pp. 325-334, Dec. 4-7, 2018.

[13] J. Zhang, L. Xu, Y. Li, "Classifying Python Code Comments Based on Supervised Learning," Proc. 2018 15th International Conference on Web Information Systems and Applications (WISA), Taiyuan, China, pp. 39-47, Sept. 14-15, 2018.

[14] H. Husain, H.-H. Wu, T. Gazit, G. Miltiadis, A. M. Brockschmidt, "CodeSearchNet Challenge Evaluating the State of Semantic Code Search," Accessed: Dec. 29, 2021. [Online]. Available: https://github.com/github/CodeSearchNet.

[15] D. Jurgens, M. T. Pilehvar, R. Navigli, "SemEval-2014 Task 3: Cross-Level Semantic Similarity," Accessed: Dec. 29, 2021. [Online]. Available: http://alt.qcri.

[16] D. Jurgens, M. T. Pilehvar, R. Navigli, "Cross level semantic similarity: an evaluation framework for universal measures of similarity," *Language Resources and Evaluation*, vol. *50*, no. *1*, pp. *5-33*, Mar 2016.

 ISBN 978-86-7466-930-3