A Large-Scale Empirical Study on Code-Comment Inconsistencies

Fengcai Wen, Csaba Nagy, Gabriele Bavota, Michele Lanza
Software Institute, Università della Svizzera italiana (USI), Switzerland
{*fengcai.wen, csaba.nagy, gabriele.bavota, michele.lanza*}*@usi.ch*

***Abstract*—Code comments are a primary means to document source code. Keeping comments up-to-date during code change activities requires substantial time and attention. For this reason, researchers have proposed methods to detect code-comment inconsistencies (*i.e.,* comments that are not kept in sync with the code they document) and studies have been conducted to investigate this phenomenon. However, these studies were performed at a small scale, relying on quantitative analysis, thus limiting the empirical knowledge about code-comment inconsistencies. We present the largest study at date investigating how code and comments co-evolve. The study has been performed by mining 1.3 Billion AST-level changes from the complete history of 1,500 systems. Moreover, we manually analyzed 500 commits to define a taxonomy of code-comment inconsistencies fixed by developers. Our analysis discloses the extent to which different types of code changes (*e.g.,* change of *selection* statements) trigger updates to the related comments, identifying cases in which code-comment inconsistencies are more likely to be introduced. The defined taxonomy categorizes the types of inconsistencies fixed by developers. Our results can guide the development of tools aimed at detecting and fixing code-comment inconsistencies.**

***Index Terms*—Software Evolution, Code Comments**

I. Introduction

Any code-related activity lays its foundations in program comprehension: before fixing a bug, refactoring a class, or writing new tests, developers first need to acquire knowledge about the involved code components. As recently shown by Xia *et al.* [1], this results in 58% of developers' time spent comprehending code. Besides the code itself, code comments are considered as the most important form of documentation for program comprehension [2]. Indeed, not surprisingly, studies showed that commented code is easier to comprehend than uncommented code [3], [4]. This empirical evidence also pushed researchers to consider code comments as a pivotal factor to study technical debt [5]–[7], or to assess code quality [8], [9].

While the importance of code comments is undisputed, developers do not always have the chance to carefully comment new code and/or to update comments as consequence of code changes [10]. This latter scenario might result in the introduction of code-comment inconsistencies, manifesting when the source code does not co-evolve with the related comments. For example, if a method comment is not updated after major changes to the method's application logic, the comment might provide misleading information to developers comprehending the method, hindering program comprehension rather than fostering it.

Given the potential harmfulness of code-comment inconsistencies, several researchers studied the co-evolution of code and comments [11]–[14], while others proposed techniques and tools able to detect code-comment inconsistencies automatically [15]–[18]. These techniques are able to identify specific types of code-comment inconsistencies. For example, @TCOMMENT [17] detects inconsistencies between Javadoc comments related to `null` values and exceptions with the behavior implemented in the related method's body, while Fraco [18] focuses on inconsistencies introduced as result of rename refactoring operations. Still, more research is needed in this area to increase the types of code-comment inconsistencies that can be automatically identified. Also, the empirical evidence provided by studies that pioneered the investigation of code-comment evolution [11]–[14] is limited to the analysis of the change history of a few software systems (less than 10).

To raise the knowledge about the co-evolution of code and comments and the introduction/fixing of code-comment inconsistencies, we present a large-scale empirical study quantitatively and qualitatively analyzing these phenomena. We mine the complete change history of 1,500 Java projects hosted on GitHub for a total of 3,323,198 analyzed commits. For each commit, we use GumTreeDiff [19] to extract AST operations performed on the files modified in it. In this way, we captured fine-grained changes performed in code (*e.g.,* change of a *selection* statement) as well as update, delete, and insert operations performed in the related comments. Overall, this process resulted in a database of $\sim$476 GB containing $\sim$1.3 Billion AST-level operations impacting code or comments. Using this data, we study the extent to which code changes impacting different code constructs (*e.g., literals*, *iteration statements*) trigger the update of the related code comments (*e.g.,* the developer adds a try statement and updates the method comment to "document" the changed code behavior).

Then, we manually analyze 500 commits identified, via a keywords-matching mechanism, as likely related to the fixing of code-comment inconsistencies. The output of this analysis is a taxonomy of code comment-related changes implemented by developers, from which we present relevant cases related to code-comment inconsistencies, and discuss implications for researchers and practitioners.

As a contribution to the research community, we make the database of fine-grained code changes publicly available. This enables the replication of this work, making also other types of investigations possible.

II. Related Work

We discuss related work concerning (i) empirical studies on code comments, and (ii) approaches for the detection of code-comment inconsistencies.

A. Empirical Studies on Code Comments

Woodfield *et al.* [3] conducted a user study with 48 programmers and showed that commented code is better understood by developers as compared to non-commented code.

Ying *et al.* [20] analyzed the usage of code comments in the IBM internal codebase. They show that comments are not only a means to document the source code, but also a communication channel towards colleagues, *e.g.,* to assign tasks and keep tracks of ongoing coding activities.

McBurney and McMillan [21] compared code summaries written by code authors and readers (*i.e.,* non-authors performing code understanding). They used the Short Text Semantic Similarity (STSS) metric to assess the similarity between source code and summaries written by the authors and compare it to the similarity between the code and the summaries written by the readers. They found that readers rely more on source code than authors when summarizing the code. Pascarella and Bacchelli [22] presented a hierarchical taxonomy of types of code comments for Java projects. Such a taxonomy, composed of six top categories and 16 inner categories, was built by manually analyzing 2,000 code comments. The taxonomy presented in this paper, differently from the one in [22], aims at classifying the types of code-comment inconsistencies fixed by software developers.

Other authors studied the evolution of code comments. Jiang and Hassan [11] conducted a study on the evolution of comments in PostgreSQL. They investigated the trend over time of the percentage of commented functions in PostgreSQL. Their results reveal that the proportion of commented functions remains constant over time.

Arafat *et al.* [23] studied the density of comments (*i.e.,* the number of comment lines divided by the number of code lines) in the history of 5,229 open source projects written in different programming languages. They show that the average comment density depends on the programming language (with the highest one of 25% measured for Java systems), while it is not impacted by the project and team size.

Ibrahim *et al.* [14] studied the relationship between comment update practices and bug introduction. Their findings show that abnormal comment update behavior (*e.g.,* missing to update a comment in a subsystem whose comments are always updated) leads to a higher probability of introducing bugs.

Fluri *et al.* [12] investigated how comments and source code co-evolved over time in three open source systems. They observed that 23%, 52%, and 43% of all comment changes in ArgoUML, Azureus, and JDT Core respectively, were due to source code changes, and in 97% of these cases the comment changes occurred in the same revision as the associated code change. However, newly added code barely got commented.

In a follow-up work, Fluri *et al.* [13] investigated the co-evolution between code and comment in eight systems. They found that the ratio between the growth of code and comments is constant but confirmed the previous observation about the frequent lack of comment updates for newly added code. They also found that (i) the type of code entity impacts its likelihood of being commented (*e.g.,* `if` statements are commented more often than other types of statements), and (ii) 90% of comment changes represent a simultaneous co-evolution with code (*i.e.,* they change in the same revision).

Our study stems from the seminal work by Fluri *et al.* [12], [13], but it is performed on a much larger scale, involving the change history of 1,500 projects. Also, we complement this quantitative analysis with a manually defined taxonomy of code-comment inconsistencies fixed by developers.

B. Automatic Assessment of Comments Quality

Researchers have developed tools and metrics to capture the quality of code comments. Khamis *et al.* [16] developed JavadocMiner, an approach to assess the quality of Javadoc comments. JavadocMiner exploits Natural Language Processing (NLP) to evaluate the "quality" of the language used in the comment as well as its consistency with the source code. The quality of the language is assessed using several heuristics (*e.g.,* checking whether the comment uses well-formed sentences including nouns and verbs) combined with readability metrics such as the Gunning Fog Index. The consistency between code and comments is also checked with a heuristic-based approach, *e.g.,* a method having a return type and parameters is expected to have these elements documented in the Javadoc with the `@return` and `@param` tags.

Steidl *et al.* [10] also proposed an approach for the automatic assessment of comments' quality. First, their approach uses machine learning to classify the "type" of code comment (*e.g.,* copyright comment, header comment). Second, a quality model is defined to assess the comments' quality. Also in this case, the model is based on a number of heuristics (*e.g.,* the coherence of the vocabulary used in code and comments). On a similar line of research, Scalabrino *et al.* [9] used the semantic (textual) consistency between source code and comments to assess code readability, conjecturing that the higher this consistency, the higher the readability of the commented code.

Other authors explicitly focused on the automatic detection of code-comment inconsistencies. Seminal in this area are the works by Tan *et al.* [15], [17]. First, they presented iComment [15], a technique using NLP, machine learning, and program analysis to detect code-comment inconsistencies. iComment is able to detect inconsistencies related to the usage of locking mechanisms in code and their description in comments. This technique was evaluated on four systems (Linux, Mozilla, Wine, and Apache) showing its ability to identify inconsistencies confirmed by the original developers.

In a follow-up work, Tan *et al.* [17] also presented @TCOMMENT, an approach able to test the consistency between Javadoc comments related to `null` values and exceptions with the behavior of the related method's body. @TCOMMENT has been experimented on seven open source projects, identifying inconsistencies confirmed by developers.

Similarly, Zhou *et al.* [24] devised an approach detecting inconsistencies related to parameter constraints and exceptions API documentation and code. The approach was able to detect 1,146 defective document directives with a ∼80% precision.

A rule-based approach named Fraco was proposed by Ratol *et al.* [18] to detect code-comment inconsistencies resulting from rename refactoring operations performed on identifiers. Their evaluation shows the superior performance ensured by FRACO as compared to the rename refactoring support implemented in Eclipse.

Liu *et al.* [7] analyzed historical versions of existing projects to train a machine learner able to identify comments that need to be updated. The approach uses 64 features capturing, for example, the *diff* of the implemented changes, to automatically detect outdated comments. The authors report a ∼75% detection precision for their approach.

Finally, a related research thread is the one presenting techniques to detect self-admitted technical debt (SATD) in code comments [5], [6], [25]–[27]. These techniques, while not directly related to the quality of code comments, use these latter to make the development team aware of SATD.

Our work, while not related to the automatic assessment of comments' quality, provides empirical knowledge useful to devise novel approaches for the detection of "problematic" code comments.

III. STUDY DESIGN

The *goal* of the study is to investigate code-comments inconsistencies from a quantitative and a qualitative perspective. The *purpose* is to (i) understand how code and comments co-evolve, to identify coding activities triggering/not-triggering the introduction of code-comment inconsistencies; (ii) define a taxonomy of inconsistencies that developers tend to fix.

The study addresses the following research questions (RQ):

RQ_1: *To what extent do different code change types trigger comment updates?* This RQ studies the code-comments co-evolution in open source projects. We investigate the extent to which different types of fine-grained code changes (*e.g., changes to selection statements*) trigger the update of the related code comments. This analysis provides empirical evidence useful to quantify the cases in which code-comment inconsistencies could possibly be introduced and to identify the types of code changes having a higher chance of introducing these inconsistencies. This evidence can be used, for example, to develop context-aware tools warning developers when code changes are likely to require code comments' updates.

RQ_2: *What types of code-comment inconsistencies are fixed by developers?* This research question aims at identifying the types of code-comment inconsistencies that are fixed by software developers *e.g.,* updating a comment as a consequence of a previously performed refactoring that renamed an identifier. Knowing the types of code-comment inconsistencies fixed by developers can guide the development of tools aimed at automatically detecting them.

TABLE I
DATASET STATISTICS

	Overall	Per Project		
		Mean	Median	Std. Dev.
Java files	1,599,323	1,068	360	2,838
Effective LOC	162,243,714	108,379	31,392	305,704
Stars	2,895,219	1,930	762	3,455
Commits analyzed	3,323,198	2,215	832	5,089

A. Data Collection and Analysis

To answer RQ_1 we mine the fine-grained changes at AST (Abstract Syntax Tree) level performed in commits from the change history of 1,500 open source Java projects hosted on GitHub. Then, we analyze the extent to which different types of code changes trigger updates in the related code comments. The 1,500 projects representing the context of our study have been selected from GitHub in November 2018 using the following constraints:

Programming language. We only consider projects written in Java since, as it will be clear later, Java is the reference language for the infrastructure used in this study.

Change history. Since in RQ_1 we study the co-evolution of code and comments, we only focus on projects having a long change history, composed of at least 500 commits.

Popularity. The number of stars [28] of a repository is a proxy for its popularity on GitHub. Starring a repository allows GitHub users to express their appreciation for the project. Projects with less than ten stars are excluded from the dataset, to avoid the inclusion of likely irrelevant/toy projects.

6,563 projects satisfy these constraints. Then, we manually filtered out repositories that do not represent real software systems (*e.g.,* JAVA-DESIGN-PATTERNS [**?**] and SPRING-PETCLINIC [**?**]), and checked for projects with shared history (*i.e.,* forked projects). When we identified a set of forked projects, we only selected among them the one with the longest commit history (*e.g.,* both FINDBUGS [**?**] and its successor SPOTBUGS [**?**] fall under our search criteria, but we only kept the latter one). Finally, considering the high computational cost of the data extraction process needed for our study (details follow), we decided to only analyze a subset of the remaining projects: We sorted the projects in descending order based on their number of stars (*i.e.,* the most popular on top), and we selected from the list the top 1,500 projects for our study. Table I reports descriptive statistics for size, change history, and popularity of the selected projects. The complete list of considered projects is available in our replication package [29].

We cloned the 1,500 GitHub repositories and extracted the list of commits performed over the change history of each project. To do so, we iterated through the commit history related to all branches of each project with the `git log --topo-order` command. This allowed us to analyze all branches of a project, without intermixing their history and avoiding unwanted effects of merge commits. We then excluded commits unrelated to Java files (*i.e.,* commits that do not impact at least one Java file). For each remaining commit c_i, we use GumTreeDiff [19] with its JavaParser generator to extract AST operations performed on the files modified in c_i.

GumTreeDiff considers the following edit actions performed both on code and comment nodes: (i) *updatedValue* replaces the value of a node in the AST; (ii) *add/insert* inserts a new node in the AST; (iii) *delete*, which deletes a node in the AST; (iv) *move*, which moves an existing node in a different location in the AST. Also, to store more details of the changed AST nodes, such as their parent method and class (needed to know the code component to which a comment AST node belongs to), we extended GumTree with our own reporter. Overall, we extracted 1.3 Billion AST-level changes, resulting in a 476 GB database (excluding indexes) we make publicly available [29].

From our analysis we disregard any file added/deleted in c_i, since our primary goal is to study how changes to different types of code constructs trigger (or not) updates in code comments. In an added/deleted file, all code and comment AST nodes would trivially be added or deleted. Also, we work at method-level granularity, meaning that we only focus on code changes affecting the body or the signature of methods, discarding code changes impacting *e.g.,* a class attribute. This is done since it is easy, from the AST, to identify the comment related to a method (and, thus, to study how changes in the method impact the related comments) while it is not trivial to precisely link a class attribute to its related comments. Finally, we ignore the *move* actions detected by GumTreeDiff because we noticed that they generate a lot of noise in the data, since also deleting a blank line can result in an AST node move.

TABLE II
CATEGORIES OF AST-LEVEL CODE CHANGES

Category	GumTreeDiff Changes
Annotation	MarkerAnnotationExpr, MemberValuePair, NormalAnnotationExpr, SingleMemberAnnotationExpr
Array	ArrayAccessExpr, ArrayCreationExpr, ArrayCreationLevel, ArrayInitializerExpr
Casting	CastExpr, InstanceOfExpr
Constructor	ConstructorDeclaration
Empty Statement	EmptyStmt
Exception Handling	CatchClause, ThrowStmt, TryStmt
Expression	AssignExpr, BinaryExpr, ClassExpr, ConditionalExpr, EnclosedExpr, FieldAccessExpr, SuperExpr, ThisExpr, UnaryExpr
Iteration	BreakStmt, ContinueStmt, DoStmt, ForeachStmt, ForStmt, WhileStmt
Lambda Expression	LambdaExpr, MethodReferenceExpr
Literal	BooleanLiteralExpr, CharLiteralExpr, DoubleLiteralExpr, IntegerLiteralExpr, LongLiteralExpr, NullLiteralExpr, StringLiteralExpr
Method Invocation	ExplicitConstructorInvocationStmt, MethodCallExpr
Method Signature	MethodDeclaration, Parameter
Name	Name, SimpleName
Others	AssertStmt, BlockStmt, InitializerDeclaration, LabeledStmt, ObjectCreationExpr, SynchronizedStmt
Return	ReturnStmt
Selection	IfStmt, SwitchEntryStmt, SwitchStmt
Type	ArrayType, ClassOrInterfaceDeclaration, ClassOrInterfaceType, IntersectionType, LocalClassDeclarationStmt, PrimitiveType, TypeExpr, TypeParameter, UnionType, VoidType, WildcardType
Variable Declaration	VariableDeclarationExpr, VariableDeclarator

Once collected the list of AST operations performed in each commit on the code and comments of modified files, we classified the code changes into categories as shown in Table II. The idea is to group together AST-level operations performed on related code constructs. For example, all operations performed on `if` and `switch` statements are grouped into the *Selection* category. Such a grouping is done for the sake of easing the RQ_1 data analysis. In particular, for each code change category CH_i in Table II, we compute $MCC(CH_i)$ as the percentage of AST changes falling in the CH_i category that triggered a Method Comment Change in comments related to the impacted method. Using the AST, we classify as comments related to the method those present in the method body plus its Javadoc comment (if any). As "Comment Changes" we consider (i) the addition of a comment inside the method or of the Javadoc; (ii) modifications applied to any of the already existing method-related comments; and (iii) deletions of any of the existing method-related comments. Important to highlight is that, to better isolate the *triggering effect* of the CH_i type of change on the method's comments, we only consider CH_i's changes performed in **isolation** on a given method when computing $MCC(CH_i)$. Let us explain this design choice with an example: In a given commit two methods are modified, M_1 and M_2. Both methods are subject to AST changes belonging to the category CH_i, but M_2 is also affected by changes of type CH_j, with $i \neq j$. When computing $MCC(CH_i)$, we consider the changes in M_1, since possible M_1's comments updates are likely to be triggered by the change type CH_i, while this is not true for possible comment updates observed in M_2, since this latter has been subject to different categories of changes.

Since a comment in a method could also have a major impact on the responsibilities implemented by a class, for each CH_i we also compute $CCC(CH_i)$ as the percentage of changes falling in the CH_i category that triggered a Class Comment Change in comments related to the class the impacted method belongs to. In this case, we only focus on the Javadoc comment of the class, since the comments related to the methods it implements are already considered by the MCC metric. Also in this case, we only consider changes performed in isolation for a given change category, as explained for the MCC.

We answer $\mathbf{RQ_1}$ by comparing the distributions of MCC and CCC for the change categories reported in Table II via bar charts, showing the percentage of times that each change category CH_i triggered comment updates. We also use the Fisher's exact test [30] to test whether the chance of triggering method's and class's comments update significantly differ across change categories. To control the impact of multiple pairwise comparisons (*e.g.,* the chance of triggering method's comment changes of the *Array* category is compared against that of 17 other categories), we adjust p-values using the Holm's correction [31]. We use the Odds Ratio (OR) [30] as effect size measure. An OR of 1 indicates that the event under study (*i.e.,* the chance of triggering comment updates) is equally likely in two compared groups (*e.g.,* Array *vs* Casting).

An OR greater than 1 indicates that the event is more likely in the first group, while an OR lower than 1 indicates that the event is more likely in the second group.

Concerning $\mathbf{RQ}_2$, we manually analyzed a set of commits in which the developers fixed code-comment inconsistencies. We extracted, from the same set of 1,500 systems used in RQ_1, all commits having a commit note matching lexical patterns likely indicating the fixing of code-comment inconsistencies. To define these lexical patterns, the first author experimented with different combinations of words and inspected the resulting commits (details in [29]). He found the following pattern to be the best suited for the identification of the commits of interest: (*update** or *outdate**) and *comment(s)*. In other words, all commit notes containing the word *update* or *outdate* in different derivations (*e.g.,* updates, updated) **and** the word *comment* or *comments* have been selected, for a total of 3,641 commits matched. From this set, we randomly selected for the manual analysis a sample of 500 commits, representing a 99% statistically significant sample with a 5% confidence interval.

The 500 commits were randomly distributed among three authors, making sure that each commit was classified by two authors. The goal of the process was to identify the exact reason behind the changes performed in the commit. If the commit was unrelated to code comments, the evaluator classified it as *false positive*. Otherwise, a tag explaining the reason for the change (*e.g., update comment to correct a wrong method's parameter description*) was assigned, even in the case the commit was not related to a code-comment inconsistency, but just to changes in a comment (*e.g., fixed a typo in a comment*). We did not limit our analysis to the reading of the commit message, but we analyzed the source code diff of the changes implemented in the GitHub commit. The tagging process was supported by a Web application that we developed to classify the commit and to solve conflicts between the authors. Each author independently tagged the commits assigned to him by defining a tag describing the reason behind the commit. Every time the authors had to tag a commit, the Web application also showed the list of tags created so far, allowing the tagger to select one of the already defined tags. Although, in principle, this is against the notion of open coding, in a context like the one encountered in this work, where the number of possible tags (*i.e.,* cause behind the commit) is extremely high, such a choice helps using consistent naming and does not introduce a substantial bias. In cases for which there was no agreement between the two evaluators (51% of the classified commits), the document was assigned to an additional evaluator to solve the conflict.

After having manually tagged all commits, we defined a taxonomy of code comment-related changes through an open discussion involving all the authors (see Fig. 3). We qualitatively answer RQ_2 by discussing specific categories of commits likely related to the fixing of code-comment inconsistencies. For each category, we present interesting examples and common solutions, and discuss implications for researchers and practitioners.

B. Replication Package

The data used in our study is publicly available [29]. We provide (i) the list of 1,500 subject projects, (ii) the database containing the fine-grained changes as extracted by GumTreeDiff for the 3,323,198 mined commits, and (iii) the link to the 500 commits we manually analyzed.

IV. Results

A. To what extent do different code change types trigger comment updates?

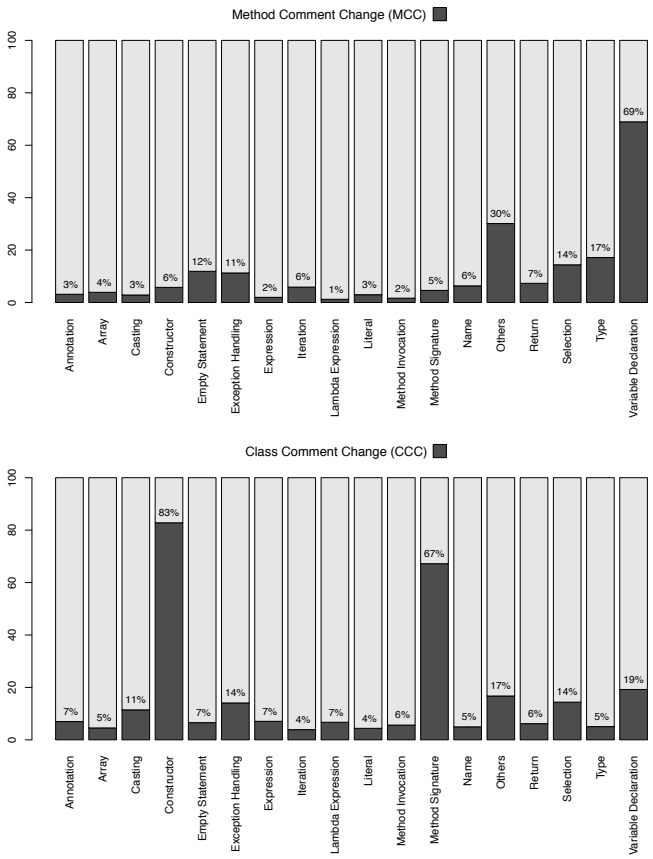

Fig. 1. RQ_1: MCC and CCC by change category (Table II)

Fig. 1 compares the MCC (top) and the CCC (bottom) for the categories of AST-level changes described in Table II. It is worth remembering that the MCC and the CCC values for a change category CH_i represent the percentage of times that a change of type CH_i triggered a change in a related method (MCC) or class (CCC) comment. Fig. 2 summarizes the results of the statistical comparison between the chance of triggering method (left) and class (right) comment changes for different categories of change categories in the form of a heatmap: A white block indicates that the difference between two categories is not statistically significant (adjusted p-value ≥ 0.05) or that the odds ratio between the two categories indicate a similar chance of triggering changes in code comments ($0.8 \leq d \leq 1.25$).

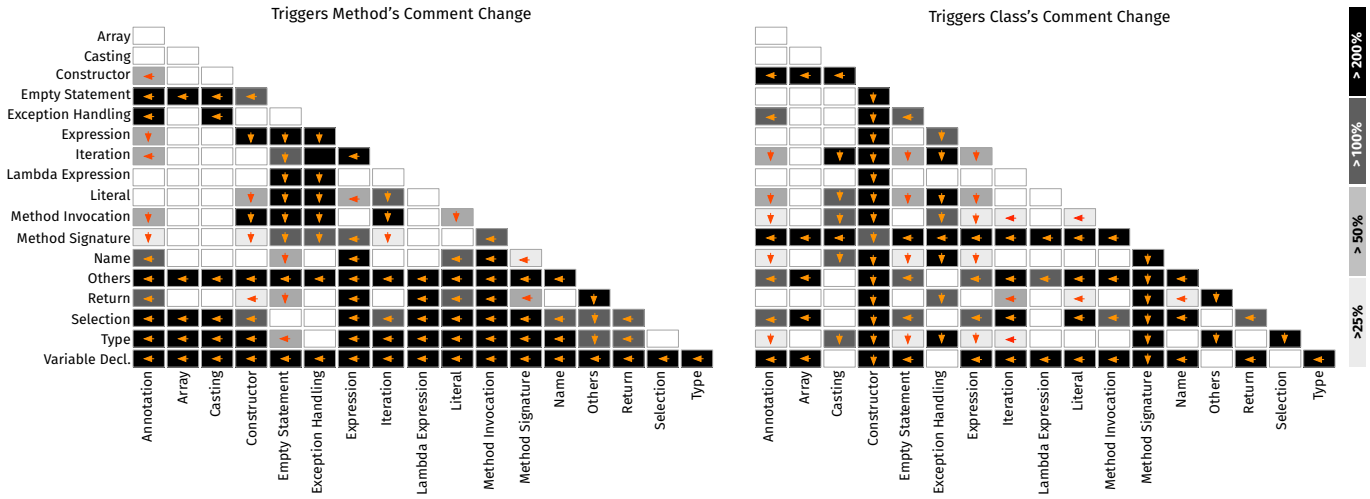

Fig. 2. RQ_1: Statistical comparison of the chance of triggering method (left) and class (right) comment update by change category (Table II)

Blocks with four different grayscale values from light to dark represent a significant difference between two change categories CH_i and CH_j accompanied by an odds ratio indicating that CH_i's changes have at least 25%, 50%, 100%, or 200% higher chance of triggering method or class comment changes than CH_j (or *vice versa*). The arrows in the heatmap point to the change category having the highest chance of triggering comment changes among the compared two. For example, when comparing the categories *Type* and *Constructor*, Fig. 2-left shows that *Type*'s changes have a higher chance of triggering updates in the related comments (at least 200% higher — black square). The detailed results with adjusted p-values and odds ratio are available in our online appendix for all comparisons [29].

From the analysis of Fig. 1 and 2 it is clear that different categories of code changes have a different likelihood of triggering updates in the related method and class comments. Also, the MCC and the CCC values show that changes to method and class comments are triggered by different categories of code changes. For example, changes impacting the *Constructor* are much more likely to trigger updates in the class comment ($CCC = 0.83$) as compared to the method comment ($MCC = 0.06$). This is expected since changes to the constructor can impact the way the whole class is instantiated and, as a consequence, are likely to require updates to the class's comment description. A similar trend can be seen for changes impacting the *Method Signature* ($CCC = 0.68$ *vs* $MCC = 0.04$), while the opposite is true for *Variable declaration*-related changes, *i.e.,* these changes trigger more frequently updates in the related method comments ($MCC = 0.69$) than in the class comment ($CCC = 0.19$). This result is reasonable, considering that we only take into account code changes affecting the methods' body, as we previously explained. Thus, changes to a variable declared inside a method are likely to only impact the logic of that method, without necessarily involving the overall class functioning.

One general trend that can be observed from Fig. 1 is that most of the code change categories rarely trigger changes in the related method and class comments. Our results point in the same direction of the findings by Fluri *et al.* [12]: They found that 23%, 52%, and 43% of all comment changes in ArgoUML, Azureus, and JDT Core respectively, were triggered by source code changes. Working on a much larger corpus of 1,500 systems, when considering all change types together we observe a co-evolution of code and comments happening in 7% of cases for method's comments and 13% of cases for class's comments. This means that, according to our data, 13% to 20% of code changes trigger a comment change in the class and/or in the methods' comments: 13% in case there is complete overlap between the two sets of changes (*i.e.,* those triggering methods' and those triggering class's comments changes), 20% in case they are completely disjointed.

Categories exhibiting low values of both MCC and CCC and showing statistically significant lower chance to trigger comment updates when compared to most of the other categories are *Array*, *Lambda Expression*, *Iteration*, *Literal*, *Method Invocation*, and *Name*. Due to the lack of space, we only discuss two exemplary cases from these categories, while more qualitative analysis will be reported in RQ_2.

The *Name* category includes changes performed on identifiers (*e.g.,* rename refactoring). We found cases of code-comment inconsistencies introduced as result of renamed identifiers. For example, in a commit performed in the `alluxio` project [**?**], the developer implements a rename refactoring on the `mIn` identifier, changing it to `mStream`. This change affects several methods implemented in the class, but only one of them, namely `openStream()`, mentions the renamed identifier in its comment. In this commit, the developer forgets to update the comment, thus referring in it to an identifier that does not exist anymore in the code. The issue is fixed 20 days later [**?**].

The second example of inconsistency refers to the

Literal category, grouping changes related to fixed values in code (*e.g.,* String literal). A commit from the `bitcoinj` project [**?**] changed the value of a String literal from `"connectionTimeoutMillis"` to `"connectTimeoutMillis"`. This literal was used as a parameter value in a call to the `setOption` method. As explained in the commit note, this change was needed to fix a bug: *"Fix typo that prevented connection timeouts from being set properly"*. Indeed, the parameter value `"connectionTimeoutMillis"` was not a valid one. While the commit fixed the problem in the code, it did not fix an example reported in one of the comments of the class including an invocation to the `setOption` method, still using the old, wrong parameter value. The problem has been fixed in a later commit [**?**].

We answer RQ_1 with the following observations:

We confirm previous findings in the literature [12], showing that between 13% and 20% of code changes trigger comment updates. This does not imply that in the remaining ∼80% of cases code-comment inconsistencies are introduced, but they represent a possibility, as we observed through manual inspection, and as further demonstrated by the qualitative analysis we present in RQ_2.

Code changes to the Array, Lambda Expression, Iteration, Literal, Method Invocation, and Name categories are the ones less frequently triggering comment updates. This is also confirmed by the statistical analysis (Fig. 2), in which these categories exhibit, as compared to other categories, statistically significant lower chance of triggering comment updates, accompanied by at least a "small" and in most cases by a "large" effect size.

Change categories Variable Declaration and Selection are among those more likely to trigger comment updates, both at the method and at the class level. This is possibly due to the fact that these changes could severely impact the application logic (*Selection*) or the data manipulated in the code *Variable Declaration*. Also, changes in the *Method Signature* and *Constructor* categories are often accompanied by changes to the class's comment.

*The other change categories (*e.g., *Return, Annotation, etc.) exhibit MCC and CCC values mostly in the range 0.1-0.2*, showing that they still represent possible scenarios for the introduction of code-comment inconsistencies.

B. What types of code-comment inconsistencies are fixed by developers?

We addressed RQ_2 by labeling 500 commits identified as candidates to fix code-comment inconsistencies (see Section III). We identified 138 false positives and 362 commits actually related to comment changes. Note that, while not all these commits are strictly related to code-comment inconsistencies, they are all related to improvement actions performed on comments. Overall, we identified 69 types of comment changes tackled by developers, 25 of which relevant for code-comment inconsistencies.

Fig. 3 presents the results in the form of a hierarchical taxonomy composed by six root categories: Application Logic, Code Design/Quality, Maintenance, Formatting/Readability, Copyright/License and Others. The more specific types of comment-related changes are represented either as intermediate nodes or leaves, and changes relevant for the fixing of code-comment inconsistencies are marked with a ❗ sign.

For each root category, we next describe representative examples and, at the end of this section, we discuss implications for researchers and/or practitioners derived from our findings.

1) Application Logic (136): This category groups comment changes in which the impacted comments are related to the implemented application logic, such as a Javadoc describing the steps of an algorithm implemented in a method, its parameters or return type. In most cases, the change occurred in the form of a comment update (113), while in a few cases (12) a new comment was added. We observed three main reasons why developers update comments: (i) the comment wrongly describes the application logic (35), due to an error done when the comment was written in the first place or to an inconsistency introduced later (in these cases we were not able to trace back to the specific cause of the problem); (ii) the comment needs to be updated as a consequence of a new implementation logic (25); (iii) the comment is improved to explain the implementation in more details (53).

For instance, in a commit of `QRCodegenerator` [**?**] an inline comment describing how an array element is calculated was updated to fix a copy/paste mistake done when the code was firstly written. The comment was copied from another line of code also calculating an array, but in a different way. This commit fixed the inconsistency between the code implementation and the comment description.

In `WordPressforAndroid` [**?**], the previously misleading comment of the `getPath()` method was replaced from "*descendants must implement this to send their specific request to the stats api*" to "*descendants must implement this to return their specific path to the stats rest api*". Similarly to the example discussed in RQ_1, also in RQ_2 we found cases in which the comment was fixed to update a code usage example reported in the comment and not aligned with the actual code implementation (see [**?**]).

Comments can also be used to explain the rationale for implementation choices (*e.g.,* to justify the usage of a specific collection type to represent data). We found cases in which after a code change, these comments became outdated, pushing developers to fix the discrepancy by simply deleting the comments (see [**?**]). In other commits, comments documenting the rationale were added, as in the case of the `ApacheCassandra` project [**?**], in which a comment was added in 2017 to explain why a variable introduced in 2015 was named `nulls`.

2) Code Design/Quality (80): This category groups comment improvements that originate as consequence of actions related to code quality and design (*e.g.,* refactoring activities).

We observed three cases in which changes to the class hierarchy resulted in inconsistent comments.

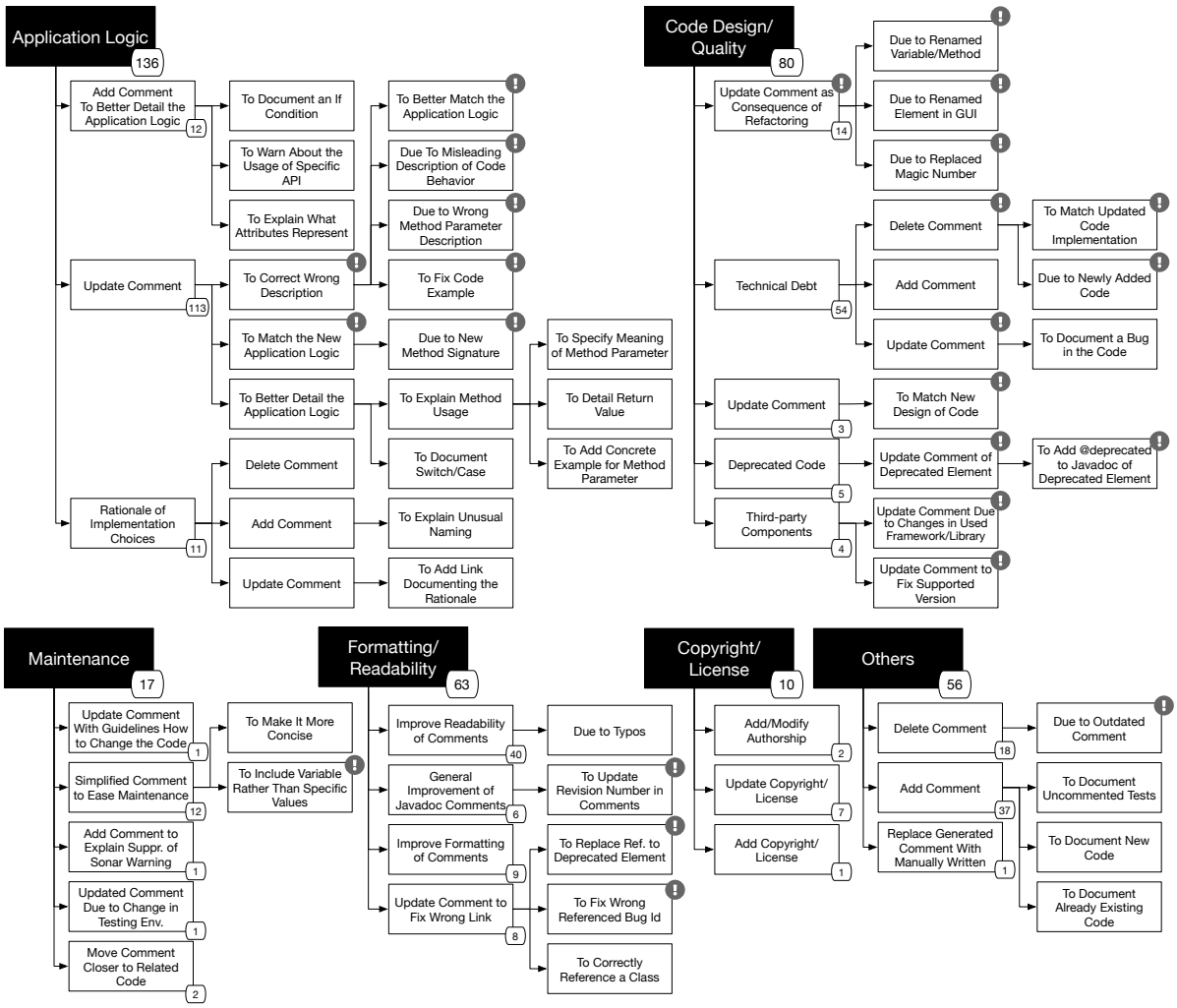

Fig. 3. RQ_2: Taxonomy of Code Comment-Related Changes

One of these is from the `ApacheCordovaAndroid` project [**?**], in which we found a commit accompanied by the note: "*Update documentation comments to match implementation*". In 2012, the developers refactored the class hierarchy and converted the abstract class `CordovaInterface` to an interface [**?**]. However, its Javadoc comment has only been updated one year later, in 2013 [**?**].

Most of the cases in this group are related to *Technical Debt* comments (54), *i.e.,* comments describing known issues or 'TODOs' in the code. Such comments are often deleted (26) as a consequence of the developers *paying back* the debt. While the comment is usually removed in the same commit in which the technical debt is paid back, we found cases in which developers fixed the technical debt issue but left the comment by mistake. This required a subsequent commit aimed at removing the comment, *e.g.,* "*Issue #326: Remove forgotten outdated comment*" [**?**], from the `JavaParser` project.

In 19 commits the 'TODO comments' were updated due to a change in the code, for example to keep track of progress done in the fixing of the documented technical debt.

Related to technical debt, there were also cases in which comments were added to document the fact that a class/method was deprecated (see [**?**], [**?**]).

Updated comments following refactorings were also frequent (14), particularly after renamed methods/variables (11), but we found interesting examples also following renamed GUI elements (2), *e.g.,* [**?**], or a replace magic number refactoring (1), *e.g.,* [**?**]. In the latter case the developer replaced the inline comment "*Keep only 1000 batches worth data in queue*" with "*Keep only numBatchesInQueuePerPartition batches worth data in queue*", to match a replace magic number refactoring performed in a previous commit, thus fixing the code-comment inconsistency.

Finally, related to *Code Design/Quality* are comment changes aimed at fixing inconsistencies originated outside of the project scope in third-party libraries. An example we found is from the `PSIProbe` project [**?**]: "*Update comment about support as TomEE now supports tomcat 8.5*". Here the code implementation already provided support for a new Tomcat version that, however, was not officially released yet.

This was documented in the code through a code comment, that became obsolete once Tomcat 8.5 was released, pushing developers to delete it.

3) Maintenance (17): In this category fall comment changes aimed to ease the future maintenance of comments, for example by making them more concise. Interesting are the changes implemented in `ApacheGroovy` to fix an outdated comment in such a way to also avoid uptodateness issues in the future: They modified the comment in order to use a newly introduced variable ("*The parameter can take one of the values in* `@link ALLOWED_JDKS`") rather than listing the complete list of supported JDK versions ("*[...] can take one of the values 1.7, 1.6, 1.5, 1.4*") [**?**] [**?**]. This makes unnecessary in the future to update these comments when new versions are supported or old versions are not supported anymore.

Another example of comment change aimed at avoiding future uptodateness issues is the commit "*remove comment that can be very easily outdated*" from `JetBrainsAndroid` [**?**]. Here the developer extracts from the Javadoc comment of the `IntellijCodeNavigator` class three paragraphs detailing the logic of its main method (`getNavigatable`), in particular related to branches of an `if` statement it implements. Each of the extracted paragraphs has then been moved closer to the specific lines of code it documents, to allow for an easier maintainability and to avoid that future changes to the code would not be reflected in the comment.

4) Formatting/Readability (63): Changes intended to improve the formatting or readability of comments are grouped in this category. Not surprisingly, we found many commits in which developers just improved the wording of the comments (31) or fixed typos (9). We also grouped in this category comments aimed at implementing general improvements in the Javadoc (6), with a mix of changes aimed at fixing typos, improving readability, formatting, etc. (*e.g.,* "*Fix comments to update javadoc for a bunch of methods*" from `Aluxio` [**?**]).

Although this type of changes is usually not related to code-comment inconsistencies, we found cases in which references (*e.g.,* related to other code elements, bug reports) became obsolete, resulting in invalid/outdated references in comments. For example, in `GoogleGuava` a commit says: "*Updated a comment in ListenerCallQueue to point at SequentialExecutor instead of the deprecated SerializingExecutor wrapper interface*" [**?**].

5) Copyright/License (10): We grouped fixes related to copyright/license comments separately under this category as we found a considerable amount of commits working on updating licensing information.

These changes were mostly related to simple maintenance of copyright headers in source files, *i.e.,* updating authorship [**?**] or copyright year [**?**].

We also spotted cases, however, where outdated copyright comments remained in source files for several years. In 2011 a developer of the `ApacheGroovy` project updated the copyright header of a Java file from a BSD variant to Apache License v2 [**?**], although the project had already changed its license back in 2007 [**?**].

6) Others (56): This category groups comment changes that, while not being false positives (*i.e.,* they are related to code comments), do not fit any of the previous categories. Comments were added in 37 cases to document new or already existing code. In one case, automatically generated comment skeletons were replaced with manually written comments [**?**], while the comment deletions were generally related to outdated comments left in the code by mistake. An example can be seen in a commit of the `CrateDB` project [**?**] where the developer deletes the description of the error handling of SQL operations that was rewritten in earlier commits.

C. Discussion and Implications

Our large-scale study in RQ_1 confirmed previous findings reported in the literature and showing that, in most of the cases, code and comments do not co-evolve. It is important to highlight that a code change does not always result in the need for updating the corresponding comment. Thus, we are not claiming that do not updating comments as a consequence of code changes is a bad practice. However, our qualitative analysis disclosed several cases in which developers introduced (RQ_1) or fixed previously introduced (RQ_2) code-comment inconsistencies, providing us with a number of lessons learned. In the following we discuss implications for researchers (indicated with the ⚗ icon) and/or practitioners (🐿 icon) derived from our findings.

The maintainability of comments is as important as that of source code 🐿. As it happens for code, comments should also deal with "functional" and "non-functional" requirements. The functional aspect here is the proper documentation of the source code, and it has always been recognized as a fundamental support for code comprehension. Not less important are, however, the non-functional aspects of code comments, such as their readability and *maintainability*. As shown in our study, a simple idea such as using a variable referenced in the comment to document the supported JDK versions as opposed to explicitly listing them [**?**] can dramatically simplify the maintenance of the comment, that will not require any future update in case of changes to the supported JDKs. Basically, as source code is often designed to isolate and minimize future changes, the same should happen for comments.

Refactoring code comments ⚗. From the researchers' perspective, our study stresses the importance of investing effort in the development of tools to support code comments refactoring. Indeed, most of the effort in this field has been devoted to the automatic assessment of comment quality (see *e.g.,* [7], [10], [16], [17]) without, however, recommending how to automatically refactor it. Our findings provide insights for the future development of approaches able to both detect and fix issues in code comments. For example, we have seen as simple copy/paste can introduce code-comment inconsistencies, due to a wrong reuse of comments across semantically different instructions (*i.e.,* the same comment is reused for two different instructions, wrongly documenting one of the two) [**?**]. Identifying different code components documented with the same comment can help in identifying these problems.

Concerning the automatic comment refactoring, a first step in this direction could be the definition of a catalogue of operations for comments, similarly to what has been done for source code [32]. For example, we observed an instance of what can be defined as an "*extract comment refactoring*" [**?**], aimed at splitting a large comment into several comments to place each one closer to the exact instruction it documents.

Code comments are first-class citizens in code refactoring ⚗ ⚲. We observed several code-comment inconsistencies introduced as consequence of refactoring activities. For example, the application of a replace magic number refactoring [**?**] caused the magic number to be removed from the code but not from the related comments. Similarly, major refactorings to the class hierarchy [**?**] caused outdated references in comments. This highlights: (i) the need for developers to consider the effects on comments when applying refactoring operations; (ii) the opportunity for researchers to investigate how to integrate better comment support into refactoring tools.

Code-comment traceability is still an open problem ⚗. Related to the previously discussed points, one major research challenge is the code-comment traceability (*i.e.,* automatically identifying the code instructions documented by a given comment). In 1988, Kaelbling argued that programming languages should not have comment statements [33], but *scoped comments*, explicitly indicating the code elements they refer to. As of today, documentation tools such as Javadoc help developers to explicitly comment certain elements by referring to them. IDEs also provide support, *e.g.,* by showing related comments of selected items. However, popular programming languages are still bound to line and block comments. There are many research opportunities here both for language designers and researchers to facilitate the code-comment traceability. Solving this problem will in turn help to make substantial steps ahead in the identification/fixing of code-comment inconsistencies.

V. Threats to Validity

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is the most important kind of threat for our study, and is related to:

RQ_1: Computation of the MCC and CCC metrics. As explained in Section III these metrics, for a specific type of code change category CH_i, measure the percentage of times that method (MCC) or class (CCC) comments are inserted/deleted/modified in response to CH_i's changes. Clearly, if a modified method/class has no comments, these metrics cannot capture the deletion or modification of the method's comments, but only the insertion of new comments. Considering the scale of our study and the focus on long-lived and popular systems unlikely to have many undocumented methods, these imprecisions should not have a major impact on the outcome of our study.

RQ_1: Imprecision introduced by GumTreeDiff. As any differencing tool, GumTreeDiff could generate wrong information. For example, we noticed that in some cases the update of a variable type (*e.g.,* from `double` to `int`) was reported as the deletion of a variable (the `double` one) followed by the addition of a new variable (the `int` one). However, GumTree is a state-of-the-art differencing tool and at least we tried to reduce possible noise caused by "move" operations.

RQ_2: Subjectivity in the manual classification. We identified through manual analysis the reasons behind commits performed by developers to (likely) fix code-comment inconsistencies. To mitigate subjectivity bias in such a process, every commit was assigned to two authors who manually analyzed it independently. Then, in the case of a disagreement, a third author was assigned to the commit to solve the conflict.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. One aspect could be related to the selection of projects being considered. As explained by Kalliamvakou *et al.* [34] mining GitHub can be risky because projects may contain very few commits. To mitigate this threat, we applied strict criteria (*i.e.,* more than 500 commits, more than 10 stars) when selecting the context of our study. To reinforce the internal validity, when possible, we integrated the quantitative analysis with a qualitative one.

Threats to *external validity* concern the generalizability of our findings. RQ_1 tries to achieve a high generalizability in terms of mined projects that, however, are all written in Java. Future work should focus on systems written in different languages to confirm or contradict our findings. RQ_2 analysis is limited to a specific set of 500 commits we randomly selected as output of a keyword-based mechanism used for the pre-selection of commits likely related to code-comment inconsistencies. Because of this procedure, our taxonomy surely omits types of code-comment inconsistencies fixed in commits we did not analyze and/or documented in diverse data sources (*e.g.,* issues on GitHub).

VI. Conclusion

We presented the largest study at date about the co-evolution of code and comments. The study involved the analysis of the complete change history of 1,500 Java systems. Then, we manually analyzed 500 commits likely related to the improvement of code comments, classifying 362 of them (the non-false positives) into a taxonomy of comment-related changes (Fig. 3). The results achieved with our quantitative and qualitative analyses have been used to distill lessons learned resulting in actionable items for both researchers and practitioners (Section IV-C).

Our future work will target two directions. First, we plan to enlarge the set of commits manually analyzed for RQ_2 to test the generalizability of the defined taxonomy. Second, we will work on the development and experimentation of approaches able to support code comment refactoring, with the goal of improving the comments' maintainability.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects PROBE (SNF Project No. 172799) and CCQR (SNF Project No. 175513).

REFERENCES

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2018.

[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting &Amp; Designing for Pervasive Information*, ser. SIGDOC '05, 2005, pp. 68–75.

[3] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81, 1981, pp. 215–223.

[4] T. Tenny, "Program readability: Procedures versus comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, Sep. 1988.

[5] G. Sridhara, "Automatically detecting the up-to-date status of ToDo comments in Java programs," in *Proc. of the 9th India Software Engineering Conference*, ser. ISEC '16. ACM, 2016, pp. 16–25.

[6] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying self-admitted technical debt in open source projects using text mining," *Empirical Software Engineering*, vol. 23, no. 1, pp. 418–451, Feb 2018.

[7] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou, "Automatic detection of outdated comments during code changes," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 01, July 2018, pp. 154–163.

[8] J. H. Hayes and L. Zhao, "Maintainability prediction: A regression analysis of measures of evolving systems," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05, 2005, pp. 601–604.

[9] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *Proc. of the 24th IEEE International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.

[10] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proc. of the 21st IEEE International Conference on Program Comprehension (ICPC)*, May 2013, pp. 83–92.

[11] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in postgresql," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. ACM, 2006, pp. 179–180.

[12] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proc. of the 14th Working Conference on Reverse Engineering (WCRE 2007)*, Oct 2007, pp. 70–79.

[13] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, Dec 2009.

[14] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293 – 2304, 2012.

[15] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*iComment: Bugs or bad comments?*/," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 145–158, Oct. 2007.

[16] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: The JavadocMiner," in *Natural Language Processing and Information Systems*, C. J. Hopfe, Y. Rezgui, E. Métais, A. Preece, and H. Li, Eds. Springer, 2010, pp. 68–79.

[17] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *Proc. of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 260–269.

[18] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 112–122.

[19] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014, pp. 313–324.

[20] A. T. T. Ying, J. L. Wright, and S. Abrams, "Source code that talks: An exploration of eclipse task comments and their implication to repository mining," in *Proc. of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. ACM, 2005, pp. 1–5.

[21] P. W. McBurney and C. McMillan, "An empirical study of the textual similarity between source code and source code summaries," *Empirical Software Engineering*, vol. 21, no. 1, pp. 17–42, Feb 2016.

[22] L. Pascarella and A. Bacchelli, "Classifying code comments in Java open-source software systems," in *Proc. of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 227–237.

[23] O. Arafat and D. Riehle, "The commenting practice of open source," in *Proc. of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. ACM, 2009, pp. 857–864.

[24] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs documentation and code to detect directive defects," in *Proc. of the 39th IEEE International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, pp. 27–37.

[25] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "SATD Detector: A text-mining-based self-admitted technical debt detection tool," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 9–12.

[26] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *Proc. of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 179–188.

[27] E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Proc. of the 7th International Workshop on Managing Technical Debt (MTD)*, Oct 2015, pp. 9–15.

[28] "About stars (GitHub). https://help.github.com/articles/about-stars/."

[29] "Replication package. https://github.com/USI-INF-Software/ICPC2019-code-comment-inconsistencies."

[30] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[31] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[32] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[33] M. J. Kaelbling, "Programming languages should not have comment statements," *SIGPLAN Not.*, vol. 23, no. 10, pp. 59–60, Oct. 1988.

[34] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proc. of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 92–101.