

On Indirectly Dependent Documentation in the Context of Code Evolution: A Study

Devika Sondhi, Avyakt Gupta, Salil Purandare, Ankit Rana, Deepanshu Kaushal and Rahul Purandare
IIIT-Delhi

New Delhi, India

{devikas, avyakt17285, salil17093, ankit18381, deepanshu18388, purandare}@iiitd.ac.in

Abstract—A software system evolves over time due to factors such as bug-fixes, enhancements, optimizations and deprecation. As entities interact in a software repository, the alterations made at one point may require the changes to be reflected at various other points to maintain consistency. However, often less attention is given to making appropriate changes to the documentation associated with the functions. Inconsistent documentation is undesirable, since documentation serves as a useful source of information about the functionality. This paper presents a study on the prevalence of function documentations that are indirectly or implicitly dependent on entities other than the associated function. We observe a substantial presence of such documentations, with 62% of the studied Javadoc comments being dependent on other entities, as studied in 11 open-source repositories implemented in Java. We comprehensively analyze the nature of documentation updates made in 1288 commit logs and study patterns to reason about the cause of dependency in the documentation. Our findings from the observed patterns may be applied to suggest documentations that should be updated on making a change in the repository.

Index Terms—code evolution, GitHub repositories, documentation, commits

I. INTRODUCTION

A software system undergoes evolution over time. Changes to software in this evolution process may be attributed to factors such as enhancements, bug fixes, refactoring, and deprecation of entities. Modifications made at one point in the source code often require alterations at various other points to ensure the expected behavior. While developers use test-suites to verify that the changes to the code do not break any functionality, in many cases, less attention is given to making appropriate changes to the documentation associated with the functions. Such inconsistencies are undesirable for developers as well as users. A developer needs to obtain knowledge about code components for performing enhancements and other code-maintenance tasks. Documentation serves as a prominent source for acquiring this knowledge [1], [2]. Additionally, from a user’s perspective, reading the documentation is preferable to deducing functionality from source code.

There have been research efforts to highlight inconsistencies between different code fragments or between code and its documentation (or comment) [3]–[6]. Past work has focused on inconsistencies with function documentation when the directly associated function is modified. However, the documentation may be dependent on several other entities. Consider two snippets from a commit made in the Spring-Framework



Fig. 1: Snippets from a commit log showing a code change that triggers a documentation update in another file.

project, as shown in Fig. 1. The commit is sourced from an issue, SPR-16130 [7], which discusses altering the default access control to make a client request to a server. The field `allowCredentials` is by default set to enabled (or `true`), allowing any site to make an XHR request with credentials. The developers realize that this is not a secure configuration and resolve the issue by setting `allowCredentials` to disabled (or `false`) by default. Fig.1 demonstrates how this code change in one file induces a critical documentation update for the `allowCredentials` method in another file.

As a project evolves, there may be several changes such as the one discussed in this example. A developer may neglect to update the documentation, especially when it is not directly linked to the entity to which the changes have been made.

In this paper, we study method documentation and commits logs of 11 open-source projects to observe trends on the prevalence of function documentations that are indirectly or

implicitly dependent on entities other than the function with which they are associated. We find a substantial presence of such documentations. We also observe that developers are likely to leave the documentation inconsistent for a long period of time in the process of making updates to the project. Motivated by the extent and relevance of the problem of inconsistent documentation caused by indirect dependencies, we build a taxonomy of the types of documentation updates that indicate dependencies on other changes. We observe that the developers primarily enhance or fix the core description of the functionality. We present patterns that indicate potential relations between the source of an update, such as a code-change, and the affected documentation, in order to infer the cause of a dependency. We further discuss the implications of these patterns for developers from the perspective of building a warning system which indicates potential inconsistencies introduced on making updates.

To summarize, this work makes the following contributions:

- A study on the prevalence and the nature of indirect dependencies in function documentations.
- A comprehensive analysis of the nature of documentation updates over 1288 commit logs extracted from 11 open-source repositories. The analysis covers the role of developers in maintaining consistent documentation and builds a taxonomy of the nature of documentation updates.
- Patterns that induce indirect dependencies in documentations. We discuss the implications of these patterns in building applications to reduce documentation inconsistencies in the context of code evolution.
- A publicly available annotated dataset of commit logs and methods with the above-mentioned analysis, to ease possible future extensions.

II. RELATED WORK

Code-comment inconsistency. Documentation-writing has traditionally been done manually. Recent research has given due attention to automated documentation generation [8], [9]. However, the aspects of their quality and maintenance are equally essential [2], [10]–[14]. Aghajani et al. study documentation-related issues reported on different platforms [10]. They build a taxonomy of such issues to suggest actionable points. Our study complements these findings.

Focusing on the correctness of the documentation, code-comment inconsistencies have been studied in the past and tools have been developed to detect such inconsistencies. Wen et al. study the change-history of GitHub projects to analyze how code and comment co-evolve [4]. They further build a taxonomy of comment-related changes observed on the analyzed commits. @tCOMMENT is a tool for testing Javadoc comments that highlights inconsistencies related to method properties specifically about null values and related exceptions [15]. iComment is another tool that detects bad comments at a function level [6]. The tool uses a template-based approach by combining NLP, machine learning, and program analysis to highlight gaps in the context of locking protocols. All these studies and techniques are scoped to method-level granularity

to analyze the inconsistencies between the method’s code and the directly associated comments. We take a step further to analyze how code-level or external changes could impact Javadoc comments at various other points.

Several refactoring tools have been proposed to minimize code-comment inconsistencies arising from refactoring [5], [16], [17]. Eclipse’s Java development tools support automated comment refactoring [17]. While Eclipse’s refactoring functionality focuses on lexical matches for renaming identifiers, Ratol et al. propose a lexical and semantic rules-based algorithm to detect fragile comments in the context of performing refactoring-based changes to identifiers in the code [5].

Risking changes in code evolution. Due to software delivery pressures, developers may be rushed into completing certain tasks, compromising the overall software quality. This phenomenon is referred to as the technical debt [18]. Changes occurring in the process of code evolution involve several risks [19], [20]. Shihab et al. study various factors and their effectiveness in detecting risky changes [21]. One of their findings suggests that developers are unreliable in identifying the risks when changes are made that require related changes. As risky changes incur debt in terms of more reviewing and testing, it becomes essential to identify these risks.

Technical debt may also be intentionally introduced by developers, referred to as self-admitted technical debt (SATD) [22], [23]. This may involve introducing workarounds or sub-optimal code to satisfy the basic requirement, or adding notes for future releases. A study on code-comments suggests that SATD may persist due to the code-comment inconsistent changes after releases [22]. Hence, efforts to warn developers about introducing SATD become essential. Our study is an effort in this direction, by highlighting patterns that can potentially aid in identifying the targets where the changes are needed when a source is updated.

Information Foraging and Program Navigation. Past works discuss programmer-navigation models on code, as learnt from the programmer’s behavior to recommend navigation actions [20], [24]. Lawrance et al. propose a model that uses a source code’s topology and its ‘scent’ to predict that the programmer will visit a source code given a bug report [24]. Relying on the programmer’s behavior can make the approach subjective, as is stated in one of these papers. Instead, our work focuses on documentation updates and primarily contributes a taxonomy of the nature of documentation-updates made in the code evolution process. The application derived from our work is a recommendation system for which we suggest objective patterns to recommend points of documentation changes.

Change patterns in the context of code evolution. Past studies obtained code evolution patterns to harness for applications such as patch generation [25], [26], language migration [27], code recommendation [28], change guidance [3], [29], [30], and code completion [31]. Nguyen et al. propose a technique to detect semantic code change patterns using their tool, CPATMINER [32]. CPATMINER builds a change-graph from the original and the altered code functions, each represented as a program dependency graph (PDG).

Listing 1: Sample of a Javadoc comment for a method.

```
/**
 * Replaces the existing container under test with
 * a new container.
 * This is useful when a single test method needs
 * to create multiple containers while retaining
 * the ability to use {@link #expectContents()}
 * expectContents()} and other related methods.
 *
 * @param newValue the new container instance
 * @return the new container instance
 * @throws IllegalArgumentException if the input
 * is not a valid encoded container
 * @since 1.0
 */
public Container resetContainer(Container newValue)
{    //..body }
```

The change-graph connects the two PDGs by drawing edges between the unchanged nodes before and after the change. This change-graph is analyzed to mine change-patterns. To infer repetitiveness in change-patterns, CPATMINER checks if the graphs are isomorphic. Mehta et al. propose a correlated change analysis technique that applies machine learning on file commits to mine change-rules based on files that frequently change together [30]. These rules are used to suggest files that should be altered once a change is made in another correlated file. Their tool supports changes at file-level granularity for configuration files. While these techniques focus on learning the changes made to the code and configurations, our study focuses on code evolution from the perspective of inferring the potential causes and patterns that result in inconsistent documentations due to changes made in other related entities.

III. EXPERIMENT DESIGN

The objective of this study is to analyze the nature of inconsistencies in documentations, as well as their causes, in the context of code evolution. We analyze method documentation and commit logs in open-source projects on GitHub that are implemented in Java. Java has been ranked as one of the five most popular languages on GitHub over the last five years [33]. We scope our study to analyzing documentation updates associated with methods, as method headers, in the code file. Listing 1 shows a Javadoc comment for a method, which is the typical documentation format for code files implemented in Java.

A. Research Questions

One expects a documentation to describe the entity with which it is directly associated. However, dependency of documentation on other entities would require maintaining the consistency of the documentation with other changes within or external to the project. We designed the first research question to study the prevalence of dependencies between a method's documentation and indirectly associated source code and other entities. These other entities include documentations associated with other methods, classes, interfaces, fields, or external sources, such as URL references to web pages. This brings us to the first RQ.

TABLE I: Dataset analyzed in the study.

Repository	Star-Count*	# Commit logs extracted	# Commit logs analyzed
elasticsearch	50.2k	12742	344
spring-boot	49.5k	6507	128
RxJava	43.2k	376	64
spring-framework	38.5k	3854	510
guava	38.3k	545	72
retrofit	36.2k	171	2
dubbo	33.1k	1823	93
MPAndroidChart	31.3k	70	2
glide	29.6k	257	24
netty	24.3k	1063	47
gson	18.3k	57	2
Total		27465	1288

* as of May 2020

RQ1: *Are documentations written independently?*

We further study whether the maintenance of documentation over time matters to the developers and users of the library project. Hence, for the second RQ, we study discussions over issue reports and pull requests to observe whether documentation-related inconsistencies are highlighted.

RQ2: *How often is the dependence of documentation highlighted through user and developer discussions?*

Inconsistent documentations in a library can cause inconvenience to the developers and users of the library with respect to program comprehension. To address the third RQ, we study how often developers miss updating dependent documentations while making alterations to the repository.

RQ3: *How often is the consistency of dependent documentation with code maintained as the repository evolves?*

After studying the existence and importance of the problem of indirectly dependent documentation in the context of code evolution, we carry out a deeper qualitative analysis of the libraries through RQ 4 and RQ 5. This analysis reasons about the patterns under which a documentation update is needed and builds a taxonomy of the nature of documentation updates made as a project evolves.

RQ4: *What is the nature of updates made to documentation to maintain consistency?*

RQ5: *What factors influence a documentation to be updated with the evolution of repository?*

B. Data Collection

We selected 11 top-starred repositories of libraries from GitHub that are implemented in Java and documented in English. These repositories are listed in Table I.

a) *Methods and Commits Extraction:* From the master branch of each repository, our script randomly chose source files and used JavaParser [34] to extract 50 methods and their

Javadoc comments. We obtained 550 methods from 11 projects to study RQ1. To study the remaining RQs, over each of the 11 repositories, we used the Git API to extract commit logs from all branches spanning 2 years from June 15, 2018 to June 15, 2020. We extracted all the files changed in a commit. GitHub does not follow a mainline integration model; developers can create branches for various purposes and merge the changes into the stable branch once they are confident about it [35]. We only consider the non-merge commits to avoid analyzing potentially duplicate updates. This resulted in the extraction of 27,465 commit logs (Table I). These commit logs were used to study the changes made to the repository and their effect on the documentations.

C. Analysis Procedure

We used a semi-automated approach to analyze the methods and the commit logs, where five of the authors performed the task of annotating the observations.

1) *RQ1: On the independence of documentation:* We manually studied the 550 methods and their Javadoc comments extracted from 11 libraries to determine whether the documentation is only influenced by the associated method’s source code or whether there exists another entity influencing the content of the documentation. The latter is a case of ‘dependent’ documentation. In order to characterize documentation dependencies, we conducted a preliminary study on a set of 100 randomly selected methods from the obtained dataset of methods. We checked whether each reference made by the target documentation could be mapped to the method source with which it is associated. In cases where we could not, we identified the content mapped to other entities, which we refer to as dependencies. In the process, we identified two key features that indicate a dependency. Accordingly, the annotator labelled a method documentation as ‘dependent’ if it satisfies at least one of the following criteria.

- The documentation has references to other entities that are object types, methods, fields, file paths, or URLs.
- The description of the documentation explains more than what the associated method directly implements. This description is either sourced from the invoked methods or other external factors.

2) *RQ2-5: On the importance of maintaining consistency and understanding the nature of and the reasons behind inconsistencies in the context of implicit dependencies:* As the focus of this study is the documentation associated with a method, we shortlisted commit logs from the 27,465 extracted logs which were more likely to highlight the indirect dependencies of the documentation. Consider a scenario where both a method’s source code and its documentation are updated; it is likely that the documentation update is induced by the change made to the associated method. However, in cases where only the documentation of a method is being updated, there are high chances of it being induced by changes made elsewhere. Further, these changes may be sourced from either the same commit in which the documentation is updated or from one of the previous commits. Based on this idea, we shortlisted

commits that contained at least one method for which only the documentation had been updated, that is, without any change made to the corresponding method’s implementation (filtration details in Section III-C2a). The annotators manually analyzed these commits, as described later in this section.

a) *Commit Filtration:* To filter out the unwanted commit logs, we followed a two-stage process. For a given list of commit IDs and the corresponding list of altered files (described in Section III-B0a), the first stage uses the Git API to get the parent commit ID of each commit. We used the `git diff` command to compute the lines changed in a file between the two commits, `commitid1` and `commitid2`. These changes are in the form of insertion and deletion of lines. A script stores all the changed lines in JSON format, with keys as `commitid1_commitid2_filename` and the values being the arrays of line numbers of insertions and deletions.

To obtain methods that have been altered in a commit, the second stage uses JavaParser alongside Git API to process the JSON file. From each key, our script parses the two commit IDs and filename. It then extracts the two versions of the file corresponding to the two commits. On each file version, we used JavaParser to visit all top-level class’s methods and get the start and end lines of the code and its documentation. A script compares these lines with the lines inserted or deleted to extract the list of methods changed between two commits for a given file. The script shortlists commits that contain at least one method in which only the documentation was changed. Note that there may still exist other methods having code-related changes in such commits. Applying this filtration process, we shortlisted 1,288 commits (Table I).

b) *Features Captured:* We perform a qualitative analysis on the shortlisted commits to capture various features associated with a commit log. As the study focuses on analyzing dependencies in documentation associated with methods, we use the following related terms throughout the paper.

A **source** is an entity, such as changed code or a referred URL, which influences the documentation of an indirectly related method.

A **target** is a method whose documentation has been affected by an indirectly related entity (the source).

Each annotator was provided with a JSON file that contained shortlisted commit IDs and corresponding lists of methods for which only documentation was modified. These methods were used to identify the sources and targets. Since we want to capture the maintenance of documentation consistency in the context of project evolution, the annotators analyzed only the differential part of the documentations in the given commit to log the observations. A documentation was labelled as ‘dependent’ as per the two criteria discussed in Section III-C1. Additionally, if the commit difference suggests that the documentation change occurred due to a change made elsewhere, such documentation was also labelled as ‘dependent’. The method associated with this documentation was labelled as a target. Note that not all shortlisted commit logs necessarily

indicate an indirect documentation dependency. Although they may appear in our shortlist, documentation updates such as grammatical fixes or formatting changes are not dependencies. Further, documentation updates may be directly sourced from previous changes made to the associated method. As we scope our study to indirectly induced documentation inconsistencies, we ignored such direct updates during the analysis.

RQ2: On documentation being discussed: We studied whether the maintenance of documentation matters to the developers and the users of the libraries. The annotators captured whether the commit log under analysis is linked to any pull request or issue report such that the documentation is the main topic of discussion over the thread.

RQ3: On maintaining the documentation consistency as the repository evolves: We studied the prevalence of cases in which a change is made to the project while the indirectly dependent documentations are overlooked and changed much later. Hence, an annotator captured whether a documentation update is sourced from a change made in the same commit or in a previous commit. The commit-change description and the discussion thread on the related pull request or issue report were used to obtain hints on the cause of a documentation update. This aligns well with the most frequently highlighted theme of ‘change description’ by developers, as listed by Ram et al. in their study on what constitutes a good code change from the perspective of change-reviewability and change-comprehension [36]. If the annotator did not find the source of documentation update in the same commit, he/she searched the linked discussion thread for potential references to previous commits and studied the commit history of the method to find the source of update. Further, on obtaining the previous commit, the annotator also noted the time between the introduction and the fixing of the documentation’s inconsistency.

RQ4: Nature of documentation updates to maintain consistency: To understand how documentation is influenced by various indirect changes, we studied the nature of updates made to the documentations. As observed in our dataset, documentation (a Javadoc comment) typically consists of a description of the method’s functionality, parameters, returned output, and extra information for detailed understanding. We identified 6 main categories of the nature of documentation updates described in Table II. We further identified 8 sub-categories for ‘description update’ (Table II).


All the categories have been defined to be exclusive of each other, as also validated over our dataset used for the analysis. This implies that every single update made to a documentation gets categorized uniquely. However, it may still be possible for different parts of a documentation to have changes of different sub-nature, in the ‘description update’ category. A typical commit log targets a single objective. Hence, every dependent documentation update was assigned with a single main category. Often, one type of documentation update triggered by a code change can result in other types of updates elsewhere. For instance, consider a case where the introduction of a new method leads to the deprecation of an older method. As a result, the developer adds a deprecation

note to the older method. This leads to the replacement of references made by the documentation of other methods to the older method (now deprecated) with the new method. Hence, one would observe two categories of documentation updates, ‘mark deprecated’ and ‘description update’. However, we know that the root type of the documentation update in this case is ‘mark deprecated’, despite the presence of other updates. Therefore, for clarity, we captured only the root update category for a commit log.

In case of a ‘description update’, it may be possible to assign multiple sub-categories to the documentation, owing to updates in multiple sections. For instance, if an update is made to the description of the functionality, and additionally, extra references are introduced for further clarification, then both sub-categories, ‘behavioral description’ and ‘extra suggestions description’, shall be noted. An annotator assigned appropriate category and sub-categories to the updates in the target’s documentation in the commit.

RQ5: Relations that trigger documentation updates: To understand the scenarios for which documentation is more likely to be updated, we studied the relations between the source and the target. We obtained patterns that can be harnessed to build applications. The annotators searched for the following relations between the source and the target and logged patterns of these relations for each commit log.

i. *Referential relation:* We captured whether the source or the target makes direct references to each other in their respective documentations. This relation indicates that if either of these entities references the other, then a change in one will potentially require updating the other’s documentation. Hence, a significant prevalence of such a pattern can be useful in automating the process of shortlisting targets, given a source. While analyzing a commit, for the extracted source and target pairs, the annotator logged one of the following observations: a) target’s documentation references the source, b) source’s documentation references the target, c) both, d) none.

ii. *Call-graph relation:* A call-graph captures the control flow relation between the methods calls made in a program. We captured this relation between the source and the target based on the idea that such relations would be reflected in the program’s behavior, and hence are likely to be documented by the developer. Therefore, in the context of alterations to the code, the documentation may also require updates. The annotator logged one of the following observations per commit: a) target in the call-graph of source, b) source in the call-graph of target, c) none. These relations  captured on the call-graph generated using the *Soot Framework* [50].

iii. *Inheritance relation:* We investigated whether it is common for the respective classes of the source and the target to share a relationship based on inheritance. Certain properties may get propagated by this relation, thereby creating a dependency between the source and the target. Hence, for each commit, the annotator captured the following observation on the classes to which the source and the target belong: a) target’s class inherits from source’s class, b) source’s class inherits from target’s class, c) classes of source and target

TABLE II: Nature of documentation updates.

	Category Description	Example of a documentation update
1.	Refactoring: Structural changes made to the references in the documentation, which do not change the description of the functions or semantics.	Elasticsearch commit 1a5e72e [37]: “...When created through #with-LocalReduction(SearchRequest, String[], String, long, boolean) , this method returns..” replaced by “...When created through #crossClusterSearch(SearchRequest, String[], String, long, boolean) , this method returns..” due to the renaming of the referenced method.
2.	Description update: Semantic update to the core description of the method.	
	2a. Behavioral description: Updates in the functionality’s intent description without making any direct references to entities. This may occur due to change in the source’s behavior without directly referencing it.	MPAndroidChart commit fcc5af7 [38]: “Callbacks when the chart is scaled / zoomed via pinch zoom gesture.” replaced by “Callbacks when the chart is scaled / zoomed via pinch zoom / double-tap gesture .” as the support for double-tap zoom is added in another class that invokes the target method.
	2b. Referential description: Updates in the references to entities for enhancing the functionality’s intent description. These references may also be to class comment to get further insights.	Glide commit ed20643 [39]: “..Previous calls to #apply(RequestOptions) ..” replaced by “..Previous calls to #apply(BaseRequestOptions) ..” where BaseRequestOptions class is added as the parent of RequestOptions.
	2c. Extra suggestions description: References to other entities for additional details.	Spring-framework commit 859923b [40]: Adding “ @see #getBeanType() ” to the documentation of getBeanName() , as extra information.
	2d. Parameter description: Updates in the description of method parameters and constraints induced on them by other entities.	Elasticsearch commit 3f2a241 [41]: Addition of “ @param requestId see ResponseHandlers#add(ResponseContext) for details ” to the parameter description, on addition of ResponseHandlers class.
	2e. Return value description: Updates in the description of the return-values and their constraints.	Guava commit 21cfbea [42]: Clarification added on the return type by referencing SortedMap in the documentation “the returned map itself is not necessarily a SortedMap”.
	2f. Example description: Updating examples to describe the method usage or functionality, which may also involve configuring other entities.	Guava commit 3dfec64 [43]: “Example: Files.fileTraverser(). breadthFirst(“/”) may return files ... [“/”, “/etc”, ...] ” replaced by “Example: Files.fileTraverser().depthFirstPreOrder(new File(“/”)) may return files ... [“/”, “/etc”, “/etc/config.txt”, ...] ” to include a more common example.
	2g. Alternative reference description: Updating the documentation with a reference to another method that can serve as an alternate to the target method under certain specific requirements.	RxJava commit 6ba932c [44]: Documentation of target updated with an alternative reference- “Similar to Objects.requireNonNull but composes the error message..”.
	2h. TODO notes description: Updating a note left in the documentation for future action having direct/indirect references to other entities.	Guava commit af3ee1c [45]: Documentation is added with a future action due to a code-change made elsewhere- “ TODO(user): remove the addNode() calls, that’s now contractually guaranteed ”.
3.	Mark deprecated: Deprecation of the target method, leading to referencing an alternative in the documentation.	Spring-boot commit a63ab46 [46]: Deprecation note added to an existing method when a new method is introduced- “ @deprecated in favor of #setRSocketServerCustomizers(Collection) as of 2.2.7 ”
4.	Exception type addition/update: Declaring or updating exception types in the documentation that may get thrown by the target method.	Spring-framework commit 71f3498 [47]: Exception was declared in the Javadoc of a method as one of the callee method’s code was updated- “ @throws DecodingException if the metadata cannot be decoded ”.
5.	Referential typo fix: Typo fixes in the documentation made in references to the source entities.	Netty commit 8f01259 [48]: Incorrect reference to a method in the documentation fixed from setMaxHeaderListSize(long) to setMaxHeaderListSize(long) .
6.	Version update: Updating the documentation in the context of certain platform or version dependencies.	Guava commit 7fdf1a6 [49]: The documentation is updated with a special instruction for Java 9 users- “ Java 9 users: use java.util.Objects.requireNonNullElse(first, second) instead ”.

share a common parent, d) source and target belong to same class, e) none.

iv. *Interface relation:* As interfaces provide abstractions that the implementing classes must adhere to, we studied the prevalence of such a relation between the source and the target’s class or interface in an attempt to infer the cause of the dependency. For every commit, the annotator noted the corresponding class or interface of the target and source

and logged one of the following observations: a) target’s class implements from source’s interface, b) source’s class implements from target’s interface, c) classes/interfaces of source and target implement/extend a common interface, d) source and target belong to the same interface, e) none.

c) *Inter-rater agreement:* An agreement mechanism was followed batch-wise in the process of analyzing the data. Initially, each of the five authors annotated 10 commit logs.

TABLE III: Free Marginal Kappa scores to evaluate the inter-rater agreement.

Feature Analyzed	Kappa Score
Documentation has indirect dependency	0.88
Documentation update triggered by change made in the same or previous commit	1
If the update is sourced from a discussion on the documentation	0.89
Update induced from the change made in same or different file	0.76
Nature of documentation update	0.84
Call-graph relation-type	0.74
Reference relation-type in the documentation	0.85
Inheritance relation-type	0.93
Interface relation-type	0.93

Then a group discussion was held among the authors to discuss all 50 logs, to come to an agreement and check consistency in the understanding. A second batch of 10 logs, analyzed by each annotator, was validated by another annotator. To capture the inter-rater agreement among the annotators, 10 randomly picked commit logs from the dataset were provided to be independently analyzed by each of them. We computed the free-marginal kappa agreement scores (Table III) [51] on these annotations [52]. A kappa score of 0.88, which indicates “almost perfect agreement” [51], was obtained for the classification of documentation updates as indirectly dependent or independent. On obtaining a thorough understanding through this exercise, all annotators continued to annotate a subset of commit logs and marked their confidence levels as high, medium, or low, to associate with each annotation. All logs marked as ‘low’ were resolved in a group discussion and the logs marked as ‘medium’ were validated by another annotator.

IV. FINDINGS

With the design and experiments to explore the mentioned RQs, we discuss the findings in this section. We have made all the findings of the study, along with the dataset and the scripts used to prepare it, available¹.

A. RQ1: Are documentations written independently?

For 341 of the 550 analyzed Javadoc comments of methods, we observed the prevalence of dependencies on indirectly associated source code and other entities. Considering the two criteria mentioned in Section III-C1 to mark a documentation as dependent, for 321 cases, the Javadoc comment had references to the source of dependencies, while for 20 cases the dependency was purely descriptive, without references to the source. Fig. 2 shows a project-wise distribution of dependencies with and without references. Across all projects, the occurrence of dependent Javadoc comments ranged from 36% to 74% (overall 62%). These numbers indicate a significant presence of dependencies that require the attention of the developers in the context of making updates to the repositories.

Listing 2 shows a method for which the dependent Javadoc comment describes the method that has been invoked by `sha256BytesToHex` without directly referencing it. Fig.2

Listing 2: Javadoc comment that is dependent on the invoked method without directly referencing it.

```
/** Returns the hex string of the given byte array
 * representing a SHA256 hash. */
public String sha256BytesToHex(byte[] bytes) {
    synchronized (SHA_256_CHARS) {
        return bytesToHex(bytes, SHA_256_CHARS);
    }
}
```

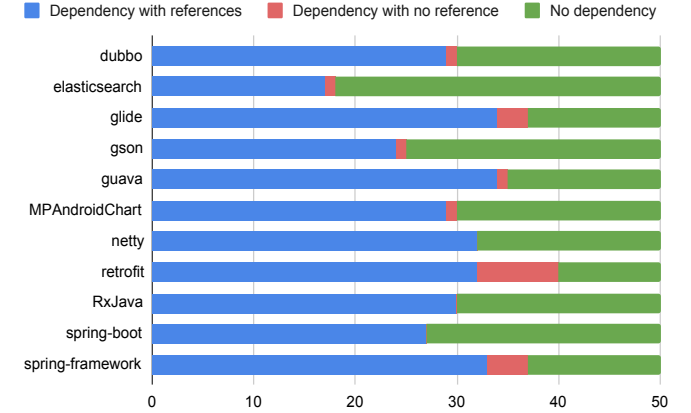


Fig. 2: Project-wise distribution of indirectly dependent and independent method Javadocs studied for RQ1 with 50 methods analysed from each of the 11 projects.

represents the share of such cases in red. Cases like these make it difficult to automate the identification of sources and potentially require support from natural language processing techniques, in addition to program analysis. The documentation containing references had the following types of references: 1) method, 2) field, 3) class object and exception types, 4) enum types, 5) URL to external resources, and 6) package path. Fig. 3 gives a taxonomy of the nature of dependent Javadoc comments observed while studying for RQ1. Most of these categories are similar to those stated in Table II; however, note that the latter describes the nature of ‘documentation updates’, discussed later, while the former describes the nature of dependent components in the whole ‘documentation’.

Finding 1: *There is a substantial presence of indirect dependencies in the documentations, which requires the attention of the developers in the context of making updates to the repositories.*

B. RQ2: How often is the dependence of documentation highlighted through user and developer discussions?

Of the 1,288 shortlisted commit logs, we observed that for 592 cases, the commit involved the updating of at least one method’s documentation (the target) due to modifications made at other places (the sources). A single commit may have multiple targets and sources arising from cases where a) a set of modifications collectively influence documentation of the target, which leads to multiple sources, or b) a single change induces documentation updates for multiple methods, resulting in multiple targets. The nature of documentation update was found to be highly similar in the case of multiple targets.

¹<https://github.com/pag-iiitd/DocDependency>

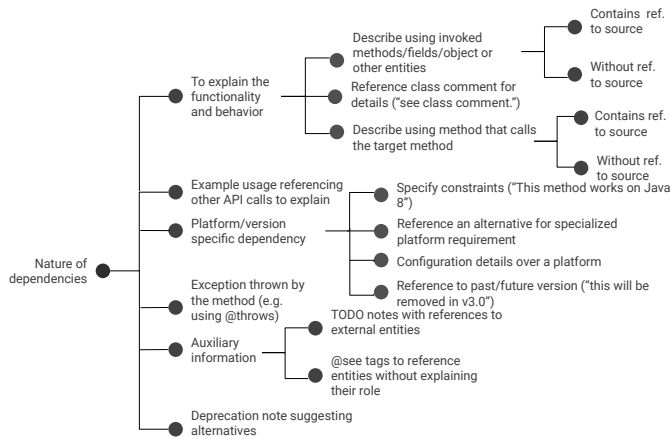


Fig. 3: Nature of dependent updates observed for RQ1.

Finding 2: 46% of the commit logs in the dataset resolved indirect dependencies in the method’s documentations.

This observation indicates that the maintenance of documentation matters to the developers. We further observed that 189 of 592 (32%) commits were sourced from discussions, available on the thread of issue reports or pull requests, that were specifically on documentation. The following are two such comments quoted from the discussion threads:

‘The link to the HLRC documentation for the async Put Lifecycle method was never updated to the correct link. This commit fixes that link.’

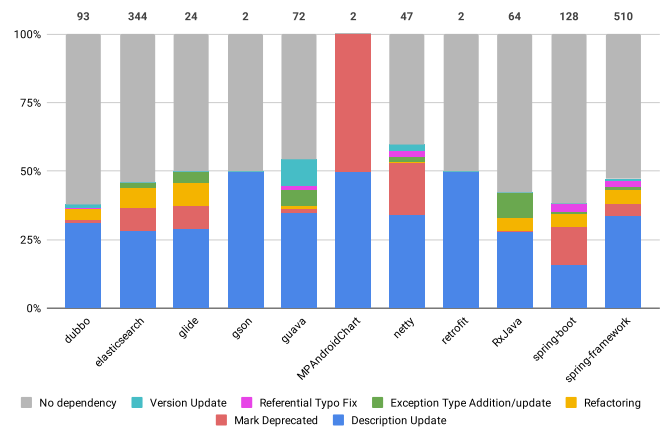
‘Questions on how to work with ActionPlugin#getRestHandlerWrapper() come up in discussion forums all the time. This change adds an example to the Javadoc how this method should/could be used.’

Finding 3: Maintenance of the documentation matters to the developers and users of the repositories, as evident from the discussion threads.

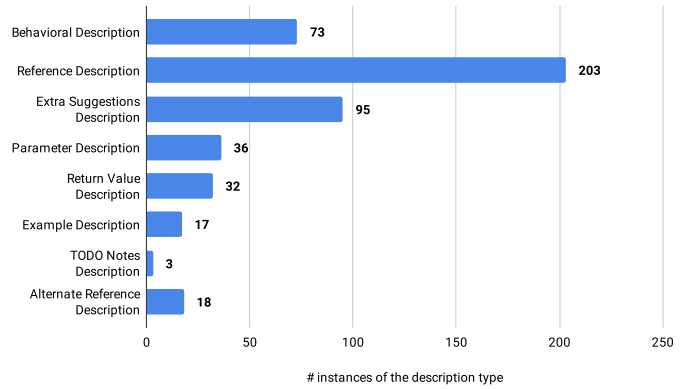
C. RQ3: How often is the consistency of dependent documentation with code maintained as the repository evolves?

Ideally, any change made to the repository should be reflected in the dependent documentations at the same time. However, we observed that in reality, the documentation at a point is often left inconsistent with the committed code. Of the 592 commit logs containing dependent documentation updates, in 77 cases (13%), the documentation update was sourced from changes made in past commits. Further, we observed that this dependent documentation may be left inconsistent for anywhere from 0 to 1988 days since the change that induced the documentation update, with a mean resolution duration of 469.8 days and standard deviation of 565.7 days.

Finding 4: As the repository evolves, dependent documentations may be left inconsistent for a long time.



(a) Project-wise distribution of nature of documentation updates.



(b) Distribution of sub-categories of ‘description update’.

Fig. 4: Nature of documentation update.

D. RQ4: What is the nature of updates made to documentation to maintain consistency?

For 325 of the 592 commit logs (54.9%) containing dependent documentation updates, the target existed in a different file from the source. 48 of the 325 cases had multiple targets of which some belonged to the same file as the source and others belonged to different files. The sources may also be external entities existing in a library or a web resource different from the target, as observed for 81 of 592 cases (13.7%). Such external dependencies can pose challenges in tracking the updates. One such documentation update containing a link to a web-page is quoted as ‘This can leave your server implementation vulnerable to “https://cwe.mitre.org/data/definitions/113.html” CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers (‘HTTP Response Splitting’).’

Finding 5: Documentation may contain updates from external sources (not present in the target repository).

Fig. 4 shows the observed distribution of the nature of updates made to method documentation. Fig. 4(a) indicates the project-wise existence of different update categories described in Table II. The grey fraction of each bar indicates the percentage of shortlisted commit logs with no indirectly dependent

documentation updates; these documentation updates were either directly influenced by the associated method or grammar or formatting fixes. Among the dependent documentation updates, ‘description update’ dominated across all projects, followed by ‘mark deprecated’.

Each commit log with a ‘description update’ (386 of 592 commits) may be further described by one or more sub-categories. We analyzed the description updates (Fig. 4(b)) and observed that in most cases, the update enhanced or fixed the description of the functionality by using references, covered by the category ‘referential description’ (203 of 386 cases). The second most popular description update type was ‘extra suggestions description’ (95 of 386 cases), to add or update supplementary details by describing or referencing entities for added clarity about the functionality. This was followed by ‘behavioral updates’ to clarify or fix the functionality description. These observations indicate that while in most cases the sources influence updates of the description of the functionality, the developers also seek to maintain additional details that normally act as supplementary references.

Finding 6: *Developers seek to update the documentation in a variety of ways. They mainly enhance or fix the core description of the functionality, indicate deprecation by referring to an alternative, or update the supplementary details for better clarity about the method.*

E. RQ5: What factors influence a documentation to be updated with the evolution of repository?

To reason about the dependency caused in documentations, we analyzed patterns in four types of relations between the source and the target methods and their classes/interfaces. Fig. 5 shows a distribution of source and the target associations across the four relations, as observed in our dataset.

Referential relation captures the association of source and target through a reference to one in the other’s documentation. Fig. 5(a) indicates that for 86.6% of the commits, either the target’s or the source’s documentation mentioned the other. In most cases, the target referenced the source (78.8% cases). Hence, an alteration to the source, such as renaming the source method, may induce an update to the target’s documentation. The presence of such references may prove to be useful in identifying the source given a target or a target given a source. The latter is especially useful when the developer needs to identify points (the targets) where documentation (or code) needs to be updated after a source is modified.

Finding 7: *The documentation of the source method often has references to entities influencing it.*

To further analyze the cause of such references, we explored three types of programmatic relations between source and target. Call-graph relation captures if the two are related by a caller-callee relation (which may occur at any depth in the call-graph). We observed that for 24% cases, such a relation existed (Fig. 5(b)), with the target calling the source in most of these cases. This indicates that the documentation of the

target often describes or derives information from the called entities, which induces a dependency.

We further analyzed if classes or interfaces corresponding to source and target are related. A dominant case was observed in which the source and the target are present in the same class/interface (Fig. 5(c),(d)). This can be explained by the usual practice of entities, such as methods or class fields, within a class interacting with each other. As a result, their documentations are likely to have details that are dependent on one another. There were a few instances where the corresponding classes/interfaces of the source and target extended/implemented the same parent class/interface, shared a parent-child relation, or implemented the other’s interface. Inheritance aims at propagating properties that get reflected in a child class. A change in either the parent or its children must not violate these properties. Similarly, an interface provides abstractions that the implementing classes must adhere to. These relations explain possible causes of dependencies in documentation, and may also be useful in mitigating inconsistent documentation, as we discuss in Section V.

Finding 8: *Several programmatic relations such as caller-callee pattern, inheritance of properties, and implementation of abstractions propagated by interface may influence the content of the documentation of an entity.*

Of the 592 commits having dependent documentation updates, we observed that at least one of these four relations categorized to a value other than ‘none’ for all but 11 cases. Hence, searching for such patterns may help in identifying sources and targets. For the 11 commits where none of the patterns existed, we analyzed how else the source and the target were related. We observed the following patterns.

- *Non-programmatic source which isn’t referenced in the target’s documentation.* Example: Example code in the documentation was changed based on the recommended convention in an RFC documentation.
- *Source and target accessing a common entity.* For instance, consider a class field which is accessed for manipulation by the source and target methods, with the documentation being derived from this behavior.

We plan to explore these patterns on the entire dataset as future work in order to draw further insights.

V. DISCUSSION

The discussed observations may find a direct use case for developers in the form of a recommendation system to suggest methods (targets) for which code or documentation should be altered when a developer makes code changes to other entities (sources). This would help prevent documentation inconsistencies that arise as a project evolves. Some patterns that can potentially be leveraged for the application are discussed in this section.

In 6% of cases with dependent documentation updates, we observed sources calling targets. This pattern is directly useful for shortlisting methods to recommend for a documentation

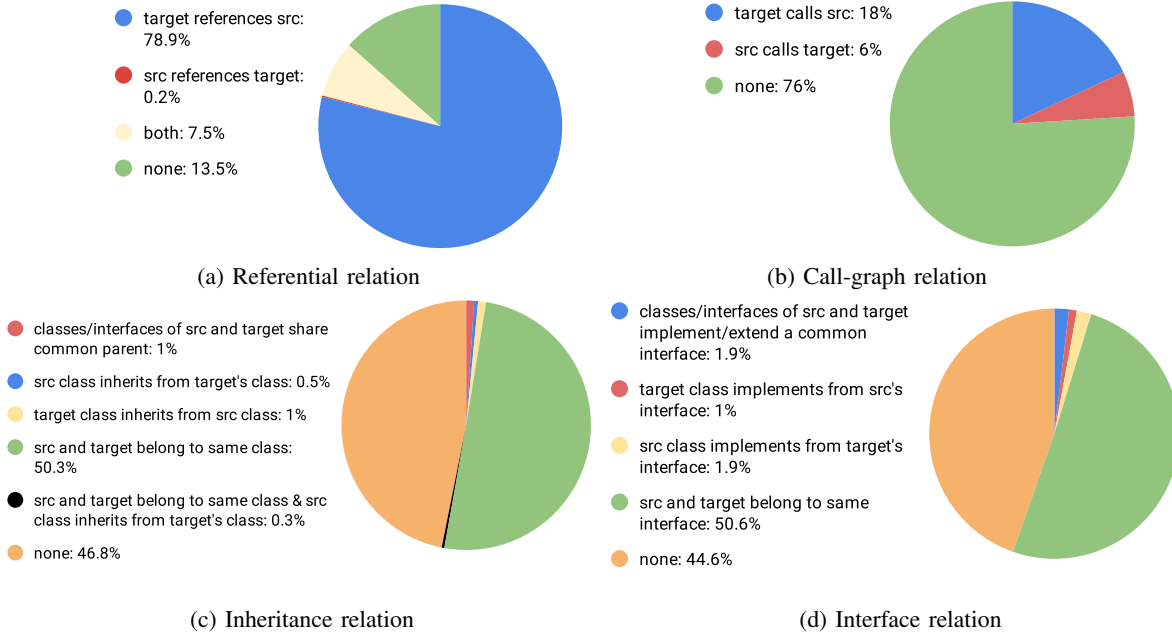


Fig. 5: Relation between the source and the target as observed in 592 commit logs.

update using the program’s call-graph. On the other hand, 18% of cases with documentation updates involve targets calling sources. This pattern could be applied to shortlist the targets for a given source by maintaining a linkage graph for the repository where edges exist from a programmatic entity to all other entities that call or use it; the graph may be dynamically updated for every code change. Hence, as a code change occurs, all nodes directly or indirectly linked to it can be shortlisted to check for documentation inconsistencies. A similar linkage graph can be maintained to reflect inheritance or interface relations. For a scalable and a meaningful recommendation, the described approach can be combined with heuristics based on referential relations, which were found to exist in abundance. For instance, the shortlisted linked entities could be further processed such that the methods referencing the source in their documentation or whose references appear in the source’s documentation can be shortlisted for updating the documentation. Several such heuristics can be developed to leverage the observed patterns.

The recommendation may be extended to suggest documentation changes for the identified targets. This would require studying relations between the nature of code or documentation change made at the source and the nature of the induced documentation update observed at the target. Combining these relational insights with natural language processing techniques could aid in building an IDE plugin for recommending documentation updates in addition to identifying methods for which a documentation update is required.

VI. THREATS TO VALIDITY

a) *Construct validity*: Inferences of the source and the nature of documentation update are purely on openly available artefacts such as discussion threads and commit history. Other

missing factors, such as in-person discussions between developers, may have influenced code and documentation changes.

The selection of only library projects was incidental. Applying our criterion to pick top-starred projects resulted in only libraries and frameworks. One explanation for this trend could be that libraries and frameworks are meant to be reusable for further development, and hence, more developers use or star them. Extending our analysis to non-library projects would make for interesting future work.

b) *Internal validity*: To mitigate the possible subjectivity in the analysis, which has been done by 5 annotators, we followed a phase-wise approach to build common understanding. We evaluated this understanding through an inter-rater agreement exercise (Table III). Further, logs marked with medium and low confidence by an annotator were given to other annotators to analyze independently (Section III-C2c).

For four cases with untraceable sources, we take a conservative approach by not considering the documentations to have indirect dependencies.

c) *External validity*: The filtration criteria used to extract the commit logs (Section III-C2a) may not cover all cases of documentation updates that have been indirectly influenced in a project. However, the criteria only give an underestimate; the actual number of cases of dependent documentations is likely to be higher, which makes the relevance of the problem even stronger.

The study has been conducted on projects implemented in Java. As Java is among the top 5 languages on GitHub, our findings may nonetheless be useful to a significant fraction of the developer community. With a considerable amount of manual analysis involved and the underlying language-dependency of the APIs used for data processing, we limited the study to 11 projects. With the discussed patterns found

by the study, it opens an opportunity to automate some components of the analysis and explore projects from more languages and diverse classes.

VII. CONCLUSION

We conducted a comprehensive study on 11 open-source projects to analyze the prevalence and nature of documentation dependencies on entities other than the associated function. We analyzed 550 Javadoc comments and observed over 62% of them to be indirectly dependent. We further analyzed 1288 commit logs to categorize the nature of updates made to documentations, as induced by code-evolution changes and external changes. To reason about the factors that induce these inconsistencies in the documentation, we studied four types of relations between the source and the target. We observed the presence of references in the documentation to be a dominant reason for the existence of dependencies. Analyzing the programmatic relations between the source and the target, we suggested applications of these patterns. One such application is a recommendation system to suggest methods for which documentation should be updated when a developer makes changes in other sections of the repository.

The findings of the study open several interesting problems. One such problem would be to track updates in external sources that can trigger documentation updates in the repository. As the domain of external sources is huge and mostly non-programmatic in nature (such as web-pages), taking note of their updates and influence on the target would be a challenge. Further, there may exist unexplored patterns relating the source and the target that can be leveraged to mitigate the problem of inconsistent documentation. Combining programmatic features, extracted by program analysis, and machine learning approaches may aid in learning these patterns.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback on this work. This work is supported in part by the Department of Science and Technology (DST) (India), Science and Engineering Research Board (SERB), Confederation of Indian Industry (CII), Microsoft Research. We offer special thanks to Sehaj Risham Kaur, who is no longer with us but she has been an inspiration behind taking the initial idea of the study to culmination.

REFERENCES

- [1] 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 & designing for pervasive information*, 2005, pp. 68–75.
- [2] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.
- [3] Y. Higo and S. Kusumoto, "How often do unintended inconsistencies happen? deriving modification patterns and detecting overlooked code fragments," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 222–231.
- [4] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 53–64.
- [5] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 112–122.
- [6] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*" in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 145–158.
- [7] Spring-projects, *Spring-framework. SPR-16130*, 2017 (October 30, 2017). [Online]. Available: <https://github.com/spring-projects/spring-framework/issues/20678>
- [8] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2015.
- [9] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.
- [10] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1199–1210.
- [11] V. Misra, J. S. K. Reddy, and S. Chimalakonda, "Is there a correlation between code comments and issues? an exploratory study," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 110–117.
- [12] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.
- [13] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 2007, pp. 70–79.
- [14] A. Corazza, V. Maggio, and G. Scanniello, "On the coherence between comments and implementations in source code," in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2015, pp. 76–83.
- [15] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@ tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 260–269.
- [16] S. Rebai, O. B. Sghaier, V. Alizadeh, M. Kessentini, and M. Chater, "Interactive refactoring documentation bot," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2019, pp. 152–162.
- [17] M. Petito, "Eclipse refactoring," <http://people.clarkson.edu/~dhou/courses/EE564-s07/Eclipse-Refactoring.pdf>, vol. 5, p. 2010, 2007.
- [18] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.
- [19] A. Sarma, G. Bortis, and A. Van Der Hoek, "Towards supporting awareness of indirect conflicts across software configuration management workspaces," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 94–103.
- [20] J. Czerwinka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev, "Crane: Failure prediction, change analysis and test prioritization in practice—experiences from windows," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 357–366.
- [21] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.
- [22] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 91–100.
- [23] 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, 2018.
- [24] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008, pp. 1323–1332.
- [25] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 802–811.

- [26] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 315–324.
- [27] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 195–204.
- [28] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 511–522.
- [29] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.
- [30] S. Mehta, R. Bhagwan, R. Kumar, C. Bansal, C. Maddila, B. Ashok, S. Asthana, C. Bird, and A. Kumar, "Rex: Preventing bugs and misconfiguration in large services using correlated change analysis," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 435–448.
- [31] R. Robbes and M. Lanza, "How program history can improve code completion," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 317–326.
- [32] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 819–830.
- [33] Octoverse, GitHub, "Github. the state of the octoverse 2019." 2019, retrieved September 30, 2019. [Online]. Available: <https://octoverse.github.com>
- [34] N. Smith, D. van Bruggen, and F. Tomassetti, "Javaparser: visited," *Leanpub, oct. de*, 2017.
- [35] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 1–10.
- [36] A. Ram, A. A. Sawant, M. Castelluccio, and A. Bacchelli, "What makes a code change easier to review: an empirical investigation on code change reviewability," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 201–212.
- [37] Elasticsearch, *Elasticsearch. Commit 1a5e72e*, 2019 (February 26, 2019). [Online]. Available: <https://github.com/elastic/elasticsearch/commit/1a5e72e2b4b>
- [38] MPAndroidchart, *MPAndroidchart. Commit fcc5af71*, 2020 (January 22, 2020). [Online]. Available: <https://github.com/PhilJay/MPAndroidChart/commit/fcc5af71>
- [39] Glide, *Glide. Commit ed20643fb*, 2018 (September 6, 2018). [Online]. Available: <https://github.com/bumptech/glide/commit/ed20643fb>
- [40] Spring-projects, *Spring-framework. Commit 859923b*, 2019 (June 11, 2019). [Online]. Available: <https://github.com/spring-projects/spring-framework/commit/859923b>
- [41] Elasticsearch, *Elasticsearch. Commit 3f2a241*, 2018 (July 4, 2018). [Online]. Available: <https://github.com/elastic/elasticsearch/commit/3f2a241b7f0>
- [42] Guava, *Guava. Commit 21cfbea*, 2019 (July 16, 2019). [Online]. Available: <https://github.com/google/guava/commit/21cfbea052>
- [43] Guava, *Guava. Commit 3dfee64*, 2018 (December 6, 2018). [Online]. Available: <https://github.com/google/guava/commit/3dfee64294>
- [44] ReactiveX, *RxJava. Commit 6ba932c*, 2019 (December 18, 2019). [Online]. Available: <https://github.com/ReactiveX/RxJava/commit/6ba932c9a>
- [45] Guava, *Guava. Commit af3ee1c*, 2018 (October 27, 2018). [Online]. Available: <https://github.com/google/guava/commit/af3ee1c598>
- [46] Spring-projects, *Spring-boot. Commit a63ab46*, 2020 (April 28, 2020). [Online]. Available: <https://github.com/spring-projects/spring-boot/commit/a63ab468a3>
- [47] Spring-framework, *Spring-framework. Commit 71f3498*, 2019 (September 2, 2019). [Online]. Available: <https://github.com/spring-projects/spring-framework/commit/71f3498>
- [48] Netty, *Netty. Commit 8f01259*, 2018 (June 26, 2018). [Online]. Available: <https://github.com/netty/netty/commit/8f01259833>
- [49] Guava, *Guava. Commit 7fdf1a6*, 2019 (October 3, 2019). [Online]. Available: <https://github.com/google/guava/commit/7fdf1a6431>
- [50] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.
- [51] A. J. Viera, J. M. Garrett *et al.*, "Understanding interobserver agreement: the kappa statistic," *Fam med*, vol. 37, no. 5, pp. 360–363, 2005.
- [52] J. J. Randolph, "Online kappa calculator," *Retrieved October*, vol. 20, p. 2011, 2008.