An Empirical Evaluation of Method Signature Similarity in Java Codebases

Mohammad Taha Khan
Washington and Lee University
Lexington, Virginia, USA
tkhan@wlu.edu

Mohamed Elhussiny
Washington and Lee University
Lexington, Virginia, USA
elhussinyh24@mail.wlu.edu

William Tobin
Washington and Lee University
Lexington, Virginia, USA
tobin24@mail.wlu.edu

Muhammad Ali Gulzar
Virginia Tech
Blacksburg, Virginia, USA
gulzar@vt.edu

Abstract

Modern programming languages have transformed software development by providing capabilities of enhancing productivity and reducing code redundancy. One such feature is allowing developers to choose meaningful method names for implementation and functionality. As programs evolve into APIs and libraries, developers often design methods with similar signatures to streamline code management and improve comprehensibility.

In this paper, we conduct a comprehensive study to evaluate the prevalence, usage, and perception of methods with similar signatures, including both conventionally overloaded and textually similar methods. Through analyzing 6.4 million lines of code across 167 well-established Java repositories on GitHub, we statistically assess the occurrence of these methods and their impact on usability and software quality. Additionally, we explore the evolution of these methods through a longitudinal analysis of historical commit snapshots. Our research reveals that both overloaded and textually similar methods are common in leading Java repositories and are primarily driven by specific software design requirements, program logic, and developer's programming habits. As software matures, development shifts towards maintenance tasks that rarely necessitate design changes. Our longitudinal analysis corroborates this by indicating minimal changes in methods with similar signatures in the later stages of a repository's life.

CCS Concepts

• **Software and its engineering** → **Maintaining software**; **Software evolution**; *Empirical software.*

Keywords

software engineering, empirical measurements, software analysis, code usability, naming conventions

ASSE 2024, September 11–13, 2024, Tokyo, Japan
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1754-3/24/09
https://doi.org/10.1145/3702138.3702152

ACM Reference Format:
Mohammad Taha Khan, Mohamed Elhussiny, William Tobin, and Muhammad Ali Gulzar. 2024. An Empirical Evaluation of Method Signature Similarity in Java Codebases. In *2024 5th Asia Service Sciences and Software Engineering Conference (ASSE 2024), September 11–13, 2024, Tokyo, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3702138.3702152

1 Introduction

In software development, method names play a critical role in conveying the underlying semantics of a method's function. As software evolves into libraries, the API interfaces, particularly their names, are often used interchangeably with documentation to understand the underlying functions. Consequently, best software engineering practices advocate for meaningful and distinct method names to enhance code management and comprehensibility. However, in the development process of large software systems, many methods end up with highly similar signatures. Commonly, one instance of this is method overloading, which results in sharing names across multiple semantically dissimilar implementations but with the same abstract functionality. For example, the methods `String.substring(int)` and `String.substring(int, int)` are considered overloaded. Similarly, some method names, while not identical, may be very similar due to the constraints of natural language representation, such as `javax.sql.Rowset.getclob` and `javax.sql.Rowset.getNclob`, which are distinct in context and implementation. Such similarities in method signatures are intended for better comprehension, code management, and meaningful variable names.

While method signature similarity is generally accepted as a beneficial concept in software development, their overuse can lead to increased code complexity, which can significantly impact usability, especially in large and growing repositories [10]. Work in other domains has also demonstrated that information overload can cause cognitive limitations affecting productivity [12, 25]. In this study, we build on this idea by specifically evaluating the influence of method signature similarity and its evolution through empirical measurements. We examine whether an extensive presence of such methods can lead to confusion, potentially resulting in unintentional misuse of the API and subsequent software bugs. This risk is especially relevant given that these methods often have largely overlapping parameters and can inadvertently introduce latent bugs that are likely to slip through manual inspection, compilation checks, and, in some cases, even test suites. Given this, it is necessary to study how the presence of these methods affects software quality.

For our study, the term *signature similarity* encompasses two categories of similarly looking methods. The first comprises conventionally overloaded methods with an identical name but varying parameters. The second category includes method pairs that are textually similar, identified by a small edit distance among their names. This metric is crucial for Java codebases, where the language's case-sensitive and object-oriented nature offers developers flexibility in naming methods. However, this flexibility can result in method names that are strikingly similar, with only subtle variations.

We design five research questions for this study, focusing on the statistical prevalence of signature similarity in methods followed by a longitudinal analysis of how such methods have evolved over time. By answering these questions, we can better understand trends among popular projects, including the long-term perceived utility of such methods especially in the context of project maturity and changes in developer perception.

We curated a selection of 167 public Java repositories from GitHub with the highest number of stars, collectively containing a total of 6.4 million lines of code. We utilize the GitHub Rest API to access the source code of these projects at various stages of their life cycle, represented by specific commit numbers. For our analysis, we employ the popular Java static analysis tool, CK [5], to extract the abstract syntax trees (ASTs) of the respective project classes. These were then utilized to record method signatures and assess their similarity. Section 3 provides a more detailed discussion of our data collection and analysis.

Our statistical analysis reveals that across the studied repositories, an average of 5.1% of methods were conventionally overloaded. This is a new finding that is slightly lower than the estimated value of 13% in prior work [15] from 2010. Furthermore, we also observed a prevalence of textually similar methods, with an average of 4.2% of methods having a close edit distance with another method. Given that these percentages are in line with the prevalence of overloaded methods, it is crucial for programmers to utilize and name methods appropriately to avoid confusion and error. While method overloading is well-suited for constructor methods, we find that a majority of overloaded methods are actually tied to specific design patterns and project use cases. More notably, we identify that textual similarity in methods frequently arises from naming conventions and, in certain cases, poor naming practices that affect code maintainability.

Our longitudinal analysis highlights significant trends in method signature evolution. Early-stage snapshots of repositories show frequent changes in overloaded methods due to *adaptive software maintenance* tasks like adding new features. As repositories mature, the number of overloaded methods stabilizes, reflecting a shift to bug fixes and optimizations. We also observe a notable decrease in textually similar methods, suggesting that developers remove them to reduce ambiguity and improve code quality.

To our knowledge, this study is the first to quantify the prevalence of method signature similarity in Java repositories, comparing overloaded and textually similar methods and exploring their evolution over time. While previous research has focused on method overloading characteristics and API misuse due to poor documentation and copy-paste editing [17], we aim to understand how similar method signatures impact code usability and their evolution over time.

Repository Details	Minimum	Average	Maximum
Methods per Repository	8	11744	126615
Classes per Repository	1	1992	20496
Methods per Class	1	6	4096
Lines of Code	13	84357	1188302
Number of Contributors	1	98	405
Repository Age (Days)	895	3303	5294
Repository Stars	3491	10263	40000

Table 1: Summary statistics of repositories in our dataset.

2 Related Work

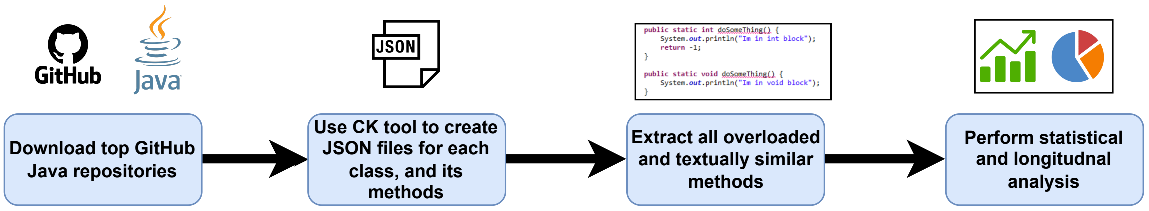
The study on understanding how method names impact developer practices has long been an active area of research [8, 14, 22, 23]. The flexibility to select a particular name not only allows for better comprehensibility [20] but also keeps the cost of code maintenance low [13] while enabling more robust testing [22]. To study the effectiveness of naming methods, a recent study by Alsuhaibani et al. conducted a survey of 1100 participants to examine the general acceptance and practical usage of source code method naming standards [3]. Their findings revealed that most participants agreed on the use of standards. Distinctively, another allied area of research explores the use of overloaded methods, especially in Java [4]. In 2010, Gil et al. surveyed overloaded methods in Java [15] and found their widespread use in code repositories. Previous work has suggested the restriction of overloaded methods [21] while others have highlighted how overloading can cause encapsulation flaws [7]. Our research builds upon previous studies and aims to quantitatively identify the impact of both overloaded methods and textually similar methods on developer practices. By broadening the scope, we seek to gain a comprehensive understanding of how different naming choices influence and impact software development processes.

3 Methodology

In this section, we describe our overall approach to collecting and analyzing the repositories for our work. To study signature similarity among methods, we selected Java as our language of choice. The motivation behind this stemmed from Java's object-oriented design and its programmers' ability to overload methods with versatility. Additionally, the high-level nature of the language makes it a popular choice for developing many projects that span multiple domains, ranging from big data technologies to financial security systems. Figure 1 provides a visual summary of our data collection and analysis approach.

3.1 Data Collection

To accurately collect a dataset representative of large-scale development in Java, we first ranked Java repositories on GitHub based on their star count. Repository stars are a commonly used indicator of a repository's popularity and usage and have been regularly used as a selection criterion in prior work [26]. From these, we shortlisted 252 repositories per GitHub's rate limits of


GitHub
Java
JSON
Download top GitHub Java repositories
Use CK tool to create JSON files for each class, and its methods
Extract all overloaded and textually similar methods
Perform statistical and longitudnal analysis

```
public static int doSomeThing() {
    System.out.println("Im in int block");
    return -1;
}

public static void doSomeThing() {
    System.out.println("Im in void block");
}
```

Figure 1: Data collection and extraction approach.

1,000 requests on searches. Upon manual review, we discovered that some repositories only contained markdown content, while others were incompatible with our analysis due to their large size. For instance, `elastic/elasticsearch` has 4.4 million lines of code that the CK [5] static analysis tool could not process. Consequently, we finalized a selection of 167 Java projects for data collection.

We primarily used a Python script incorporating the GitHub REST API to collect the latest *master* branch of the Java repositories. Our repositories had between 3,491 and 40,000 stars, which strongly indicated widespread adoption. Despite some repositories being incompatible with our analysis framework, our methodical selection process guaranteed us access to large amounts of data for analysis. Table 1 summarizes the characteristics of the final 167 repositories based on their most recent state at the time of our data collection. In addition to performing statistical analysis on the state of methods with similar signatures, we also intended to perform a longitudinal analysis of how these repositories had evolved over time. From the total 167 repositories, we shortlisted 43 and collected 25 historic commits for each, evenly distributed across their lifetime. This approach enabled us to precisely examine the changes in the repository's contents while catering to variations in development pace. By including commits from various stages, we accounted for the tendency of most repositories to experience rapid growth during their initial development phases and other developmental spurts.

3.2 Method Extraction Approach

After collecting the source code of the repositories, we used the CK code analysis tool [5] to extract relevant repository and class metrics. Our choice of CK was grounded in its general acceptability among research academia for analyzing Java repositories. To tailor CK for our specific goals, we modified the main driver file, *Runner.java*, to iterate over all the methods in the repositories. We then created a hash table of the method-level metrics for all Java classes that existed in the collected repositories. For each method, we gathered detailed metadata, including its name, frequency within the class, input parameter order and types, return type, and inheritance hierarchy specifics.

Under the umbrella term of signature similarity, we categorized our methods into two distinct groups. The first group included conventionally overloaded methods, which have the same name but different parameters. Figure 2a provides an example of such methods in the `utils.Array` class of the `OpenJDK11` [2] repository. The second group comprised methods with textually similar names, as shown in Figure 2b. While identifying overloaded methods was straightforward, we needed to devise a strategy to detect these textually similar methods. To accomplish this, we employed the *Levenshtein* distance approach [19] and measured the edit distance between pairs of methods within the same Java class. This technique allowed us to account for deletions, insertions, and substitutions in method names. We then isolated pairs of methods with an edit distance of one or two, indicating a high degree of textual similarity. This methodology enabled us to categorize and analyze methods based on their signature similarities effectively.

We further filtered the pairs by matching their argument names, types, and order, as well as their return type and the type of the associated class. This additional step enabled us to uniquely identify pairs that could potentially be misused by developers, who might inadvertently swap their usage due to human error. Such mistakes can remain undetected by compilers, which are typically only adept at identifying incompatible parameters and return types. Figure 2b provides two distinct methods, both belonging to the `ClearOperation` class in the `HazelCast` [1] repository, a popular Java library for data processing. Although we also extracted combinations with an edit distance of three and four, we excluded them from our analysis because, at such distances, the methods are no longer textually similar and are unlikely to be mixed up.

4 Results

We performed an extensive empirical analysis to evaluate the prevalence and evolution of methods with similar signatures and also provided a comparison between conventionally overloaded and textually similar methods. We curated five distinct research questions (RQs) that allowed us to gather insights from the collected data. The following list provides the details of each investigative question:

- **RQ 1:** What is the prevalence of methods with similar signatures in top Java repositories?
- **RQ 2:** For methods with similar signatures, how many instances of such methods exist?
- **RQ 3:** What are the most common methods with similar signatures across repositories?
- **RQ 4:** Is there a correlation between repository attributes and the prevalence of methods with similar signatures?
- **RQ 5:** How often does the number of methods with similar signatures change across the lifecycle of a software repository?

Our research questions can be divided into two distinct categories. RQs 1 through 4 focus on statistical analysis of the present state of the repositories. RQ 5 enables us to perform a longitudinal analysis, allowing us to understand the evolution of methods with

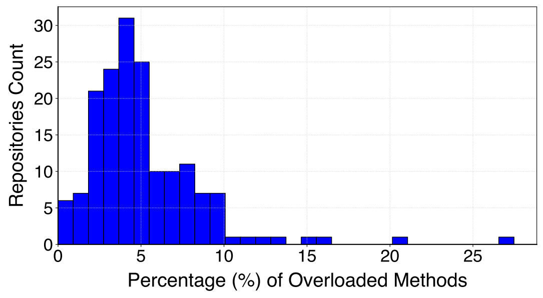
```java
package java.util;
class Array {
    // . . .
    public static void parallelSort(short[] a) {
     int n = a.length, p, g;
        if (n <= MIN_ARRAY_SORT_GRAN ||
        // . . .
    }

    public static void parallelSort(int[] a) {
    int n = a.length, p, g;
        if (n <= MIN_ARRAY_SORT_GRAN ||
        // . . .
    }
```

(a) Overloaded methods `parallelSort` in `utils.Array` class in `OpenJDk11` [2].

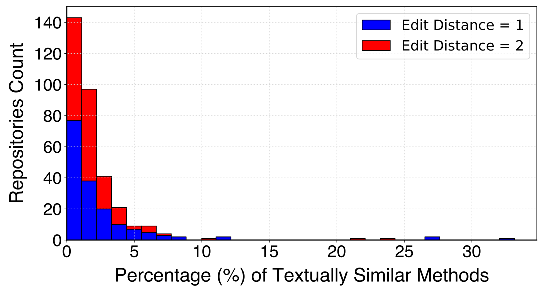
```java
public class ClearOperation {
    @Override
    public int getSyncBackupCount() {
        return mapServiceContext
            .getMapContainer(name)
            .getBackupCount();
    }

    @Override
    public int getAsyncBackupCount() {
        return mapServiceContext
            .getMapContainer(name)
            .getAsyncBackupCount();
    }
```

(b) Textually similar methods `getAsyncBackupCount` and `getSyncBackupCount` in the `HazelCast` [1] repository.

Figure 2: Example instances of overloaded and textually similar methods in classes, extracted from our repository dataset.



(a) Overloaded Methods



(b) Textually Similar Methods

Figure 3: Histogram showing the distribution of repositories based on the percentage of methods with similar signatures. The x-axis represents the percentage of overloaded methods, while the y-axis shows the count of repositories in the dataset.

similar signatures. This approach allows us to identify trends and changes in coding practices, which can serve as indicators of utility and code quality. We next report our findings for each of the research questions.

RQ1 : What is the prevalence of methods with similar signatures in top Java repositories?

To better understand the use of methods that are either conventionally overloaded or textually similar, we performed static analysis on the codebase metadata. We extracted all the methods present in each class across all repositories and then performed an aggregation. We primarily report our results on a repository level for ease of comprehension.

Figure 3a provides insight into the distribution of overloaded methods in our dataset. On average, a total of 5.13% of methods were overloaded per repository. This value is slightly lower than previous work from 2010 [15], showing a slight decrease in the prevalence of overloaded methods in modern repositories. Additionally, it can be noted that the majority of repositories had an overloaded percentage between 0% and 10%, with some repositories having as high as 27% overloaded methods. These outliers exist due to certain repositories having significantly more overloaded methods. For instance, the project `jhy/jsoup` is an HTML parser that uses a visitor pattern that solely relies on Java method overloading to define unique operations for different node types in HTML. As a result, the project uses a high percentage of overloaded methods.

Figure 3b shows the distribution of methods that were textually similar. On average, 4.1% of methods per repository were textually similar, indicating a presence comparable to that of overloaded methods. While the majority of repositories had less than 10% textually similar methods, there were some exceptions with over 30%. One particular outlier was the GraalVM's repository `graal`, a high-performance JDK distribution. The source code of this repository contained variants of the `run` method with different numerical suffixes, likely invoked for parallel computation.

Further observing Figure 3b, there is a notable peak around the lower percentage of textually similar methods, indicating that while such methods are prevalent in many repositories, their percentages are generally low. Additionally, when comparing different edit distances, there are more repositories with a low percentage of edit distance 1 compared to edit distance 2. This is intuitive, as increasing the edit distance value generates more similar method pairs.

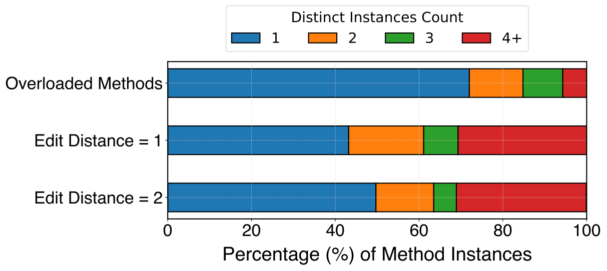


Figure 4: Number of instances of methods with similar signatures.

Finding. *Both overloaded and text textually similar methods have a prevalence in leading Java repositories. Textually similar methods tend to follow a similar distribution trend as overloaded methods.*

RQ2: For methods with similar signatures, how many instances of such methods exist?

Multiple similar methods are likely to cause confusion and potential errors in development. To explore this, we analyzed unique instances of methods with similar signatures. For example, if a method is overloaded multiple times, it can lead to confusion and incorrect method calls. Developers face challenges distinguishing such methods due to subtle differences in input parameters. The same is true for textually similar methods, where a slight typographical error may not raise compiler errors if another valid textual variant method with the same parameters and return type exists.

Figure 4 quantifies the number of instances of methods with similar signatures. For overloaded methods, we look at the number of times a given method is overloaded across all repositories. For textually similar methods, we count how many times a given method appeared in distinct method pairs generated at a fixed edit distance of 1 and 2. 94% of the methods are overloaded three times or less, suggesting a drop-off in the number of methods overloaded a higher number of times. This trend is slightly divergent for textually similar methods. For both edit distance variants (1 and 2), we see that approximately 70% of methods are present in three or fewer distinct pairs, whereas 30% of the methods exist in four or more distinct pairs. This higher percentage suggests a diverse range of method variations with only minor textual differences. This finding suggests as that edit distance among method pairs increases, the overall trend in the frequency of distinct pairs remains consistent, suggesting that when developers are naming new methods, they are likely to choose names that are closely related to those that already exist rather than creating entirely new names.

Finding. *It is uncommon for a method to be overloaded more than three times. However, method variations with small edit distances are more widespread. This suggests developer practices involve naming multiple methods with closely related names.*

RQ3: What are the most common methods with similar signatures across repositories?

Overloaded Methods	Methods with Edit Distance 1	Methods with Edit Distance 2
`visit`	`run*`	`getM**`
`create`	`test*`	`m**`
`malloc`	`get*`	`**activate`
`Buffer`	`set*`	`**serializeStateData`
`fetch`	`component*`	`**register`

Table 2: Top methods with similar signatures in our dataset in descending order. The * represents the replacement, deletion, or substitution of a character in the name.

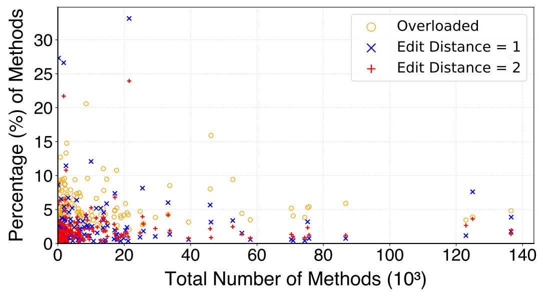
Following up on the previous research question, we study the most frequent methods with similar signatures. This analysis involves extracting the names of the methods, followed by a manual investigation of the codebase to understand the project's purpose. Table 2 enumerates the names of the methods with similar signatures that were most prevalent across the repositories.

One of the commonly listed overloaded methods is `visit`, often used in visitor design patterns. Similarly, the `create` method is associated with the factory design pattern, commonly used to construct appropriate objects in an application. Other examples of prevalent methods include `malloc` and `Buffer`, which are frequently overloaded for variations in memory management, as well as methods related to I/O operations such as `fetch`.

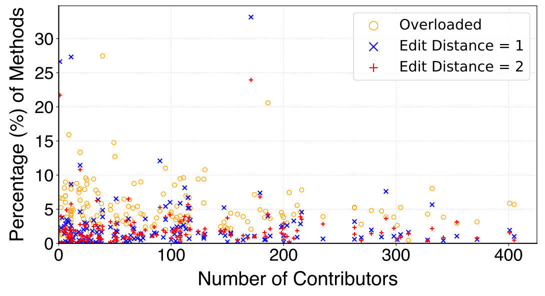
Almost all such methods with close textual similarity were artifacts of naming conventions. When writing code, developers often append a suffix to an existing method name to create a new method with a similar but slightly different name, avoiding the effort to pick a semantically representative name. For instance, the methods `component1` and `component2` in the Java repositories `JOOQ` and `web3j` were used to manipulate and access elements in the tuple structure. Most textually similar methods with an edit distance of 1 fall into this category. Since this practice is often applied to internal private methods not exposed as external APIs, developers are less inclined to follow meaningful naming conventions for such methods, opting to add suffixes. However, prior work has shown that poor naming conventions can significantly harm code usability and maintenance and lead to improper method usage [9, 14].

Another interesting insight into textually similar method names is the extensive use of the camel case naming convention, which uses a mixture of upper and lower case letters without spaces or underscores, e.g., getUserName. Camel case method names tend to have closer edit distances due to the absence of delimiters between words. In contrast, snake case naming uses lowercase letters with words separated by delimiters, which are usually underscores, e.g., get_user_name. As snake case naming results in larger edit distances due to the presence of delimiters, we suggest that when working with larger codebases, developers should adopt snake case as it leads to more distinction between names by increasing the edit distance and consequently improving code readability, while also minimizing sources of error and confusion.

Finding. *Methods with similar signatures are often tied to specific use cases and design patterns, with overloading primarily driven by these requirements. Textual similarity in methods frequently arises from naming conventions like camel case and suffix addition, leading*

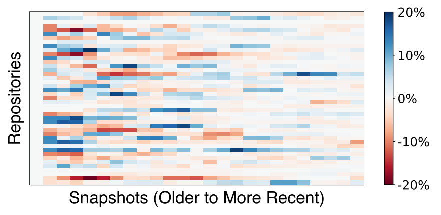


(a) Total Methods in a Repository

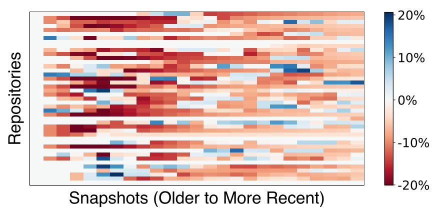


(b) Number of Contributors in a Repository

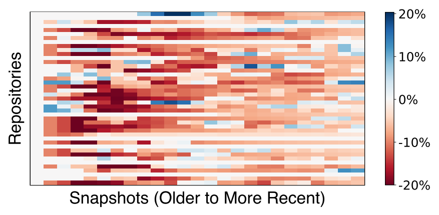
Figure 5: Scatter plot showing the percentage distribution of methods with similar signatures across repository metrics. The x-axis represents the studied metric, while the y-axis indicates the percentage of methods in the repositories.



(a) Overloaded Methods



(b) Pairs with Edit Distance 1



(c) Pairs with Edit Distance 2

Figure 6: Heatmaps showing the percentage change in methods with similar signatures across 25 sampled snapshots spaced over the lifetime of the repositories.

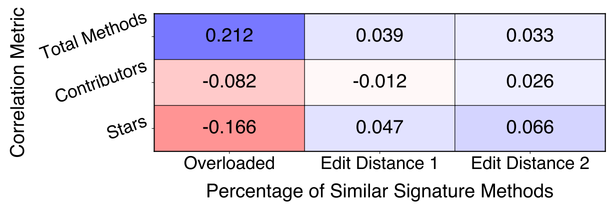


Figure 7: Correlation coefficients across repository metrics.

to poor naming practices that can negatively impact code usability and maintenance.

RQ4: Is there a correlation between repository attributes and the prevalence of methods with similar signatures?

We evaluate how methods with similar signatures correlate with certain characteristics of a repository, such as the total methods, contributors, and repository stars. The total method metric relates to repository growth, whereas contributors are a proxy for collaborative projects, and the stars represent popularity. Figure 7 provides a heat map of correlation coefficients across the studied metrics.

The percentage of methods with similar signatures in a repository shows a weak correlation with both the total number of methods and contributors, with the highest coefficient value being just 0.212. This is understandable, as such methods are usually created due to programming requirements and constraints rather than the collaborative nature or overall growth of the projects.

We further investigate this phenomenon through scatter plots shown in 5. While the total number of methods increases across various repositories, this does not influence the frequency of either overloaded or textually similar methods, as seen in 5a. This indicates that code size does not affect the frequency of these methods, and their percentages tend to stay consistent under 10% for most repositories. This suggests that developers apply overloading judiciously within specific contexts and design patterns rather than as a general practice across all methods. Additionally, textually similar methods are used in specific contexts like getters, setters, or state toggles rather than as a core software design practice such as polymorphism in general-purpose software.

The same is true for total contributors, as shown in 5b. The number of individuals collaborating on a project does not necessarily determine the number of methods with similar signatures. This suggests that the nature of the project influences the methods' signatures more than the number of contributors. For example, simpler projects or those with well-defined scopes may naturally have fewer similar methods, while more complex projects might require more unique and varied methods.

Finding. *The size of the repository, number of contributors, or repository's popularity does not determine the usage of methods with similar signatures. It is based on the programming context, coding structure, and the requirements of a project.*

RQ5: How often do methods with similar signatures change across the lifecycle of a software repository?

We extend our empirical investigation with a longitudinal analysis of the percentage of methods with similar signatures across a repository's entire lifecycle. We take 25 evenly distributed snapshots of a one-quarter random sample of the repositories in our dataset. In total, we apply our analysis to 1075 snapshots across 43 repositories. We aim to learn two insights from this longitudinal investigation: (1) when and why methods with similar signatures were introduced in the repository lifecycle and (2) how the usage of the new and existing methods with similar signatures changes as repositories evolve. Figure 6 summarizes this analysis in three heatmaps, representing the percentage of chance of methods with signature similarity across the 25 snapshots. On a color spectrum of red to blue, red represents a high percentage decrease, and blue represents a high percentage increase.

Figure 6a presents the results from the longitudinal analysis of the usage of overloaded methods. We observe frequent noticeable changes, both increase and decrease, in the number of overloaded methods in the earlier phases of the lifecycle of repositories. Such noticeable changes are rarely seen in the later stages, i.e., more recent snapshots. These observations clearly reflect the evolution and maintenance practices of the software. Earlier snapshots of the repositories often represent *adaptive software maintenance*, which entails adding new features and new functionality [11]. These types of changes warrant the use of overloaded methods owing to software logic and design needs, which are the primary reasons for using method overloading principles.

As repositories mature, the number of overloaded methods stabilizes. This is intuitive because most code changes in mature repositories are *preventative*, *corrective*, and *perfective software maintenance* tasks that include bug fixes and performance optimizations [11]. These tasks rarely require software design changes, and thus, new overloaded methods are rarely introduced during these stages.

Figure 6b and 6c present results from analyzing methods textually similar methods across 25 snapshots. Generally, changes in the number of methods textually similar are observed to be dispersed across the entire lifecycle of the repositories. However, a key observation is that both figures are dominated by red slots, indicating a significant decrease in methods with edit distances of 1 and 2 in their signatures. In most of these repositories, developers tend to remove methods with similar signatures, suggesting that such methods are discouraged as repositories evolve. Initially, many of these methods may be introduced for quick prototyping and during agile software development cycles. However, as the software maintenance phase begins, as early as the initial commit, developers tend to remove methods with similar signatures to reduce ambiguity and API misuse in the code, thereby improving code quality.

In two specific repositories, `awesome-java-leetcode` and `from-java-to-kotlin`, we find absolutely no change in the number of textually similar methods. Upon manual investigation, we find that both repositories are training projects that were initially created by uploading the entire source code in one commit. All later commits only include documentation changes.

***Finding**. Both overloaded and textually similar methods are often introduced early in development, reflecting frequent design changes. As the code matures, maintenance tasks rarely involve modifying overloaded methods, and developers tend to remove textually similar methods, showing a preference for cleaner code.*

5 Discussion

Our study reveals that carefully designed overloaded methods enhance utility by maintaining consistent naming for different use cases. We suggest that developers standardize naming practices early on in large-scale projects to avoid later confusion and fixes. Project managers should consider using build tools like Maven [6] with code style checkers like CheckStyle [24] to enforce stricter naming conventions. Additionally, to prevent misuse, test methods with similar signatures should be implemented in separate classes as part of test suites. We also suggest that developers use automated tools for generating variable names and documentation [16, 18]. These tools analyze methods' code to suggest suitable names and documentation. This minimizes cognitive load, reduces the occurrence of textually similar methods, and ultimately leads to improved code quality from the beginning, ensuring long-term maintainability as the project evolves.

Our study offers a detailed empirical overview of method signature similarity but was limited by computational resources, restricting our longitudinal analysis to a subset of repositories. Additionally, it does not account for the diverse developer practices and coding standards across various teams and projects, which can influence the use and management of method signatures. Building on our current findings, future work will involve user studies where participants program in sandboxed environments with similar method signatures to analyze their impact on usability and error rates. We also plan to identify commits with significant changes in method signature similarity and examine related GitHub issues and commit notes. This will help us understand the reasons and motivations behind these changes, providing deeper insights into the practical implications and informing best practices in software development.

6 Conclusion

This study provides a comprehensive analysis of method signature similarity in Java codebases, focusing on both overloaded and textually similar methods. By examining 167 repositories, we have highlighted the prevalence of these methods and their implications for software usability and quality. Our findings reveal that while method overloading and textual similarity are common practices, they often arise from specific use cases and developer conventions. As projects mature, the occurrence of these methods stabilizes, emphasizing the importance of careful method naming and design choices to enhance code maintainability and reduce potential errors.

References

[1] 2023. Hazelcast: In-Memory Data Grid. https://github.com/hazelcast/hazelcast.
[2] 2024. OpenJDK 11. https://openjdk.java.net/projects/jdk/11/.
[3] Reem S. Alsuhaibani, Christian D. Newman, Michael J. Decker, Michael L. Collard, and Jonathan I. Maletic. 2021. On the Naming of Methods: A Survey of Professional Developers. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 587–599. https://doi.org/10.1109/ICSE43902.2021.00061
[4] Davide Ancona, Elena Zucca, and Sophia Drossopoulou. 2000. Overloading and inheritance in Java. In *2th Workshop on Formal Techniques for Java Programs*.
[5] Mauricio Aniche. 2024. CK: Code Metrics for Java code by means of static analysis. https://github.com/mauricioaniche/ck. GitHub repository.
[6] Apache Maven Project. 2024. Maven - Welcome to Apache Maven. https://maven.apache.org/. Accessed: 2024-06-24.
[7] Antoine Beugnard and Salah Sadou. 2007. Method Overloading and Overriding Cause Distribution Transparency and Encapsulation Flaws. *J. Object Technol.* 6, 2 (2007), 31–45.
[8] Simon Butler. 2012. Mining Java class identifier naming conventions. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1641–1643.
[9] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *2010 14th European Conference on Software Maintenance and Reengineering*. 156–165. https://doi.org/10.1109/CSMR.2010.27
[10] Jürgen Börstler, Ulrike Mettin, Marian Petre, and Marie Nordström. 2018. Developers Talking About Code Quality: The Role of Structure and Complexity in Software Maintainability. *Empirical Software Engineering* 23, 4 (2018), 2075–2105. https://doi.org/10.1007/s10664-017-9571-3
[11] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. 2001. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 13, 1 (2001), 3–30.

https://doi.org/10.1002/smr.220

[12] Martin J. Eppler and Jeanne Mengis. 2004. The Concept of Information Overload: A Review of Literature from Organization Science, Accounting, Marketing, MIS, and Related Disciplines. *The Information Society* 20, 5 (2004), 325–344. https://doi.org/10.1080/01972240490507974

[13] Len Erlikh. 2000. Leveraging legacy system dollars for e-business. *IT professional* 2, 3 (2000), 17–23.

[14] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 286–296. https://doi.org/10.1145/3196321.3196347

[15] Joseph Gil and Keren Lenz. 2010. The use of overloading in Java programs. In *ECOOP 2010–Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings 24*. Springer, 529–551.

[16] GitHub. 2024. GitHub Copilot. https://github.com/features/copilot. Accessed: 2024-06-24.

[17] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. 2019. An empirical study on api-misuse bugs in open-source c programs. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, Vol. 1. IEEE, 11–20.

[18] Xu Hu, Guoliang Li, Xin Xia, and et al. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25 (2020), 2179–2217. https://doi.org/10.1007/s10664-019-09730-9

[19] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady* 10, 8 (1966), 707–710.

[20] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1–12.

[21] Bertrand Meyer. 2001. Overloading vs. object technology. *Journal of Object Oriented Programming* 14, 4 (2001), 3–7.

[22] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Christian D Newman. 2021. Using grammar patterns to interpret test method name evolution. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 335–346.

[23] Armstrong A Takang, Penny A Grubb, Robert D Macredie, et al. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.* (1996), 143–167.

[24] Checkstyle Team. 2024. Checkstyle. http://checkstyle.sourceforge.net/. Accessed: 2024-06-24.

[25] Mari Vartiainen, Riitta Hakala, Eeva Hakulinen, and Lasse Keskinen. 2021. Effects of a Cognitive Ergonomics Workplace Intervention (CogErg) on Cognitive Strain and Well-being: A Cluster-randomized Controlled Trial. A Study Protocol. *BMC Psychology* 9, 1 (2021), 54. https://doi.org/10.1186/s40359-021-00550-3

[26] Ratnadira Widyasari, Zhou Yang, Ferdian Thung, Sheng Qin Sim, Fiona Wee, Camellia Lok, Jack Phan, Haodi Qi, Constance Tan, Qijin Tay, and David Lo. 2023. NICHE: A Curated Dataset of Engineered Machine Learning Projects in Python. arXiv:2303.06286 [cs.SE] arXiv preprint.